From EROS to Coyotos/BitC: Open Source Meets Open Proofs

Jonathan S. Shapiro, Scott Doerrie, Eric Northup, Swaroop Sridhar
Johns Hopkins University

Abstract

While software verification has been used successfully in critical applications such as flight control systems, it remains beyond the state of the art for general purpose operating systems. Unless some way can be found to automate the validation of a general-purpose operating system, maintaining a high-assurance operating system is prohibitively expensive. The essential impediments to software verification are scale (both in size and complexity) and the absence of a systems programming language with a specified formal semantics. The architecture of today's commodity operating systems significantly increases the problems of scale and verification complexity. In addition, the current economic structure of high-assurance evaluation is fatally flawed. The Coyotos effort relies in part on open-source technology to resolve this problem.

The goal of the Coyotos project is to produce the first verified general-purpose operating system implementation. As part of this effort, we are creating a new systems programming language, BitC, that combines the semantic clarity of ML or Scheme with the expressive control of C or C++. The BitC design relies intrinsically on the integration of theorem proving technology into the compiler, enabling the programmer to demonstrate safety properties that cannot be derived by the compiler.

This paper provides an overview of the Coyotos system design and the BitC programming language that we believe will make a verified implementation feasible. In particular, we give examples of the kinds of programming and language idioms that have created difficulty in previous verification efforts, and show how we propose to address them in a system relying on explicitly managed storage and having a large aliasing burden.

1 Introduction

This paper accompanies a talk for the 2005 *Libre Software Meeting* in Dijon, France. It provides an introduction to the Coyotos/BitC project. Coyotos is the name of a new operating system – the successor to the EROS kernel. BitC is a new programming language that we are using to construct Coyotos so that we can apply software verification technology.

In a narrow sense, Coyotos is simply the successor to the EROS operating system.[1] In the larger view, the Coyotos project is attempting to fundamentally change the way high assurance systems are constructed and maintained. Just as open source is changing the economics of software production, we believe that "open proofs" can change the economics of high-assurance system construction. In this paper we will try to provide some motivation and context for the Coyotos project, explain the essential technical difficulties, and describe the approach that we plan to take.

The Coyotos/BitC project is still in its early stages, and much of what we think can be accomplished still remains to be demonstrated. In contrast to our other paper at this meeting, this paper describes a research effort that has not yet shown major results. So far, the work has shown only very preliminary success, and most of this success would be meaningful only to someone who was knowledgeable about software verification. However, we hope that this will change very shortly with the release of the first BitC compiler.

While verification is frequently used in critical software systems, attempts to construct verified

This paper was first published as part of the *Libre Software Meeting*, Dijon, France, July 5-9, 2005

1 Information on the EROS project is available at www.eros-os.org

general-purpose operating systems have not yet succeeded. This paper discusses in detail why we believe that the Coyotos effort may make greater progress then previous attempts.

The Coyotos effort inherits two useful successes from the EROS effort:

- A verification proof that the architecture can enforce a security property known as "confinement."[2] Confinement can be used as a universal foundation for information flow based security policies. This verification means that if we can succeed in verifying the operating system implementation, we will actually have an "end to end" verification of security policy enforcement.
- Our successful experience with model checking certain properties of the implementation itself.[3] It is this work that suggested to us that full verification might be feasible in connection with the EROS kernel.

One of the authors, Scott Doerrie, has nearly completed an effort to automate the confinement proof using the TWELF theorem prover. In collaboration with Eric Northup, Scott has also identified the essential elements needed to prove the correctness of the address translation algorithm in EROS, which is by far the most complex algorithm in that kernel.

Given these early results, and the level of understanding of the problem that they have allowed us to establish, we are now confident that it is feasible to verify substantial security and correctness properties for the Coyotos kernel.

2 The Coyotos Architecture

From an architectural perspective, Coyotos is trying very hard to be a conceptually small step from the EROS architecture. Our goal is to simplify and regularize certain aspects of the EROS design, and to resolve some important issues relative to the EROS interprocess communication (IPC) mechanism. In this section we briefly summarize the differences between the two systems. Readers interested in a more comprehensive treatment of the EROS architecture may wish to read *EROS: A Fast Capability System*.[4] and also *Design Evolution of the EROS Single-Level Store*.[5]

2.1 Minor Differences

There are several minor differences between EROS and Coyotos. In our view, each of these changes is a small refinement to the existing EROS system, and does not introduce significant new architectural or conceptual challenges.

2.1.1 Virtual Registers

One of the challenges in an efficient invocation interface is the need to manage registers. Any invocation arguments must be specified by the application. The argument descriptor vector is large enough that it does not fit in registers on the Pentium family, and on other architectures it would be inconvenient to unload a large number of registers for this purpose.

Coyotos therefore borrows a design idea from L4, and extends the hardware data registers with software-defined "virtual registers." These "registers" are a per-process data area (a page) that is memory mapped into the process address space at a virtual address specified by the process. The kernel maps this page independently, and does not rely on a valid application virtual address for references.

From the kernel implementation perspective, a significant benefit of the virtual register design is that the kernel does not need to perform address validation before referencing these values. While the EROS implementation was able to optimize this validation check efficiently, the larger invocation descriptor used by Coyotos will not be able to easily support the same optimization. In addition, the

2 Jonathan S. Shapiro and Sam Weber. "Verifying the EROS Confinement Mechanism." *2000 IEEE Symposium on Security and Privacy*, Oakland, CA, 2000.

3 Hao Chen and Jonathan S. Shapiro, "Using Build-Integrated Static Checking to Preserve Correctness Invariants," *Proc 11th ACM Conference on Computer and Communications Security*, Washington, D.C., 2004

4 J.S. Shapiro, J. M. Smith, and D. J. Farber. "EROS: A Fast capability System." *Proc. 17th ACM Symposium on Operating Systems Principles*. Published as *Operating Systems Review*, **34**(5):170-185. Dec, 1999.

5 J. S. Shapiro and J. Adams. "Design Evolution of the EROS Single-Level Store." *Proc. 2002 USENIX Annual Technical Conference*. Monterey, CA, 2002.

EROS strategy complicated the page fault path in a way that was difficult to explain and maintain.

Coyotos virtual registers are used both to describe the arguments to an invocation and to provide a small region of "auxiliary registers" for return of data values from the kernel. The design goal is to ensure that all data results returned by the kernel can be returned in virtual registers without requiring an outgoing string operation. This eliminates the complications of validating the destination addresses of outgoing strings in the kernel path.

Virtual data registers provide a second practical advantage. Consider a call such as

```
fd->write(offset, count, buf);
```

our measurements indicate that it is *very* important for the content of the `buf` argument to be received at a carefully aligned address in the server that implements files. If incoming buffer alignment cannot be achieved, an additional copy of the file block is required within the file server, which reduces file server performance by nearly a factor of two.

The difficulty is that the `offset` and `count` arguments must also be transferred. On architectures like the IA-32, there aren't enough hardware registers to transfer these values. They must be added to the "string" argument, destroying the desired alignment. The introduction of virtual registers in Coyotos eliminates this problem in most cases.

2.1.2 Capability Address Spaces

In EROS, every process had a limited number (32) of software-defined capability "registers." This design model was inherited from the KeyKOS system.[6] The idea in KeyKOS was that protection domains (processes) should be small, and that a small protection domain should not require access to a large number of resources.

