NICTA Formal Methods Program

Workshop on Operating Systems Verification

University of New South Wales

5–8 October 2004

Proceedings

Gerwin Klein (editor)

NICTA Technical Report 0401005T-1

National ICT Australia is funded through the Australian Government's *Backing Australia's Ability* initiative, in part through the Australian Research Council.

Table of Contents

Preface

With society's expanding reliance on information and communication technology, the need to increase our level of trust in computing and networking infrastructure is one of the critical challenges facing the ICT field. A number of groups around the world address this challenge by conducting research projects on the development of formally verified operating systems kernels.

The NICTA workshop on operating systems verification brought together invited researchers engaged in these projects. It was held in Sydney, Australia, at the Kensington campus of the University of New South Wales. The formal presentations on 5 October 2004 gave an overview of the state of current research and approaches; the rest of the week was marked by very productive informal presentations and discussions.

These proceedings contain the following workshop contributions. Jonathan Shapiro et al (Johns Hopkins University) attempt to create a verified general purpose operating system implementation, and show why they believe that there is a reasonable chance of success. Kevin Elphinstone (NICTA) introduces L4 as an example microkernel, overviews selected embedded applications benefiting from memory protection (focusing mostly on security related applications), and examines L4's applicability to those application domains. David von Oheimb and Volkmar Lotz (Siemens AG, Munich) describe a framework for modeling and verifying reactive systems, and demonstrate it on two examples: the LKW model of the Infineon SLE 66 smart card chip and Lowe's fix of the Needham-Schroeder Public-Key Protocol. Harvey Tuch and Gerwin Klein (NICTA) use the theorem prover Isabelle/HOL to build an abstract model of the virtual memory subsystem in L4, prove safety properties about this model, and then refine the page table abstraction, one part of the model, towards C source code. Norbert Schirmer (Technical University Munich) develops a general language model for sequential imperative programs together with a Hoare logic, integrated into Isabelle/HOL.

Gerwin Klein
Sydney, November 2004

Towards a Verified, General-Purpose Operating System Kernel[†]

Jonathan Shapiro, Ph.D., Michael Scott Doerrie, Eric Northup, Swaroop Sridhar, and Mark Miller

Systems Research Laboratory
Department of Computer Science
Johns Hopkins University

Abstract. Operating system kernels are complex, critical, and difficult to test systems. The imperative nature of operating system implementations, the programming languages chosen, and the usually selected implementation style combine to make verification of a general-purpose operating system kernel impractical. While security policies have been verified against *models* of general-purpose operating systems, no verification has ever been accomplished for a general purpose operating system *implementation*.
This paper summarizes how we are attempting to create a verified general purpose operating system implementation for **Coyotos**, the successor to the EROS system, and why we believe that there is a reasonable chance of success.

Introduction

The current state of affairs in computer security and reliability is unsupportable. We *must* find ways to build software systems that are robust and survivable, and develop techniques and tools that can bring these development practices into mainstream product development. The problem is foundational: there exists, in principle, no evolutionary path from current operating systems technology to a secure or survivable alternative. Without an operating system on which applications can rely, secure applications and defensible systems are impossible to build.

The only technique currently known that will allow us to build an operating system of the required robustness is formal verification. Verification has been successfully applied to various special-purpose critical software systems (most

notably critical flight control software), and it has become a key part of commercial microprocessor development, but it has not been successfully applied to a general purpose operating system kernel. There are several reasons for this:

- Few operating system designs incorporate a rigorous notion of what constitutes a "correct" or "consistent" state of the system.
- Few people who write operating systems understand verification.
- Current systems programming languages have no formally defined semantics, and suffer from problematic ambiguities.
- Most operating systems have no clearly identifiable "unit of operation" boundaries in the execution of the system where the system state is (alleged to be) consistent. This makes correctness verification difficult or impossible.
- Most kernels use non-preemptive multithreading within the kernel.[1] Even on single processor systems, multithreading creates an exponential explosion of the state space that the prover must consider — far beyond what is currently feasible to verify.
- Imperative programming languages *also* create an exponential explosion of the state space that the prover must consider.
- Provers, with the notable exception of ACL2, might fairly be said to present a "programmer hostile" interface. The language of expression used by the prover needlessly departs from the language used by the programmer, imposing a significant conceptual translation burden on the developer.
- Projects that contemplate the use of verification tools often relegate responsibility for verification to a side team in order to relieve programmers of the "burden" of verification. One result is systems that cannot be verified because their authors don't know how.

Given these issues as initial conditions, it is understandable that verification is not a high priority for operating system developers in the wild. In spite of this, there are at least three results which suggest that verifying a suitably structured microkernel system may now be feasible:

PSOS (1980) While the system was never completed, a substantial framework for verification was crafted for the PSOS system. This work had heavy influence on the subsequent evolution of *nqthm* and later *ACL2*.
See: *A Provably Secure Operating System: The System, Its Applications, and Proofs* [9].

KIT (1989) Bevier's verification of the KIT kernel against a simple microprocessor model is of approximately the same order of complexity as the verification required for a modern microkernel.
See: *Kit: A Study in Operating System Verification* [2].

[1] The UNIX kernel's `sleep()` call, for example, typically causes a context switch into a different kernel continuation.

VLISP (1995) The VLISP project's successful verification of the pre-scheme compiler and runtime system similarly suggests that programs of the size and complexity of modern microkernels should be "within reach" of modern automated provers, provided they can be implemented in a suitable language. See: *The VLISP Verified Scheme System* [5].

The Hopkins *Systems Research Laboratory* is starting work on the **Coyotos** kernel, a successor to the EROS system [13]. As part of this, we are trying to achieve a verified implementation of the kernel and the system's key utilities. We are pursuing this for several reasons:

- It is an intrinsically interesting research challenge.
- We are attempting to build a system that exceeds the requirements for EAL7 evaluation under the *Common Criteria* evaluation scheme. We believe that a fully verified correspondence argument for the implementation is easier to achieve, more rigorous, and easier to maintain than the semi-formal correspondence required for EAL7 evaluation.
- An "open proofs" system (one in which both the system code and the verification of correctness are public) would serve as a public example of how to go about building a robust, secure system. It would allow customers to hire independent experts to validate the verification. It would allow experimenters to attempt changes to code and proof as a learning vehicle.
- An open source, open proofs demonstration that verified systems are possible may fundamentally change both user expectations about critical systems and the "standard of diligence" that must be established to sustain claims of non-liability for critical system software flaws.
- We don't see any other way to get to long-term survivable software systems, especially for critical infrastructure.

For kernels and similar critical systems, we need to know that all operations terminate in a (tightly) bounded number of steps. This means that verification must be concerned with establishing *total correctness* properties. The EROS system is unusual in having a rigorous notion of consistency, an existing formal system model (with a successful paper verification [14]), and a well-defined notion of "unit of operation" (it is an interrupt-style kernel). Bounded time operations, and therefore termination, were a specific and pervasive concern in the EROS design (and its predecessors). Every invocation on the EROS kernel honors the ACID properties, which allows us to express system call semantics as atomic, consistency-preserving transformations on a well-defined system state. Coyotos, the EROS successor, retains these properties and significantly reduces both the semantic and implementation complexity of the kernel.

A key problem we face is the problem of programming language. There exist languages such as ML that are strongly typed and formally specified. While considerable work would be required, it is in principle straightforward to take an approach similar to that of ACL2 [7]: capture a full semantics for an ML language

subset (notably exluding the module system) in an automated prover, and reason about programs written in this subset. Unfortunately, ML and similar languages have several key limitations from the perspective of kernel development:

- They do not provide machine-level, fixed size representation types.
- They provide insufficient control over low-level data layout. In particular, systems codes require the ability to specify both unboxed composite types and unboxed references. This is both a performance and a correctness issue; the layout of certain data structures is dictated by the underlying hardware.
- The incorporation of full tail recursion in the language specifications means that high-performance compiler implementations cannot exploit C as a structured assembly language [1] [16]. This significantly increases the cost of implementing a suitably modified subset of these languages. Fortunately, full tail recursion is not a real-world requirement. In practice, a more constrained form of tail recursion is probably sufficient.
- Most safe languages rely intensively on dynamic memory allocation. In some cases this reliance is embedded so deeply that it is impossible to write programs that do *not* allocate memory dynamically. Kernels must be capable of operating with predictable variance in a fixed-memory environment. Dynamic allocation renders this problematic.
- With the exception of ACL2, no existing language provides means to integrate theorems and their proofs into the body of the program. From an assurance and robustness perspective, these meta-statements about the program are as important as the program itself.

In light of this, a key challenge for the Coyotos effort will be defining a programming language whose unambiguous semantics can be formally specified in mechanical form, is capable of capturing the efficiencies of low-level representation, and can be successfully used by hardcore systems programmers.

The balance of this paper briefly highlights some relevant attributes of the EROS system architecture, our current plans for the evolution to Coyotos, our approach to building an implementation language, and some of the properties that we would ultimately like to verify.

EROS

EROS is a high-performance, capability-based operating system that runs on conventional microprocessors [13]. It minimally requires a processor that provides paged memory management hardware and a reliable separation between user and supervisor execution. The current version of EROS executes on the Pentium processor family. The predecessor system, KeyKOS [4] has been ported to the Motorolla 88000, the IBM System/360, and the Sun SPARC processor families.

Along with the L4 system [11], EROS stands as one of two major remaining microkernel-based research systems. Where L4 has historically focused on sys-

temic performance issues in microkernel-based systems, EROS has focused primarily on security. Where L4 has shown that microkernel-based systems can be fast, EROS has shown that they can also be protected without sacrificing performance.[2]

System Model

The architectural model of EROS is that the kernel provides a protected extension of the underlying microprocessor, augmenting the hardware features with support for kernel-protected capabilities and implementing a canonical interface to the system's memory mapping and exception handling mechanisms using capabilities as the fundamental protection mechanism.

One may view the execution of an EROS system as the steps of a sequential state machine whose transitions consist of:

- Execution of a single, user-mode machine instruction, *or*
- Delivery of an exception notification to a user-level fault handler via IPC, *or*
- Execution of an application-initiated "invoke capability" exception, *or*
- Processing of some pending interrupt event, which may cause a preemption of the current user process.

This view of EROS in terms of an explicit operational semantics is foundational in the EROS security model. The execution of an EROS system begins in a hand-constructed consistent state, and the continued security of the system rests on an inductive argument that every instance of the operations identified above performs an atomic, consistency-preserving transformation on the global system state. To support the logic of this argument, EROS is transparently persistent. Every few minutes, an instantaneous global checkpoint is taken of the entire system state. This snapshot is then incrementally written to disk as execution proceeds. When the system powers up, it resumes execution from the most recently saved checkpoint image.

A key underlying aspect of this model is that EROS kernel invocations are atomic. Every kernel invocation (including IPC) proceeds in two phases:

Prepare During the prepare phase, all required resources are determined to be in memory and are pinned in memory for the duration of the current operation. If an object is to be mutated by the current operation, the prepare phase reserves sufficient space in the system checkpoint area to hold the modified version of the object.

[2] This characterization is not entirely fair. The L4 effort has pursued a number of areas, including real time systems and control of systemic performance, that have not been addresed by the EROS effort.

Action During the action phase, the requested operation is performed. By both design and requirement, the action phase is not permitted to fail. More precisely, the only form of failure permitted during the action phase is to halt the machine. This may occur, for example, if memory is discovered to have an ECC error. During the action phase, the process is not permitted to block, and the kernel is obligated to execute the current invocation to completion.

During the prepare phase, no "semantically observable" modification to the system state is permitted. Changes to kernel caches, rewriting of internal kernel data structures into alternative representations, and queuing of invoking processes on event completion queues in the kernel are not considered to be semantically observable events.[3]

EROS is an interrupt-style kernel. In the event that some action occurs during the prepare phase that might violate a correctness precondition previously established during the prepare phase, the current system call is restarted from scratch. No kernel stack is ever retained by a blocked process. Because no semantically observable mutations have been allowed during the prepare phase, this "abandon and restart" policy is always safe (though it does introduce proof obligations concerning liveness properties).

At some well-defined point on every static control flow path in the kernel, there is a conceptual boundary line that we refer to as the "commit point." This line marks the transitional control point between the prepare phase and the action phase. In the current kernel implementation, there is an explicit call to an inlined, empty procedure at every commit point. This allows us to use static control flow model checking to verify both that semantically observable mutations occur only after the commit point and that no process performs any action after the commit point that might block.

Finally, there is a global design requirement that every kernel path must complete in O(1) steps — that is, within a known constant number of instructions. In fact, we require that this be a (somewhat fuzzily expressed) small constant bound. Indeed, one of the significant changes between the earlier KeyKOS system and the current EROS design was the elimination of the last kernel operation that lacked a small constant time bound.

Though neither the KeyKOS nor the EROS designers realized it at the time, both groups informally but rigorously introduced measure conjectures into the system implementations. With the benefit of deeper hindsight, *all* of the recursive and iterative algorithms of the EROS kernel have straightforwardly stated and veri-

[3] Process en-queuing *is* observable in the form of latency, but this type of observation has no effect on the overt security properties of the machine.

fiable measure conjectures. KeyKOS, with the exception of a single scheduling-related algorithm, also had this property.[4]

Capability-Based Protection and Access Control

Without exception, every operation performed by an EROS application, including the execution of non-privileged instructions, may be expressed as a capability invocation. For normal instructions, the process is implicitly invoking a process capability to itself in order to rewrite its register state. For kernel calls, the capability invoked is directly identified in the invocation. For memory operations, the capability to the object ultimately manipulated (the page) is reachable by traversing a path beginning from the per-process address space capability, and computing the path access rights as an aggregation of the stepwise permissions granted by each capability in the path. Finally, exceptions may be modeled as an invocation of a capability to the appropriate fault handler. In consequence, the permission to perform any action is straightforwardly defined, easily checked, and conveniently accumulated during the traversal of a referencing path that needs to be traversed in any case to locate the target object of the operation.

In most capability systems, the permission accumulation rule is to begin with maximal permission and compute the intersection of these initial permissions with the stepwise permissions as the path is traversed. In the EROS system, the *weak* access restriction somewhat complicates this rule.

The weak access restriction provides a "transitive read-only" permission. There is (transitively) no way to obtain any capability that conveys mutate authority by proceeding from a weak capability. In practice, this restriction is performed stepwise: fetching a capability (even within the kernel) from an object named by a weak capability returns a weakened variant of the fetched capability: one whose access restrictions include both *read only* and *weak*. In some cases, this downgrade is performed conservatively by returning an invalid capability. Because the next capability at each step in the path traversal is conditionally transformed based on the permissions of the currently traversed path prefix, the simple accumulation rule must be replaced by fusing the accumulation of permissions into the operational definition of path traversal. This fusion proves to be useful, as it helps to reduce the possibility of a discrepancy between the traversals *performed* by the machine and the traversals *permitted* by the machine.

The weak access right is *not* essential from the standpoint of expressive power. A system without it can be constructed in such a way as to preserve overt confinement and partitioning. The weak access right *is* essential from the standpoint of resource efficiency and performance. Using the weak access right, for example,

[4] In KeyKOS, the flush algorithm for the meter tree could hypothetically visit every meter node in main memory. While this visitation is bounded by the size of memory, and a bounding measure conjecture can therefore be stated for it, it does not satisfy the "small constant bound" design objective shared by the two systems.

it becomes possible for two processes to share access in copy-on-write form to a common read-only graph of objects, even if the shared graph contains internal, write-authorizing capability references. This proves to be a significant enabler for some common microkernel design patterns, most notably the use of user-defined memory fault handlers. The weak access right also significantly reduces the number of operations that must be monitored and interposed by a reference monitor to implement mandatory access control policies.

The EROS confinement mechanism is constructed on top of the weak right. Lampson defines confinement as inability to communicate over unauthorized channels [8]. The EROS constructor mechanism enforces *overt* confinement (i.e. confinement ignoring covert channels). While this mechanism does not completely satisfy the Lampson definition, the enforcement of covert channel restrictions is largely an orthogonal problem. In EROS, a process is overtly confined *iff* it can be shown that all of its authority to mutate originated with capabilities provided by the instantiating client. That is, all of the *initial capabilities* held by the program instance at instantiation time are (transitively) immutable. This test is implemented by a user-mode, trusted application: the *constructor*. The constructor performs a static test prior to instantiation to validate that all of the immediate initial capabilities (as opposed to those that are transitively reachable from these) are either:

- Trivially safe kernel-implemented capabilities, *or*
- Weak (therefore transitively immutable), *or*
- Capabilities to another constructor that in turn certifies its instantiations as confined. This is acceptable because the constructor is trusted code and one constructor is able to authenticate another. This case provides an inductive extension of the previous two rules, and enables instantiation of complex confined subsystems with rich behavior.

It has been demonstrated in the KeySafe [10] system that the constructor provides a sufficient foundational mechanism to implement mandatory access controls such as multilevel security. Of perhaps greater pragmatic importance, pervasive use of the constructor as a process instantiation mechanism provides a foundation for defense in depth, as demonstrated in the EROS network stack [12] and the EROS trusted window system [15].

Resource Allocation

If we are to reason about the total correctness of a system, resource allocation is a critical concern. In order to return a correct result, a process must have sufficient space and compute time. This introduces a proof obligation concerning resource allocation that must be discharged. Determining resource sufficiency is possible if the maximal resource requirements of all processes are fully known and the system is sufficiently provisioned. This specialized solution can be extended using

temporal non-interference reasoning to further cases, but in general the resource sufficiency problem is intractible.

From the kernel perspective, it is necessary either to reason explicitly about resource allocation or to somehow *avoid* such reasoning. EROS takes the latter approach by eliminating kernel resource allocation altogether. The kernel is responsible for the *safety* of kernel resources, but it is not responsible for the *allocation* of these resources. Responsibility for resource allocation is delegated to (trusted) application level code.

In EROS, this property is slighty relaxed by allowing the kernel to cache protected state for performance reasons. In some sense, this form of caching multiplexes a fixed resource over unbounded usage demand, but the kernel design ensures that every cache can either be discarded without observable semantic consequence (again barring latency) or written back into some definitive representation object that was allocated by a user-level allocator. The end result is that the kernel is entirely deadlock-free.

This design approach extends to address space mapping structures as well. In EROS, the address space of a process is defined by explicitly user-allocated data structures called *nodes*. The hardware mapping tables are constructed by the kernel on demand by traversing the node structures, which are the definitive statement of the mapping. The hardware structures are managed as a discardable cache. In addition to its role in address space definition, the EROS node structure is also used as the persistent representation of process state.

The EROS address space definition approach is in contrast to the L4 *map* operation, which implicitly allocates a kernel mapping database node. The difficulty with the mapping database node is not that it is implicitly allocated,[5] but that its state is definitive and unaccounted. If the mapping database node is discarded, it is not always possible to reconstruct the mappings that depended on that mapping database node. This induces restrictions on application use of the L4 map operation in order to ensure that the mapping database nodes *can* be discarded. To our knowledge, no current L4 implementation treats the mapping database as a cache, and the practical design implications of such treatment have not been explored in current L4-based systems.

From EROS to Coyotos

Coyotos is the successor to the EROS system. While EROS has satisfied most of our research objectives, the system suffers from several practical impairments:

[5] The *map* operation can be implemented in such a way that every *map* invocation allocates exactly one mapping database node, so the database node allocation may be viewed as explicit rather than implicit. The L4 specification, however, does not *require* such an implementation.

– Though it simplifies security reasoning, transparent persistence is not cleanly compatible with translucent network operations.
  Persistence is removed in Coyotos.
– The EROS *node* data structure, which was introduced to support persistence, complicates both the implementation and the specification of the system:
  - In effect, EROS nodes reify capability storage, and require us to reason about kernel memory type safety in layered fashion. While the atomicity properties of the kernel interface make this possible, the constraints are difficult to understand and to reason about (formally or informally), and they significantly complicate the kernel implementation by introducing what may be thought of as cache coherency constraints across different representation caches.
  - EROS nodes do not provide a convenient representation of address spaces. Nodes have 32 slots, and this induces a structural constraint that shared subspaces must be expressed as aggregations of 32^k page units. In practice, this constraint has proven onerous for applications.

  The node structure is replaced in Coyotos by first-class kernel process structures and a new memory mapping structure called a *prefixed address translation tree*.
– EROS and KeyKOS intentionally omitted non-blocking messaging from the system primitive set. Similar effects can be achieved by using additional threads as message posting agents. Unfortunately, the result is both slow and pragmatically complicated. Without non-blocking notify, certain commonly used mutual exclusion patterns involving transmissions combining data and capability payloads through shared memory are very difficult to construct efficiently.
  Coyotos incorporates a non-blocking event posting mechanism.
– The current EROS IPC mechanism does not handle multithreaded receivers gracefully, and therefore fails to adequately encapsulate details of server implementation.
  Coyotos will incorporate explicitly named communication endpoints.
– EROS implemented (in the kernel) per-process capability registers. This proved to be constraining for applications that needed to manage large numbers of capabilities. A capability address space model was introduced late in the EROS design cycle, but was never integrated effectively into the capability invocation operation.
  Coyotos will provide more direct support for capability address spaces.

Explicit communication endpoints are a new feature in Coyotos, but with this exception all of the differences mentioned above are simplifications of the existing system that preserve all of the existing EROS design properties and constraints. The revised invocation mechanism can likewise be implemented without violating the EROS design constraints. While there are "systems" experiments that we intend to conduct with Coyotos, we are explicitly trying to restrict the core Coyotos architecture to refinement and simplification rather than invention.

By far the most significant change in the Coyotos effort is that the kernel implementation, and the re-implementation of critical system services borrowed from EROS, will proceed using a systems programming language with a mechanically specified formal semantics.

BitC: A Language for Systems Programmers

We have identified in the introduction the main deficiencies of existing languages from the perspective of kernel development: the absence of machine-level representation types and data layout control, and the inability to write programs that run without dynamic allocation. Clearly, we require a language with an unambiguous formal semantics that can be mechanically captured.

It is often stated that aliasing is a fundamental impediment to analysis in languages such as C. While true, we suggest that this view is misleading. C introduces many unnecessary aliasing concerns, but the problem of aliasing cannot be eliminated by subsetting the C language. Kernels are inherently alias-intensive programs, and reasoning about the effects of assignments through aliases is an unavoidable part of the problem of kernel verification. For this reason, we have *not* listed alias elimination as a language requirement. Pragmatically, it is extremely helpful to have a language in which idiomatic "false" aliasing can be eliminated or reduced. It is also helpful to have a language in which the use of idiomatically induced assignment can be eliminated, e.g. through use of tail recursion as an alternative to looping constructs.

One advantage to writing a workshop paper *after* the workshop is that the paper has the opportunity to reflect some of what has been learned in the workshop. In our case, the impact has been substantial. At the workshop, we introduced BitC as a language in the intersection between Scheme and C. We added machine-level representation types and C-style structures to Scheme, eliminated operations that allocated storage (including closure values), and prohibited mutation of local variables. One goal of the BitC design was to arrive at a language that could be directly emitted to C in a *very* small code generator — small enough to be credibly validated by inspection. Eventually, we intend to generate machine code directly.

Following the discussions at the workshop, our ideas about BitC evolved significantly. We came to realize that significant transformations on BitC programs would be required to rewrite programs into a form suitable for direct code generation, and that these transformations would ultimately need to be verified. While the kernel subset language must still avoid dynamic allocation, the key issue in the language from a verification perspective is termination reasoning rather than dynamic allocation. This led us to reframe some of our restrictions:

– BitC must enable the developer to straightforwardly author programs that do not dynamically allocate storage (and we need to provide tool support to check this). BitC need not *prohibit* dynamic storage allocation.

– The complete elimination of closure values was excessive. The actual requirement for kernel programs is to prohibit *upward escaping* closure values that capture local bindings (because these require dynamic allocation). There is no difficuty in using closure values defined at top level, or closure values that might be hoisted to top level without alteration of meaning. This alteration allows us to re-introduce some idioms for data structure traversal that must otherwise be expressed less directly.

Subsequent to the workshop, we introduced several new design elements into BitC:

– Parametric polymorphism supported by pattern matching and a type inference mechanism.
– Tail recursion, but limited to the case in which all of the procedures participating in the tail recursion requirement are bound simultaneously in the same LETREC-like form.
– Higher-order procedures, but using a surface syntax that discourages curried invocations. Because the use of escaping closure values is prohibited in the kernel subset, curried procedures cannot be used within the kernel.
– An ML-like tuple and datatype model.
– Vector types

Finally, there was one obvious issue that we addressed only obliquely (as "C-style structures") at the workshop: the need for unboxed aggregates *and unboxed references*. These have now been incorporated into BitC. These features in turn require us to ensure that the temporal scope of references must be bounded by the temporal scope of the referenced object, but this appears to be straightforward.

The provisional result appears to be a language with an unambiguous formal semantics that can be used to implement critical applications (including the BitC compiler). BitC has a clean subset in which the kernel can be implemented. The resulting language should probably no longer be thought of as "an intersection of C and Scheme." Rather, BitC is best viewed as ML with representation types and unboxing, the module system excised, and a parsable, LISP-like concrete syntax. Also, BitC discourages currying in favor of tuplization. As the language design progressed, we gained new appreciation for the "minimal mechanism" character of ML. The foundational semantics of the current BitC language is not substantially larger that than of the workshop version. Matters are now sufficiently far along that we are examining how to introduce measure conjectures and theorem statements into the language, and considering how to mechanically capture the reasoning about termination of an `eval()` procedure when it is applied to a known-terminating program in a language that does not intrinsically impose total types.