KeyKOS implemented 16 capability registers per process. We found in EROS that this number was too small: it did not provide a sufficient number of capabilities to efficiently support a standard runtime environment for processes. To address this, we increased this number to 32 capability registers. The basic model of KeyKOS, where processes should have a small number of capabilities, remained unchanged. From the standpoint of implementation, the register model was very convenient, and provided for an efficient capability invocation path. The ability to perform parallel validation of capability register indexes in the capability invocation path was one of the reasons the REOS implementation competed favorably with other high performance IPC implementations, most notably L4.

As EROS matured, we found that the register model was limiting in several regards.

Scratch Registers. A small number of library operations required the use of "scratch registers" for intermediate capability values during computation. Because the EROS process model did not define a capability address space, it is not possible to save capability registers to a "capability stack." In consequence, these library routines effectively imposed global register allocation constraints.

Shared Object Servers As the EROS project started to implement a more complete system, we found that certain object servers needed to retain capabilities to the objects they manage. This is necessary in some cases to support object destruction. Here again, it became necessary to have a large space of capabilities. Initially, we implemented a library to manipulate a tree of capabilities by hand. This is an excellent example of a library that requires *two* global capability registers: one to point to the top of the tree and the other used for tree traversal.

Changes in Coyotos Late in the EROS design cycle, we added support for a kernel-supported address space, but this was never part of the process model, and was never integrated into the invocation path.

In Coyotos, we initially decided to abandon capability registers and use a capability address space instead. Subsequent consideration of how to simplify the kernel interface, implementation, and verification has convinced us that the Coyotos process model should include both capability registers and a capability address space.

Preserving capability registers in the architecture eliminates all of the cases where the kernel must traverse a user address space in order to return a

6 Information about the KeyKOS system may be found at www.cis.upenn.edu/~KeyKOS.

kernel result, which simplifies address space management within the kernel. Capability space traversal must still be performed for *incoming* arguments, but never for capability return values.

2.1.3 Mapping Structures

The EROS system used a software-defined hierarchical mapping structure to describe address spaces. Address spaces are defined as a tree (actually, a lattice) of "nodes" whose leaves are pages. Sharing is permitted at any level of the software hierarchy. The implementation guarantees that hardware page tables are shared wherever the permission mechanisms of the hardware permit correct sharing to occur. Optimizations exist in the representation to allow redundant layers of the translation tree to be eliminated in some cases, but the structure is, in hindsight, relatively inefficient.

Like EROS, Coyotos uses a software-defined address space structure, but the structure has been redesigned to provide faster and more efficient translation. Where EROS uses a lattice of nodes, Coyotos uses a lattice of prefixed address translation trees (PATTs). Each PATT encodes the height h of the sub-space that it dominates. The valid byte addresses within the subspace are from 0 to 2^h-1. Each PATT also encodes a residual value r, which describes the size 2^r of the "holes" *beneath* this PATT. Thus, each individual PATT completely translates the bit positions h:r within an address.

The interesting part about the PATT design is that PATTs are fully associative. PATT translation is accomplished by pattern matching rather than indexing. Each PATT contains 16 capability slots that name sub-spaces (or, at the bottom, pages). These are paired with a vector of 16 patterns that select the appropriate capability slot during traversal. In order to be matchable, a pattern must be a multiple of 2^r (that is, the least significant r bits must be zero) and must be less than 2^h (that is, must match some address whose value does not exceed h significant bits).

The fully-associative property of the PATT structure is expected to provide significant improvements in address translation efficiency. Coyotos processes broadly divide into two categories: processes with a very small total number of pages (often less than 16), which can be described by a single PATT, or more conventional processes that consist of sparse, densely populated clusters of pages. PATTs are efficient for describing both the sparse and the dense part of such structures. In modern instruction sets, the pattern comparison can be unrolled and pipelined so that it is nearly as efficient (on some architectures, *more* efficient) than an indexing computation. Since full associativity reduces the total number of PATTs traversed, this is expected to provide an improvement in translation performance. Clever abuses of the pattern values also allow us to encode user-level fault handlers and background spaces without any need for the "wrapper" nodes of the EROS translation system.

2.2 Major Changes

There are two major architectural changes in the Coyotos architecture as compared to EROS.

2.2.1 Endpoints

The EROS and KeyKOS invocation mechanisms are strongly oriented to interprocess procedure call rather than generalized message send. One of the important results of the EROS effort was the conclusive demonstration that this was a mistake. As discussed in *Vulnerabilities in Synchronous IPC Designs*,[7] there are a variety of ways that synchronous interprocess communication systems can be exploited wherever two parties are not mutually trusting. A broader consequence of that work was the realization that one particular feature of the EROS/KeyKOS design, resume capabilities, was an unnecessary complication given the need for higher level diligence in the usage model. This is very helpful, because resume capabilities were inconvenient to implement.

Where the EROS interprocess communication used capabilities that directly name the destination of an IPC, Coyotos replaces this with a first class communication endpoint abstraction. The sending process performs a send to an endpoint write descriptor. The receiving process performs a receive on an endpoint read descriptor. In comparison to

7 J. S. Shapiro, "Vulnerabilities in Synchronous IPC Designs." *Proc. 2003 Symposium on Security and Privacy*, Oakland, CA. 2003.

the EROS design, this requires one additional argument to the receive phase of the invocation mechanism (to describe the receive descriptor), but in exchange we gain the ability to build services using multiple threads or multiple processes efficiently. Endpoints also allow the set of serving threads to be shrunk or expanded transparent to the client. Finally, a server can use multiple receive endpoints to deal with distinct service classes or services.

This change introduces minor modifications into the Coyotos exception handling protocol. It is expected that the L4ng ("next generation") architecture will also use an endpoint-based design.

2.2.2 Non-Blocking Notification

The EROS invocation mechanism was synchronous and blocking. It provided no simple means for one process to say to another "when you get around to it, there is some data waiting for processing in some previously established shared buffer." Constructing this mechanism at user level requires the introduction of additional threads of control and associated context switches. Our work on high-performance networking indicates that the absence of an atomic notification incurs a 15% penalty on overall gigabit networking performance.[8]

Coyotos incorporates a signal-like event mask into the endpoint architecture. Like UNIX signals, these should be imagined as a software-defined edge-triggered event mechanism. In contrast to the UNIX design, Coyotos events are *not* preemptive; they do not interrupt the normal flow of execution in the receiving process. Instead, events are delivered to the process as a kernel-formulated message when the process next performs a receive operation.

2.2.3 Open Issues

An open issue in the design of the Coyotos interface is the need for per-process watchdog timers. There are many circumstances where processes would like to receive a preemptive timer notification. A scheduler activation design similar to the one used in Nemesis is under consideration.[9] It is unclear how well this approach will integrate with the Coyotos invocation mechanism, and it may turn out that the use case is rare enough that this function should not be directly supported by the process model.

2.3 Summary

While the differences between Coyotos and EROS imply the need for a completely new kernel design, the only large change from a conceptual point of view is the introduction of the endpoint architecture. The verification of the confinement property for EROS should hold equally well for Coyotos, which was a primary objective of the revision. Also, Coyotos retains the "atomic kernel" restriction of the EROS design. As we will discuss below, this restriction is critical for successfully verifying the implementation.

3 The Challenge of High Assurance

Since the early 1980's, attempts have been made to produce high-assurance operating systems and applications. While three operating systems exist that meet the A-1 certification criteria of the U.S. *Trusted Computer Systems Evaluation Criteria* (TCSEC) standard, none were certified or deployed for general-purpose use, and one was canceled without ever coming to market.