High-Level Objectives

Creating a new operating system using a new programming language involves an absurd amount of work. It seems only reasonable to ask: what are we trying to achieve? Before answering this question, it is useful to describe the current context of high-assurance systems.

Our group was initially drawn to verification as a means of increased security assurance. We are dissatisfied with the level of confidence achievable under currently standardized assurance schemes, and would like to establish a stronger foundation for robust and secure systems.

Role of the *Common Criteria*

The most widely accepted statement of security assurance criteria today is the *Common Criteria* [6]. The highest assurance evaluation (therefore highest confidence) level in the *Common Criteria* scheme is known as EAL7. Its requirements may be summarized as:

- Rigorously state your threat model and functional requirements.
- Formally state your security policy and a model of the system. Rigorously state how the security policy addresses your threat model, and how the system model addresses your functional requirements.
- Verify formally that the policy is enforced in the model of the system
- Show rigorously (as opposed to formally) that the implementation corresponds to the model.

The approach is basically sound. The last step can (and should) be strengthened to require formally verified correspondence. The decision to settle for rigorous correspondence demonstration was a pragmatic compromise reflecting the perceived state of the art in program verification circa 1980.

Our group has spent a fair bit of time laying the groundwork for this type of evaluation for EROS. As our understanding of the process has increased, and we have reached several conclusions:

1. The *Common Criteria* process is exceedingly difficult, not because it is conceptually hard to do but because it imposes an overwhelming burden of paperwork. The majority of this paperwork can be eliminated if formal methods are used where merely rigorous methods are currently required.
2. The process as currently defined has limited real-world value. At the end of the day, the customer isn't running the formal system model. They are running the code. Long experience shows that human inspection of code — and we believe this applies to *rigorous* inspection as well — is simply an inadequate source of practical security.
   One solution to this is to extend the use of formal methods all the way to the code.

3. There are serious problems in the *Common Criteria* scheme and also in the evaluation process:
   - A few evaluation requirements of the scheme induce functional requirements that *reduce* the security of the final system.
   - No evaluation guidelines for evaluation above the EAL4 assurance level (soon: EAL5) exist.[6] In consequence, no public confidence is possible because it isn't understood what higher levels of assurance evaluation *mean*.
4. In the absence of widely and publicly deployed systems that have undergone a well-defined high-assurance evaluation process and been demonstrated by practical experience to be defensible, there exists no empirical evidence that the *Common Criteria* process works at all. In light of this, the cost of high assurance evaluation is not objectively justified.
   There is clear evidence from other domains, notably FAA Level-A flight control systems, that the use of full formal methods is an effective means of achieving robustness. It is likely, but not known, that this success extends to situations where proactive attempts at compromise enter the picture, *provided* that the threat model and requirements have been adequately captured. Unfortunately, no technique is known that ensures exhaustive threat or requirement modeling.
5. The *Common Criteria* process embodies a fundamental and irreconcilable conflict of interest: the party who creates the software has fiduciary influence over the party who evaluates the software, and the absence of transparency in the process lends itself to misuse and even abuse.
   Evaluation customers (the software providers) form a "buyers cartel." The absence of a large supply of business for evaluators creates an economic environment in which the diligence of the evaluation itself becomes negotiable. Evaluators are understandably reluctant to confirm this publicly, but it is widely acknowledged privately as a pervasive problem. The quality standards of the U.S. certified evaluation providers have been steadily deteriorating since the *Common Criteria* process was first deployed.
6. Taking these issues together, the *Common Criteria* serves primarily as a means of protecting incumbent vendors to governments rather than a tool for improving objectively measurable security.

As a result of these issues, Shapiro recommended in response to inquiries from members of the United States Senate during the Clinton administration that the U.S. government "evaluated product" purchasing requirement be dropped for all but the most sensitive applications, and that the latter insist on and fund the mechanisms to produce EAL6 or better evaluation processes. The first recommendation appears to have been accepted. The second was not.

[6] As calibration, EAL4 is the current evaluation assurance level of Microsoft's Windows XP (and many other products). No EAL4 system can be reliably deployed in hostile environments (such as open networks).

An Alternative

Taken as a methodology, there is much in the *Common Criteria* that is worth borrowing. We propose to overcome some of its weaknesses by extending the concept of open source systems to open proof. By "open proof," we mean systems in which:

- Source code for the software artifact is publicly accessible.
- A public statement of the requirements met by the system exists in both definitive formal specification and non-normative informal language.
- A full formal verification that the implementation meets these requirements has been performed using a publicly available proof engine.
- The resulting proof trail, sufficient to allow a third party to independently re-execute the verification, is published in machine-readable form.

The last point bears emphasis. In an environment where software must be adapted and customized by the customer, proof checking is insufficient. The customer must be able to re-execute the entire proof process on locally modified versions of the system.

Realistically, we do not expect that software customers will re-execute these proofs, nor that they would understand directly what the proofs mean. We *do* expect that customers facing potential liability in critical deployments may hire domain experts to check the results as part of software acceptance qualification.

Ultimately, our objective is to redefine the standards of acceptable practice in critical software by demonstrating publicly that formal methods are *not* "too hard" or somehow impractical. If we succeed, the "trust me" approach to software security will become economically non-viable.

Verification Goals for Coyotos

The properties that we would like to verify for Coyotos can be divided into low-level (tactical) properties about the implementation and overall system model correspondence properties.

Implementation Properties

Design Rules The Coyotos kernel inherits a (relatively short) list of design rules that help to reinforce both the atomicity and correctness objectives of the kernel. Among these, the most important is the "two phase" rule. We would like to formalize and rigorously check this rule and several others that devolve from it. A few of these have recently been validated using control flow model checking [3].

Access Check Enforcement We would like to formalize the access rules for each type of system object, and verify that the actual implementation honors the capability-defined access rights at all appropriate points.

Semantic Observability A difficult check we would like to validate is to formalize what is meant by "semantic observability" and verify that the prepare phase does not make semantically observable modifications to the system state.

The Constructor Assumptions The constructor verification relies on the assumption that certain kernel-implemented capabilities were trivially safe and that weak capabilities are transitively read only. Both of these assumptions should be verified.

Address Translation Because address translation data structures might violate both the type safe heap of the kernel and the overall security of the machine, we wish to verify that the algorithm by which the hardware memory map is constructed implements a correctness-preserving translation from the software-defined mapping structures.

Serializability In the SMP version of the kernel, we would like to verify at the end of each kernel invocation that there exists some sequential, non-overlapping sequence of kernel calls that can account for the system state.

Memory Safety We would like a kernel that is known to be "mostly memory safe." Certain hardware data structures, most notably the process and memory management structures, necessarily require low-level manipulation.

Space Bank Isolation Contract The Coyotos storage allocator must ensure that no resource is simultaneously allocated to more than one requestor. This so-called "exclusively held" property is foundational for confinement and higher level mandatory policy. It should be possible to formalize and verify this property.

System Model Correspondence Properties

Our earlier work on confinement verification yielded a formal system model in which the system state and the key system operations were formalized in an operational semantics for an abstracted machine. As a practical matter, this formalization was too high level to be useful for correspondence checking of the real kernel implementation.

We would like to create a more detailed abstract system model, and show that there are fairly direct correlations between this abstract system model and our actual implementation. What this means in practical terms is something we are reluctant to speculate on until we understand this part of the process better.

Conclusion

Last year, Shapiro authored a controversial column for IEEE Software entitled *Understanding the Windows EAL4 Evaluation*. While the column was widely

(and correctly) taken as an indictment of the Microsoft evaluation, astute readers recognized in it a much deeper indictment of the *Common Criteria* process and the current state of computer security in the wild. The current state of affairs in both security and reliability is unsustainable.

Progressive adoption of software verification techniques in critical systems offers the possibility of a major improvement in the robustness and security of day-to-day systems. Our hope with the Coyotos project is to demonstrate that these methods are much more realistic today than is widely understood. We intend to craft a set of tools for creating more robust, high-efficiency system code and provide a publicly accessible, well-documented exemplar for how the tool is applied. Along the way, we intend to create an operating system platform that might be suitable for use in critical applications including critical infrastructure, life-critical systems, and operationally critical business applications.

References

1. Bartlett, J. F: Scheme!C: A Portable Scheme-to-C Compiler. Technical Report WRL Research Report 89/1, Digital Western Research Laboratory, Jan 1989.
2. Bevier, W. R.: Kit: A Study in Operating System Verification. IEEE Transactions on Software Engineering. **15**(11). 1989. pp. 1382–1396.
3. Chen, H., Shapiro, J. S.: Using Build-Integrated Static Checking to Preserve Correctness Invariants. Proc. 2004 ACM Symposium on Computer and Communications Security. Oct. 2004.
4. Hardy, N.: The KeyKOS Architecture. Operating Systems Review **4**(19), Oct. 1985, pp. 8–25.
5. Guttman, J. D., Ramsdell, J. D., Swarup, V.: The VLISP Verified Scheme System. Lisp and Symbolic Computation, **8**(1-2), 1995, pp. 33–110.
6. —: Common Criteria for Information Technology Security, International Standards Organization. International Standard ISO/IS 15408, Final Committee Draft, version 2.0, 1998
7. Kaufmann, M., Moore, J. S.: Computer Aided Reasoning: An Approach, Kluwer Academic Publishers, 2000.
8. Lampson, B. W.: A Note on the Confinement Problem. Comm. ACM. **16**(10), 1973, pp. 613–615.
9. Neumann, P. G., Boyer, R. S., Feiertag, R. J., Levitt, K. N., Robinson, L.: A Provably Secure Operating System: The System, Its Applications, and Proofs. Computer Science Laboratory Technical Report CSL-116, 2nd ed., May 1980, SRI International.
10. Rajunas, S. A.: The KeyKOS/KeySAFE System Design Tehnical Report SEC009-01, Key Logic, Inc., March 1989.
11. —: L4 *eXperimental* Kernel Reference Manual. System Architecture Group, Dept. of Computer Science, UniversitÃ¤t Karlsruhe. 2004
12. Sinha, A., Sarat, S, Shapiro, J. S.: Network Subsystems Reloaded. Proc. 2004 USENIX Annual Technical Conference. Dec. 2004
13. Shapiro, J. S., Smith, J. M., Farber, D. J.: EROS, A Fast Capability System. Proc. 17th ACM Symposium on Operating Systems Principles. Dec 1999, pp. 170–185. Kiawah Island Resort, SC, USA.

14. Shapiro, J. S., Weber, S.: Verifying the EROS Confinement Mechanism. Proc. 2000 IEEE Symposium on Security and Privacy. May 2000. pp. 166–176. Oakland, CA, USA
15. Shapiro, J., Vanderburgh, J. Northup, E, Chizmadia, D: Design of the EROS Trusted Window System. Proc. 13th USENIX Security Symposium. 2004
16. Tarditi, D., Lee, P., Acharya, A.: No Assembly Required: Compiling Standard ML to C. Letters on Programming Languages and Systems. June 1992.

Future Directions in the Evolution of the L4 Microkernel

Kevin Elphinstone

National ICT Australia Ltd.
Sydney, Australia
kevin.elphinstone@nicta.com.au

Abstract. L4 is a small microkernel that is used as a basis for several operating systems. L4 seems an ideal basis for embedded systems that possess and use memory protection. It could provide a reliable, robust, and secure embedded platform. This paper examines L4's suitability as a basis for trustworthy embedded systems. It motivates the use of a microkernel, introduces L4 in particular as an example microkernel, overviews selected embedded applications benefiting from memory protection (focusing mostly on security related applications), and then examines L4's applicability to those application domains and identifies issues with L4's abstractions and mechanisms.

1 Introduction

Microkernels have long been espoused as a basis for robust extensible operating systems. A small, efficient, flexible kernel that provides high assurance as to its correctness provides the foundation of the system. System services are provided by applications running on the microkernel, normal applications receive those services via interacting with the system applications via interprocess communication. Such a system is modular, robust as faults are isolated within applications, flexible and extensible via removing, replacing, or adding system service applications. The efficiency of such a system structure has been demonstrated to be sufficiently close to their monolithic counterparts [9], largely as result of improved efficiency of the microkernel's fundamental primitives [19,26].

There are strong arguments for applying the microkernel approach to systems constructed in the embedded space. Embedded systems are becoming more powerful and feature the memory protection required to facilitate constructing protected systems, as exemplified by personal digital assistants, digital cameras, set-top boxes, home networking gateways and mobile phones. These platforms are no longer sufficiently resource constrained to warrant a built-from scratch, unprotected construction approach that forgoes the robustness and re-usability of basing development on an operating system.

An operating system for such embedded devices must be modular to ensure its applicability to a wide range of devices. It must the reliable as even in the absence of safety critical or mission critical requirements, embedded systems are

expected to perform their function reliably, and usually do not have a skilled operator present to correct their malfunctions. It must be robust in the presence of external and local influences, including those of a malicious nature, given a device's potential presence on the Internet or the ability to download and execute arbitrary code. It should provide strong integrity, confidentiality, and availability guarantees to applications on the embedded device both to protect data supplied by the user, and data and applications of the manufacturer, or content and service providers.

These requirements are strong motivation for a microkernel-like approach, as opposed to a monolithic approach to constructing an embedded operating system. A single monolithic operating system that contains all OS functionality is more difficult to assure as it is both larger and requires all OS functionality to be assured at the minimum level required for the most critical component. Fault-isolation is non-existent. Inevitable OS extensions make the situation worse, even to the point of allowing a well designed base system to be compromised or malfunction.

The L4 microkernel might provide a capable basis for an embedded operating system. It is both efficient, flexible, and small. It is currently undergoing formal verification [29] which would provide a high degree of assurance of correctness. One of its goals is to provide a basis for OS development for as many classes of systems as possible: "all things to all people". It has been successfully used in systems ranging from the desktop [9], to those with temporal requirements [8], virtual machine monitors [17], to high-performance network appliances [20]. Such broad success is strong motivation for exploring L4's application to the embedded space.

In the paper, we examine L4's applicability to the embedded space, and hence a potential direction in its future evolution. We first provide some background to L4 in Section 2. When the go on to examine selected application domains for embedded systems that would stand to benefit significantly from a protected operating system in Section 3, and summarized important properties required of an operating system in those domains. In Section 4, we critically examine L4's applicability to constructing systems with the identified properties by examining relevant conceptual model in both past and current versions of L4.

2 L4 Background

L4 is a small microkernel that aims to provide a minimal set of mechanisms suitable for supporting a large class of application domains. The basic abstractions provided are address spaces and threads. A classical process is the combination of the two. Interprocess communication (IPC) is the basic mechanism provided for processes to interact. The IPC mechanism is synchronous, threads themselves are the sources and destinations of IPC, not the process (address spaces) that encapsulates them. The IPC mechanism has a basic form and an extended form. The basic form simply transfers up to 64 words between source and destination in a combination of processor registers and memory dedicated to the purpose,

with the exact combination being architecture specific. The extended form of IPC consists of *typed messages* sent via the basic mechanism which are interpreted by the kernel as requests to transfer memory buffers or establish virtual memory regions.

Address space manipulation is via the *map*, *grant*, and *unmap* model as illustrated in Figure 1. The figure consists rectangular boxes representing address spaces. σ_0 initially possesses all non-kernel physical memory; A is an operating system server; C and D are two clients of A. L4 implements a recursive virtual address space model which permits virtual memory management to be performed entirely at user level. It is recursive in the sense that each address space is defined in term of other address space with initially all physical memory being mapped within the root address space σ_0, whose role is to make that physical memory available to new address spaces (in this case, the operating system server A and another concurrently support OS B). A's address space is constructed by mapping regions of accessible virtual memory from σ_0's address space to the next such that rights are either preserved or reduced.

Memory regions can either be *mapped* or *granted*. Mapping and granting is performed by sending typed objects in IPC messages. A *map* or *grant* makes the page specified in the sender's address space available in the receiver's address space. In the case of *map*, the sender retains control of the newly derived mapping and can later use another primitive (*unmap*) to revoke the mapping, including any further mappings derived from the new mapping. In the case of *grant*, the region is transferred to the receiver and disappears from the granter's address space (see Figure 1).

Page faults are handled by the kernel transforming them into messages delivered via IPC. Every thread has a pager thread associated with it. The pager is responsible for managing a thread's address space. Whenever a thread takes a page fault, the kernel catches the fault, blocks the thread and synthesizes a page-fault IPC message to the pager on the thread's behalf. The pager can then respond with a mapping and thus unblock the thread.

This model has been successfully used to construct several very different systems as user-level applications, including real-time systems and single-address-space systems [21, 5, 10, 9].

Device drivers are outside of the kernel. The kernel enable drivers to run as normal applications by allowing the registers (or ports) required for device access to be mapped into the address space (or port space) of applications. Device interrupts are transformed into messages from apparent kernel threads, they are acknowledged by sending a reply IPC to the identity of the sending kernel thread.

We can see that the basic concepts and mechanisms L4 provides are few, while at the same time, are quite powerful enablers of higher-level systems constructed on the kernel. The few concepts and mechanisms (including the lack of device drivers in the kernel) means L4 is relatively small kernel (10,000 lines of code) that could be a highly assured basis of an embedded system.

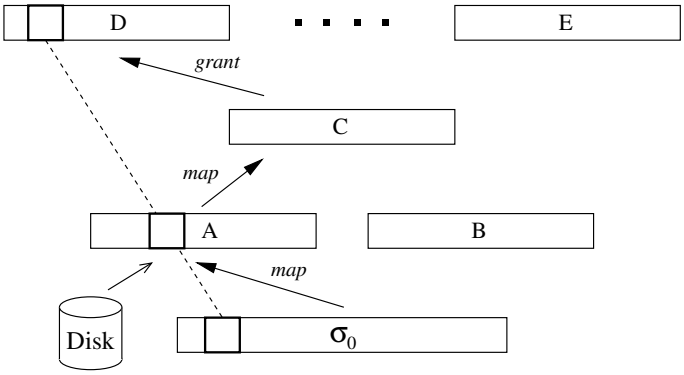


Fig. 1. Virtual Memory Primitives

3 Future Embedded Applications

In this section we examine three application domains of embedded systems: dependable systems, secure systems, and digital rights management. These application domains stand to benefit significantly when the embedded operating system can provide protection between components in the system.

3.1 Dependable Systems

Dependable systems are systems where there is justifiable grounds for having faith in the service the system provides [15]. Dependability is a desirable property of many past, existing and future embedded systems. Methods for obtaining dependable systems can be broadly classified [16] into (or combinations of) the following:

fault prevention where fault occurrence or introduction is avoided,
fault tolerance where expected service is maintained in the presence of faults,
fault removal where the number or impact of faults is reduced,
fault forecasting where the number of present and future faults, and their consequences, is estimated.

As dependable embedded systems are scaled up in terms of overall complexity, the above methods become increasingly difficult to apply. It is essential that it be possible to construct and validate subsystems independently to make the problem more tractable, while at the same time ensure that the validated properties of subsystems remain when they are composed as a whole. This approach is especially applicable to independent subsystems and leads to the idea of *partitioning* or a partitioning kernel [23].

A partition is an execution environment in which an application is isolated from all other activities on the system. One can consider a partition a virtual machine that provides exactly the same level of service to its application independent of other activities on the system. Partitions provide impenetrable barriers

between subsystems in order to guarantee fault containment within a partition. If faults can propagate between independent subsystems, the problem of assuring dependability becomes significantly more difficult. One study reported that the length of fault chains (the sequence of faults leading to a failure) was two or more 80% of the time, and three or more 20% of the time [6]. When fault chains can cross subsystem boundaries it creates extremely complex failure modes that should be avoided if possible.

Partitioning can be divided into *spatial* and *temporal* partitioning. Rushby [23] defines them as follows.

Spatial partitioning must ensure that software in one partition cannot change the software or private data of another partition (either in memory or in transit) nor command the private devices or actuators of other partitions.

Temporal Partitioning must ensure that the service received from shared resources by the software in one partition cannot be affected by the software in another partition. This includes the performance of the resource concerned, as well as the rate, latency, jitter, and duration of scheduled access to it.

One method for achieving spatial partitioning is to use hardware-based memory protection available on a processor complete with memory management unit (MMU). The MMU can be used to control access to physical memory to ensure partitions boundaries are enforced. Note that this assumes that the hardware itself is dependable which in some applications (or hardware arrangements) may not be warranted. A partitioning kernel needs to ensure its own memory is inaccessible, and also enforce a partitioning access control policy between partitioned subsystems. This is analogous to secure systems enforcing a mandatory access control policy which is a mature, well understood field of research.

Temporal partitioning is a much more challenging property to enforce. While resource sharing can be minimized by design, some resource sharing is unavoidable, such as processor time, cache memory, the translation look-aside buffer (TLB), etc. Temporal partitioning is related both to scheduling and security. The scheduling discipline has a direct role in processor time sharing, and indirectly to cache and TLB sharing, and potentially on other shared resources (disk, network bandwidth, etc.). From the field of security, the existence of covert timing channels implies the existence of temporal partition violations. Hence, various techniques for identifying covert timing channels (such as shared resource matrix methodology [14]) are applicable for detecting potential violations of temporal partitioning. However, unlike covert timing channels whose utility can be reduced or practically removed via adding noise to the channel [11], temporal partitioning is violated by any external partition-induced variance in temporal observation of a service. It is clear for the security literature that non-trivial covert-timing-channel-free systems have proved elusive, which implies complete temporal partitioning, while extremely desirable, will prove at least equally elusive.

L4 has been examined previously in the context of dependable systems [3]. It was observed that it has shortcomings in the areas of real-time predictability and in communication control. Solutions that address these issues are proposed

that improve the situation, but do not completely solve the partitioning problem in the context of L4. The most notable omission from the work is any proposals for kernel resource management. Rather than introduce and analyze L4's partitioning issues here, we postpone the discussion to Section 4.

3.2 Secure Systems

A secure system is a system that can ensure some specific security policy is adhered to, usually expressed in terms of confidentiality, integrity, and availability. Security in embedded systems differs from tradition secure systems in several ways as described by Ravi et. al. [22]. Secure embedded systems are constrained in the processing power available, and hence there is a trade-off between the strength of cryptographic algorithms employed by a device and the bandwidth of communications. Battery life is limited and thus security-related processing also limits a devices availability. Embedded systems are potentially deployed in hostile environments, which requires tamper resistance to defend against potentially sophisticated and invasive attacks. An embedded operating system is deployed in a wide variety of application domains, resulting in the need to support a wide variety of hardware and software configurations, which in turn make assurance of security properties more difficult.

Security in embedded systems has received renewed interest with the proliferation of personal computing devices such as PDAs, mobile phones, and the like. Embedded systems vendors have to balance the "apparent" power, features and flexibility the software platform provides to device users against the likelihood of devices being compromised by the users themselves or third-parties. The recent Symbian "cabir" worm demonstrates that embedded environments are not immune to the current mayhem that exists in the desktop personal computer market [1]. A substantial change is required from the large, monolithic, feature-rich, OS and application development cycle. An approach that considers security systematically and holistically is required to avoid history repeating itself.

One approach to address some of the issues described above is the small kernel, small components and small interfaces approach. This is another way of stating the *principle of least privilege* and the *principle of economy of mechanism* [24]. The operating system kernel typically has full privilege on the system it supervises. Applying the above principles strongly argues for a small kernel, with a small, well-understood interface, with careful management of the resources it arbitrates over. Such a kernel is also more conducive to high levels of assurance compared to larger, more complex kernels. A small kernel with appropriate mechanisms enabling security-policy enforcement can provide a system basis that warrants a high degree of confidence in operation.

A small kernel does not necessarily provide a secure system. The same principles that motivated the adoption of a small kernel must be applied holistically to the entire system. Such a system would consist of small components implementing well-understood functionality with the minimum privilege required to do so. The components would provide their services through well understood small

interfaces. Small components may also provide the flexibility required of embedded system to be deployed in widely varying application domains via component composition, substitution, and subtraction.

3.3 Digital Rights Management

The Internet is enabling new methods of content distribution for the entertainment industry beyond traditional methods such as physical media (DVD, CD), or broadcast or cable TV. However, the Internet is also dramatically reducing the barrier to wide-scale copyright infringement. For content providers to embrace the Internet as a distribution medium, they require confidence that the users of their content adhere to the conditions of use of the content. Conditions of use can be represented by a set of rights the end user receives with respect to the content delivered to him. Ideally, content providers would like guarantees that the set of rights granted for content are enforced, and restricted to only those authorized. The concept of specifying, enforcing, and limiting rights associated with digital content is encompassed by the term *digital rights management* (DRM). Note that we have used the entertainment industry as the motivator for DRM, however businesses requiring access-right enforcement for their own internal documents also stand to benefit from digital rights management.

To elaborate on the future role L4 might play in the DRM space, an introduction to a typical generic DRM architecture is warranted. Note that many methods can be used to directly or indirectly perform digital rights management (e.g. watermarking), however we will focus on the architecture depicted in Figure 2.

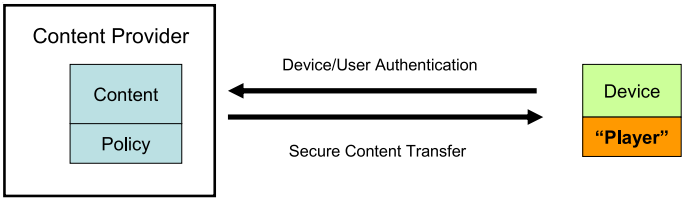


Fig. 2. General DRM architecture

Figure 2 depicts a content provider complete with content and a policy with respect to that content which he wishes respected. The user possesses a device upon which he wishes to view the content. The are many facets to this picture which require solutions prior to the user viewing the content.

– The content provider must be able to specify the policy he wishes respected. XrML [2] is one emerging standard for expression of digital rights, however for the purposes of this paper we assume a policy exists, is expressible, and interpretable by software on the end-user device.

– The user (or user's device) must be authenticated to the content provider. Again, we do not focus on the issue of authentication and simply assume it can be achieved.
– The content (and policy) must be securely transfered to the device. To prevent the content from being stolen by a third party (or even the user itself), the content is usually encrypted to ensure it remains confidential outside the player. Again, we assume this can be achieved.
– The player decrypts the content when viewing is required by the user. The player is expected to honor the content-use policy, not leak the unencrypted content, nor the key to decrypt the content.

We can see that successful enforcement of the content use policy is contingent on the player (where the content is in plain text form) respecting the policy. Content providers have in the past placed their trust in hardware solutions such as satellite TV set-top boxes where their single purpose nature, trusted manufacture, and tamper resistance of the device has mostly proved sufficient to justify the content provider's faith in their ability to honor the content provider's use policy.

One can see that in a general-purpose computing environment, where the user has complete control of the device, the content-use policy can be violated by the end user in many ways, ranging from reverse engineering the player, modifying the player, or running the player on a modified operating system such that is renders the plain text content to a file, hence the reluctance of content providers to widely embrace the Internet as a distribution medium.

One approach to tackling this problem (as exemplified by Microsoft's recently renamed NGSCB [4]), is to provide the content provider assurance that a trusted player on a trusted operating system is the only software that has access to the plain text content. The fundamental idea is to have tamper-resistant hardware provide direct or indirect *attestation* of the software stack required to view the content. The hardware attests that the software running is what it claims to be (alternatives include hardware only exposing decryption keys to trusted software). If the content provider has faith in the identity of the software stack, it is in a position to determine if is trusts the particular software stack to honor the provider's content-use policy.

Figure 3 depicts an exemplary OS architecture for DRM that support both a legacy OS with its applications and legacy kernel extensions like device drivers, and a new trusted mode of operation expected to enforce DRM policy. Very briefly, the system functions by introducing a new trusted processor mode that is more privileged than *kernel* mode. Hardware enforces a boundary between the trusted mode and all other modes (including precluding DMA from untrusted mode devices and their drivers). As expected, only the trusted mode kernel can influence what is in trusted mode or not. The secure storage chip provides attestation for the trusted kernel, which can in turn attest to the trusted nature of the applications it support, forming a chain of trust back to the tamper resistant hardware. Therefore, content providers can obtain assurance that the

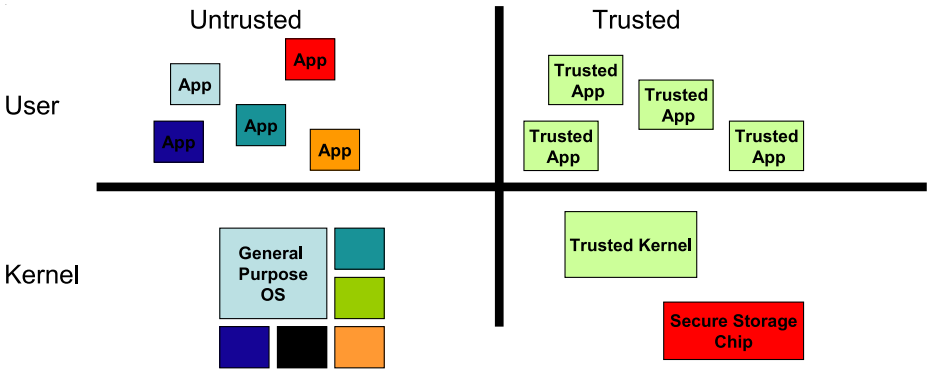


Fig. 3. An example digital-rights-management operating system.

player of their content has a chain of trust rooted in hardware, and hence can expect their content policy to be honored.

A major issue with this approach is that trust is really a label applied to software running in trusted mode, it is not a guarantee that it will always behave in a trusted manner. One would expect that as more and more software acquires trusted status, eventually the trusted partition will approach the size and complexity of the existing legacy system, unless an alternative construction paradigm is employed. The original motivation for kernel mode was to enable the execution of untrusted applications in a controlled way, trusted mode is little different.

A promising approach to building a secure *trusted* DRM OS is similar to the approach for building a secure system in general — small kernel, small components, and small interfaces, as outlined in Section 3.2. While only those components authorized would be permitted to execute in trusted mode, trusted mode itself should be a secure system in its own right, capable of defending itself against compromised trusted applications. In addition to security, the trusted-mode kernel needs to participate in the attestation process. Given an attested secure kernel, content providers can have a high level of confidence in their content-use policies being honored.

3.4 Summary

We have examined three important application domains for embedded systems possessing hardware memory management functionality: dependable systems, secure systems, and digital rights management capable systems. All three application domains require very similar properties from an operating system kernel for that application domain. A secure kernel capable of enforcing confidentiality, integrity, and availability policy for the kernel services itself might be a capable basis for all three application domains. A small secure and assured kernel when combined with a set of application domain specific operating system components

running as applications on the secure kernel is a promising direction to explore in developing a new embedded operating system for future embedded applications.

4 Impediments to L4's Adoption in Secure Embedded Systems

While L4 has been successfully employed as a research vehicle, and as the basis of systems in a variety of application domains, it has not been targeted specifically for secure systems with strong confidentially and availability requirements. Broadly speaking, the current L4 version has serious issues in the areas of communication control and kernel resource management, for which mature solutions are yet to emerge.

4.1 Communications Control

Interprocess communication forms the basis of all explicit interaction between processes running on L4. Note that shared memory regions are established via IPC, hence includes such interaction. To provide a basis for secure communication requires at least:

- control of the set of potential destinations that a process can send to, which implies control of the set of senders a process can receive from. Ideally, knowledge of the existence of other processes is limited to those processes to or from which communication is explicitly authorized.
- An unforgeable identifier must be delivered with the message to enable authorization to be performed in the recipient.

There have been at least five models investigated resulting in implementations in at least 3 cases. The models are *clans and chiefs* [18], *redirection* [13], *redirectors* [28], *virtual threads* [27], and *pclans* [3]. We will briefly examine each in turn and raise issues with them.

Clans & Chiefs The basic system (ignoring clans & chiefs for ease of introduction) consists of threads with in processes. Each thread within the system has a unique system-wide identifier which is used to specify the destination thread for IPC, can be used for authorization of requests in the recipient by receiving the sender's identify. Such a system provides integrity via the unforgeable thread identifiers, however confidentiality policy is unenforceable as any threads can communicate if they can guess the destination thread identity, a small, easily scannable name space.

To control communication flow, clans & chiefs introduces the idea of a *chief* of a *clan*. Every process has a chief statically assigned to it on process creation. The set of processes assigned to a chief is referred to as its clan.

Communication within a clan is unrestricted as before. Communication across a clan boundary is redirected to the chief for inspection. The chief can act as a

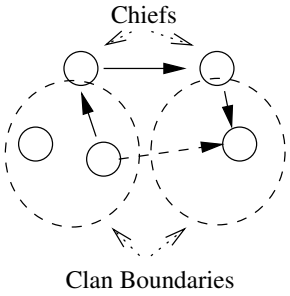


Fig. 4. Clans & Chiefs

reference monitor and enforce a communications policy between clans. In order to monitor transparently, chiefs are permitted to forge the sender identifier received by the recipient in a controlled way. A chief can forward a message that is redirected to it by impersonating the sender if and only if the apparent source and intended destination lie on different sides of a clan boundary.

In the most general case, each process has its own chief which mediates all communication sent and received by the process. Chiefs in such a scenario can enforce confidentiality policies[1], integrity is based on the integrity of the chiefs a message traverses which is determinable by the eventual message recipient (though normally the monitoring chiefs are within the trusted computing base).

The major issue with clans & chiefs is that of performance. In the general case, at least three IPCs are required between a source and destination: source → source's chief → destination's chief → destination. IPC can be avoided by placing processes with the exact same security classification within the same clan, however, as soon as the classification differs, the processes must be in distinct clans and suffer the penalty of extra IPCs via the chief. A smaller issue with clans & chiefs is that they are assigned statically at process creation, and hence cannot be changed if the security policy modified (and thus modifying the security classification of processes).

Having a chief on the IPC path between source and destination also changes the semantics of IPC, as IPC completion at the sender no longer implies delivery with a chief interposed on the path between source and destination [12]. Proposals to address this issue include postponing sender completion until delivery at the eventual destination which results in an unduely complex IPC model that seems prone to denial-of-service attacks. An alternative is to assume intermediaries are in place for all IPC, which implies delivery acknowledgment IPC in cases where the sender requires notification of successful message delivery. Neither solution is entirely satisfactory.

[1] We acknowledge and ignore for now that thread identifiers are allocated within global name space which could form a covert channel.

Redirection The redirection model was proposed and implemented to address the shortcomings of the clans & chiefs model. The model provided a mechanism that enabled for each potential source-destination pair of threads in the system, that IPC between the two could be disallowed, allowed, or redirected to an intermediary which could perform monitoring in a similar fashion to chiefs. One strong advantage of redirection is that it can enforce basic communications control without requiring an intermediary to be in place to forward or discard messages, thus the issues raised previously regarding preserving IPC semantics with intermediaries can be avoided. Another advantage was that redirection is dynamically configurable.

The major issue with redirection is that thread identifiers are still allocated in a global name space which will be prone to covert channels. Another issue is that if intermediaries are required for monitoring, a method for transparently forwarding messages is required. The restriction on impersonation required for forwarding is that an intermediary can impersonate a source to a destination if the intermediary is on the path of intermediaries between source and destination. This check is no longer as simple as the trivial clan & chiefs check, as it requires a search (hopefully short) for membership of a node within a path of a graph. However, in the worst case the length of the path is only bounded by the number of threads in the system. The issue of preserving IPC semantics in the presence of intermediaries, as described with clans and chiefs, also remains.

Redirectors The redirector model has been implemented in *L4Ka::Pistachio*. The basic model is that each process (termed *address space* in Pistachio) has a redirector which can be *nil* or an intermediary. If the redirector is nil, IPC is uncontrolled. If an intermediary is specified, all cross-address-space IPC is redirected to the intermediary independent of the destination. The intermediary can perform monitoring, auditing, debugging, etc.

Redirectors is a simplified model of redirection that avoids the bookkeeping and lookup required to redirect on a source-destination basis. However, doing so requires the single intermediary to handle all monitoring functionality required on any path from a source, as opposed to having potentially separate monitors that enforce security policy, audit, debug, etc. Typically, the intermediary was a single central OS personality, and requiring a single intermediary was not problematic. Control of communication without an intermediary in place is not possible, unlike with redirection.

Redirectors suffer from most of the issues described for clan & chiefs and redirection. If any communication control is required on a source, an intermediary must be used for all communication from the source. Hence in the general case (assuming a single central intermediary), at least two IPCs are required per message for delivery. Having intermediaries in place changes the semantics of IPC. The check of permitting impersonation to enable forwarding has similar problems to the check for impersonation in redirection, it can result in searching a chain. The name space of thread identifiers is a likely covert channel.

Pclans Pclans is a hybrid between clans& chiefs and redirection. Each pclan has at least one process within it, and each process is a member of exactly one pclan. Within a pclan, communication is uncontrolled. Communication across a clan boundary is dealt with by a model similar to redirection, where an IPC can be blocked or forwarded to an intermediary. The motivation for pclans is based on the assumption that there will be significantly fewer pclans than processes, and hence the redirection table will be significantly smaller. Even if this assumption is true in general, some of the issues associated with redirection remain: covert channels over thread identifiers, semantics of IPC with intermediaries, and the requirement of an intermediary even if communication is permitted to processes external to the clan.

Virtual Threads Virtual threads is a model in which threads are named by virtual identifiers in a processes' local *thread space*, not by global system-wide thread identifiers. Note that processes are not identified explicitly, all communication is between threads whether it is intra- or inter-address-space. Each process's thread space is managed using mechanisms similar to the mechanisms provided to manage a process's virtual memory address space. Access to a thread is given by *mapping* or *granting* a reference to the real thread. The reference to the real thread is placed in and referred to by a location in thread space, its virtual thread identifier. The references to real threads in thread space can be considered IPC capabilities to threads that are indistinguishable in the recipient. Access to the thread can be removed via *unmapping* it.

To distinguish between senders, the IPC call delivers the index of the sender's virtual identifier in the recipient's thread space, if it exists, otherwise the IPC is denied. To speed up the search for the appropriate index and to distinguish between potential aliases of the sender in the recipient, the sender is expected to specify the thread index of itself in the recipients thread space.

Given that virtual threads implements a many-to-one mapping between virtual thread identifiers in thread space and actual instances of threads, one can use it to permit, block, or redirect IPC by controlling the mapping from thread space to threads. By having a local name space, a potential covert channel via global allocation of thread identifiers is avoided.

The main issue with the virtual thread model is requirement for the sender to provide its virtual identifier in the recipient. The coordination of name spaces required is cumbersome, precludes name space re-arrangement in the recipient, makes transparent insertion of intermediaries problematic, and creates a shared name space between all potential senders to a destination (a potential covert channel). In general, sender provided identifiers violate the principle of encapsulation of implementation of the recipient.

Summary It is clear that existing and proposed communications control mechanisms are unsatisfactory for secure communications control. The proposal closest to being satisfactory is virtual threads, however, requiring sender-provided iden-

tifiers required to be valid in the recipient's thread space violates encapsulation of implementation of the recipient.

If the virtual thread model was modified such that the virtual identifiers themselves were distinguishable (not the sender itself), then the distinguished virtual identifiers could be used for authorization. In such a system, we are not actually dealing with virtual identifiers, but distinguished capabilities conveying the right to IPC to a particular thread. Such a capability-based IPC authorization model appears to be the most promising direction to explore in providing L4 with a secure communication model.

4.2 Resource Management

Precisely controlled resource allocation for kernel operations is a requirement for secure system construction. Poor resource management within the kernel can lead to denial of service when resources are exhausted, or covert channels when resource availability is widely visible.

The default resource allocation for L4 is a central allocator that allocates from a fixed memory pool allocated at boot time. The default kernel makes no claims to being suitable for an environment with strong confidentiality or availability requirements. In fact, it is trivial to mount a denial-of-service attack on the kernel-memory allocator.

Given the well-known limitations of the existing allocator, alternative strategies have been proposed. The initial proposal [19] was motivated with the goal of preventing denial of service attacks when executing downloaded web content. Its approach is to introduce a mechanism to provide physical memory (frames) to the kernel for a specific process in the event of resource exhaustion, which I will term the *lend to kernel* model. Examining *map* as an illustrative example, let's assume we have pagers P_1 and P_2, a client C, and σ_0 as illustrated in Figure 5. If C requires a virtual memory mapping established, but has insufficient resources to allocate a page table. P_2's map operation will fail and it then can choose to either deny service to C due to C's insufficient resources, or P_2 can choose to allocate and account one of its own pages (and corresponding frame) to C to supply service to C based on some resource management policy P_2 applies to C. P_2 calls to P_1 with a "lend the chosen page to C" message. Given that the chosen page in P_2 was original supplied by P_1, P_1 can apply a resource management policy to P_2, and if the page donation to the kernel is permitted, P_1 sends a similar request to σ_0 which lends the underlying frame to the kernel. When C is eventually deleted, the frames lent to the kernel on its behalf are freed and are available to the resource pools in σ_0, P_1, and P_2 on demand.

The above lend-to-kernel resource management design has several problems. The first one is that revocation of frames allocated to a process is only possible upon deletion of that process, even though the frame contents might be discarded and later reconstructed via redundant data elsewhere. A page table node is the obvious example of such a frame that could be revoked and re-established depending on memory demands. As a result, a long running process will consume its peak kernel memory requirements, not its current requirements.

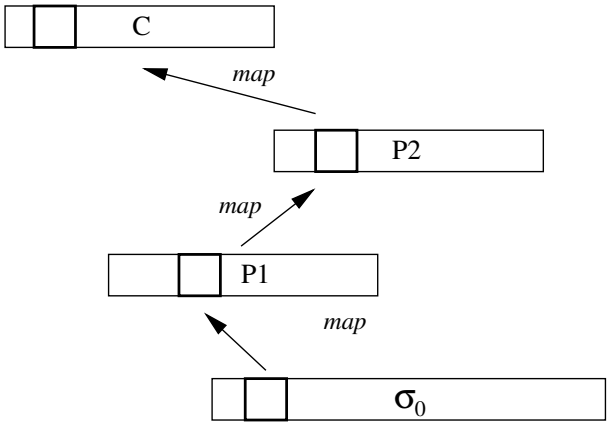


Fig. 5. Example pager hierarchy

Another issue is that it is not clear how one could modify the design to limit the actual kernel memory consumption of a process such that its available kernel memory acts a cache of a larger memory space that is paged to disk, i.e. using part of physical memory as a fixed size cache of a process's kernel objects.

Great care is also required in a system based on the above as (i) the resources of a client consumed by a server performing a service on its behalf is only indirectly controlled by the client; (ii) there are no kernel enforced restrictions on who one can donate kernel memory to, it's only limited by resource management policy; and (iii) one does not necessarily have the right to revoke memory given to a client as it requires *delete rights* to the client.

An example of where issue (i) manifests itself is when a client receives a mapping from a server, the address of the mapping indirectly determines the page table requirements. Issue (ii) manifests itself with simplistic resource management policies such as "the OS personality can have as much as it needs, all normal processes are limited to X frames", upon which a denial-of-service attack can be mounted by donating all available memory to the OS personality. Issue (iii) occurs when a server supplies memory to a client (a peer) and does not posses the right to delete the client. This can be avoided by requesting a resource provider with delete rights to provide the memory, but this requires careful resource management co-ordination and accounting which may in turn result in more issues.

The lend-to-kernel model might be workable in theory, and the previous three issues are pragmatic ease-of-use issues, not necessarily flaws, however it remains to be seen how to build a practical system with precise kernel resource management based on the loan-to-kernel model.

A more recent proposal is user-level management of kernel memory [7]. In this proposal, kernel memory is managed on a per-process basis. Each process starts with zero memory and consequently must obtain all memory in a controlled way via the mechanisms provided. The mechanism associates a *kpager* with each thread. The kpager receives faults generated on behalf of a thread by the kernel. Faults (like page faults) are used to signal resource exhaustion of the thread's process and suspend the thread until the fault is resolved. The kpager can chose to supply a frame to the kernel based on a user-level resource policy or deny the request. If the frame is mapped to the kernel, it becomes opaque to the kpager, but still revocable via *unmap*.

Unlike the lend-to-kernel model, the kpager can revoke memory from the kernel. This is achieved via the kernel either zeroing the content if it is redundant, or exporting it back to user-level in a form that can be validated upon return. As an illustrative example of the utility of revocation, a kpager can implement a cache policy for kernel memory by preempting kernel memory (and potentially storing it to disk) and re-assigning to another process. Each thread is assigned a kpager, which may be distinct if kernel memory should be managed differently for different concurrent processes, e.g. real-time versus best effort.

The issues with the user-level management of kernel memory proposal include the lack of precision of what the kernel uses the memory for, and the potential for all operations requiring an object to be allocated to block on a kpager fault. Devising a scheme to accurately reflect in the fault the subsequent use of the memory provided to the kernel was left as future work. Without accurately being able to determine the use of the memory, revocation has unknown consequences. Even without considering revocation, the kernel may use the memory for providing a kernel object for which the kpager might have delayed the fault handling (or denied it completely) had it known the eventual purpose of the memory required.

Having a thread block on faults when resources are unavailable creates a denial of service issue similar to that created when memory is copied from one task to another, where a page missing in the source or destination blocks both the source and destination. An example of the problem is when a server maps a page to a client who does not have memory for the needed for the page table node required. A kpager fault is generated blocking the server on a kpager related to the client. The kpager has a indeterminable trust relationship with the server, which leads to the server's reliance on timeouts to prevent the potential denial of service, but timeouts other than *zero* or *never* are problematic [25] and should be avoided as a concept fundamental to the design.

Summary Precise, controlled kernel-resource management is something that has eluded L4 to this point in time. Without a coherent, practical, precise mech-

anism for kernel memory allocation, L4 will remain unsuitable as a basis for systems requiring strong availability and confidentiality guarantees.

5 Conclusion

One potential direction for L4's future evolution is into the domain of trustworthy embedded systems, as exemplified by dependable systems, secure systems, and some classes of digital-rights-management-capable systems. We have examined L4 (both the current design, and previous designs) for its suitability for supporting trustworthy embedded systems. We have identified two general areas where L4 is lacking: communications control and kernel-resource management. We have examined all existing schemes proposed or implemented (to the best knowledge of the author) for communication control and kernel-resource management for L4 in particular, and no scheme is entirely satisfactory.

Given we have clearly identified what we believe are the major obstacles to L4's adoption in the domain of trustworthy embedded systems, and that we are confident we can resolve these issues, we expect L4 to become an ideal basis for the development of future trustworthy embedded systems.

References

1. http://www.symbian.com/press-office/2004/pr040618.html.
2. http://www.xrml.org/.
3. M. D. Bennett and N. C. Audsley. Partitioning support for the l4 kernel. techreport YCS-2003-366, Dept. of Computer Science, University of York, 2003.
4. P. England, J. D. DeTreville, and B. W. Lampson. Digital rights managements operating system. US Patent 6,330,670, December 2001.
5. A. Gefflaut, T. Jaeger, Y. Park, J. Liedtke, K. Elphinstone, V. Uhlig, J.E. Tidswell, L. Deller, and L. Reuther. The SawMill multiserver approach. In *9th SIGOPS European Workshop*, Kolding, Denmark, September 2000.
6. Jim Gray. A census of Tandem system availability between 1985 and 1990. *IEEE Transactions on Reliability*, 39(4), October 1990.
7. Andreas Haeberlen and Kevin Elphinstone. User-level management of kernel memory. In *Advances in Computer System Architecture (Proc. ACSAC'03), Lecture Notes in Computer Science*, volume 2823. Springer-Verlag, October 2003.
8. H. Härtig, R. Baumgartl, M. Borriss, C. J. Hamann, M. Hohmuth, F. Mehnert, L. Reuther, S. Schönberg, and J. Wolter. DROPS - OS support for distributed multimedia applications. In *Proc. 8th SIGOPS European Workshop*, Sintra, Portugal, 1998.
9. Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. The performance of μ-kernel-based systems. In *Proc. 16th Symp. on Operating Systems Principles*. ACM, 1997.
10. Gernot Heiser, Kevin Elphinstone, Jerry Vochteloo, Stephen Russell, and Jochen Liedtke. The Mungi single-address-space operating system. *Software Practice and Experience*, 28(9), July 1998.
11. Wei-Ming Hu. Reducing timing channels with fuzzy time. In *IEEE Symposium on Research in Security and Privacy*, May 1991.

12. T. Jaeger, J.E. Tidswell, A. Gefflaut, Y. Park, J. Liedtke, and K. Elphinstone. Synchronous IPC over transparent monitors. In *9th SIGOPS European Workshop*, Kolding, Denmark, September 2000.
13. Trent Jaeger, Kevin Elphinstone, Jochen Liedtke, Vsevolod Panteleenko, and Yoonho Park. Flexible access control using IPC redirection. In *7th Workshop on Hot Topics in Operating Systems*, Rio Rico, Arizona, March 1999.
14. Richard A. Kemmerer. Shared resource matrix methodology: an approach to identifying storage and timing channels. *IEEE Transactions on Computer Systems*, 1(3), August 1983.
15. Jean-Claude Laprie. Dependable computing and fault tolerance: Concepts and terminology. In *Proc. 15th Fault Tolerant Computing Symposium*, Ann Arbor, MI, June 1985.
16. Jean-Claude Laprie. Dependability of computer systems: concepts, limits, improvements. In *Proc. 6th Symposium on Software Reliability Engineering*, October 1995.
17. Joshua LeVasseur, Volkmar Uhlig, Jan Stoess, and Stefan Götz. Unmodified device driver reuse and improved system dependability via virtual machines. Unpublished OSDI Submission, 2004.
18. Jochen Liedtke. Clans & chiefs. In *12. GI/ITG-Fachtagung Architektur von Rechensystemen*, Kiel, March 1992. Springer Verlag.
19. Jochen Liedtke, Kevin Elphinstone, Sebastian Schönberg, Hermann Härtig, Gernot Heiser, Nayeem Islam, and Trent Jaeger. Achieved IPC performance. In *6th Workshop on Hot Topics in Operating Systems (HotOS)*, Chatham, Massachusetts, May 1997.
20. Jochen Liedtke, Vsevolod Panteleenko, Trent Jaeger, and Nayeem Islam. High-performance caching with the lava hit-server. In *USENIX Annual Technical Conference*, New Orleans, June 1998.
21. Frank Mehnert, Michael Hohmuth, and Hermann Härtig. Cost and benefit of separate address spaces in real-time operating systems. In *Proc. 23rd Real-Time Systems Symposium*, December 2002.
22. S. Ravi, A. Raghunathan, P. Kocher, and S. Hattangady. Security in embedded systems: Design challenges. *ACM Transactions on Embedded Computing Systems*, 3(3), August 2004.
23. John Rushby. Partitioning for safety and security: Requirements, mechanisms, and assurance. NASA Contractor Report CR-1999-209347, NASA Langley Research Center, June 1999. Also to be issued by the FAA.
24. J. Saltzer and M. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9), September 1975.
25. Jonathan Shapiro. Vulnerabilities in synchronous IPC designs. In *Proc. IEEE Symposium on Security and Privacy*, Oakland, CA, 2003.
26. Jonathan S. Shapiro, David J. Farber, and Jonathan M. Smith. The measured performance of a fast local IPC. In *Proc. 5th Int'l Workshop on Object-Orientation in Operating Systems*, Seattle, WA, 1996.
27. Espen Skoglund. Confinement, virtualization and rights delegation using virtual threads. Personal Communication, 2004.
28. L4Ka Team. *L4 eXperimental Kernel Reference Manual*. System Architecture Group, Dept. of Computer Science, Universität Karlsruhe, revision 6 edition, October 2004.
29. Harvey Tuch and Gerwin Klein. Verifying the L4 virtual memory subsystem. In Gerwin Klein, editor, *Proc. NICTA workshop on OS verification 2004, Technical Report 0401005T-1*, Sydney, Australia, October 2004. National ICT Australia.