With the exception of the Coyotos project, we are aware of no current attempt to produce a *general purpose* system that could be certified at the EAL7 level or better under the *Common Criteria*. Michael Hohmuth is leading an effort to create a verified implementation of L4 in collaboration with the NICTA group at the University of New South Wales, but we do not know whether that project is part of a larger effort to deliver a complete, certifiable system. Until recently, there were two publicly acknowledged efforts in the United States to produce an EAL7 system for specialized

8 Anshumal Sinha, Sandeep Sarat, and Jonathan S. Shapiro, "Network Subsystems Reloaded: A High-Performance, Defensible Network Subsystem." *Proc. 2004 USENIX Annual Technical Conference.*

9 I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns and E. Hyden. "The Design and Implementation of an Operating System to Support Distributed Multimedia Applications." *Proc. IEEE Journal on Selected Areas in Communications*. **14**(7):1280-1297. September 1996.

applications – one being constructed by Green Hills, Inc. and the other by a team at the Naval Postgraduate School in Monterey, California. There are at least two additional projects underway that are not publicly acknowledged. Of these projects, Coyotos and L4 are the only ones attempting a verified implementation.

In early 2005, the EAL7 design objective for the Green Hills, Inc. kernel intended to support the U.S. Joint Strike Fighter (JSF) program has been reduced after concluding that an EAL7-certifiable system cannot be cost-effectively achieved in a timely fashion. It is now alleged in hindsight that high assurance was not really required in the JSF project, but the scale of the software integration challenge, the amount of software reuse, and the large number of software developers involved suggests that this is wishful thinking, and private discussions with several parties involved suggest that this is a case of "*post hoc, propter hoc*." In reality, the Green Hills effort was approached unrealistically and simply cannot achieve any credible high assurance certification. Faced with a choice of reducing objectives or (further) delays to the JSF program, the decision was made to reduce the objectives.

If high-assurance systems are important, it seems useful to ask why they have not appeared in the market. In particular, does the failure lie in the absence of a market, the absence of a technology, restrictions of export control, or the absence of an economically feasible path for a maintainable product deployment?

3.1 Non-Impediments

Until 2000, U.S. export controls on secure operating systems effectively guaranteed that they would not be produced. Software vendors operate in a global marketplace. Since 2000, the responsibility for export approval concerning high-assurance operating systems has been controlled by the U.S. Department of Commerce.[10] Today, there is no meaningful impediment to the export of a secure operating system, provided that is either open source or a "shrink wrapped" (commodity) product. In practical terms, this means that no proprietary secure operating system is likely to succeed, but an open source effort could be made widely available.[11] The challenge today is to get such a system released before the export rules are tightened.

In the absence of any generally-available product offering, one can only speculate about whether a market for high-assurance systems actually exists. In the view of consumers, the major impediment will be issues of software compatibility, but there exist many applications where this is not (yet) a requirement. In particular, the convergence of cell phones, PDAs, MP3 players, and similar devices still has no strongly defined application base, and digital content providers might very well prefer to deploy content on secure devices. There are also markets for distributed applications that cannot be exploited effectively in the absence of an attested platform running an operating system with meaningful security guarantees.

3.2 Real Impediments

The real impediments to high assurance systems are a combination of technical, economic, and social factors.

In my discussions with Brian Snow (formerly with the U.S. National Security Agency) and several people associated with the U.S. National Information Assurance Program (NIAP), the cost of certifying an EAL7 operating system has been consistently estimated in the range of $7 Million (USD). While this number is high, the *real* problem is that it is not a one-time cost.

10 The original 2000 export control revision considered only the liberalization of encryption software, which was restricted under U.S. munitions export control provisions. The same provisions restricted the export of any operating system enforcing mandatory access controls. Comments on the draft regulations by Shapiro led to an initial decision that operating systems enforcing mandatory control would be evaluated as "encryption items" under the revised rules. The rules were later extended to explicitly permit export of secure operating systems as long as these are either open source or commodity systems.

11 A proprietary, shrink wrapped product would need to be sold through consumer channels, where it would be forced to compete with Microsoft Windows and/or Linux. Given the significantly higher cost associated with current high-assurance development practices, and the fact that this product would be incompatible with existing operating systems, it is unlikely that it could compete successfully in a retail sales environment.

In current standards such as the *Common Criteria*, (CC) the highest specified level of assurance calls for a formally specified security policy and system model, and a formal verification that the model enforces the policy. In 1985, when TCSEC was codified, verifying the actual implementation of an operating system was significantly beyond the state of the art. This was still true when the content of CC was frozen. In consequence, CC requires only that the implementation of the system must be "semi-formally" validated in relationship to the model. The term "semi-formal" is not adequately defined in the standard, but is generally taken to mean "with mathematical rigor, but with correspondence arguments made using semi-formal human language rather than mathematics."

The difficulty with semi-formal correspondence is that certification cannot be automated. In particular, the last step of correspondence must be validated by human analysis, which is both expensive and prone to error. Seemingly innocent, local changes in code can have significant non-local consequences. Because of this, each new release of the system requires an expensive re-evaluation and re-certification. The "assurance maintenance" process is certainly not as expensive as the initial effort, but it adds sufficient cost (and more importantly, delay) to the process to make a product non-competitive. Imagine that before each new release of an operating system, the vendor had to pay for an expensive and uncertain six month re-certification process whose outcome might add a year of development effort before the product could be released, to accomplish objectives that were not part of the product specification and had limited value to customers. This was typical in the TCSEC evaluation process, and continues (to a lesser degree) in the CC evaluation system.

Imagine further that the results of this process could be shockingly arbitrary. In the TCSEC process, ambiguities in the standards themselves, coupled with the fact that the evaluators are *not* (as a rule) experienced software engineers, caused significant and expensive differences in opinion about what the standards actually mean and whether a given product satisfies the requirements. It is instructive to read Marv Shaeffer's retrospective comments on the shortcomings of the TCSEC evaluation processes. In the United States, at least, potential vendors of high-assurance systems concluded that the evaluation process was inconsistent with cost-effective or timely product delivery.

3.3 The Essential Issues

Much of the problem can ultimately be traced to three issues:

- The absence of a viable software verification technology.
- A flawed incentive structure in the evaluation process.
- The assumption that the source code of a secure operating system must be proprietary, and

A few years ago, I co-developed with David Chizmadia a course on high-assurance software development. In addition to being an experienced high-assurance evaluator. David was the editor and primary author of the TCSEC "Pink Book," which defined the process for assurance maintenance. The process was educational on both sides, but one conclusion from our effort stands out:

> Unless the *implementation* of the system can be verified to meet requirements, both time to market and life cycle maintenance costs are prohibitive.

Advances in automated theorem provers since 1985 may now have crossed the threshold of viability for fully verified implementations.[12] If high-assurance validation can be effectively automated through verification, the entire structure of the current assurance certification process becomes obsolete for high-assurance systems.

In particular, the economic incentives of the current evaluation process are badly flawed. Today, the vendor pays the evaluator. At least in the United States, the qualify of evaluation under CC has progressively decayed, because the vendors are able to "bid down" the demands of the certification process by making evaluation companies compete for business. What is needed is a system in which

12 See, in particular: J. D. Guttman, J. D. Ramsdell, and V. Swarup. "The VLISP Verified Scheme System." *Lisp and Symbolic Computation*, **8**(1-2), 1995, pp. 33-110.

the *customer* pays for the evaluation – or at least can independently check the result at reasonable cost.