Formal Security Analysis with Interacting State Machines

David von Oheimb and Volkmar Lotz

Siemens AG, Corporate Technology, D-81730 Munich,
{David.von.Oheimb|Volkmar.Lotz}@siemens.com

Abstract. We introduce the ISM approach, a framework for modeling and verifying reactive systems in a formal, even machine-checked, way. The framework has been developed for applications in security analysis. It is based on the notion of Interacting State Machines (ISMs), kind of high-level Input/Output Automata. The ISM framework is used to define system models and present them graphically with the AutoFocus tool, to let them be checked for consistency and translated to a representation within the theorem prover Isabelle/HOL (or alternatively to define them directly as Isabelle theory sections), and finally to employ the theorem prover for performing any kind of syntactic and semantic checks, in particular semi-automatic verification. We demonstrate that the framework can be fruitfully applied for formal system analysis by two classical application examples: the LKW model of the Infineon SLE 66 smart card chip and Lowe's fix of the Needham-Schroeder Public-Key Protocol.

Keywords: security, formal analysis, Interacting State Machines, Isabelle/HOL, AutoFocus, smart cards, protocols

1 Introduction

1.1 Motivation

In industrial environments, there is an increased demand for rigorous analysis of security properties of systems. Due to restrictions imposed by the application domain, the system environment, and business needs, new security mechanisms and architectures have to be invented frequently, with time-to-market pressure and intellectual property considerations obstructing the chance to gain confidence by exposing a proposed solution to the security community (which has been shown to be appropriate for cryptographic algorithm assessment). Formal analysis of suitable abstractions of systems has instead turned out to be extremely helpful in reasoning about a system's security, since the mathematical precision of the arguments allows for maximal confidence in the results obtained and, thus, in the security of the system being modeled.

The importance of formal analysis – on top of open review – in security assessment is, for instance, reflected by the requirements stated for high assurance levels of criteria like ITSEC [ITS91] and CC [CC99], which include formal security modeling and formal system development steps, and the achievements of the security protocol verification community, which discovered flaws in protocols that failed to be detected by informal approaches.

However, even in a formal setting it is easy to make – minor and sometimes even major – mistakes: undefined expressions, type mismatches, inconsistent specifications, missing evidence in proofs, false conclusions etc. Therefore, pure pen-and-paper formalizations cannot be considered fully reliable. Machine-checking of formal objects and structures has to be employed in order to significantly reduce the occurrence of such mistakes. Machine support additionally gives the opportunity to represent and deal with formal objects – both specifications and proofs – in an easy-to-comprehend way, which is a prerequisite for introducing formal approaches in an industrial environment characterized by time and cost restrictions.

1.2 Goals

A framework for machine-assisted formal security analysis that is particularly suited for industrial use should enjoy a number of properties:

Expressiveness. It should be possible to describe any typical security sensitive computation, storage, and communication system in an abstract way. This requires in particular the notions of state transformation, concurrency, and message passing.

Flexibility. Since IT systems and their security threats evolve quickly, the models produced within the framework should be easily adaptable and extendable as necessary to reflect the changes.

Simplicity. Modeling a system, stating its properties and proving them should require as little expertise and time as possible while maintaining the rigor of a fully formal approach.

Graphical capabilities. System models should be representable as diagrams that provide a good overview of the system structure and advance a quick intuition about its behavior.

Maturity of the semantics. The specification formalism should build upon a well-understood logic and have a well-defined semantics that supports reasoning about, e.g., invariants and refinement.

Availability of tools. The framework should be built from existing widely available (open-source) software like editors and proof tools and require at most minor modifications or extensions to them.

Since we did not find an existing framework that fulfills all these requirements to a satisfactory extent, we decided to build our own.

1.3 Related Work

The *IOA Language and Toolset* [GL98,Kay01] is a framework for analyzing computational processes with aims very similar to ours. It consists of a specification language and tool support for simulation, theorem proving, model checking, and code generation, where by now the simulation aspect is developed most and theorem proving support is limited to PVS. Its semantic foundation is the notion of *I/O Automata (IOAs)* [LT89] modeling asynchronous distributed computation with synchronous communication. Since the notion is based on transition systems augmented by communication primitives (rather than e.g. a process algebra augmented by local computation primitives), it is fairly easy to understand. It is equipped with a well-developed meta theory supporting refinement and compositional reasoning. System properties, both safety and liveness ones, may be described using temporal logics and proved by model checking and interactive theorem proving.

The only — but severe — drawback of IOAs from our perspective, in particular when modeling system security in an abstract way, is that their interaction scheme is rather low-level: buffered communication has to be modeled explicitly, and transitions involving several related input, internal processing, and output activities cannot be expressed atomically. Instead, each high-level transition has to be split into multiple low-level transitions, and between these, any number of further input events may take place due to the input-enabledness of IOAs. The solution to this problem is to add input buffers that accumulate messages asynchronously. An automaton may retrieve messages from multiple buffers, process them and send output to multiple buffers, and all this can be done simultaneously within a single atomic[1] transition. Our notion of ISMs, first described in [Ohe02], provides for that.

A further related framework that provided inspiration for ours is AutoFocus [HSSS96] – see §2.2 for more details. Even though developed primarily for modeling and verifying functional properties of embedded systems, it is used also for the security analysis of general distributed systems [WW01, JW01].

Other related approaches combine state-oriented and message-oriented description methods, for example translating CSP to B [But99] or Z to CSP [Fis00]. The drawback of such hybrids is that the user has to deal with two different nontrivial formalisms. Moreover, theorem proving support respecting the structure of the mixed-style specifications seems not to be available.

2 Preliminaries

In this section, we briefly introduce the two software tools we rely on and comment on their suitability for the ISM approach.

[1] Even though these high-level transitions are atomic, the corresponding I/O events are independent of each other because of the buffered asynchronous output semantics; thus there is no need for action refinement.

2.1 Isabelle/HOL

Isabelle [NPW02] is a generic theorem prover that has been instantiated to many logics, in particular the very practical *Higher-Order Logic (HOL)*. Isabelle/HOL [PNW+] is a predicate logic based on the simply-typed λ-calculus and thus in a sense combines logical and functional programming. Being quite expressive and supporting automatic type inference, it is the most important and best supported logic of Isabelle. The lack of dependent types introduces a minor nuisance for applications like ours: for systems consisting of more than one ISM, there has to be a single type of message contents into which all message data is injected, and analogously for the local states of the automata composed in parallel.

Proofs are conducted primarily in an interactive fashion where automatic and semi-automatic methods are available to tackle the routine parts. The Isabelle system is well-documented and well-supported, is freely available (including sources) and comes with the excellent user interface Proof General [AGKS99]. We consider it the most flexible and mature verification environment available. Using Isabelle/HOL, security properties can be expressed easily and adequately and verified with powerful proof methods.

2.2 AutoFocus

AutoFocus [HSSS96] is a freely available specification and simulation tool for distributed systems. Components and their behavior are specified by a combination of *System Structure Diagrams (SSDs)*, *State Transition Diagrams (STDs)* and auxiliary *Data Type Definitions (DTDs)*. Their execution can be visualized using *Extended Event Traces (EETs)*. Various back-ends including code generators and interfaces to model checkers may be acquired by purchase from Validas [S+].

We employ AutoFocus for its strengths concerning graphical design and presentation, which is important when setting up models in collaboration with clients (where strong familiarity with formal notations cannot be assumed), when documenting our work, and publishing its results. For abstract security modeling, there are currently two problems with AutoFocus. First, expressiveness is limited concerning the type system and the handling of underspecification. Second, due to the original emphasis of AutoFocus on embedded systems, the underlying semantics is still clock-synchronous. In contrast, for the most of our applications, an asynchronous (buffered) semantics is more adequate, which is under consideration also for future versions of AutoFocus. Using an alternative semantics implies that we cannot make use of the simulation, code generation and model checking capabilites of current AutoFocus and its back-ends. Yet this is not a real obstacle for us since we are interested mainly in its graphic capabilities and the offered specification syntax is general enough to cover our deviating semantics as well.

3 The ISM approach

ISMs are the core of our modeling and verification framework. In this section we explain the ISM concepts and semantics both in an intuitive way and as rigorous mathematical definitions. Moreover, we comment briefly on the ISM representation in AutoFocus and define the syntax of ISM sections in Isabelle/HOL theories. In the subsequent sections we present two classical case studies.

We use ISMs as building blocks for defining system models of a wide range of IT systems and expressing and verifying their security properties. At the time of writing, we have applied the ISM formalism in three major projects. They include the analysis of a complex database access control system for Siemens Medical Solutions and of the Infineon SLE88 smart card processor memory management [OLW04]. More information on the current status of the ISM framework, including the sources, a manual, and all publications, may be found at the project home page, `http://ddvo.net/ISM/`.

The ISM formalism has been extended to include global state [OL03]. This can be used, for instance, to provide for dynamic activation state and communication topology [OL03] or ambient-like administrative domains [KO03] or even their combination [KO03].

3.1 Concept of Interacting State Machines

An *Interacting State Machine (ISM)* is an automaton whose state transitions may involve multiple input and output simultaneously on any number of ports. As the name suggests, the key concepts of ISMs are states (and in particular the transitions between them) and interaction. By *interaction* we mean explicit buffered communication via named ports (which are also called connections), where on each *port*, (typically) one receiver listens to possibly many senders. Figure 1 gives the basic ISM structure.

Any number of ISMs may be composed in parallel by interleaving their transitions and forming I/O connections among peer ISMs. The local state of the resulting ISM is essentially the Cartesian product of the local states of its components. The top-level composition is called an ISM *system*. In [OL03] we extend the ISM concept by the notion of global state, which is not directly visible to ISMs but can control the whole system structure. The global state is affected by commands contained in transitions of elementary ISMs.

A *configuration* of an ISM consists of its input buffer state and local state. The *local state* may have arbitrary structure but typically is the Cartesian product of a *control state* which is of finite type and a *data state* which is a record of named fields representing local variables. Each ISM has a single[2] local *initial state*.

[2] If a non-singleton set of initial states is required, this may be simulated by nondeterministic spontaneous transitions from a single dummy initial state.

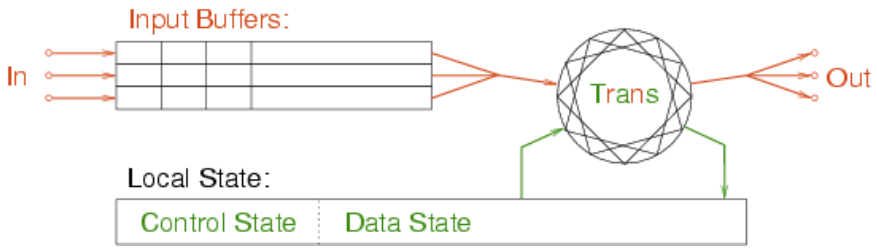


Fig. 1. ISM structure

The input buffers of an ISM are a family of (unbounded) message FIFOs, indexed by port names. The buffers are not part of elementary ISMs but are introduced by the parallel composition. Input ports can – but in most applications should not – be shared among ISMs, which leads to nondeterministic competition on each input item, without fairness guarantees.

Message exchange is triggered by an output operation of any ISM within the system. Input from the environment may be modeled with suitable ISMs. Inputs cannot be blocked, i.e. they may occur at any time, appending the received value to the corresponding FIFO. Values stored in the input buffers related to an ISM are received and processed by the ISM when it is ready to do so.

The actions of ISMs are given as user-defined *transitions*, which may be nondeterministic and can be specified in any relational style. Thus for each transition the user has the choice to define it in an operational (i.e., executable) or axiomatic (i.e., property-oriented) fashion or a mixture of the two. Transition rules specify that – potentially under some precondition that typically includes matching of messages in the input buffers – the ISM consumes some input, makes a local state transition, and produces some output. The output is appended to the respective input buffers specified by port names. Direct or indirect feedback is possible. Multicast is not directly supported but may be explicitly modeled easily.

An ISM system *run* is any prefix of the sequence of configurations reachable from the initial configuration. The length of a run is not bounded but finite. Finiteness allows for a simple trace semantics, but on the other hand implies that we cannot handle liveness properties. Yet we do not feel this as a real restriction because most relevant properties are essentially safety properties: practical guarantees about the existence of future events typically involve timeouts.

Transitions of different ISMs that are composed in parallel cannot directly interfere with each other but are related only by the causality wrt. the messages interchanged. Execution gets stuck (i.e., deadlocks) when there is no component that can perform any step. As is typical for reactive systems, there is no built-in notion of final or accepting states.

3.2 ISM Semantics

This subsection gives the logical meaning of ISMs, which is both an extension and a slight simplification of the definitions given in [Ohe02]. As the modifications pervade all parts of the ISM definitions, and for self-containedness, it appears mandatory to rephrase all of them.

First some general remarks on the presentation: all definitions and proofs have been developed as a hierarchy of Isabelle/HOL theories and machine-checked using this tool. One important effect of this approach is that many kinds of mistakes like type mismatches can be ruled out. Using the LaTeX documentation feature of Isabelle would even preclude typographic slips in the presentation but on the other hand would introduce some technicalities many readers would not be familiar with. Therefore, we give the semantics in traditional "mathematical" style in order to enhance readability. We sometimes make use of λ-abstraction borrowed from the λ-calculus, but write (multi-argument) function application in the conventional form, e.g. $f(a, b, c)$. Occasionally we make use of partial application (aka. *currying*), such that, in the example just given, $f(a, b)$ is an intermediate function that requires a third parameter before yielding the actual function result.

Message Families Let $\mathcal{M}$ be the type of all messages potentially exchanged by ISMs and $\mathcal{P}$ the type of port names. Then the *message families*, which are used to denote both input[3] buffers and input/output patterns, have type $MSGs = \mathcal{P} \to \mathcal{M}^*$ where $\mathcal{M}^*$ is any finite sequence of elements of $\mathcal{M}$. We will make use of the following operations on message families:

- the term *bla* denotes the empty message family $\lambda p.\ \langle\rangle$ where $\langle\rangle$ denotes the empty sequence
- the term $mdom(m)$ abbreviates $\{p|\ m(p) \neq \langle\rangle\}$, i.e. the domain of m
- the infix operation .@. concatenates two message families m and n on a port by port basis: $(m\ .@.\ n)(p) = m(p)\ @\ n(p)$

States and Transitions A set of ISM transitions has type $TRANS(\Sigma) = \wp((MSGs \times \Sigma) \times (MSGs \times \Sigma))$ where the parameter Σ stands for the type of the local state and the two occurrences of $MSGs$ stand for input and output patterns, respectively. Each element has the form $((i, \sigma), (o, \sigma'))$ and means that the ISM can (possibly nondeterministically) perform a step from local state σ to σ', consuming input i and producing output o. Simultaneous input and/or output on multiple channels can be specified because both i and o each denote whole message families. In contrast to the original definition of ISMs [Ohe02], within a transition, input is described by patterns of messages consumed in the given step — not by a transition between the state of the input buffer before and after the transition. This simplifies the definition of single ISMs and shifts the concept of input buffering to the places where it is indispensable: at the definitions of parallel composition and automata runs.

[3] Recall that output buffers are not required.

Elementary ISMs An ISM is given as a quadruple[4] $a = (In(a), Out(a), \sigma_0(a), Trans(a))$ of type $ISM(\Sigma) = \wp(\mathcal{P}) \times \wp(\mathcal{P}) \times \Sigma \times TRANS(\Sigma)$ where

- $In(a)$ is the set of input port names
- $Out(a)$ is the set of output port names
- $\sigma_0(a)$ is the initial local state
- $Trans(a)$ is the transition relation

Such an ISM is *well-formed* iff all the port names actually used in the transitions for input or output respect the I/O interface of the ISM, i.e. $ipns(a) \subseteq In(a)$ and $opns(a) \subseteq Out(a)$ where

- $ipns(a) = \bigcup_{t \in Trans(a)} mdom((\lambda((i,\sigma),(o,\sigma')).\ i)(t))$
- $opns(a) = \bigcup_{t \in Trans(a)} mdom((\lambda((i,\sigma),(o,\sigma')).\ o)(t))$

Note that $In(a)$ and $Out(a)$ may overlap, which allows for direct feedback within parallel composition.

Runs Below we will define composite ISM runs, i.e. the parallel composition and execution of a family of ISMs, directly in one step. Nevertheless, we first define the two notions of ISM runs and parallel composition independently. Defining parallel composition in isolation not only makes it easier to understand but also enables hierarchical analysis and design.

The *open runs* of an ISM a, denoted by $Runs(a) \in \wp(\Sigma^*)$, are finite sequences of states that are inductively defined as

$$\frac{}{\langle \sigma_0(a) \rangle \in Runs(a)}$$

$$\frac{ss^\frown\sigma \in Runs(a) \quad ((i,\sigma),(o,\sigma')) \in Trans(a)}{ss^\frown\sigma^\frown\sigma' \in Runs(a)}$$

The operator $^\frown$ appends elements to a sequence.

This form of runs is called *open* because in each step the environment provides arbitrary input to the ISM, and any output of the ISM is discarded. If feedback from output to input is desired, one can achieve this by applying the parallel composition operator to the singleton family of ISMs consisting just of a, described next.

Parallel Composition Any number of ISMs can be combined in parallel to form a single composite ISM, which may be further combined with others, etc. By identifying input and output buffers of ISMs to be combined, internal communication including feedback loops can be introduced as shown in Figure 2.

[4] The definition pattern $x = (sel_1(x), sel_2(x), \ldots)$ should not be understood as a recursive definition of x but as a shorthand introducing a tuple with typical name x and with selectors (i.e., projection functions) sel_1, sel_2, ...

The *parallel composition* $\|_{i\in I} A_i$ of a family of ISMs $A = (A_i)_{i\in I}$ is an ISM of type $ISM(CONF(\Pi_{i\in I}\Sigma_i))$ where I is any index set I and for any X, the type of an ISM *configuration* $CONF(X)$ is defined as $MSGs \times X$. Here $MSGs$ stands for the type of internal buffers. The composite ISM is defined as the quadruple $(AllIn(A)\backslash AllOut(A),\ AllOut(A)\backslash AllIn(A),\ (bla, S_0(A)),\ \ PTrans(A))$ where

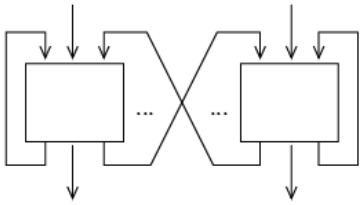


Fig. 2. General communication pattern within parallel composition

- $AllIn(A) = \bigcup_{i\in I} In(A_i)$
- $AllOut(A) = \bigcup_{i\in I} Out(A_i)$
- bla gives the initial value of the internal buffers, which are used to handle I/O among peers as well as direct feedback
- $S_0(A) = \Pi_{i\in I}(\sigma_0(A_i))$ is the Cartesian product of all initial local states
- $PTrans(A)$ of type $TRANS(CONF(\Pi_{i\in I}\Sigma_i))$ is the parallel composition of their transition relations.

The pre- and post-states in the composed transition relation refer not only to the Cartesian product of all local states but also to a message family b. As already mentioned above for the initial state, the role of b is to buffer internal I/O. Apart from this, the composed transition relation is defined simply as the interleaving of the transitions of the component ISMs:

$$\frac{\begin{array}{c} j \in I \\ ((i,\sigma),(o,\sigma')) \in Trans(A_j) \end{array}}{\begin{array}{l} ((i_{|\overline{AllOut(A)}}, (i_{|AllOut(A)}\ .@.\ b, S[j{:=}\sigma\,])), \\ \ (o_{|\overline{AllIn(A)}},\ (b\ .@.\ o_{|AllIn(A)},\ \ S[j{:=}\sigma'])))\in PTrans(A) \end{array}}$$

where

- $S[j{:=}\sigma]$ denotes the replacement of the j-th component of the tuple S by σ
- $m_{|P}$ denotes the restriction $\lambda p.$ if $p \in P$ then $m(p)$ else $\langle\rangle$ of the message family m to the set of ports P
- $i_{|\overline{AllOut(A)}}$ denotes those parts of the input i provided not by the output of peer ISMS but by outer ISMs
- $i_{|AllOut(A)}$ denotes the internal input from peer ISMs or direct feedback, which is taken from the current buffer contents b
- $o_{|\overline{AllIn(A)}}$ denotes those parts of the output o provided to outer ISMs
- $o_{|AllIn(A)}$ denotes the internal output to peer ISMs or direct feedback, which is added to the current buffer contents b.

A parallel composition is *well-formed* iff the inputs of the individual components do not overlap: $\forall i\ j.\ i \neq j \longrightarrow In(A_i) \cap In(A_j) = \emptyset$. On the other hand, outputs may overlap, which allows the outputs of different ISMs to interleave nondeterministically.

A family A of ISMs is called *closed* iff $AllIn(A) = AllOut(A)$, i.e. there is no interaction with any outside ISMs. If a system is modeled with a closed ISM family and input from the environment is important, this may be modeled with an ISM that belongs to the family and does nothing but generating all possible input patterns.

When composing ISMs, it is occasionally necessary to prevent name clashes or to hide connections, which can be achieved by suitable renaming of ports.

Composite Runs We define ISM runs not only for single (possibly composite) ISMs but also directly for closed families of ISMs intended to run in parallel. The above definition of parallel composition may be used in combination with composite runs to describe inner (possibly nested) levels of parallel composition.

The set of all possible *composite runs* is denoted by $CRuns(A)$ and has type $\wp((CONF(\Pi_{i\in I}\Sigma_i))^*)$ corresponding to the ISM type $ISM(\Pi_{i\in I}\Sigma_i)$. Its elements are finite sequences of configurations, inductively defined as

$$\frac{}{\langle(bla, S_0(A))\rangle \in CRuns(A)}$$

$$\frac{\begin{array}{c} j \in I \\ cs^\frown(i\ .@.\ b, S[j:=\sigma]) \in CRuns(A) \\ ((i,\sigma),(o,\sigma')) \in Trans(A_j) \end{array}}{cs \frown (i\ .@.\ b, S[j:=\sigma]) \frown (b\ .@.\ o, (S[j:=\sigma'])) \in CRuns(A)}$$

Traces of composite runs have the form $\langle(bla,S_0(A)),\ (b_1,S_1),\ (b_2,S_2),\ \ldots\rangle$ where each element of the sequence is a pair of the current internal buffer contents and the Cartesian product of all the currently relevant local states.

One can show that composite runs of any closed family A of well-formed ISMs are equivalent to the runs of the parallel composition of the same family: $wf\_isms(A)\ \wedge\ closed(A)\ \longrightarrow\ Runs(\|_{i\in I}A_i) = CRuns(A)$.

3.3 AutoFocus representation

By design, ISMs have almost the same structure as the automata definable with AutoFocus [HSSS96], and thus we can use AutoFocus as a graphical front-end to our Isabelle implementation. We will employ AutoFocus diagrams when introducing the application examples below.

In a typical application of our framework, ISMs are first specified[5] as standard non-hierarchical AutoFocus automata, saved in the so-called *Quest* file format, and then translated into suitable Isabelle theory files by a tool program [Nan02, ON02].

[5] see the online tutorial `http://autofocus.in.tum.de/nelli/englisch/html/`

3.4 Isabelle representation

ISMs can be defined in dedicated sections of Isabelle theories. Syntactically, this abstract representation has essentially a one-to-one correspondence to the AutoFocus representation. Its semantics is the one defined in §3.2.

An ISM section is introduced by the keyword **ism** and has the following general structure[6]:

ism *name* *((*(*param_name* **::** *param_type*)*)** =
  ports *pn_type*
    inputs *I_pns*
    outputs *O_pns*
  messages *msg_type*
  states *[state_type]*
  *[***control** *cs_type [***init** *cs_expr0]]*
  *[***data** *ds_type [***init** *ds_expr0] [***name** *ds_name]]*
 *[***transitions**
  *(tr_name [attrs]***:** *[cs_expr* `->` *cs_expr']*
  *[***pre** *(bool_expr)*$^+$*]*
  *[***in** *([***multi***]** *I_pn I_msgs)*$^+$*]*
  *[***out** *([***multi***]** *O_pn O_msgs)*$^+$*]*
  *[***post** *((lvar_name* **:=** *expr)*$^+$ | *ds_expr')]*
)$^+$*]*

The meaning of the individual parts is as follows.

- The ISM definition will be referred to by *name*. It may have any number of parameters, each declared by *param_name* and a corresponding *param_type*. The parameters may be used throughout the definition body.
- The type expression *pn_type* gives the Isabelle/HOL type of the port names, while *I_pns* and *O_pns* denote the set of input and output port names, respectively.
- The type expression *msg_type* gives the type of the messages, which is typically an algebraic datatype with a constructor for each kind of message.
- The optional *state_type* should be given if the current ISM forms part of a parallel composition and the state types of the ISMs involved differ. In this case, *state_type* should be a free algebraic datatype with a constructor for each state type of the ISMs involved.
  The type expressions *cs_type* and *ds_type* give the types of the control and data state, respectively, while the optional terms *cs_expr0* and *ds_expr0* specify their initial values — if not given, they default to some arbitrary value. Either (i.e., not both) the control state or the data state may be absent.
  The optional logical variable name *ds_name*, which defaults to `s`, may be used to refer to the whole data state within transition rules.