The current evaluation system is designed around the assumption that software is highly proprietary. This assumption seemed plausible in 1985, but must now be called into question. Today, there exists a thriving ecology of open source and "source available" software systems, including operating systems. In 1985, it was assumed that direct validation by the customer was economically prohibitive, and the CC evaluation guidelines do not require that customer re-evaluation be enabled. On the contrary, the trend in the US evaluation process has been to accept an increasing degree of non-disclosure. Today, *even the requirements list need not be disclosed*! The customer must accept the word of a certifying agency paid by the vendor who says that a system meets undisclosed requirements and is therefore trustworthy. This is simply absurd.

3.4 An Alternative Approach

Consider, however, how this market might operate under revised assumptions. Suppose we assume that:

- The source code of the system, including verification objectives, theorem database, and verification system are cheaply available to the customer,
- The assurance objectives are required to be public, *and*
- The process of re-executing a verification, given the theorem database and the verification system, is largely automated.

Under these conditions, it becomes possible to restructure the assurance problem. The role of the certifier is now to certify the *proof objectives* and the system design[13] rather than the implementation. Given these, it is possible for a customer to directly verify the implementation, and even to apply limited local repairs without fear of violating security requirements. It is also possible for the vendor to make changes, revisions, and enhancements within fairly broad parameters. Time to market is no longer impeded by a recurring and arbitrary human process. Total life cycle costs go dramatically *down* because verified software is dramatically less prone to flaws than conventional software.

In short, I am proposing that the best path to realistic high assurance is to extend the concept of open source with "open proofs." Not only must the software itself be openly available, but its formally stated requirements, theorem database and the "proof trail" that demonstrates its correctness (with respect to these requirements) must *also* be openly available. Strictly speaking this could be accomplished with a "source available" approach, but I do not believe that a half measure of this sort is likely to succeed in the marketplace.

Many readers of my column *Understanding the Windows EAL4 Evaluation*[14] have noted correctly that the column is not so much a criticism of the Windows evaluation as an indictment of the CC assurance process. The practical demonstration of software verification in the context of high-assurance creates an interesting dilemma. There is overwhelming evidence that existing *low* assurance approaches simply do not work at all. Existing legacy systems, including Windows, Linux, the BSD systems, and MacOS X, will not be within reach of software verification technology for at least another 30 years – and will not survive the process of verification without change.

The overall objective of the Coyotos project is to demonstrate that this "open proofs" approach is practically feasible, and to provide an operating system built around this concept.

4 The Need for BitC

If we are going to verify properties about programs, it is necessary to know what the programs *mean*. That is, it is necessary to know in a mathematical sense how every expression or statement is evaluated and what effect it has on the program's execution state.

13 And, of course, the existence of at least one proof.

14 J. S. Shapiro, "Understanding the Windows EAL4 Evaluation," *IEEE Computer*, February 2003, pp. 103-105.

4.1 The Problem with C/C++

Regrettably, verification is impossible in C or C++. Both language standards acknowledge a large number of semantic ambiguities in their specifications. For example, the order of argument evaluation during procedure call is not specified, and the behavior of the operators ++ and – is largely left as an exercise for the compiler writer. Issues such as these can be avoided to some extent by programming in a restricted language subset, but this is not entirely satisfactory. The optimizer is entitled to exploit the ambiguities in the specification for purposes of optimization, and is free to rearrange code as long as the result does not violate the language specification.

More problematically, neither C nor C++ are type safe. There is no guarantee that a pointer of type `T*` actually points to a memory location containing an object of type `T`. Worse, there is no guarantee that the object is *still* a well-formed object of type `T` after any *other* pointer operation. In any code sequence of the form:

```
char *s;
T *t;
...
s[3] = 'a';
use t->x
```

it is impossible to know in general whether `x` is a valid member of `t` unless we can establish that `s` and `t` point to non-overlapping locations. Because we cannot rely on the C type system, this must be accomplished by reasoning about memory locations. Unfortunately, the layout of objects in memory is one of the things that is *not* adequately specified by the C and C++ programming language standards.

Michael Hohmuth is attempting to define a semantics for a sufficiently restricted subset of C and C++ for use in the L4ng ("next generation") capability system. It will be interesting to see how well he succeeds, and whether the result can be used successfully by C programmers who are accustomed to relying on unsafe idioms.

An alternative approach has been taken by the SPARK Ada system, which relies on a restricted, unambiguous subset of the Ada programming language. This language has been successfully used in a number of critical applications such as flight control systems. Unfortunately, the restrictions imposed include removing pointers and recursion from the language. After a great deal of thought, we concluded that a Coyotos kernel written in SPARK would be difficult to maintain. Also, it would rely on a closed proof system that would make our "open proofs" objective difficult to demonstrate. Ultimately, the success of SPARK builds on the fact that SPARK is targeted at a narrowly selected (and important) class of applications. For Coyotos, we wanted to build on something more general.

4.2 Why Not ML?

If there is one programming language for which a large body of successful verifications exists, it is ML – or more precisely, the purely functional subset of ML. ML has a precisely defined semantics, a horrible syntax, and a large number of powerful programming features including strong typing, first-class procedures, and closures.

Regrettably, ML is unsuited to systems programming. It has no mechanism for specifying the memory representation of data structures and weak mechanisms for dealing with mutable fields in these data structures. ML simply cannot express some things that are essential in operating system codes – such as the layout of a page table entry. Adding the necessary features to an ML-like language raises subtle and difficult language design issues.

At the runtime level, ML relies strongly on garbage collection to avoid certain semantic difficulties arising from explicit storage management. Garbage collection may be a tool whose time has come for general purpose programming, but there exists no concurrent, real-time collector today with a high enough mutator rate to be appropriate for kernel and low level systems programming. Further, all of the real-time collectors we know about have unusual cases where their real-time guarantees are violated. Our sense as system builders is that we *want* garbage collection for many of the applications we plan to build, but that we also require the ability to write programs such as the kernel that rely on managed storage. As we will shortly see, this is a demanding challenge.

At the implementation level, some of the semantic requirements of ML cannot be efficiently expressed in C. In particular, ML requires "proper tail recursion," so that looping constructs can be expressed using recursion without requiring a very large stack. Unfortunately, generalized tail recursion cannot be implemented efficiently in C.[15] Because we want to use C as an optimizing assembly language for our early compilers, this requirement needs to be relaxed.

Finally, the ML surface syntax is not designed to be embeddable. Because software verification is an essential part of our effort, it is important to be able to write programs about other programs. To do this, it is exceptionally convenient to have a syntax in which a program fragment can be expressed as data, but written in a notation that is convenient to the programmer. Current ML-based theorem provers such as Isabelle/HOL have attempted to achieve this type of syntactic embedding for ML. The results are singularly unpleasant to use.

Historically, the language family that has excelled in this regard is the LISP family of languages, but we require a language that is statically typed. One way to view the BitC programming language is that it is a reworked ML that provides explicit control over representation and mutability, adds a richer base of fixed-precision integer and floating point types, and presents a LISP-like surface syntax to better support embedding. BitC does not quite achieve the degree of transparency that is achieved in LISP, but there is a direct, immediate, and unambiguous translation from the BitC surface syntax to the corresponding AST syntax, and BitC (like Scheme and ML) has a small enough number of syntactic forms to make reasoning feasible.