[6] *[...]* marks optional parts, *(...)*$^+$ means one or more comma-delimited occurrences

Transitions are given via named rules where *attrs* is an optional list of attributes, e.g. **[intro]**. The control states (if any) before and after the transition are specified by the expressions[7] *cs_expr* and *cs_expr'*.

Expressions within a rule may refer to the logical data state variable mentioned above. In particular, assuming that `s` is the name of the data state variable, then the value of any local variable `lvar` of the ISM may be referred to by `lvar s`. The scope of free variables appearing in a rule is the whole rule, i.e. free variables are implicitly universally quantified (immediately) outside each rule. All the following parts of a transition rule are optional:

- The **pre** part contains guard expressions *bool_expr*, i.e. preconditions constraining the enabledness of a transition.
- The **in** part gives input port names (or sets of them if preceded by **multi**) *I_pn*, each in conjunction with a list *I_msgs* of message patterns expected to be present in the corresponding input buffer(s). When an ISM executes a transition, any free variables in message patterns are bound to the actual values that have been input. Each port names should appear at most once within a **in** part. Any input port not explicitly mentioned is left untouched.
- The **out** part gives output port names *O_pn*, each in conjunction with an expression *O_msgs* denoting a list of values designated for output to the corresponding port. The variant using **multi** is used to specify multicasts. Each port name should be used at most once within each **out** part. Any output port not mentioned does not obtain new output.
- The **post** part describes assignments of values *expr* to the local variables *lvar_name* of the data state. Variables not mentioned remain invariant. Alternatively, an expression *ds_expr'* may be given that represents the entire new data state after the transition. Assignments to the local variables suit an operational style, whereas an axiomatic style can be achieved using *ds_expr'* (in conjunction with suitable constraints in the preconditions).

An **ism** theory section is translated to Isabelle/HOL concepts in a straightforward way using an extension to Isabelle, as described in [Nan02]. In particular, each ISM section is translated to a record definition with the appropriate fields, the most complex one being the transition relation, which is defined via an inductive (but not actually recursive) definition.

The meta theory of ISMs that we have defined in Isabelle/HOL includes all concepts mentioned in §3.2, in particular well-formedness, renaming, parallel composition, runs, and composite runs. Further auxiliary concepts are introduced as well, in particular reachability and induction schemes related to ISM runs. The characteristic properties of these concepts, as required for system verification, are derived within Isabelle/HOL. All details of the meta theory may be found in [ON02].

Example **ism** sections will be given in §4.4 and §5.2.

[7] These need not be constant but may contain also variables, which is useful for modeling generic transitions. In this case, one such transition has to be represented by a set of transitions within AutoFocus.

4 LKW Model for the Infineon SLE 66

We give a slightly extended and improved version of the LKW formal security model for the Infineon SLE 66 smart card processor.

4.1 The SLE 66 family

SLE 66 is the short name of a family of smart card chips by Infineon. Each chip consists of a CPU including an encryption unit, RAM, ROM, and EEPROM, which stores e.g. firmware and personalization data.

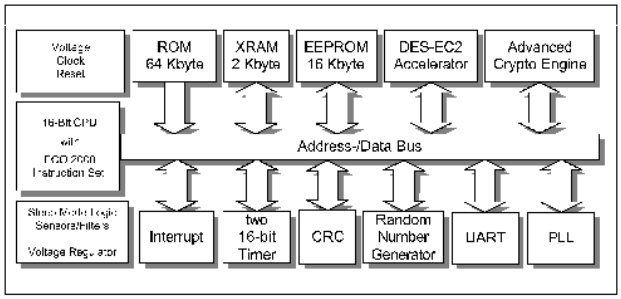


Fig. 3. SLE 66 Block Diagram

The chip has been designed as a general-purpose microprocessor with special hardware supporting security-sensitive applications like electronic passport or payment systems. In contrast to the successor family, SLE 88, these processors do not provide separation of memory via a MMU [OLW04] or any operation system functionality but provide a secure platform for a customized BIOS and essentially a single application. Therefore, security has to be dealt with at a very elementary level where nothing can be assumed about higher-level functionality.

The most important security objective is to preserve the security of information stored in the memory components. In more detail:

- The data items stored in any of the memory components shall be protected against unauthorized disclosure or modification.
- The security relevant functions stored in ROM or EEPROM shall be protected against unauthorized disclosure or modification.
- Hardware test routines shall be protected against unauthorized execution.

The objectives are achieved by implementing a set of security enforcing functions which mainly perform the following two tasks:

- The system runs through several phases during its lifetime. Entry to the phases is controlled by *test functions*, which check different flags and give a specified level of authorization.
- Additionally, all data stored in the memory components is encrypted by hardware means, utilizing several keys and key sources with a chip specific random number among them.

4.2 LKW formal security model

The *LKW model* [LKW00] has been one of the first formal models for security properties of hardware chips. It has been used very successfully within the security evaluation process for the whole SLE 66 family on ITSEC level E4 high and the corresponding Evaluation Assurance Level 5 (*semi-formally designed and tested*, which includes a formal security model) [CC99]. A slight extension has been introduced [OLW02] in order to reflect additional application-oriented security objectives defined in the Smart Card IC Platform Protection Profile [AHIP01]. More recently, we have added an analysis of nonleakage [Ohe04].

Developing the original LKW model took about two months of work, including understanding and discussing the system design and security target, investigating modeling alternatives, discussing the model with the chip developers, and supporting the evaluation process. The formal parts made up about ten percent of the whole evaluation and certification effort which was even based on existing development documents. Re-stating the model with the ISM approach took about two weeks. Incorporating the extension mentioned above took just a few days including discussions etc. These numbers may serve as an indicator for estimating formal modeling efforts in future evaluation processes.

Meanwhile we have developed also a security model of the SLE 88 memory management unit [OWL03] following the ISM approach as well.

The formal security policy model of the SLE 66 consists of two parts: a system model describing the processor's behavior on an abstract level by means of a state transition automaton with input and output, and a set of security objective specifications given as properties of automata runs. Thus one can prove that the security objectives are met by the system model. Interpreting the system model in terms of the real processor then allows one to conclude with some evidence that the processor indeed meets its security objectives as required by ITSEC E4 assessment criteria.

The style of the LKW security model is ad-hoc, but using classical formal access control models instead would not be appropriate because they introduce notational overhead that would not be justified in the context of the SLE 66 evaluation and because they are not flexible enough to handle phase transitions and the like adequately.

The LKW model has been done originally as a pen-and-paper work, i.e. without tool assistance. Inevitably, even fully reviewed descriptions of the model contained many (mostly minor) syntactical, typographical and semantical slips as well as type errors, but also omissions like missing assumptions and incomplete proofs. Therefore it was desirable to formalize the model in a machine-checked way, applying a well-developed meta theory. At first, using the Isabelle implementation of IOAs [Mül98] for this purpose seemed promising, yet the weak structure of IOA transitions appeared inappropriate, which became one of our motivations to invent ISMs. Using the ISM approach, the LKW model can be represented adequately and with maximal quality, as demonstrated on the following pages.

4.3 AutoFocus Diagrams

On the abstract level of the LKW model, the system architecture of the SLE 66 is rather trivial: there is one component with one input port named `In` and one output port named `Out`, as depicted by Figure 4. The data state of the component consists of two stores mapping names of functions to the corresponding function code and data objects to corresponding data values.

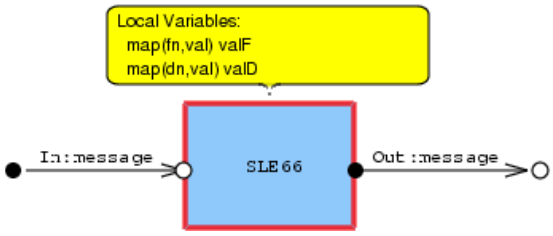


Fig. 4. SLE66 System Structure Diagram

Much more involved is the structure of the state transitions. There are four control states corresponding to the *phases* of the SLE 66 life cycle:

Phase *0*: construction of the chip
Phase *1*: upload of Smartcard Embedded Software and personalization
Phase *2*: normal usage
Phase *Error*: locked mode from which there is no escape

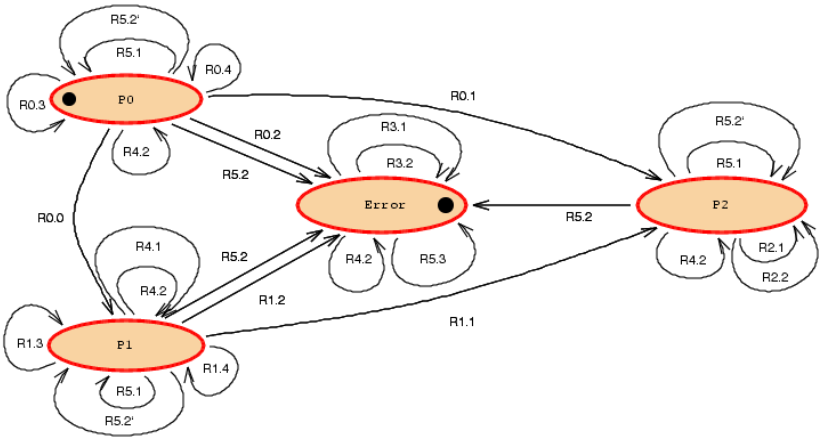


Fig. 5. SLE 66 State Transition Diagram

In order to keep the state transition diagram clear, Figure 5 contains all control states and transitions, but instead of showing the preconditions, inputs, outputs, and changes to the data state, we just label the transitions with the names of the corresponding transition rules. These are described in detail in §4.4, while here we give an informal general description:

R0.0 thru R0.4 describe the execution of functions in the initial phase 0. Only the processor manufacturer is allowed to invoke functions in this phase and the requested function must be present.

R0.0 states that if the function belongs to class `FTest0` and the corresponding test succeeds, phase 1 will be entered, and the test functions of that class are disabled.

R0.1 describes a shortcut leaving out phase 1: if the function belongs to class `FTest1` and the test succeeds, phase 2 will be entered, and all test functions are disabled.

R0.2 states that if a test fails, the system will enter the error state.

R0.3 models the successful execution of any other function, in which case the function may change the chip state and yield a value.

R0.4 states that in all remaining cases of function execution, the chip responds with `No` and its state remains unchanged.

R1.1 thru R1.4 describe the execution of functions in the upload phase 1 analogously to R0.1 thru R0.4.

R2.1 and R2.2 describe the execution of functions in the usage phase 2 analogously to R0.3 and R0.4.

R3.1 and R3.2 describe the execution of functions in the error phase analogously to R0.3 and R0.4, except that the only function allowed to be executed in this phase is chip identification.

R4.1 and R4.2 describe the effects of a specific operation used for uploading new (operating system and application) functionality on the chip. This must be done by subjects trusted by the processor manufacturer and is allowed only in phase 1.

R4.1 describes the admissible situations, and

R4.2 describes all other cases.

R5.1 thru R5.3 describe the effects of attacks. Any attempts to tamper with the chip and to read security-relevant objects via physical probing on side channels (by mechanical, electrical, optical, and/or chemical means), for example differential power analysis or inspecting the silicon with a microscope, are modeled as a special "spy" input. Note that modeling physical attacks in more detail is not feasible because this would require a model of physical hardware. In particular, the conditions (and related mechanisms) under which the processor detects a physical attack are beyond the scope of the model.

R5.1 describes the innocent case of reading non-security-relevant objects in any regular phase, which actually reveals the requested information.

R5.2 describes the attempt to reading security-relevant objects in any regular phase. The chip has to detect this and enters the error phase, while the requested object may be revealed or not. This concept is called "destructive reading": one cannot rule out that attacks may reveal information even about security-relevant objects, but after the first of any such attacks, the processor hardware will be "destroyed", i.e. cannot be used regularly.

R5.3 states that in the error phase no (further) information is revealed.

4.4 Isabelle Definition

We describe in detail our ISM model of the SLE 66, which is based on the original LKW model plus the slight extension introduced in [OLW02]. We do this employing the automatic LaTeX documentation facility of Isabelle that can be used like a "literate programming" environment: the user augments an Isabelle theory (in this case representing our SLE 66 model) with comments and other text sections in LaTeX format that may refer (via a special quotation mechanism) to the type declarations, constant definitions, theorems, etc. When Isabelle processes the theory, it generates LaTeX output for all parts of the theory that are marked as relevant for documentation and merges them with the chunks of text supplied by the user. The great advantage of this approach is that the theory (and proof) development and its documentation are always with each other and mistakes typically resulting from typesetting formulas with LaTeX manually are avoided.

The Isabelle theory sources, including the documenting text, may be obtained from [ON02]. For the original description of the LKW model containing, among others, a more general discussion on the benefits of formal modeling, refer to [LKW00].

theory *SLE66 = ISM_package:* — we build on the general ISM definitions

First we have to define a number of entities (types, logical constants, etc.) acting as building blocks for the actual ISM theory section. In order to keep the model as abstract as possible, which makes it less bulky to read and simplifies the proofs, we often use underspecification. This important modeling technique means that for part of the types and constants we do not give full definitions but only declarations of their names. We even do not make the encryption of data in the memory components explicit.

Names Objects stored on the chip may be either functions or data and are referred to by object names. The type of these names, *on*, is the disjoint sum of function names *fn* and data object names *dn*, which are not further specified:

typedecl *fn* — function name
typedecl *dn* — data object name
datatype *on = F fn | D dn* — object name

Objects are classified as security-relevant (demanding secrecy and integrity) by including their names in the sets *F_Sec* or *D_Sec*, whose disjoint union is called *Sec*. In order to meet the additional requirements of [AHIP01], the domain of security relevant functions *F_Sec* of the original LKW model has been refined to the disjoint union of *F_PSec* and *F_ASec*, which control the protection of the processor and application functionality, respectively.

In the following theory sections, we declare a list of constants together with their types. We define only part of them, and for part of the remaining ones we give the essential properties in the form of axioms:

consts
```
  f_SN   :: "fn"      — the name of the function giving the serial number
  FTest0 :: "fn set"  — the names of test functions of phase 0
  FTest1 :: "fn set"  — the names of test functions of phase 1
  FTest  :: "fn set"  — the names of all test functions
  F_Sec  :: "fn set"  — the names of all security-relevant functions
  F_PSec :: "fn set"  — the subset of F_Sec relevant for the processor
  F_ASec :: "fn set"  — the names of F_Sec relevant for applications
  F_NSec :: "fn set"  — the names of all non-security-relevant functions
  D_Sec  :: "dn set"  — the names of all security-relevant data objects
  D_PSec :: "dn set"  — the subset of D_Sec relevant for the processor
  D_ASec :: "dn set"  — the names of D_Sec relevant for applications
  D_NSec :: "dn set"  — the names of all non-security-relevant data objects
    Sec  :: "on set"  — the names of all security-relevant objects
```
defs
```
  FTest_def:  "FTest  ≡ FTest0 ∪ FTest1"
  F_ASec_def: "F_ASec ≡ F_Sec - F_PSec"
  D_ASec_def: "D_ASec ≡ D_Sec - D_PSec"
  F_NSec_def: "F_NSec ≡ -F_Sec"
  D_NSec_def: "D_NSec ≡ -D_Sec"
  Sec_def: "Sec ≡ {F fn |fn. fn ∈ F_Sec} ∪ {D dn |dn. dn ∈ D_Sec}"
```
axioms
```
  FTest01_disjunct: "FTest0 ∩ FTest1 = {}"
  f_SN_not_FTest:   "f_SN ∉ FTest"
  F_PSec_is_Sec:    "F_PSec ⊆ F_Sec"
  FTest_is_PSec:    "FTest ⊆ F_PSec"
```

State The abstract state of an SLE 66 chip is a pair, where the first component is the phase in the processor life cycle:

datatype `ph = P0 | P1 | P2 | Error`

We introduce the type `val` for any values, i.e. function code or data stored or processed by the chip. The only thing we need to know about the type `val` is that the serial number of the chip belongs to it.

typedecl `val` — data and function values
consts `SN :: val` — serial number

The second state component is a record of two partial functions, `valF` and `valD`, mapping function and data object names to values:

record `chip_data =`
```
  valF :: "fn ⇀ val"
  valD :: "dn ⇀ val"
```

The function `val` takes an argument of type `chip_data` and yields a partial function lifting `valF` and `valD` to general object names of type `on`:

constdefs
```
  val :: "chip_data ⇒ on ⇀ val"
 "val s on ≡ case on of F fn ⇒ valF s fn  | D dn ⇒ valD s dn"
```

Having defined the two components of the processor state, we can now give the definition of the overall state:

```
types SLE66_state = "ph × chip_data"
```

We will often need to refer to the set of functions available in the current state, therefore we introduce an auxiliary function `fct` that yields the domain of `valF`:

```
constdefs
  fct :: "chip_data ⇒ fn set"
 "fct s ≡ dom (valF s)"
```

We declare three further auxiliary functions that denote the results and state changes of a processor function (including test functions):

```
consts
 "output"   :: "fn ⇒ chip_data ⇒ val"
 "change"   :: "fn ⇒ chip_data ⇒ chip_data"
                                  — change is unused for test functions
 "positive" :: "val ⇒ bool"  — check for positive test outcome
```

Further ISM section ingredients We need only two port names, one for input to the chip and one for its output:

```
datatype interface = In | Out
```

SLE 66 commands provide information on the subjects issuing them. There is a special subject `Pmf` denoting the processor manufacturer.

```
typedecl sb
consts Pmf :: sb
```

Possible input consists of either the two kinds of SLE 66 commands modeling function execution and function code loading operations or the `Spy` operation, which models attacks that may reveal information stored on the chip and may corrupt the chip memories. Output of the SLE 66 may be the result value of a (regular) function or an indication of success or failure.

```
datatype message =
  Exec sb fn | Load sb fn val | Spy on  — input
| Val val | Ok | No                    — output
```

The subjects performing regular commands identify themselves to the chip via physical means. The actual authentication mechanism, as well as many other implementation details, is confidential and beyond the scope of this article anyway. Here we just declare an auxiliary function that yields the subject issuing a (regular) command:

```
consts subject :: "message ⇒ sb"
primrec
  "subject (Exec sb fn  ) = sb"
  "subject (Load sb fn v) = sb"
```

ISM definition Having defined its various parameters, we can finally give the theory section that specifies the SLE 66 model as an ISM:

```
ism SLE66 =
  ports interface
    inputs   "{In}"
    outputs "{Out}"
  messages message
  states
    control ph init "P0"
    data    chip_data name "s" — The data state variable is called s. Note that
```

the initial data state is left unspecified and thus is arbitrary, which is a good example of underspecification since its actual value is immaterial for the security properties we are interested in.

```
  transitions
```

— Rule R00 specifies execution of a test function f from the set FTest0 by the processor manufacturer Pmf in the initial phase P0. If the test is successful then the SLE 66 enters the next phase P1, answers with Ok, and disables the test functions FTest0. As specified by the data theory subsection just above, the variable s denotes the current data state of the ISM at the beginning of the transition. Thus, for example, fct s means the functions currently available. The operator '⌊' below restricts a partial function, in this case valF s, to the given set, in this case the complement of FTest0.
Rule R00 is typical for interactions of the SLE 66 in the sense that a single input triggers a single output. Note that the direct relation of input and output is expressed easily using ISMs, whereas using IOAs, two transitions would be required whose relation would be cumbersome to express and to use during verification.

```
  R00: P0 → P1
    pre "f ∈ fct s∩FTest0", "positive (output f s)"
    in  In  "[Exec Pmf f]"
    out Out "[Ok]"
    post valF := "valF s⌊(-FTest0)"
```

— Rule R01 is analogous to R00, but specifies that if the test function f is from FTest1 rather than FTest0 then phase P1 is skipped and the chip enters P2 immediately, disabling all test functions FTest:

```
  R01: P0 → P2
    pre "f ∈ fct s∩FTest1", "positive (output f s)"
    in  In  "[Exec Pmf f]"
    out Out "[Ok]"
    post valF := "valF s⌊(-FTest)"
```

— If in P0 a test function gives a negative result then the chip enters the Error phase and the output is No:

```
  R02: P0 → Error
    pre "f ∈ fct s∩FTest0", "¬positive (output f s)"
    in  In  "[Exec Pmf f]"
    out Out "[No]"
```

— Any other function call issued by the processor manufacturer in P0 has the standard consequences: The function result is output and the data state changed (according to the semantics of the function which is not further specfied). Note that by the form of

postcondition used, the whole data state (consisting of *valF* and *valD* here) is replaced by the given value: the denotation of *change f s*.

R03: P0 → P0
  pre *"f ∈ fct s - FTest"*
  in *In "[Exec Pmf f]"*
  out *Out "[Val (output f s)]"*
  post *"change f s"*

— In all remaining cases for phase 0, the attempted function execution is ignored and the output is *No*:

R04: P0 → P0
  pre *"sb ≠ Pmf ∨ f ∉ fct s"*
  in *In "[Exec sb f]"*
  out *Out "[No]"*

— This ends the specifications of transitions originating in *P0*.

The specifications of transitions originating in *P1* are fully analogous to the rules R00, R02, R03, and R04, just replacing *P0* by *P1*, *P1* by *P2*, and *FTest0* by *FTest1*:

R11: P1 → P2
  pre *"f ∈ fct s∩FTest1", "positive (output f s)"*
  in *In "[Exec Pmf f]"*
  out *Out "[Ok]"*
  post *valF := "valF s⌊(-FTest1)"*

R12: P1 → Error
  pre *"f ∈ fct s∩FTest1", "¬positive (output f s)"*
  in *In "[Exec Pmf f]"*
  out *Out "[No]"*

R13: P1 → P1
  pre *"f ∈ fct s - FTest1"*
  in *In "[Exec Pmf f]"*
  out *Out "[Val (output f s)]"*
  post *"change f s"*

R14: P1 → P1
  pre *"sb ≠ Pmf ∨ f ∉ fct s"*
  in *In "[Exec sb f]"*
  out *Out "[No]"*

— The rules R21 and R22 specify function calls in *P2* analogously to R03 and R04, except that any subject is allowed to issue them:

R21: P2 → P2
  pre *"f ∈ fct s"*
  in *In "[Exec sb f]"*
  out *Out "[Val (output f s)]"*
  post *"change f s"*

R22: P2 → P2
  pre *"f ∉ fct s"*
  in *In "[Exec sb f]"*
  out *Out "[No]"*

— In the *Error* phase, the only function that may be called is chip identification, yielding the serial number *SN*. All other cases yield *No*:

```
R31: Error → Error
  pre "f_SN ∈ fct s"
  in  In  "[Exec sb f_SN]"
  out Out "[Val SN]"
R32: Error → Error
  pre "f ∉ fct s∩{f_SN}"
  in  In  "[Exec sb f]"
  out Out "[No]"
```

— The rules R41 and R42 specify the behavior of the *Load* operation, which is allowed only for the processor manufacturer and only in the upload phase *P1*. If allowed, *valF* is updated at the position *f* with the new function value *v*.
In contrast to the original LKW model [LKW00], the *Load* operation may upload not only non-security-relevant functions but also functions of the application security domain (as long as no such function of the same name is already present).

```
R41: P1 → P1
  pre "f ∈ F_NSec ∪ (F_ASec - fct s)"
  in  In  "[Load Pmf f v]"
  out Out "[Ok]"
  post valF := "valF s(f↦v)"
R42: ph → ph
  pre "f ∉ F_NSec ∪ (F_ASec - fct s) ∨ sb ≠ Pmf ∨ ph ≠ P1"
  in  In  "[Load sb f v]"
  out Out "[No]"
```

— Note that the rule R42 is generic in the sense that it applies to more than one control state of the ISM, namely all phases except *P1*.

— The rules R51 thru R53 specify the possible reactions of the chip to attacks, modeled by the *Spy* operation. If the attacker attempts to read a non-secret object whose name is *on* and the chip is not in the *Error* phase, the access may be granted, yielding the desired value (if any):

```
R51: ph → ph
  pre "on ∉ Sec", "ph ≠ Error"
  in  In  "[Spy on]"
  out Out "case val s on of None ⇒ [] | Some v ⇒ [Val v]"
```

— Rule R52 specifies the typical reaction of the SLE 66 upon attacks trying to read a secret object while tampering with the chip: it may be unable to prevent that the desired value is output, but in any case it reaches the *Error* phase from which no further secrets may be obtained, as specified by the rules R31, R32, and R53.

```
R52: ph → Error
  pre "on ∈ Sec", "v ∈ {[],[Val (the (val s on))]}", "ph ≠ Error"
  in  In  "[Spy on]"
  out Out "v"
  post "any"
```

— Note that R52 describes two sorts of nondeterminism: *v* denotes either the empty output or the singleton output giving the desired value, and the attack may corrupt the function and data stores arbitrarily.

There are also cases where the chip can resist an attack without any damage and without any leakage of secrets, such that there is no need to enter the *Error* phase:

```
R52':ph → ph
  pre "on ∈ Sec", "ph ≠ Error"
  in  In  "[Spy on]"
  out Out "[]"
```

— If the chip is already in the *Error* phase, no further secrets can be obtained. The chip state may be corrupted further, but it makes sure that it stays locked in the *Error* phase:

```
R53: Error → Error
  in  In  "[Spy on]"
  out Out "[]"
  post "any"
```

As expressed by the rules R52 and R53, the attacker may obtain (the representation of) at most one secret object from the chip memory. It is interesting to observe that the leakage of one item is harmless because all data stored on the chip is encrypted. There are two cases to consider:

- The secret obtained is the de-/encryption key itself, which is not helpful to the attacker because no further data item, in particular none encrypted with the key, can be obtained.
- The secret obtained is an encrypted value, which is not helpful because the attacker cannot any more obtain the decryption key.