4.3 Assembly Language

With all of this discussion about programming languages, a word should be said about the issue of assembly language code.

As UNIX demonstrated, there is a very small portion of any kernel that must be written in assembly language. EROS and Coyotos have even less assembly code than UNIX does. These assembly language fragments consist exclusively of short, straight-line sequences with very simple control flow. For example, the register save sequence saves the machine registers and branches immediately to code in a higher-level language. In most cases these sequences are quite short – no more than 40 instructions. In each case, the required behavior is dictated precisely by the specification of the microprocessor's exception handling interface.

In principle, it is possible to analyze such restricted assembly code for correctness.[16] We do not plan to do so. The ultimate purpose of verification is to establish confidence in the correctness of a system. We believe that short, straight-line assembly language fragments can be adequately validated by human inspection.

5 Verification and Coyotos

Coyotos is a capability-based microkernel. It is the successor to the EROS microkernel,[17] and inherits from the EROS kernel both high performance and high security.

For the purposes of this paper, the structure of the Coyotos system as a whole is not greatly important. In this section, we focus on selected architectural properties of Coyotos that make it friendly to verification.

5.1 Mostly-Static Allocation

The Coyotos kernel does not perform dynamic allocation of data structures in the style of typical kernels. At system startup time, the size of system memory is determined and an initial allocation of data structures is performed. Once this initialization-time allocation is established, the total number of data structures of each type is (mostly) fixed, and is not (generally) altered. The kernel proper does not contain calls to either *malloc*() or *free*().

15 See J.F. Bartlett. *Scheme!C: A Portable Scheme-to-C Compiler*. Technical Report WRL Research Report 89/1, Digital, Western Research Laboratory, Jan 1989.
Also: D. Tarditi, P. Lee, A. Acharya. "No Assembly Required: Compiling Standard ML to C." *Letters on Programming Languages and Systems*. June 1992.

16 See, for example, Steve Crocker's dissertation or some of the assembly language algorithm verifications performed by Computational Logic, Inc.

17 J. S. Shapiro, Jonathan M. Smith, and David J. Farber. "EROS: A Fast Capability System." *Proc. 17th ACM Symposium on Operating Systems Principles*. Charleston, SC. 1999

When data structures are reclaimed, they are placed on a type-specific free list.

The reason this is important is that there exist algorithms that iterate over all kernel resources of some type *T*, and it is necessary to know that these algorithms terminate. The simplest way to demonstrate termination for these algorithms is to show that (a) there existed at the start of the algorithm a bounded number of resources of type *T*, (b) this bound did not grow during the operation, and (c) the algorithm visited each of these resources at most *k* times (for some bound *k*).

Of course, when the problem is described in this way it seems obvious that the algorithm must terminate. Note, however, that these requirements can very easily be violated if the kernel performs general dynamic allocation. For example, if a network stack dynamically allocates a new packet buffer, then the set of packet buffers has grown during the current operation. Unfortunately, memory allocation tends to trigger aging algorithms, and aging algorithms tend to iterate over resources, so the addition of a new packet buffer causes the strategy for establishing termination to fail.

There are a small number of cases where Coyotos allows run-time changes to the system resource configuration, primarily involving the configuration of cards that provide a memory-mapped interface. For example, a memory region descriptor for the frame buffer is inserted dynamically by the video driver, and this has the effect of adding "pages" (which are a kernel resource) to the overall system image long after initialization is completed. In a similar way, resources may be erased by removal of a "hot plug" device.

The Coyotos kernel handles isolates these operations into carefully restricted system call paths to ensure that they do not cause a termination bound to be violated. For example, there is no operation that adds a page and also modifies any page in the same operation.

5.1.1 Verification Implications

It is tempting to think that the absence of dynamic allocation (or more precisely, the absence of dynamic *free*) in the Coyotos kernel simplifies verification. The absence of *malloc*() and *free*() does remove the need to consider memory allocation failure, and it eliminates problems of type safety (because the type of storage does not alter when an object is reclaimed) and certain issues of termination. However, it does *not* eliminate problems of alias analysis. No object can be reallocated until it is known that there are no live pointers to the previous object that occupied the same data structure. This turns out to be the essential impediment for verification of managed storage.

However, because storage is typed we need only consider as candidates those pointers that are type compatible with the object being freed. Also, the atomic structure of the Coyotos kernel greatly simplifies the aliasing problem by largely eliminating the need to consider pointers from the stack to the heap (see below).

5.2 Capabilities

Coyotos is a capability system. A capability is a protected pair consisting of a resource name and a set of authorized operations (typically described as "permissions"). In Coyotos, these capabilities are the *only* permissions mechanism recognized and enforced by the kernel.

5.2.1 Representation

Inside the kernel, capabilities have two representations: the "object on disk" format, in which the capability contains the unique object ID of the object that it names, and the "in memory" format, where the capability points directly to the in-memory object. All capabilities to an in-memory object reside on a circular list that is rooted at the target object. This means that if the target object is removed from memory or explicitly destroyed, it is possible to efficiently locate all of the outstanding pointers to the object.[18]

Indeed, *all* of the kernel data structures have the property that they can be pointed to from a very small number of locations, and *most* of the kernel data structures have the property that they reside in

18 To improve capability copy efficiency, we are considering an alternative design similar to the Smalltalk object table design. This would change several data structures in the kernel, but would not eliminate the ability to efficiently invalidate capability-embedded pointers.

some form of circular list that allows these pointers to be efficiently eliminated if the object needs to be freed. As we will discuss below, this is an important property for verification in the fact of managed storage.

5.2.2 Access Checking

The basic rule for determining permission in a capability system may be stated as follows:

> In order to perform an operation on an object, an application must designate a capability that it holds, the capability must name the object, and the requested operation must be permitted by that capability.

Coyotos extends this rule slightly, because it implements a capability address space with an unusual access right (the "weak" right). In Coyotos, the permissions rule is therefore augmented:

> In order to perform an operation on an object, an application must designate a capability (by giving an address) contained in the application's address space. The capability must name the object. The *effective* permissions are computed by taking the permissions encoded in the capability and applying any restrictions expressed along the path. The requested operation must be permitted under the resulting effective permissions.

While there are other ways of expressing permissions, capabilities are a very pleasant mechanism to reason about. The permissions rule given above can be expressed formally as a path traversal in a graph, which is easy to specify formally. Because the code must actually visit each capability in the path, it is relatively straightforward to show that the path restriction is satisfied by the implementation.

A Coyotos kernel operation *accesses* only those objects that are designated by the invocation. This designation is always explicit – even the traversal of the invoking application's address space is directed by some application-specified address. When an invocation is made on an object that is directly implemented by the kernel, at most two objects are modified: the object that is actually invoked by the operation, and the process that receives the result from the kernel.

Interprocess communication may modify a larger number of objects, because string transmission may involve up to 64 kilobytes of contiguous data. However, each modified receiver page is designated by a receiver-specified address range, and each page address within the receive range names a capability (and an address space path) that permits modification.

Surprisingly, the confinement verification does *not* rely on any property other than this type of path-based restriction. The most difficult part of the Coyotos kernel verification is showing the correctness of the memory management mechanism, not the basic enforcement of permissions.