Obviously, sophisticated techniques are required to implement the specified reaction to physical attacks modeled by the *Spy* operation.

ISM runs The SLE 66 ISM just defined models the static interface of the chip as well as all possible single state transitions that it can perform. In order to describe the overall behavior of the chip during its life-cycle, we can refer to the notions that our Isabelle implementation provides for ISMs in general:

```
types
  SLE66_trans = "(unit, interface, message, SLE66_state) trans"
constdefs
  Trans :: "SLE66_trans set"          — the set of all possible transitions
 "Trans ≡ trans SLE66.ism"

  TRuns :: "(SLE66_trans list) set"  — all possible transition sequences
 "TRuns ≡ truns SLE66.ism"

  Runs :: "(SLE66_state list) set"   — all possible sequences of states
 "Runs ≡ runs SLE66.ism"
```

This concludes the system model of the SLE 66.

4.5 Properties

The second part of the SLE 66 security model deals with the security properties derivable from the system model.

Security Objectives In the (confidential[8]) original security requirements specification by Infineon, the security objectives for the SLE 66 had been stated as follows.

SO1. "The hardware must be protected against unauthorised disclosure of security enforcing functionality."
SO2. "The hardware must be protected against unauthorised modification of security enforcing functions."
SO3. "The information stored in the processor's memory components must be protected against unauthorised access."
SO4. "The information stored in the processor's memory components must be protected against unauthorised modification."
SO5. "It may not occur that test functions are executed in an unauthorised way."

Later, an additional requirement concerning the confidentialiy and integrity of Smartcard Embedded Software, which is not part of the security enforcing functionality, has been added [AHIP01, §4.1].

Having formally defined the SLE 66 system model, these informal statements can now be expressed formally as predicates on the system behavior, describing unambiguously and in detail which states may be reached under which circumstances, which data may be modified, and which output may appear on the output channel.

After formalizing the security objectives, it is natural to ask if the chip behavior, as specified in the system model, actually fulfills these requirements. The corresponding proofs have been conducted first using pen and paper, as reported in [LKW00]. Within the ISM framework, we meanwhile have verified these properties even mechanically using Isabelle, discovering two major flaws that will be reported in this subsection. Below we give all the required auxiliary definitions, the most important lemmata, and all theorems, together with an abstract informal description of the machine-checked proofs.

Model Assumptions Due to the abstract specification style where e.g. the semantics of parts of the chip functionality is not fully specified, it turns out that in order to prove the properties, a few general axioms that augment the system model are required. The first one of them asserts that security-relevant functions do not modify security-relevant functions:

```
Axiom1: "f∈fct s∩F_Sec ⟹ valF (change f s)⌊F_Sec = valF s⌊F_Sec"
```

[8] quotations with permission.

In comparison to the version of this axiom in the original model, the scope of functions `f` has been extended from "initially available" to "security-relevant", reflecting the changes to rule `R41`. Part of the lemmas as well as the formalized security objective `FSO21` change accordingly.

The second axiom is very similar, stating that also non-security-relevant functions do not modify security-relevant functions:

```
Axiom2: "f∈fct s∩F_NSec ⟹ valF (change f s)⌊F_Sec = valF s⌊F_Sec"
```

In order to formalize the security objective SO1 and `Axiom3`, we define the set `ValF_Sec r` holding all code of security-relevant functions in a given run (i.e., sequence of states) `r`.

constdefs

```
  ValF_Sec :: "SLE66_state list  ⇒ val set"
 "ValF_Sec r ≡ ⋃{ran (valF s⌊F_Sec) |ph s. (ph,s) ∈ set r}"
```

The third (and last) axiom introduced in the LKW model states that in phase 2, a function cannot reveal (by intentional "guessing" or by accident) any members of `ValF_Sec r`. This rather self-evident requirement is needed for technical reasons in the proof of SO1.

```
Axiom3: "⟦r∈Runs; (P2,s)∈set r; f∈fct s⟧⟹ output f s∉ValF_Sec r"
```

A notational remark is in order here: in Isabelle formualas, multiple premises are bracketed using '⟦' and '⟧' and separated using '`;`'.

When machine-checking the proofs contained in [LKW00] with Isabelle, we noticed that a fourth axiom was missing that makes an implicit but important assumption explicit: if a function object may be referenced in two (different) ways and one of them declares the function to be security-relevant, the other has to do the same.

```
  Axiom4: "⟦r ∈ Runs;
            (ph, s) ∈ set r; (ph', s') ∈ set r;
            val s n = Some v; val s' n' = Some v;
            n ∈ Sec⟧ ⟹ n' ∈ Sec"
```

The exposure of missing critical assumptions demonstrates how important machine support is when conducting formal analysis.

Theorems Finally, we translate the five informal security objectives to Isabelle formulas and prove them within the system. It is instructive to compare the formal versions of the security objectives FSOx below with the informal ones, SOx, given above.

The formalization of SO1, called `FSO1`, states that in any sequence `ts` of transitions performed by the chip, if the chip outputs any value `v` representing the code of any security-relevant function during its hitherto life, then the error state is entered or the output was in response to a function execution request by the processor manufacturer:

theorem *FSO1:* "⟦*ts* ∈ *TRuns*; ((*p*,(*ph*,*s*)),*c*,(*p'*,(*ph'*,*s'*))) ∈ *set ts*; *p' Out* = [*Val v*]; *v* ∈ *ValF_Sec* (*truns2runs ts*)⟧ ⟹ *ph'* = *Error* ∨ (∃*fn*. *p In* = [*Exec Pmf fn*])"

The proof of FSO1 proceeds by unfolding some definitions, e.g. of the SLE 66 ISM, applying properties of auxiliary concepts like *truns2runs*, and a case split on all possible transitions. Isabelle can solve most of the cases automatically (with straightforward term rewriting and purely predicate-logical reasoning), except for two: the case of rule R21 is handled using Axiom3, and for R51 we rely on the property "⟦*r* ∈ *Runs*; (*ph*, *s*) ∈ *set r*; *v* ∈ *ValF_Sec r*; *val s n* = *Some v*⟧ ⟹ *n* ∈ *Sec*" which in turn relies on Axiom4.

A more elaborate formalization of SO1 and SO3 taking into account also indirect and partial information flow is motivated and sketched in [Ohe04].

Like in the original LKW model, the translation of SO2 splits into two parts. FSO21' states that for any (even unreachable) transition not ending in the error phase, if a security-relevant function *g* is present in both the pre-state and the post-state, the code associated with it stays the same:

theorem *FSO21':* "⟦((*p*,(*ph*,*s*)),*c*,(*p'*,(*ph'*,*s'*)))∈*Trans*; *ph'* ≠ *Error*; *g* ∈ *fct s*∩*fct s'*∩*F_Sec*⟧ ⟹ *valF s' g* = *valF s g*"

This property is a generalization of the original FSO21, reflecting the extensions made to the *Load* operation in rule R41: Here we do not compare the initial and current value of *g* but the previous and current one, which takes into account also functions added in the meantime.

The proof of this property is — as usual — by case distinction over all possible transitions. Most cases are trivial except for those where function execution may change the stored objects, which are described by the rules R03, R13, and R21. Here an argumentation about the invariance of security-relevant functions *g* is needed, which follows easily from Axiom1 and Axiom2.

Similarly to FSO21', FSO22 states that for any transition within the same phase that is not the error phase, the set of existing security-relevant functions is non-decreasing:

theorem *FSO22:* "⟦((*p*,(*ph*,*s*)),*c*,(*p'*,(*ph'*,*s'*)))∈*Trans*; *ph'* ≠ *Error*; *ph* = *ph'*⟧ ⟹ *fct s*∩*F_Sec* ⊆ *fct s'*∩*F_Sec*"

Not surprisingly, the proof of this property is completely analougous.

FSO3 states that if the attacker obtains a result trying to get hold of a security-relevant data object *on*, then the chip enters the error phase:

theorem *FSO3:*"⟦((*p*,(*ph*,*s*)),*c*,(*p'*,(*ph'*,*s'*)))∈*Trans*; *p In* = [*Spy on*]; *on* ∈ *Sec*; *p' Out* ≠ []⟧ ⟹ *ph'* = *Error*"

The proof is done simply by case distinction.

FSO4 states that any transition not entering the error phase but changing the state does this in a well-behaved way: *s'* is derived from *s* via the desired effect of executing an existing function, or there is a phase change where only the test functions may be modified, or only a single function *f* is changed due to a *Load* operation:

theorem *FSO4:*

```
"⟦((p,(ph,s)),c,(p',(ph',s'))) ∈ Trans; ph' ≠ Error⟧ ⟹
 s' = s ∨
 (∃sb f  . p In = [Exec sb f  ] ∧ f ∈ fct s ∧ s' = change f s) ∨
 (ph'≠ph ∧ valD s' = valD s ∧ valF s'⌊(-FTest) = valF s⌊(-FTest))∨
 (∃sb f v. p In = [Load sb f v] ∧
           valD s' = valD s ∧ valF s'⌊(-{f}  ) = valF s⌊(-{f}  ))"
```

The proof is also straightforward by case distinction.

A second omission of the LKW model was that in the proof of the security objective FSO5 an argumentation about the accessibility of certain functions was not given in a rigorous way. We fix this by introducing an auxiliary property (where, as typical with invariants, finding the appropriate one is the main challenge) and proving it to be an invariant of the ISM. The invariant states that in phase 1, the test functions from *FTest0* have been disabled, and in phase 2, all test functions have been disabled:

constdefs

```
  no_FTest_invariant :: "SLE66_state ⇒ bool"
 "no_FTest_invariant ≡ λ(ph,s).
  ∀f ∈ fct s. (ph = P1 ⟶ f ∉ FTest0) ∧ (ph = P2 ⟶ f ∉ FTest)"
```

When proving that the invariant holds, 14 of the 19 cases are trivial, and the remaining ones require simple properties of the set *FTest*, and two of them require additionally Axiom1 and Axiom2. The invariant implies

lemma *P2_no_FTest:*

```
  "⟦(P2,s) ∈ reach SLE66.ism; f ∈ fct s⟧ ⟹ f ∉ FTest"
```

Wxploiting this property for the case of rule R21, we can prove FSO5 in the usual way. This theorem states that in any sequence of transitions performed by the chip, any attempt to execute a test function not issued by the processor manufacturer is refused:

theorem *FSO5:*

```
"⟦ts ∈ TRuns; ((p,(ph,s)),c,(p',(ph',s'))) ∈ set ts;
  p In = [Exec sb f]; f ∈ FTest⟧ ⟹
  sb = Pmf ∨ s' = s ∧ p' Out = [No]"
```

The Isabelle proofs of all six theorems formalizing the security objectives and the two lemmas required are well supported by Isabelle: each of them takes just a few steps, about half of which are automatic.

end

This finishes our detailed presentation of the SLE 66 case study. It demonstrates that the ISM approach can be fruitfully applied to both model and prove the security properties of state transition systems. The use of mechanical type checks and theorem proving system ensures a level of accuracy hardly reachable in a pen-and-paper analysis.

5 Needham-Schroeder Public-Key Protocol

In contrast to the high-level requirements analysis of the rather state-oriented SLE 66 model described in the previous section, we now turn to a more low-level analysis of a communication-oriented system. Our aim is to demonstrate that the ISM approach is capable of handling such quite different systems in a both rigorous and elegant way as well.

As a typical example for such a distributed system, we take Lowe's fix of the *Needham-Schroeder public-key authentication protocol* [Low96], which we call *NSL*. The emphasis here is not to provide new insights to the protocol, but to use a well-known benchmark system that makes our approach easy to compare with many other approaches that have been used to model (essentially) the same system.

We base our ISM model on the formalization by Paulson [Pau98]. His so-called "inductive approach" is tailored to semi-automated verification of cryptographic protocols. Its great advantage is a high degree of automation, due to abstraction to the core semantics of the protocols: event traces. On the other hand, this makes both the models and the properties at least cumbersome to express: state information is implicit, yet often it has to be referred to, which is done by repeating suitable parts of the event history and sometimes even by introducing auxiliary events.

5.1 AutoFocus Diagrams

As usual, our model of the NSL system consists of an agent called `Alice` aiming to establish an authenticated session with another agent called `Bob` in the presence of an `Intruder` according to the Dolev-Yao attacker model [DY83]. As

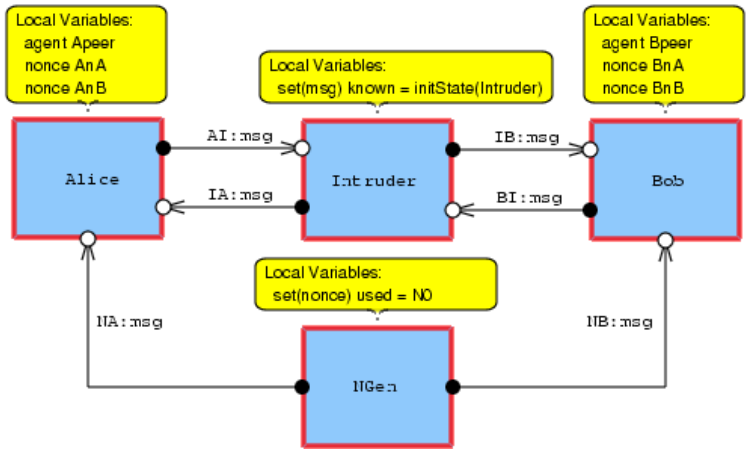


Fig. 6. NSL System Structure Diagram

will be motivated in §5.2, we furthermore introduce a server ISM called `NGen` that generates nonces for all honest agents. The corresponding system structure diagram in Figure 6 shows the four components with their data state (reflecting the expectations of the two agents, the set of messages the intruder knows of, and the set of already used nonces, respectively) and the named connections between them.

Even if sometimes neglected, agents involved in communication protocols do have state: their current expectations and knowledge. This is made explicit in a convenient way by describing their interaction behavior with state transition diagrams. Figure 7 shows the three states of the agent `Alice` and the transitions between them, which have the general format `guard : inputs : outputs : assignments`.

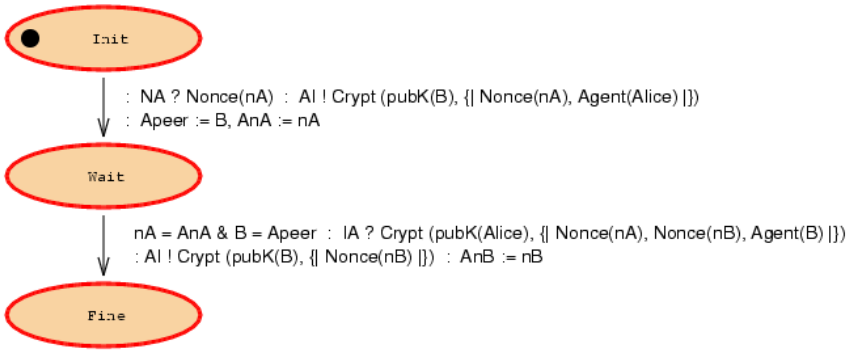


Fig. 7. NSL State Transition Diagram: `Alice`

In the initial state, `Alice` decides which agent `B` she wants to talk to and sends the corresponding request consisting of a fresh nonce `nA` (which she has obtained from the nonce server via her port `NA`) and her identity `Alice`, encrypted under the public key of the intended receiver. This message is actually sent to the port `AI` of the intruder. Alice remembers her intended peer in the local variable `Apeer` and the nonce she has used in the variable `AnA`. In the next state she awaits a response from the prospective peer, decrypts it and checks its authenticity by comparing the nonce value `nA` and agent name `B` with the corresponding items in her memory. Only if the decryption and the two comparisons are successful, the transition to her final state actually takes place, sending an appropriate acknowledgment to her peer and storing the nonce `nB` just received in her variable `AnB`. The third state represents (hopefully) successful session establishment where all essential parameters of the session may be referred to by the local variables of `Alice`.

From the example of `Alice`'s transitions, one realizes that ISM control state information is a natural way of fixing the order of protocol steps.

`Bob`'s state transitions are analogous and thus not shown here.

The transitions of the `Intruder` are quite different from the regular protocol participants: it stores all messages received on its ports `AI` and `BI` in the local variable *known* and can send any message derivable from its current knowledge (by analyzing the messages contained in the set *known*, utilizing the decryption keys it knows of, and synthesizing messages from the resulting pieces) to the ports `IA` and `IB`, as depicted by Figure 8. The figure reveals a weakness of

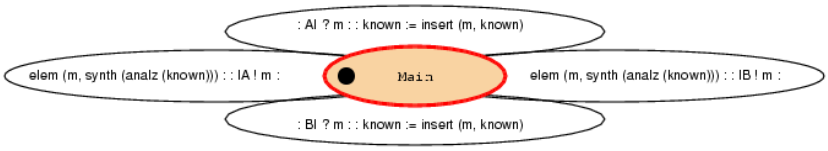


Fig. 8. NSL State Transition Diagram: `Intruder`

modeling with AutoFocus: there is a lot of redundancy among each of the two pairs of transitions (where the difference is just in the port names used), which can be avoided in the Isabelle representation by using generic transitions (where the port used for input or output is a variable that may hold either of the two possible values, as shown below).

Note that the intruder may take part in any number of sessions simultaneously. If the analysis needs to include the possibility that a regular agent takes part in more than one protocol run simultaneously, this can be modeled by multiple instantiation of the respective agent — under the assumption that from that agent's perspective the protocol runs are independent of each other.

The transition diagram of `NGen` is similar to the one of the intruder, except that there are no transitions with input.

5.2 Isabelle Definition

This section gives parts of our Isabelle representation of NSL. Refer to §3.4 for the details of ISM sections. We do not show the definitions of the various state and message components here since they are straightforward and analogous to the SLE66 model. Moreover, we give only the ISM definitions of those components for which we have not already given an AutoFocus STD above.

ism `Bob` =
  ports *channel*
    inputs *"{NB,IB}"*
    outputs *"{BI}"*
    messages *msg*
  states *state* — this is the sum of the four state types of the system components, required because of the type problem mentioned in §2.1
    control *B_control* **init** *"Idle"*
    data *B_data* **init** *"B0"*

```
  transitions
Resp: Idle → Resp
     in   NB "[Nonce nB]",
          IB "[Crypt (pubK Bob) {|Nonce nA, Agent A|}]"
     out  BI "[Crypt (pubK A) {|Nonce nA, Nonce nB, Agent Bob|}]"
     post "Bpeer := A, BnA := nA, BnB := nB"
Ack': Resp → Conn
     pre  "nB = BnB s"
     in    IB "[Crypt (pubK Bob) (Nonce nB)]"
```

Note that `Bob`'s first transition `Resp` takes two inputs, from the nonce generator and the intruder, and produces one output. If we modeled this transition using IOAs, we would have needed three transitions with intermediate states. The precondition of transition `Ack'` could have been made implicit by moving the comparison as a pattern to the **in** part, yet we make it explicit in order to emphasize its importance. The local variable `BnB` is used to store the value of the nonce expected, while the other two variables express `Bob`'s view to whom he is connected in which session. In Paulson's model, this state information is implicit in the event trace.

Modeling the freshness of nonces is an interesting problem, for which we are aware of essentially four solutions, each with their pros and cons.

- In Paulson's model [Pau98], nonces are generated non-deterministically under the side condition that they do not already appear in the current message/event history. This criterion refers to the semantic and system-global notion of event traces — something not available from the (local) perspective of ISMs.
- One could combine local and global freshness conditions and let each agent generate its own nonces: by producing fresh values locally and combining them with the globally unique agent identifier. The drawback of this solution is that each nonce issuer has to implement the mechanism just described.
- One could enforce global freshness by adding an axiom restricting system runs in the desired way. We prefer a more constructive approach here and derive the required freshness property as a lemma.
- Our solution is to introduce a nonce server component called `NGen` that performs the generation of nonces for all agents in a centralized fashion. In this way we can ensure global freshness with a constructive local criterion.

A further motivation to us for selecting the fourth solution just mentioned was that it makes the communication patterns of the agents more interesting because `Bob` has a transitions that inputs from two sources simultaneously. Note that `NGen` is just a modeling aid and thus its correct interplay with the agents, including authentication issues, does not need to be analyzed.

The ISM definition of `NGen` is rather simple because `NGen` does not require control state information and its local state consists only of the single variable storing the set of all nonces that already have been used. Therefore, we may identify the whole local state with this variable and call it *used*, eliminating

the need to define a record type and use the corresponding record selectors and updates.

```
ism NGen =
   ports channel
     inputs    "{}"
     outputs   "{NA,NB}"
     messages  msg
   states state
     data "nonce set" init "N0" name "used"
   transitions
 Cackle:
      pre   "ch ∈ {NA, NB}", "n ∉ used"
      out    ch "[Nonce n]"
      post "insert n (used)"
```

Note that the output port `ch` is (non-deterministically) selected from the set of two distinct names, which ensures the exclusive use of each nonce.

The family of all four ISMs is composed in parallel to form the NSL system. It is easy to prove that this ISM family is closed and all its members, as well as their parallel composition, are well-formed.

5.3 Properties

Properties of protocols specified with ISMs may be expressed with reference to both the state of agents and the messages exchanged. In the case of NSL, the most interesting property is authentication of `Alice` to `Bob` (actually, even session agreement [Low97] from `Bob`'s view), which we formulate as

```
⟦Alice ∉ bad;  Bob ∉ bad; (b,s)#cs ∈ NSL_Runs⟧ ⟹
 (∃nA. Bob_state   s  = (Conn, (|Bpeer=Alice, BnA=nA, BnB=nB|))) ⟶
 (∃(b',s') ∈ set cs.
 (∃nA. Alice_state s' = (Wait, (|Apeer=Bob  , AnA=nA, AnB=nB|))))
```

This can be quite intuitively read as: if in the current state `s` of the system `Bob` believes to be connected to `Alice` within a session identified by the nonce `nB` then there is an earlier state `s'` where `Alice` was in the waiting state referring to the same nonce `nB` after initiating a connection with `Bob`.

It is interesting to compare the above formulation with Paulson's[9]:

```
⟦A ∉ bad;  B ∉ bad;  evs ∈ ns_public;
 Crypt (pubK B) (Nonce NB) ∈ parts (spies evs);
 Says B A (Crypt (pubK A) {|Nonce NA,Nonce NB,Agent B|}) ∈ set evs
⟧ ⟹
 Says A B (Crypt (pubK B) {|Nonce NA,Agent A|}) ∈ set evs
```

[9] http://isabelle.in.tum.de/library/HOL/Auth/NS_Public.html

This statement is necessarily more indirect than ours since the beliefs of the agents have to be coded by elements of the event history. At least in this case, all messages of a protocol run have to be referred to. Note further that this formulation makes stronger assumptions than ours because an agreement on the value of the nonce `NB` is involved.

Due to the extra detail concerning agent state and the input buffers (which are not actually required for the NSL protocol), the proofs within the ISM approach are more painful and require more lemmas about intermediate states of protocol runs than Paulson's inductive proofs. On the other hand, the semi-automatic proofs within the ISM approach probably scale better.

There are about a dozen lemmas proved by rule induction, most of which deal with the freshness and usage of nonces generated by `NGen`. The main theorem is proved employing a variant of Schneider's rank function approach [Sch97], which we describe in detail in [Ohe02, §3].

6 Conclusion

ISMs are designed as high-level I/O automata, with additional structure and communication facilities. Like IOAs, ISMs are suitable for describing typical state-based communication systems relevant for security analysis, where ISM provide increased simplicity wrt. specifying component interaction via buffered communication and means to directly relate input and output actions within a single transition.

We have shown that the ISM approach is equally applicable to a variety of security analysis tasks, ranging from high-level security modeling and requirements analysis, typically showing less system structure but increased complexity of state transitions, to security analysis of distributed systems including cryptographic protocols, likely to exhibit advanced system structuring. The examples explicate the importance of a fully formalized strategy and mechanized proofs. In particular, the LKW model has been significantly improved by identifying hidden assumptions and completing sloppy argumentation.

The ISM approach offers graphic representation by means of AutoFocus System Structure Diagrams and State Transitions Diagrams. A tool program closely relates these graphical development and documentation capabilities with the formal system specification and verification capabilities of the mechanical theorem prover Isabelle/HOL.

Further work on ISMs includes the extension of the proof support in the ISM level concerning e.g. refinement and the provision of a specification language based on temporal logic. Additional AutoFocus capabilities may be made available, including further systems views like event traces and simulation, as well as test case generation.

In brief, the Interacting State Machines approach turns out to offer good support for formal security analysis in the way required within an industrial environment, meeting the goals stated in §1.2.

Acknowledgements. We thank Guido Wimmel, Thomas Kuhn and several anonymous referees for their comments on earlier versions of this article.

References

AGKS99. David Aspinall, Healfdene Goguen, Thomas Kleymann, and Dilip Sequeira. *Proof General*, 1999. http://www.proofgeneral.org/.

AHIP01. Atmel, Hitachi Europe, Infineon Technologies, and Philips Semiconductors. Smartcard IC Platform Protection Profile, Version 1.0, July 2001. http://www.bsi.de/cc/pplist/ssvgpp01.pdf.

But99. Michael Butler. csp2B : A practical approach to combining CSP and B. In *Proc. of FM'99: World Congress on Formal Methods*, pages 490–508, 1999.

CC99. Common Criteria for Information Technology Security Evaluation (CC), Version 2.1, 1999. ISO/IEC 15408.