5.2.3 Verification Implications

From a verification perspective, a capability-based system is extremely helpful. Capability access restrictions are defined by path traversals in a graph. Conceptually, this is a problem class for which we have a wealth of formal modeling experience. One of the major previous verification efforts, PSOS, was also constructed on a capability-based design.[19]

From the standpoint of verifying the *implementation* of Coyotos, a path-based model is convenient. The relationship between the resources and capabilities traversed and the algorithms used to traverse them is fairly direct. In particular, it is very easy to show that the resources traversed were all "legal" in the view of the permissions system.

This description identifies a significant difference between verification of security properties and verification of total correctness. It is relatively straightforward to show that the resources traversed in a given operation were permitted by the permissions system. It is considerably harder to show that the *correct* resources were traversed. The distinction lies in the fact that the access check does not care *which* path is traversed, so long as it is legal – an "all paths" analysis is sufficient (and necessary). The correctness check requires a much stronger specification of what constitutes the "correct" path, and a more careful analysis of the algorithm to ensure that this path was actually visited.

19 R. Feiertag and P. Neumann. "The Foundations of a Provably Secure Operating System." *Proc. 1979 National Computer Conference*. pp. 379-334, 1979.

While we ultimately plan to verify selected correctness properties in the Coyotos implementation, our initial focus will be on *security* properties. That is, we will initially show that the Coyotos implementation is safe, even where it is not necessarily correct. Later, we plan to extend our model of the system to allow the definition and verification of total correctness properties.

5.3 Atomicity

Like EROS, Coyotos is designed around the notion of atomic operations. Every kernel operation is indivisible, and the externally visible effects of a kernel operation either happen completely or not at all.[20] Every kernel invocation has two clearly separated phases: the setup phase and the action phase.

5.3.1 The Setup Phase

During the *setup* phase, preconditions are checked, locks are acquired, and objects to be modified are marked dirty. Data structures that have alternative representations may be converted from one representation to another. Internal caches may be modified by loading and flushing entries. All necessary permissions are checked. If necessary, these activities may trigger the aging logic of the kernel and/or initiate the fault-in of objects (conceptually equivalent to reloading pages from the swap area). However, *during the setup phase, no semantically observable modification to system state is permitted.* No error or result codes can be issued during this phase, because no operation has logically occurred.

During the setup phase, it is possible to initiate activities such as aging or object loads that cause the requesting process to block. In Coyotos, processes that are blocked do not sleep with an active per-process kernel stack. On wakeup, a sleeping process restarts its entire system call from scratch. This implies that a sleeping process holds no locks and retains no pointers to kernel resources. A pleasant side effect of this design is that kernel deadlock is impossible.

20 Atomicity cannot be perfectly achieved for system calls involving behavior that is dependent on real-world time or certain processor features such as global performance monitoring, where values may be returned that are not based on observably deterministic behavior.

5.3.2 The Action Phase

On every control flow path through the kernel, there is an explicit place where the setup phase ends and the *action* phase begins. We refer to this point as the "commit point." After the commit point, the operation in progress must complete – either by executing successfully or by returning an error code. Additional resource acquisition is *not* permitted after the commit point. A resource error during this phase is necessarily a design error in the kernel, and results in a kernel halt.

5.3.3 Verification Implications

The atomic kernel design has two extremely useful properties for verification purposes: the empty stack property and a degree of freedom from difficult aliasing problems.

Empty Stack Because sleeping processes do not retain a per-process kernel stack, an atomic kernel design has a property that is very useful for verification purposes: at the beginning and end of each kernel invocation there is no live state on the kernel stack, because the kernel stack is empty.

This property simplifies reasoning about actions such as object destruction and/or pageing. If we can show that an object is not referenced from the heap (including globals), and we can show that an object is not referenced by the remainder of the current kernel invocation, it is safe to delete or remove it.

Alias Freedom There are many difficult types of analyses that arise in verification as a consequence of "aliasing." Aliasing is simply the existence of multiple pointers (or references) to an object. It is very easy for aliases of an object to come into existence, and it is very difficult to determine from static analysis whether two pointers may point to the same object. This becomes a problem when there is some sequence of control flow of the form:

```
ptr1->establishConstraint();
... // intervening procedure calls
ptr2->maybeInvalidateConstraint();
ptr1->actionRequiringConstraint();
```

That is, a sequence where some requirement is established by a call using one pointer, a second call is later made that might cause this constraint to be

violated, and then an operation is performed that requires the constraint to still be in force. An obvious example is paging in an object followed by running the kernel cleaning logic. The difficulty is to establish that `ptr2` cannot name the same object as `ptr1`. This is known to compiler writers as "alias analysis." In some cases, non-aliasing can be difficult to establish even within a single procedure. It is *extremely* difficult to establish the absence of aliasing over a whole program, and pragmatically impossible when there exists a large number of global pointers that might point to these objects.

However, the problem is largely eliminated in an atomic action kernel by other means. Operations that might invalidate a constraint are permitted only during the setup phase, and are divided into two categories:

- Invalidating operations that are not permitted on any object that is required by the current invocation (e.g. Aging).
- Operations that cause the current invocation to be restarted from scratch, guaranteeing that no subsequent reliance on the constraint occurs.

In either case, the original motivation for alias analysis is eliminated.

5.4 Bounded Storage and Time

All operations in the Coyotos kernel have constant bounds on both storage and time. There are no unbounded recursions in the kernel code, and no algorithms that operate over more than a small constant number of objects. This guarantees that all kernel operations terminate, and that the worst-case depth of the kernel stack can be computed. In principle, the kernel code could be automatically transformed to eliminate the need for a stack entirely; the stack exists as a convenience to support higher-level programming languages.

6 Challenges in Verification

Software verification is generally considered to be difficult. It is especially difficult for programs (like operating systems) that make extensive use of global state. More precisely, it is difficult for programs that cannot be viewed as relatively pure state machines. Microprocessor designs are pure state machines, and there have been some very large, very successful verifications of microprocessor designs. Similarly, critical flight control systems are typically designed as state machines, and there have been successful verifications in this domain as well. Both types of systems are characterized by atomic action designs, and the issues in verifying these systems are problems of specification and scale.

Current commodity operating systems typically are *not* viewed as state machines. If a kernel operation can block with a retained kernel stack, then the behavior of the kernel can no longer be characterized as a pure state machine. Instead, it must be characterized by concurrent, communicating pushdown automata with significant amounts of shared state, and we believe that this is the essential reason why verifying their implementation has been intractable. Verification of concurrent programs remains in its infancy, and the combination of concurrency and shared state raises the complexity of verification by several orders of magnitude.

An atomic kernel very carefully straddles the boundary between the pure state machine and the communicating pushdown automata view: there *is* a stack, but it is temporary (and could, in principle, be eliminated by automated program transformation). There is statically shared state in the multiprocessor case, but there is no semantically important *dynamic* sharing because of locking discipline.

6.1 Formal Specification

For any successful verification, both the system state and the meaning of the operations on that state must be formally specified.

6.1.1 Specifying System State

In the context of operating systems, specifying the system state is not unusually difficult, but there are some aspects that may not be obvious.

In particular, the Coyotos specification will need to very carefully deal with the difference between "semantically observable" kernel state and internal kernel state. By "semantically observable," we mean "state that can alter the result of a kernel in-

vocation." This excludes internal state changes such as modifications to software caches that are not exposed by the kernel interface specification.

However, the definition of "exposed" is a bit tricky, because things like cache modifications *are* visible as changes in latency. The presence or absence of an entry in a software cache changes the kernel control flow, and we will need to show that in all cases these paths return the same answer.

Further, we will need to deal with the observability of blocking. If a process sleeps in the kernel, this has observable latency. It may also cause the behavior of other processes to change as a consequence of queuing: process A invokes process B, and blocks until process B is in a receiving state.

Finally, there are elements of the processor such as the cycle counter and performance monitoring registers that may give different answers when an operation is restarted.

Because of these issues, we anticipate that defining formally our notion of "observable system state" will prove to be challenging.

6.1.2 Specifying the Meaning of Operations

In order to answer questions about correctness, we must first define some standard. That is, "correctness with respect to *what*." Each operation performed by the kernel must be formally defined, and the proper range of outputs for each operation under various conditions must be established. This is tricker than it appears, because there are operational results that cannot be defined in terms of the externally visible system state.

For example, if an operation uses an address, then an address space must be traversed. Suppose that somewhere in the traversal we encounter an object that we must bring in from the disk. Suppose further that the sector of the disk has become damaged so that the object cannot be recovered. We must formalize this situation in order to account for one of the error codes that might occur from this operation. At the same time, we would like to know that this error can only arise if the broken object is actually referenced by the operation.

And yet, the external system specification does not really have any notion of disk sectors, which means that we cannot specify these conditions without appealing to knowledge of the implementation – exactly the kind of thing that a specification should not do. One possibility is to formalize the notion of disk storage. Another is to add a boolean value to each object indicating whether the object has become "broken." Deciding where to stop in this type of modeling can be an interesting challenge.

6.2 Termination and State

In most software verification efforts, it is critical to show that programs terminate. Typically, this requirement arises with recursive algorithms, and it is dealt with by showing that (a) the input is finite, (b) each recursive step solves a smaller problem, and (c) eventually, there must be some step at the bottom that generates a result for the minimal case. That is, the strategy is to show that there is an inductive bound on the recursion depth.

Because all of the algorithms in the Coyotos kernel have constant bounded space and time, it would be relatively straightforward to show the existence of such bounds. Unfortunately, operating system kernels rely very heavily on the presence of mutable state, and this forces us to make a difficult choice in our verification approach:

- We could transform the kernel for verification purposes into a "monadic" kernel, in which the global state of the kernel is carried and returned as a procedure argument.
- We can use a "small step" evaluation logic, and examine how the state of the kernel evolves inductively as the steps of the execution proceed.

This is a difficult choice. The monadic transform approach would allow us to use verification techniques that are in some respects simpler, and it reduces the search space that must be examined by the verifier search engine. Unfortunately, the correspondence of the transformed program to the original is *not* obvious to the eye, and we are concerned about maintainability under this approach.

The small step approach is the approach that we have decided to take, primarily because it is better suited to the verification of programs with state.

This approach will certainly work, but it must search a relatively large space of system state.

We are hopeful that the atomic operation nature of the Coyotos kernel will significantly reduce the search space that must be examined by the verifier, Whether this will turn out to be true in practice remains to be seen.

6.3 Proof Objectives

One reason that software verification is difficult has to do with the types of things that people are attempting to prove. Typical verification efforts have focused on showing "total correctness" properties. The term "total correctness" means showing that (a) the answer is correct, and (b) the algorithm terminates. A second type of verification is "partial correctness," where the verification shows only that *if* the program terminates it returns the correct result. In kernels like the Linux kernel (or Windows, or FreeBSD), showing termination is difficult because of deadlock and concurrency issues. In atomic action kernels, showing termination is generally simple.

However, defining correct behavior is more challenging, and we plan to approach our verification efforts in three stages:

1. Show that certain control flow properties are honored. For example that there is a commit point on every path and that no call is restarted after the commit point. These properties have simple specifications, and we know from our earlier model checking work that they can be accomplished.
2. Show that memory safety and type safety are preserved.
3. Show that access checks are performed and information flow restrictions are enforced. This is also relatively simple to specify and check.
4. Begin to look at broader correctness properties (total correctness).

The first set of properties can be achieved using pure control flow reasoning. This allows us to check certain global properties of the implementation that are difficult to validate by hand. The second class comes "for free" from our use of BitC, though it requires us to demonstrate that BitC is type safe. The distinction between (3) and (4) is that in (3) we are not concerned with the question "Is the answer correct?" We are only concerned to determine whether authorized data sources are consulted as input to the computation. In (4) we must define precisely what the correct answer should be.

Our observation from our previous work on model checking is that a great deal of confidence can be obtained from the first three categories. In particular, demonstrations of total correctness are *not* required to verify security properties. This raises the question "where should verification stop?" It may turn out that security properties should be shown by verification and correctness properties should be shown by conventional testing.

7 A Look at BitC

Since we have now mentioned BitC several times, it may be useful to give some sense of what BitC looks like. Originally, our goal was to have a language that a C programmer might look at and say to themselves "I may not like the syntax but I understand how to write efficient programs in this language." We hope that this is still possible, but as we proceeded deeper into the design of BitC the coherency of the language forced us into choices that may not be comfortable to C programmers.

From the programmer perspective, BitC is really two languages that are mixed together in the source code. BitC/P is the programming language. This is the language that is used to write programs. BitC/L is the *logic* language, which is used to write statements *about* programs. The two languages share a common syntactic framework so that they can be mixed in a fairly natural way.

Today, the definition of BitC/P is fairly complete. We are in the process of formalizing its type system and semantics, after which we will define BitC/L. The examples of BitC/L given here should be taken as suggestive – the language may change at any time.

Developers interested in taking an early look at BitC can obtain a copy of Swaroop Sridhar's bootstrap compiler from the Coyotos website.

7.1 A Simple Example

To give a simple example of BitC/P, here is the C implementation of a quasi-factorial using fixed precision integer arithmetic:

```
int qfact(int x)
{
  if (x < 0)
    return -qfact(-x);
  if (x == 0)
    return 1;
  return x * qfact(x - 1);
}
```

The equivalent BitC code would be:

```
(define (qfact x:int32)
  (cond ((< x 0) (qfact (- 0 x)))
        ((= x 0) 1)
        (#t (* x (qfact (- x 1))))))
```

Like ML, BitC makes extensive use of type inference. The result type of `qfact` must be `int32`, because the input argument is of type `int32`, and in consequence the only possible type selections for the remaining variables and operations must be `int32`. It is possible for the programmer to qualify an arbitrary expression by declaring what its result type must be, but allowing the compiler to infer the types helps to automate the propagation of changes when types are revised.

In practice, of course, we would not implement this algorithm recursively. In C, we would write:

```
int qfact(int x)
{
  int result = (x < 0) ? -1 : 1;
  x = abs(x);
  while (x > 0)
    result *= x--;
  return result;
}
```

In BitC, the inner loop is accomplished using tail recursion:

```
(define (qfact x:int32)
  (letrec
    ((qfact1
      (lambda (x result)
        (if (= x 0)
            result
            (qfact1 (- x 1)
                    (* result x)))))
    (qfact1 (abs x)
            (if (< x 0) -1 1))))
```

But in practice this would be written using the "named let" convenience syntax:

```
(define (qfact x:int32)
  (let loop ((x (abs x))
             (result
               (if (< x 0) -1 1)))
    (if (= x 0)
       result
       (loop (- x 1) (* x result)))))
```

The generated code is just as efficient as the C version.