DY83. Danny Dolev and Andrew C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, IT-29(12):198–208, March 1983.

Fis00. Clemens Fischer. *Combination and implementation of processes and data: from CSP-OZ to Java*. PhD thesis, Univ. of Oldenburg, 2000.

GL98. Stephen J. Garland and Nancy A. Lynch. The IOA language and toolset: Support for designing, analyzing, and building distributed systems. Technical Report MIT/LCS/TR-762, Laboratory for Computer Science, MIT, August 1998.

HSSS96. Franz Huber, Bernhard Schätz, Alexander Schmidt, and Katharina Spies. Autofocus - a tool for distributed systems specification. In *Proceedings FTRTFT'96 - Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 1135 of *LNCS*, pages 467–470. Springer-Verlag, 1996. See also http://autofocus.in.tum.de/index-e.html.

ITS91. Information Technology Security Evaluation Criteria (ITSEC), 1991.

JW01. Jan Jürjens and Guido Wimmel. Formally testing fail-safety of electronic purse protocols. In *Automated Software Engineering*. IEEE Computer Society, 2001.

Kay01. Dilsun Kirli Kaynar. IOA language and toolset, 2001. http://theory.lcs.mit.edu/tds/ioa.html.

KO03. Thomas Kuhn and David von Oheimb. Interacting State Machines for mobility. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *Proc. of the* 12^{th} *International FME Symposium (FM'03)*, volume 2805 of *LNCS*. Springer, September 2003. http://ddvo.net/papers/ISMfM.html.

LKW00. Volkmar Lotz, Volker Kessler, and Georg Walter. A Formal Security Model for Microprocessor Hardware. In *IEEE Transactions on Software Engineering*, volume 26, pages 702–712, August 2000.

Low96. Gavin Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Proc. of TACAS*, volume 1055 of *LNCS*, pages 147–166. Springer-Verlag, 1996.

Low97. Gavin Lowe. A hierarchy of authentication specifications. In *10th Computer Security Foundations Workshop (CSFW)*. IEEE Computer Society Press, 1997.

LT89. Nancy Lynch and Mark Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2(3):219–246, 1989. http://theory.lcs.mit.edu/tds/papers/Lynch/CWI89.html.

Mül98. Olaf Müller. *A Verification Environment for I/O Automata Based on Formalized Meta-Theory*. PhD thesis, Technische Univerität München, 1998. See also http://isabelle.in.tum.de/IOA/.

Nan02. Sebastian Nanz. Integration of CASE tools and theorem provers: a framework for system modeling and verification with AutoFocus and Isabelle. Master's thesis, TU München, 2002. http://www.doc.ic.ac.uk/~nanz/publications/csthesis/.

NPW02. Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002. See also http://isabelle.in.tum.de/docs.html.

Ohe02. David von Oheimb. Interacting State Machines: *a stateful approach to proving security*. In Ali E. Abdallah, Peter Ryan, and Steve Schneider, editors, *Formal Aspects of Security*, volume 2629 of *LNCS*, pages 15–32. Springer-Verlag, 2002. http://ddvo.net/papers/ISMs.html.

Ohe04. David von Oheimb. Information flow control revisited: Noninfluence = Noninterference + Nonleakage. In P. Samarati, P. Ryan, D. Gollmann, and R. Molva, editors, *Computer Security – ESORICS 2004*, volume 3193 of *LNCS*, pages 225–243. Springer, 2004. http://ddvo.net/papers/Noninfluence.html.

OL03. David von Oheimb and Volkmar Lotz. Generic Interacting State Machines and their instantiation with dynamic features. In Jin Song Dong and Jim Woodcock, editors, *Formal Methods and Software Engineering (ICFEM)*, volume 2885 of *LNCS*, pages 144–166. Springer, November 2003. http://ddvo.net/papers/GenISMs.html.

OLW02. David von Oheimb, Volkmar Lotz, and Gerog Walter. An interpretation of the LKW model according to the SLE66CX322P security target. Unpublished, January 2002.

OLW04. David von Oheimb, Volkmar Lotz, and Georg Walter. Analyzing SLE 88 memory management security using Interacting State Machines. *International Journal of Information Security*, 2004. To appear; preprint: http://ddvo.net/papers/SLE88_MM.html.

ON02. David von Oheimb and Sebastian Nanz. *ISM Homepage: Documentation, sources and distribution*, 2002. http://ddvo.net/ISM/.

OWL03. David von Oheimb, Georg Walter, and Volkmar Lotz. A formal security model of the infineon SLE 88 smart card memory management. In *Proc. of the 8th European Symposium on Research in Computer Security (ESORICS)*, volume 2808 of *LNCS*. Springer, 2003. http://ddvo.net/papers/SLE88_MM.html.

Pau98. Lawrence C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6:85–128, 1998.

PNW+. Lawrence Paulson, Tobias Nipkow, Markus Wenzel, et al. The Isabelle/HOL library. http://isabelle.in.tum.de/library/HOL/.

S+. Oscar Slotosch et al. Validas Model Validation AG. http://www.validas.de/.

Sch97. Steve Schneider. Verifying authentication protocols with CSP. In *10th Computer Security Foundations Workshop (CSFW)*. IEEE Computer Society Press, June 1997.

WW01. Guido Wimmel and Alexander Wisspeintner. Extended description techniques for security engineering. In M. Dupuy and P. Paradinas, editors, *International Conference on Information Security (IFIP/SEC)*. Kluwer Academic Publishers, 2001. http://www4.in.tum.de/papers/WW01.html.

Index

Verifying the L4 Virtual Memory Subsystem

Harvey Tuch and Gerwin Klein

National ICT Australia*, Sydney, Australia
{gerwin.klein|harvey.tuch}@nicta.com.au

Abstract. We describe aspects of the formalisation and verification of the L4 μ-kernel. Starting from an abstract model of the virtual memory subsystem in L4, we prove safety properties about this model, and then refine the page table abstraction, one part of the model, towards C source code. All formalisations and proofs have been carried out in the theorem prover Isabelle.

1 Introduction

L4 is a second generation microkernel based on the principles of minimality, flexibility, and efficiency [12]. It provides the traditional advantages of the microkernel approach to system structure, namely improved reliability and flexibility, while overcoming the performance limitations of the previous generation of microkernels. With implementation sizes in the order of 10,000 lines of C++ and assembler code it is about an order of magnitude smaller than Mach and two orders of magnitude smaller than Linux.

The operating system (OS) is clearly one of the most fundamental components of non-trivial systems. The correctness and reliability of the system critically depends on the OS. In terms of security, the OS is part of the trusted computing base, that is, the hardware and software necessary for the enforcement of a system's security policy. It has been repeatedly demonstrated that current operating systems fail at these requirements of correctness, reliability, and security. Microkernels address this problem by applying the principles of minimality and least privilege to operating system architecture. However, the success of this approach is still predicated on the microkernel being designed and implemented correctly. We can address this by formally modelling and verifying it.

The design of L4 is not only geared towards flexibility and reliability, but also is of a size which makes formalisation and verification feasible. Compared to other operating system kernels, L4 is very small; compared to the size of other verification efforts, 10,000 lines of code is still considered a very large and complex system. Our methodology for solving this verification problem is shown in Fig. 1. It is a classic refinement strategy. We start out from an abstract model of the kernel that is phrased in terms of user concepts as they are explained in

* National ICT Australia is funded through the Australian Government's *Backing Australia's Ability* initiative, in part through the Australian Research Council

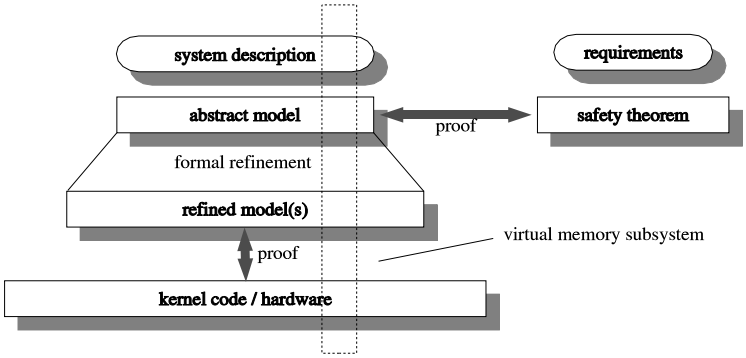


Fig. 1. Overview

the L4 reference manual [10]. This is the level at which most of the safety and security theorems will be shown. We then formally refine this abstract model in multiple property preserving steps towards the implementation of L4. The last step consists of verifying that the C++ and assembler source code of the kernel correctly implements the most concrete refinement level. At the end of this process, we will have shown that the kernel source code satisfies the safety and security properties we have proved about the abstract model.

In this paper we give an overview of some of the steps in this refinement process. L4 provides three main abstractions: threads, address spaces, and interprocess communication (IPC). We have chosen to start with address spaces. This is supported by the virtual memory subsystem of the kernel and is fundamental for implementing separation and security policies on top of L4. We first show an abstract model of address spaces, describe the framework in which the refinement process proceeds and then concentrate on the implementation of one particular operation of the abstract model. This operation is implemented in the Kernel using page tables of which we again first show an abstract view and then provide an implementation of some of its operations in a programming language that in its level of abstraction is close to C.

Earlier work on operating system kernel formalisation and verification includes PSOS [15] and UCLA Secure Unix [20]. The focus of this work was on capability-based security kernels, allowing security policies such as multi-level security to be enforced. These efforts were hampered by the lack of mechanisation and appropriate tools available at the time and so while the designs were formalised, the full verification proofs were not practical. Later work, such as KIT [2], describes verification of properties such as process isolation to source or object level but with kernels providing far simpler and less general abstractions than modern microkernels. There exists some work in the literature on the modelling of microkernels at the abstract level with varying degrees of completeness. Bevier and Smith [3] specify legal Mach states and describe Mach system calls using temporal logic. Shapiro and Weber [17] give an operational

semantics for EROS and prove a confinement security policy. Our work differs in that we plan to formally relate our model to the implementation. Some case studies [6, 4, 19] appear in the literature in which the IPC and scheduling subsystems of microkernels have been described in PROMELA and verified with the SPIN model checker. These abstractions were not necessarily sound, having been manually constructed from the implementations, and so while useful for discovering concurrency bugs do not provide guarantees of correctness. Finally, the VFiasco project, working with the Fiasco implementation of L4, has published exploratory work on the issues involved in C++ verification at the source level [7].

After introducing our notation in the following section, we first present the abstract conceptual model of virtual memory in L4 in section 3, and then show parts of the refinement of the memory lookup operation in this model towards a page table implementation in section 4.

2 Notation

Our meta-language Isabelle/HOL conforms largely to everyday mathematical notation. This section introduces further non-standard notation and in particular a few basic data types with their primitive operations.

The space of total functions is denoted by $\Rightarrow$. Type variables are written $'a$, $'b$, etc. The notation $t :: \tau$ means that HOL term t has HOL type τ.

The cons of an element x to a list xs is written $x \# xs$, and $[]$ is the empty list. Pairs come with the two projection functions $fst :: 'a \times 'b \Rightarrow 'a$ and $snd :: 'a \times 'b \Rightarrow 'b$. We identify tuples with pairs nested to the right: (a, b, c) is identical to $(a, (b, c))$ and $'a \times 'b \times 'c$ is identical to $'a \times ('b \times 'c)$.

$$\textbf{datatype}\ 'a\ option = None \mid Some\ 'a$$

adjoins a new element *None* to a type $'a$. For succinctness we write $\lfloor a \rfloor$ instead of *Some a*.

Function update is written $f(x := y)$ where $f :: 'a \Rightarrow 'b$, $x :: 'a$ and $y :: 'b$.

Partial functions are modelled as functions of type $'a \Rightarrow 'b\ option$, where *None* represents undefinedness and $f\ x = \lfloor y \rfloor$ means x is mapped to y. We call such functions *maps*, and abbreviate $f(x := \lfloor y \rfloor)$ to $f(x \mapsto y)$. The map $\lambda x.\ None$ is written *empty*, and $empty(\ldots)$, where $\ldots$ are updates, abbreviates to $[\ldots]$. For example, $empty(x \mapsto y)$ becomes $[x \mapsto y]$.

Implication is denoted by $\Longrightarrow$ and $[\![\ A_1; \ldots; A_n\ ]\!] \Longrightarrow A$ abbreviates $A_1 \Longrightarrow (\ldots \Longrightarrow (A_n \Longrightarrow A)\ldots)$.

Records in Isabelle [14], as familiar from programming languages, are essentially tuples with named fields. The type declaration

$$\begin{array}{l}\textbf{record}\ point = \\ \quad X :: nat \\ \quad Y :: nat\end{array}$$

creates a new record type *point* with two components X and Y of type *nat*. The notation $(\!|X{=}0,\ Y{=}0|\!)$ stands for the element of type *point* that has both components set to 0. Isabelle automatically creates two selector functions $X :: point \Rightarrow nat$ and $Y :: point \Rightarrow nat$ such that, e.g. $X\ (\!|X{=}0,\ Y{=}0|\!) = 0$. Updating field Y of a record p with value n is written $p\ (\!|\,Y := n|\!)$. As for function update, multiple record updates separated by comma are admitted.

3 Abstract Address Space Model

The virtual memory subsystem in L4 provides a flexible, hierarchical way of manipulating the mapping from virtual to physical memory pages of address spaces at user-level. We now present a formal model for address spaces. A first description of this model has already appeared in [9]. For completeness, we repeat parts of it in sections 3.1 and 3.2. The treatment of abstract datatypes in section 3.3 is updated to incorporate operations with output.

3.1 Address Spaces

Fig. 2 illustrates the concept of hierarchical mappings. Large boxes depict virtual address spaces. The smaller boxes inside stand for virtual pages in the address space. The rounded box at the bottom is the set of physical pages. The arrows stand for direct mappings which connect pages in one address spaces to addresses in (possibly) other address spaces. In well-behaved states, the transitive closure of mappings always ends in physical pages. The example in Fig. 2 maps virtual page v_1 in space n_1, as well as v_2 in n_2, and v_4 in n_4 to the physical page r_1.

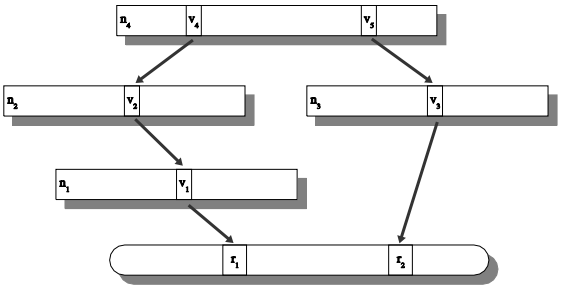


Fig. 2. Address Spaces

Formally, we use the types R for the physical pages (r_1, r_2, etc.), V for virtual pages (v_1, v_2, etc.), and N for the names of address spaces (n_1, n_2, etc.).

A position in this picture is determined uniquely by either naming a virtual page in a virtual address space, or by naming a physical page. We call these the mappings M:

$$\textbf{datatype}\ M = \mathit{Virtual}\ N\ V \mid \mathit{Real}\ R$$

An address space associates with each virtual page either a mapping, or nothing (the nil page). We implement this in Isabelle by the *option* datatype:

$$\textbf{types}\ \mathit{space} = V \Rightarrow M\ \mathit{option}$$

The machine state is then a map from address space names to address spaces. Not all names need to be associated with an address space, so we use *option* again:

$$\textbf{types}\ \mathit{state} = N \Rightarrow \mathit{space}\ \mathit{option}$$

To relate these functions to the arrows in Fig. 2, we use the concept of *paths*. The term $s \vdash x \rightsquigarrow^1 y$ means that in state s there is a direct path from position x to position y. There is a direct path from position $\mathit{Virtual}\ n\ v$ to another position y if in state s the address space with name n is defined and maps the virtual page v to y. There can be no paths starting at physical pages. Formally,

$$s \vdash x \rightsquigarrow^1 y = (\exists\, n\ v\ \sigma.\ x = \mathit{Virtual}\ n\ v \wedge s\ n = \lfloor\sigma\rfloor \wedge \sigma\ v = \lfloor y\rfloor)$$

We write $\_ \vdash \_ \rightsquigarrow^+ \_$ for the transitive and $\_ \vdash \_ \rightsquigarrow^* \_$ for the reflexive and transitive closure of the direct path relation.

3.2 Operations

The L4 kernel exports the following basic operations on address spaces: *unmap*, *flush*, *map*, and *grant*. The former two operations remove mappings, the latter two create or move mappings. We explain and define them below.

Fig. 3 illustrates the *unmap* n v operation. It is the most fundamental of the operations above. We say a space n unmaps v if it removes all mappings that depend on $\mathit{Virtual}\ n\ v$, or in terms of paths if it removes all edges leading to $\mathit{Virtual}\ n\ v$.

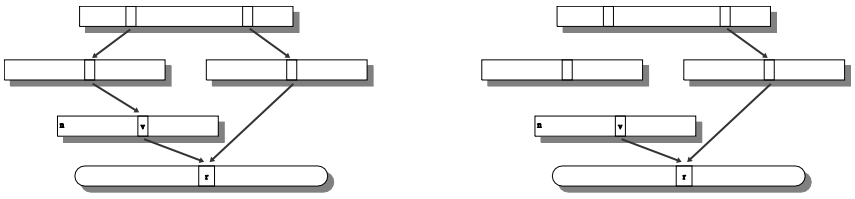


Fig. 3. The *unmap* operation (before and after)

To implement this, we use a function *clear* that, given name n, page v, and address space σ in a state s, returns σ with all v' leading to $\mathit{Virtual}\ n\ v$ mapped to *None*.

$clear :: N \Rightarrow V \Rightarrow state \Rightarrow space \Rightarrow space$
$clear\ n\ v\ s\ \sigma \equiv$
$\lambda v'.\ \mathsf{case}\ \sigma\ v'\ \mathsf{of}\ None \Rightarrow None$
$\quad \mid \lfloor m \rfloor \Rightarrow \mathsf{if}\ s \vdash m \rightsquigarrow^* Virtual\ n\ v\ \mathsf{then}\ None\ \mathsf{else}\ \lfloor m \rfloor$

An *unmap* $n\ v$ in state s then produces a new state in which each address space is cleared of all paths leading to $Virtual\ n\ v$.

$unmap :: N \Rightarrow V \Rightarrow state \Rightarrow state$
$unmap\ n\ v\ s \equiv \lambda n'.\ \mathsf{case}\ s\ n'\ \mathsf{of}\ None \Rightarrow None \mid \lfloor \sigma \rfloor \Rightarrow \lfloor clear\ n\ v\ s\ \sigma \rfloor$

For updating a space with name n at page v with a new mapping m we write $n,v \leftarrow m$, where m may be *None*.

$n,v \leftarrow m \equiv \lambda s.\ s(n := \mathsf{case}\ s\ n\ \mathsf{of}\ None \Rightarrow None \mid \lfloor \sigma \rfloor \Rightarrow \lfloor \sigma(v := m) \rfloor)$

With this, the flush operation is simply *unmap* followed by setting n,v to *None*.

$flush :: N \Rightarrow V \Rightarrow state \Rightarrow state$
$flush\ n\ v \equiv n,v \leftarrow None \circ unmap\ n\ v$

The remaining two operations *map* and *grant* establish new mappings in the receiving address space. To ensure a consistent new state, this new mapping must ultimately be connected to a physical page. We call a mapping m *valid* in state s (written $s \vdash m$) if it is a physical page, or if it is of the form $Virtual\ n\ v$ and is the source of some direct path. We show later that in all reachable states of the system, this definition is equivalent to saying that the mapping leads to a physical page.

$s \vdash m \equiv \mathsf{case}\ m\ \mathsf{of}\ Virtual\ n\ v \Rightarrow \exists x.\ s \vdash m \rightsquigarrow^1 x \mid Real\ r \Rightarrow True$

Before the kernel establishes a new value, the destination is always flushed. This may invalidate the source. The operation only continues if the source is still valid, otherwise it stops. We capture this behaviour in a slightly modified update notation $\leftharpoonup$:

$n,v \leftharpoonup m \equiv \lambda s.\ \mathsf{let}\ s_0 = flush\ n\ v\ s\ \mathsf{in}\ (\mathsf{if}\ s_0 \vdash m\ \mathsf{then}\ n,v \leftarrow \lfloor m \rfloor\ \mathsf{else}\ id)\ s_0$

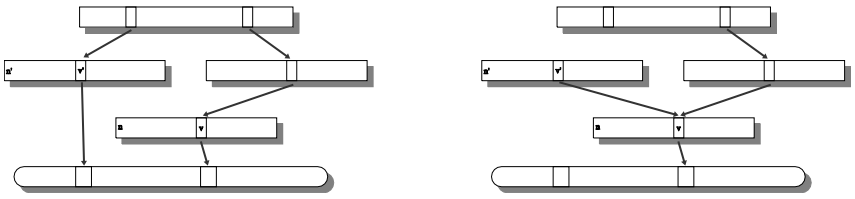


Fig. 4. The *map* operation (before and after)

In L4, an address space n can *map* a page v to another space n' at page v'. Again, the operation only goes ahead, if the mapping $Virtual\ n\ v$ is valid:

$map :: N \Rightarrow V \Rightarrow N \Rightarrow V \Rightarrow state \Rightarrow state$
$map\ n\ v\ n'\ v'\ s \equiv$ *if* $\neg\ s \vdash Virtual\ n\ v$ *then* s *else* $(n',v' \leftharpoonup Virtual\ n\ v)\ s$

Fig. 4 shows an example for the *map* operation. Address space n maps page v to n' at v'. The destination n',v' is first flushed and then updated with the new mapping $Virtual\ n\ v$.

A space n can also *grant* a page v to v' in n'. As illustrated in Fig. 5, granting updates n',v' to the value of n at v and flushes the source n,v.

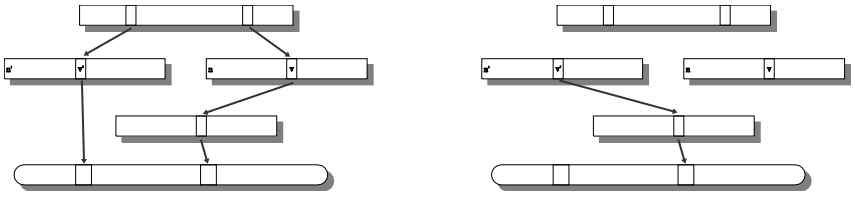


Fig. 5. The *grant* operation (before and after)

$grant :: N \Rightarrow V \Rightarrow N \Rightarrow V \Rightarrow state \Rightarrow state$
$grant\ n\ v\ n'\ v'\ s \equiv$
if $\neg\ s \vdash Virtual\ n\ v$ *then* s
else let $\lfloor\sigma\rfloor = s\ n$; $\lfloor m\rfloor = \sigma\ v$ *in* $(flush\ n\ v \circ n',v' \leftharpoonup m)\ s$

This concludes the kernel operations on address spaces. We have also modelled the hardware memory management unit (MMU). On this abstract level, all the MMU does is lookup: it determines which physical page needs to be accessed for each virtual page v and address space n. We write $s \vdash n,v \rhd \lfloor r\rfloor$ if lookup of page v in the address space with name n in state s yields the physical page r. As we already have the concepts of paths, this is easily described formally:

$s \vdash n,v \rhd \lfloor r\rfloor = s \vdash Virtual\ n\ v \rightsquigarrow^+ Real\ r$
$s \vdash n,v \rhd None = (\exists \sigma.\ s\ n = \lfloor\sigma\rfloor \wedge \sigma\ v = None) \vee s\ n = None$

The model in this section is based on an earlier pen-and-paper formalisation of L4 address spaces by Liedtke [12]. Formalising it in Isabelle/HOL eliminated problems like the mutual recursive definition of the update and flush functions being not well-founded. It would be well-founded—at least on reachable kernel states—if the model had the property that no loops can be constructed in address spaces. This is not true in the original model. The operation $map\ n\ v\ n'\ v'$ followed by $grant\ n'\ v'\ n\ v$ is a counter example. We have introduced the formal concept of valid mappings to establish this no-loops property as well as the fact that any page that is mapped at all is mapped to a physical address.

3.3 An abstract data type for virtual memory

In the following we phrase the model of virtual memory and of the MMU hardware in terms of an abstract data type consisting of the type *state* and the

operations detailed above. This data type (not to be confused with Isabelle's keyword **datatype**) is used implicitly by any user-level program. Even if the program does not invoke any mapping operations directly, the CPU performs a lookup operation with every memory access.

Putting the operations in terms of an abstract data type enables us to formulate refinement explicitly: if the data type of the abstract address spaces model is replaced with the data type of more concrete models (and finally the implementation) the program will not have any observable differences in behaviour.