7.2 Differences from C

Ignoring issues of syntax, BitC can be used in a fashion that is quite similar to C. There are three differences relative to C:

- BitC implements tagged unions, and makes it syntactically impossible to use the wrong tag.
- BitC is polymorphic, which has an effect similar to C++ templates, but with stronger and more general type checking.
- BitC has first-class, higher-order procedures.

Polymorphism and higher-order procedures tend to introduce a need for garbage collection. The BitC compiler has a mode in which restrictions are imposed on the escape of higher-order procedures to ensure that no inadvertent storage allocation is performed.

One particular issue in BitC is the need to deal with managed storage. It is important to know that when an object is destroyed there are no live pointers to it remaining in the program. We do *not* expect that such verifications will be feasible in general, and we are not attempting to design a language that can accomplish this. Instead, BitC programs can be compiled in two modes:

- A mode that relies on garbage collection to ensure type safety and object reference safety.
- A mode that allows use of managed storage, but where the compiler requires a human-supplied proof that every object destruction is "safe."

The proofs required in the second category are generally considered to be extremely difficult. Much of the motivation for region-based memory allocation in languages like Cyclone is the desire to avoid these verifications where possible. Because of its atomic system call structure, and because the circular linkage structures provide explicit pointer traceability, we are hopeful that these proof obligations can be successfully discharged for Coyotos in particular.[21]

7.3 Representation

In contrast to Scheme or ML, BitC can be used to describe data structures that require explicit control over representation. For example, the IA32 GDT structure can be declared as:

```
(defstruct SegDescriptor
  loLimit     : (fixint 16 32 #f)
  loBase      : (fixint 16 32 #f)
  midBase     : (fixint 8  32 #f)
  type        : (fixint 4  32 #f)
  system      : (fixint 1  32 #f)
  dpl         : (fixint 2  32 #f)
  present     : (fixint 1  32 #f)
  hiLimit     : (fixint 4  32 #f)
  avail       : (fixint 1  32 #f)
  zero        : (fixint 1  32 #f)
  size        : (fixint 1  32 #f)
  granularity : (fixint 1  32 #f)
  hiBase      : (fixint 4  32 #f))
```

Where the `fixint` form indicates the size and alignment (in bits) of the field and whether it is signed. Given this structure declaration, one can write code that manipulates the bit-level fields of the hardware segment descriptors. For data structures like these whose alignment restrictions cannot be described within the language, BitC provides a mechanism to declare variables as externally defined. This mechanism is considered unsafe, and incurs an obligation for visual (human) inspection of correspondence.

21 When a size-classified allocator can be assumed, Mark Miller has identified a "fat pointer" based design that allows reliable use of managed storage without requiring garbage collection. This approach eliminates the problem of unsafe references at the cost of introducing the possibility of runtime exceptions by using versioned storage allocation in a fashion similar to EROS "allocation counts."

7.4 The Logic Language

The BitC compiler is both a compiler and a theorem prover. In consequence, it is possible in BitC to write judgments in the BitC logic language BitC/L. A BitC program is a combination of program declarations written in BitC/P and proof obligations (judgments) state in BitC/L.

There are three types of judgments that can be expressed in BitC:

1. Judgments about things that are stateless, such as procedures that do not reference mutable global variables.
2. Judgments about state-dependent code that should be true for *all* programs.
3. Judgments about the evolution of state in some *particular* program.

For example, suppose we have the standard definition of lists and lengths of lists:

```
; List with elements of like type:
(defunion (list 'a):ref
  (cons 'a (list 'a))
  nil)

(define (first l)
  (case l
    ((nil nil))
    ((cons x y) x)))

(define (rest l)
  (case l
    ((nil nil))
    ((cons x y) y)))

(define (length l)
  (if (= l nil)
      0
      (+ 1 (length (rest l)))
```

These procedures make no reference to mutable global variables, so they fall into the first category of theorem. It is possible to write things like:

```
; theorem: length of rest <=
; length of list:
(defthm sublist-le
  (>= (length l)
      (length (rest l))))
```

```
; theorem: rest of non-empty list
; is smaller than list
(defthm nonempty-less-smaller
  (implies (> (length l) 0)
           (> (length l)
              (length (rest l)))))
```

As an example of a theorem that should hold for all programs, we might write a theorem about linked list elements:

```
(defthm safety-preserved
  (forall s:(state-of
             '(main ?argc ?argv))
    (implies
      (eval-result '(safe-state?) s)
      (eval-result '(safe-state?)
        (eval `(do-invoke ,?msg) s)))))
```

That is, for all states *s* that are valid states for the program invoked by the expression:

```
(main argc argv)
```

(for any arguments), and for all messages, if evaluating the procedure `safe-state?` yields true for the initial state, then evaluating the procedure `safe-state?` will yield true after the next kernel invocation is performed. That is, operations performed by the kernel are safety preserving.

These examples are admittedly contrived, and the exact syntax for stating theorems in BitC/L is not yet defined. However, this type of theorem introduction and discharge can be built up into very powerful statements about overall execution, and also constraints about intended execution contexts.

Systems like ACL2 and Isabelle/HOL have made it possible to reason about stateless programs (and in the case of ACL2, restricted stateful programs) for some time. The novel aspect of BitC/L is that it integrates into the compiler the ability to reason about stateful programs in unmodified form. The programmer is therefore provided with three alternatives for implementation purposes that can be mixed according to need:

- Restrict programs to constructs that are known to be safe, but may not be suitable for some requirements.
- Use features that are not obviously safe from the perspective of the compiler, but structure the application in such a way that the safety of these idioms can be independently demonstrated by the programmer.
- Use the ability to introduce theorems to supply global program invariants that restrict the use of a type or an algorithm, providing additional checks of correctness and/or usage.

To our knowledge, BitC is the first attempt to integrate general purpose theorem proving into a programming language in a way that provides a continuum of engineering choices for the developer.

8 Conclusion

It is too early to know whether the general verification aspects of the Coyotos/BitC project will be successful. At a minimum, we should be able to verify security properties subject to the *assumption* of type safety, we should be able to reason unambiguously about global control flow, and we should be able to achieve the same degree of verification capability that exists in systems like ACL2. We should also be able to define a language for systems programs where the developer can use unsafe idioms and managed storage, subject to the requirement that these idioms must later be shown to be used in a safe way. If this is all that we accomplish, it will be a substantial step forward toward demonstrably robust system construction.

Ultimately, the test of Coyotos and BitC will be to learn whether we can establish a logic language that is rich enough to enable a fairly complete system specification, prove properties with respect to this specification, and establish a mostly automated verification of correspondence between the system model and the implementation. Coyotos itself has several design and architectural properties that merit optimism, and our discussions with several people more experienced in verification than we are confirm that these properties simultaneously increase the feasibility of verification and reduce the scale of what must be verified.

Ideally, it will be possible to deeply restructure the process of high assurance evaluation, placing the customer back in a reasonable state of control over the acquisition of high assurance and high reliability systems.

Further information on Coyotos can be found online at www.coyotos.org.

9 Acknowledgements

Coyotos/BitC is a team effort. We are grateful for some of the productive interactions we have had with Benjamin Pierce, Frank Pfenning, Michael Hohmuth, and Gerwin Klein.

We are also grateful for an extended set of interactions with the L4ng (next generation) architecture team, including the groups in Dresden and UNSW, and some members from Karlsruhe. We view the convergence of the L4ng and Coyotos designs as extremely encouraging, since it gathers a critical mass of attention on a critical class of system and security design challenges.