Formally we define an abstract data type as a record consisting of an initial set of states and of a transition relation that models execution with return values of type $'o$:

$$\begin{array}{l}\textbf{record}\ ('a,\ 'j,\ 'o)\ \mathit{DataType} = \\ \mathit{Init} :: 'a\ \mathit{set} \\ \mathit{Step} :: 'j \to ('a \times 'a \times 'o)\ \mathit{set}\end{array}$$

For our virtual memory model, the operations are enumerated in the index type *VMIndex*:

$$\begin{array}{l}\textbf{datatype}\ \mathit{VMIndex} = \mathsf{create}\ N \mid \mathsf{unmap}\ N\ V \mid \mathsf{flush}\ N\ V \mid \mathsf{map}\ N\ V\ N\ V \\ \qquad\qquad \mid \mathsf{grant}\ N\ V\ N\ V \mid \mathsf{lookup}\ N\ V\end{array}$$

The abstract model $\mathcal{A}$ in terms of a (*state*, *VMIndex*, *R option*) *DataType* is then:

$\mathit{Init}\ \mathcal{A} = \{[\sigma_0 \mapsto \sigma]\ |\sigma.\ \mathit{inj}_p\ \sigma \wedge \mathit{ran}\ \sigma \subseteq \mathit{range}\ \mathit{Real}\}$
$\mathit{Step}\ \mathcal{A}\ (\mathsf{lookup}\ n\ v) = \{(s,\ s',\ r) \mid s = s' \wedge s \vdash n{,}v \rhd r\}$
$\mathit{Step}\ \mathcal{A}\ (\mathsf{create}\ n) = \{(s,\ s',\ r) \mid r = \mathit{None} \wedge s\ n = \mathit{None} \wedge s' = s(n \mapsto \mathit{empty})\}$
$\mathit{Step}\ \mathcal{A}\ (\mathsf{unmap}\ n\ v) = \{(s,\ s',\ r) \mid r = \mathit{None} \wedge s\ n \neq \mathit{None} \wedge s' = \mathit{unmap}\ n\ v\ s\}$
$\mathit{Step}\ \mathcal{A}\ (\mathsf{flush}\ n\ v) = \{(s,\ s',\ r) \mid r = \mathit{None} \wedge s\ n \neq \mathit{None} \wedge s' = \mathit{flush}\ n\ v\ s\}$
$\mathit{Step}\ \mathcal{A}\ (\mathsf{map}\ n\ v\ n'\ v') =$
$\{(s,\ s',\ r) \mid r = \mathit{None} \wedge s\ n \neq \mathit{None} \wedge s\ n' \neq \mathit{None} \wedge s' = \mathit{map}\ n\ v\ n'\ v'\ s\}$
$\mathit{Step}\ \mathcal{A}\ (\mathsf{grant}\ n\ v\ n'\ v') =$
$\{(s,\ s',\ r) \mid r = \mathit{None} \wedge s\ n \neq \mathit{None} \wedge s\ n' \neq \mathit{None} \wedge s' = \mathit{grant}\ n\ v\ n'\ v'\ s\}$

The boot process creates an address space σ_0 that is an injective mapping from virtual to physical pages. The functions *ran* and *range* return the codomain of a function, where *ran* works on functions $'a \Rightarrow 'b\ \mathit{option}$ and *range* on total functions. Injectivity is constrained to the part of the function that returns $\lfloor x \rfloor$: $\mathit{inj}_p\ f \equiv \mathit{inj\text{-}on}\ f\ \{x \mid \exists\, y.\ f\ x = \lfloor y \rfloor\}$.

The lookup operation is the only operation that returns a value. All other operations return *None*.

Creating a new address space n is modelled by updating the state s at n with the predefined map *empty*. The other mapping operations have been defined above. All of them require the address spaces they operate on to exist. This condition is ensured automatically in the current L4 implementation as the address spaces are determined by sender and receiver of an IPC operation.

The correctness of the implementation with respect to the abstraction is established by showing the concrete model to be a refinement of the abstract

model. Here refinement is taken to mean *data refinement* [5] and we use the proof technique of simulation. Simulation between an abstract $('a, 'j, 'o)$ *DataType* and a concrete $('c, 'j, 'o)$ *DataType* is formalized as follows.

The step relations for each operation are of type $('a \times 'a \times 'o)$ *set*. It is convenient to have a relation for these operations of type $('a \times 'o) \times ('a \times 'o)$ below, so we introduce the function *up*. This gives the semantics of the operations on the state space $'a \times 'o$. Since the value of the $'o$ component in the pre-state has no effect on the semantics of the operations in an ADT it can be left unrestricted.

$$up\ r \equiv \{((a,\ i),\ b,\ k) \mid (a,\ b,\ k) \in r\}$$

The type of the abstraction relation r is $('a \times 'c)$ *set*. id_o lifts this to $('a \times 'o) \times ('c \times 'o)$.

$$id_o\ r \equiv \{((s,\ k),\ s',\ k') \mid k = k' \wedge (s,\ s') \in r\}$$

A relation c is an L-subset of a relation a under the relation r if the following holds, where $a\ ;\ b$ is relational compososition of a and b.

$$r \vdash c \subseteq^L a \equiv r\ ;\ c \subseteq a\ ;\ r$$

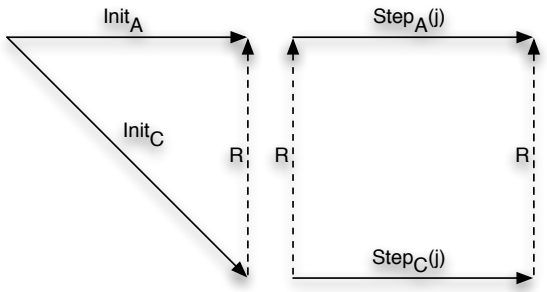


Fig. 6. Simulation

A forward simulation exists if the diagrams in Figure 6 commute. That is, there exists a relation such that the initial states of the concrete model are a subset of those in the abstract model under the relational image of the lifted abstraction relation, and if for each step operation the concrete step is an L-subset of the abstract step relation under r.

$Lr\ r\ C\ A \equiv$
let $r_o = id_o\ r$
in $Init\ C \times UNIV \subseteq r_o\ ``\ (Init\ A \times UNIV) \wedge$
$(\forall j.\ r_o \vdash up\ (Step\ C\ j) \subseteq^L up\ (Step\ A\ j))$

We write $C \leq_F A$ when concrete data type C simulates abstract data type A:

$$C \leq_F A \equiv \exists r.\ Lr\ r\ C\ A$$

3.4 Properties

We have shown a number of safety properties about the abstract address space model. They are formulated as invariants over the abstract datatype. A set of states I is an invariant if it contains all initial states and if execution of any operation in a state of I again leads to a state in I. We write $\mathcal{D} \models I$ when I is an invariant of data type $\mathcal{D}$.

Theorem 1. *There are no loops in the address space structure.*

$$\mathcal{A} \models \{s \mid \forall x.\ \neg\ s \vdash x \rightsquigarrow^+ x\}$$

The proof is by case distinction on the operations and proceeds by observing how each operation changes existing paths. Theorem 1 is significant for implementing the lookup function efficiently. It also ensures that internal kernel functions can walk the corresponding data structures naively. Together with the properties below it says that address spaces always have a tree structure.

Theorem 2. *All valid pages translate to physical pages.*

$$\mathcal{A} \models \{s \mid \forall x.\ s \vdash x \longrightarrow (\exists r.\ s \vdash x \rightsquigarrow^*\ Real\ r)\}$$

The proof is again by case distinction on the operations. Together with the following theorem we obtain that address lookup is a total function on data type $\mathcal{A}$.

Theorem 3. *The lookup relation is a function.*

$$[\![s \vdash n,v \rhd r;\ s \vdash n,v \rhd r']\!] \Longrightarrow r = r'$$

This theorem follows directly from the fact that paths are built on functions.

That address lookup is a total function may sound like merely a nice formal property, but it is quite literally an important safety property in reality. Undefined behaviour, possibly physical damage, may result if two conflicting TLB entries are present for the same virtual address. The current ARM reference manual [1, p. B3-26] explicitly warns against this scenario.

3.5 Simplifications and Assumptions

The current model makes the following simplifications and assumptions.

– The L4Ka::Pistachio API stipulates two regions per address space that are shared between the user and kernel, the *kernel interface page* (KIP) and *user thread control blocks* (UTCBs). These should have a valid translation from virtual to physical memory pages, but can not be manipulated by the mapping operations.

– The mapping operations in L4 work on regions of the address space rather than individual pages. These regions, known as *flexpages*, are $2^k b, k \geq 0$ aligned and sized where b is the minimum page size on the architecture. This introduces significant complexity in the implementation and has a number of boundary conditions of interest, so adding this to the abstract model would be beneficial. At the same time, it is possible to create systems using L4 that only use the minimum flexpage size so this omission does not pose a serious limitation to the utility of the model.
– *map* and *grant* are implemented through the IPC primitives in L4 and involve an agreement on the region to be transferred between sender and receiver. This can be added when the IPC abstraction is modelled.
– Flexpages also have associated read, write and execute access rights. At present the model can be considered as providing an all or nothing view of access rights.
– We assume that all of the mapping operations are atomic, which is the case in the current non-preemptable implementation, and a single processor, hence a sequential system.

4 Page Tables

The model in the previous section provides an abstract model of address spaces in L4 but does not bear much resemblance to the kernel implementation. This is not surprising since the kernel must provide an efficient realisation of the mapping operations and the code supporting this executes under time and space restrictions.

Below we consider the refinement of one component of the virtual memory subsystem necessary for the implementation of address spaces. The models and interfaces below are based on the existing page table implementation in the L4Ka::Pistachio [11] kernel.

4.1 Abstract model

The implementation of address spaces is provided by the hardware and OS virtual memory mechanisms. The lookup relation corresponds to the virtual-to-physical mapping function provided by the MMU on the CPU. This translation is carried out on every memory access and so is critical to system performance. It is typically hidden in the processor pipeline by an associative cache, called the *translation-lookaside buffer* (TLB). This holds a subset of mappings from the *page table* data structure which is located in memory. The TLB caches *page table entries* (PTEs), as in Figure 7 — a PTE for a page in the virtual address space specifies the corresponding physical page, access rights, and other page specific information, shown in Figure 8. On a TLB miss a hardware mechanism[1] traverses the page table data structure to perform address translation.

[1] On the ARM architecture. Other architectures might also rely on software mechanisms to achieve the same goal.

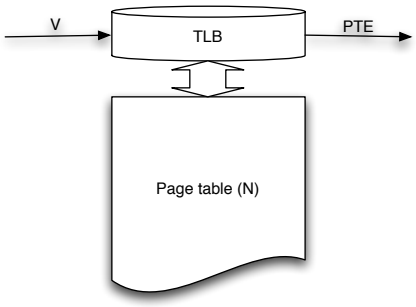


Fig. 7. PTE lookup through the TLB

The design of page table implementations is influenced by the direct and indirect performance costs of this operation.

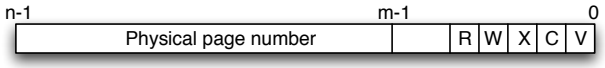


Fig. 8. Page table entry (PTE)

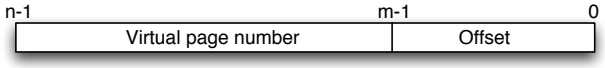


Fig. 9. Virtual address

While we treated virtual page numbers and virtual addresses interchangeably in the previous section, this will no longer be sufficient when considering the specifics of page table implementations, since modern TLBs usually support multiple page sizes, called *superpages* [18], in order to improve the coverage of the TLB; a single PTE (and TLB entry) can then cover large regions of the address space. Hence many virtual pages may be associated with a single virtual address. An n-bit virtual addresses can be considered as consisting of an $(n-m)$-bit virtual page number and an m-bit offset, as in Figure 9, where 2^m is in the set of page sizes supported by the the architecture. Mappings are then from 2^m

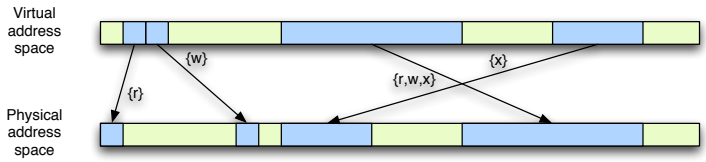


Fig. 10. Mappings from virtual to physical pages

sized, aligned regions of the virtual address space to the physical address space, shown in Figure 10.

Virtual addresses are modelled using a theory of fixed-width words in Isabelle, where the word type is a quotient type with equivalence classes derived by taking the natural numbers modulo the word size. The theory is imported from the HOL4 system. We use the *word*32 type for virtual addresses, although nothing should depend on this particular value for word size and the model and proofs presented here should be the same for any size of virtual addresses.

$$\textbf{types}\ V = \mathit{word}32$$

The function *page-bits ps* gives the value of m for page size *ps*. We introduce the type *PTESize* of which values are supported (super)page sizes.

$$\textbf{consts}\ \mathit{page\text{-}bits} :: \mathit{PTESize} \Rightarrow \mathit{nat}$$

The *vpn* for a virtual address is its virtual page number as a *nat*. The function $w2n$ converts from a value of type *word*32 to the corresponding *nat*. $n2w$ does the reverse.

$$\mathit{vpn}\ v\ \mathit{ps} \equiv w2n\ v\ \mathit{div}\ 2\ \hat{}\ \mathit{page\text{-}bits}\ \mathit{ps}$$

At a given page size, two virtual addresses with identical virtual page numbers are of the same page.

$$\mathit{page\text{-}equiv}\ l\ v\ v' \equiv \mathit{vpn}\ v\ l = \mathit{vpn}\ v'\ l$$

page-set v ps gives the set of virtual addresses for the page of size *ps* containing the virtual address v. Figure 11 gives two example *page-set*s for a virtual address v with superpage sizes m_0 and m_1.

$$\mathit{page\text{-}set}\ v\ l \equiv \{v' \mid \mathit{page\text{-}equiv}\ l\ v\ v'\}$$

We begin with the description of the state space for the abstract model by introducing several new types. We model page tables as function from $N \times V$ to a pointer to a PTE stored in a heap. We choose to model explicitly at the abstract

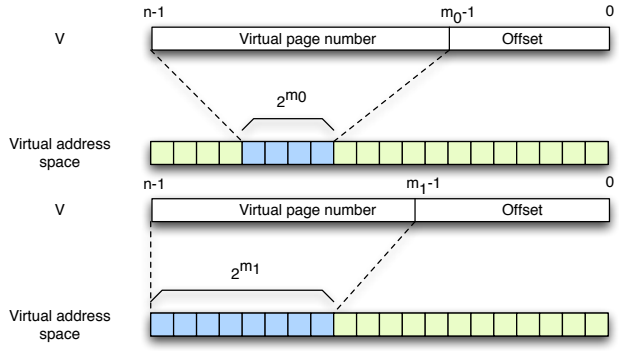


Fig. 11. Example *page-set*s

level indirection with respect to PTEs, based on the interface observed in the L4Ka::Pistachio linear page table implementation. This allows for efficient implementation of operations that modify PTEs since unnecessary traversal of the page table can be avoided. The type of PTE pointers is *PTEName*. In addition, a pointer type *TreeListNodeName* is introduced. The page table ADT is utilised by the mapping database (MDB) that stores the map/grant relationships between address spaces as illustrated in section 3.1. In addition to the fields that are usually present in a PTE, the MDB requires that each PTE has the corresponding virtual address and a pointer of type *TreeListNodeName* associated with it.

An abstract PTE is modelled as a record type. *Paddr* contains the physical page number for the page, R, W, X specify the access rights for this mapping, and *Cached* indicates whether the data accessed through this mapping may be stored in the data or instruction caches. As part of the MDB required interface we also conceptually associate the two additional fields *MapNode* and *Vaddr* with the PTE as explained above.

$$
\begin{array}{l}
\textbf{record}\ PTE_a = \\
\quad Paddr :: R \\
\quad R :: bool \\
\quad W :: bool \\
\quad X :: bool \\
\quad Cached :: bool \\
\quad MapNode :: TreeListNodeName \\
\quad Vaddr :: V
\end{array}
$$

The state space is then a partial function from $N \times V$ to the PTE pointer for the mapping and the size of the mapping, and a heap for PTEs. In addition, the N field of *PTState* stores which address spaces are currently active (have been created).

types $PageTable = N \times V \Rightarrow (PTEName \times PTESize)\ option$
types $PTEHeap = PTEName \Rightarrow PTE_a$

record $PTState =$
$N :: N\ list$
$Heap :: PTEHeap$
$PageTable :: PageTable$

The operations provided by the page table ADT and their return types are enumerated in the following two type declarations.

datatype $PTIndex =$ insert $N\ V\ PTESize$
| lookup $N\ V$
| getpaddr $PTEName\ PTESize$
| setpaddr $PTEName\ R\ PTESize$
| setlinknode $PTEName\ TreeListNodeName\ V\ PTESize$
| getmapnode $PTEName\ V\ PTESize$
| createspace

datatype $PTResult = RInsert\ (PTEName \times PTESize)\ option$
| $RLookup\ (PTEName \times PTESize)\ option$
| $RGetPaddr\ R \mid RSetPaddr \mid RSetLinkNode$
| $RGetMapNode\ TreeListNodeName \mid RCreateSpace\ N$

The ADT definition follows, with the semantics of these operations described further below. The *get* and *set* operations for the physical page number (getpaddr and setpaddr) are given, but omitted for the other fields (R,W,X,$Cached$) with the exception of the last two, since they are identical in all but name.

$Init\ \mathcal{P} = \{x \mid N\ x = [] \wedge PageTable\ x = empty\}$
$Step\ \mathcal{P}$ createspace $= create\text{-}space_a$
$Step\ \mathcal{P}$ (lookup $n\ v$) $= lookup_a\ n\ v$
$Step\ \mathcal{P}$ (insert $n\ v\ ps$) $= insert_a\ n\ v\ ps$
$Step\ \mathcal{P}$ (setpaddr $p\ r\ ps$) $= set\text{-}paddr_a\ p\ r\ ps$
$Step\ \mathcal{P}$ (getpaddr $p\ ps$) $= get\text{-}paddr_a\ p\ ps$
$Step\ \mathcal{P}$ (setlinknode $p\ m\ v\ ps$) $= set\text{-}link\text{-}node_a\ p\ m\ v\ ps$
$Step\ \mathcal{P}$ (getmapnode $p\ v\ ps$) $= get\text{-}map\text{-}node_a\ p\ v\ ps$

In the initial state there are no valid address spaces and hence no mappings.

Before mappings can be added, new address spaces need to be created. The createspace operation picks and returns the name of the new address space non-deterministically. It must be distinct from the name of any existing address space.

$create\text{-}space_a \equiv$
$\{(s,\ s',\ r) \mid \exists n.\ n \notin set\ (N\ s) \wedge s' = s(\!|N := n\ \#\ N\ s|\!) \wedge r = RCreateSpace\ n\}$

Lookup returns the PTE pointer and size of the mapping that contains v assuming a valid address space n is specified. The onus is on the caller to supply a valid address space to avoid a potentially unnecessary check for validity on each invocation of this operation.

$$lookup_a\ n\ v \equiv$$
$$\{(s,\ s',\ r) \mid n \in set\ (N\ s) \longrightarrow s' = s \wedge r = RLookup\ (PageTable\ s\ (n,\ v))\}$$

Insertion of new mappings is the most complicated of the operations in this model. Assuming a valid address space argument is supplied, there are two possibilities here, depending on whether the mapping overlaps an existing mapping. If an overlap exists then a conflicting mapping is not inserted. Conflicts occur if the *page-set* for v at the given page size ps has a non-empty intersection with the set of currently mapped virtual addresses. Figure 12 shows two examples of this.

$$valid\text{-}vaddr\ pt\ n \equiv \{x \mid pt\ (n,\ x) \neq None\}$$

$$conflict\ n\ v\ ps\ pt \equiv valid\text{-}vaddr\ pt\ n \cap page\text{-}set\ v\ ps \neq \{\}$$

If there is no conflict, *update-page-table* gives the new state, where an unused location in the PTE heap is selected and all virtual addresses in *page-set v ps* are set to this value with the given page size. The choice of location in the heap is non-deterministic in this model.

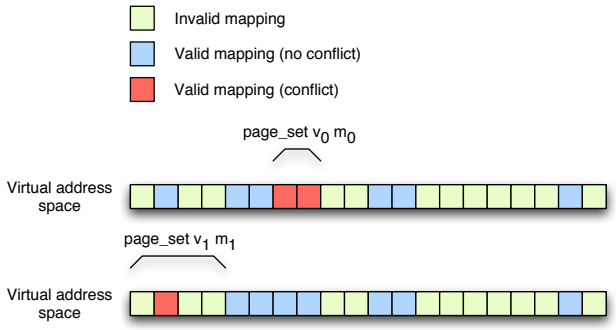


Fig. 12. Example conflicts with existing mappings

$$\begin{aligned}
&update\text{-}page\text{-}table\ n\ v\ ps \equiv \\
&\{(s,\ s') \mid \exists p.\ \neg\ \ PageTable\ s \vdash_p p\ \wedge \\
&\qquad s' = \\
&\qquad s(\!|PageTable := \\
&\qquad\quad \lambda(n',\ v'). \\
&\qquad\quad\ \textit{if}\ n' = n \wedge v' \in page\text{-}set\ v\ ps\ \textit{then}\ \lfloor (p,\ ps) \rfloor \\
&\qquad\quad\ \textit{else}\ PageTable\ s\ (n',\ v')|\!)\}
\end{aligned}$$

The result of this operation in the case of a successful insertion is the PTE pointer for the new mapping. If insertion was not successful, but the existing mapping fully contains the new one, a pointer to the existing mapping is returned. Otherwise the operation returns *None*.

$$
\begin{array}{l}
\mathit{insert\text{-}result}\ n\ v\ ps\ s \equiv \\
\mathsf{case}\ \mathit{PageTable}\ s\ (n,\ v)\ \mathsf{of}\ \mathit{None} \Rightarrow \mathit{None} \\
\mid \lfloor (p,\ ps') \rfloor \Rightarrow \mathsf{if}\ \mathit{w2n}\ ps < \mathit{w2n}\ ps'\ \mathsf{then}\ \lfloor (p,\ ps') \rfloor\ \mathsf{else}\ \mathit{None}
\end{array}
$$

$$
\begin{array}{l}
\mathit{insert}_a\ n\ v\ ps \equiv \\
\{(s,\ s',\ r) \mid n \in \mathit{set}\ (N\ s) \longrightarrow \\
\qquad (\mathsf{if}\ \mathit{conflict}\ n\ v\ ps\ (\mathit{PageTable}\ s) \\
\qquad\ \mathsf{then}\ (s,\ s') \in \mathit{update\text{-}page\text{-}table}\ n\ v\ ps\ \mathsf{else}\ s = s') \wedge \\
\qquad r = \mathit{RInsert}\ (\mathit{insert\text{-}result}\ n\ v\ ps\ s')\}
\end{array}
$$

The fields of the PTE can be set and retrieved through the heap in the following operations. For each operation it is a precondition that the PTE pointer supplied is valid and the page size at which it is mapped is also correctly provided as an argument. A PTE pointer p is considered valid, $pt \vdash_p p$, if it is in the image of the page table pt for some page size ps, $pt \vdash_p p,ps$ if the pair $(p,\ ps)$ is in this image. The notation $f\ `\ A$ stands for the image of set A under f.

$$pt \vdash_p p \equiv p \in \mathit{fst}\ `\ \mathit{ran}\ pt$$

$$pt \vdash_p p,ps \equiv (p,\ ps) \in \mathit{ran}\ pt$$

$$
\begin{array}{l}
\mathit{set\text{-}paddr}_a\ p\ \mathit{paddr}\ ps \equiv \\
\{(s,\ s',\ r) \mid \mathit{PageTable}\ s \vdash_p p,ps \longrightarrow \\
\qquad s' = s(\!|\mathit{Heap} := (\mathit{Heap}\ s)(p := \mathit{Heap}\ s\ p(\!|\mathit{Paddr} := \mathit{paddr}|\!))|\!) \wedge \\
\qquad r = \mathit{RSetPaddr}\}
\end{array}
$$

$$
\begin{array}{l}
\mathit{get\text{-}paddr}_a\ p\ ps \equiv \\
\{(s,\ s',\ r) \mid \mathit{PageTable}\ s \vdash_p p,ps \longrightarrow s' = s \wedge \\
\qquad r = \mathit{RGetPaddr}\ (\mathit{Paddr}\ (\mathit{Heap}\ s\ p))\}
\end{array}
$$

Associated with each PTE is a *link node*. While not part of the fundamental page table abstraction, this is required for the mapping database and other operations in the kernel. The link node stores the virtual address and a pointer to a corresponding node in the MDB for the mapping. In the implementation this is optimised to be a single field, with the bitwise exclusive-OR of the values stored in the link node. We model this by requiring the complementary value to be passed as an argument to the inspection operations.

$$
\begin{array}{l}
\mathit{set\text{-}link\text{-}node}_a\ p\ m\ v\ ps \equiv \\
\{(s,\ s',\ r) \mid \mathit{PageTable}\ s \vdash_p p,ps \longrightarrow \\
\qquad s' = \\
\qquad s(\!|\mathit{Heap} := (\mathit{Heap}\ s)(p := \mathit{Heap}\ s\ p(\!|\mathit{MapNode} := m,\ \mathit{Vaddr} := v|\!))|\!) \wedge \\
\qquad r = \mathit{RSetPaddr}\}
\end{array}
$$

$$
\begin{array}{l}
\mathit{get\text{-}map\text{-}node}_a\ p\ v\ ps \equiv \\
\{(s,\ s',\ r) \mid \mathit{PageTable}\ s \vdash_p p,ps \wedge v = \mathit{Vaddr}\ (\mathit{Heap}\ s\ p) \longrightarrow s' = s \wedge \\
\qquad r = \mathit{RGetMapNode}\ (\mathit{MapNode}\ (\mathit{Heap}\ s\ p))\}
\end{array}
$$

Some features that should be present in a complete page table model have been omitted here and are currently being added to the model. These range from fairly trivial changes to those necessary to increase the generality of the model.

An example of a small omission is the status and cache control bits in the PTE which are not included for brevity. These do not differ conceptually from other fields in the PTE such as permission bits for the purpose of this model. A more important limitation is that on some architectures it may not be possible to insert a mapping even if no conflicts exist due to the page table structure being affected by nearby mappings. A hardware model for the TLB and page table walker can be added to provide a *lookup* operation as described in the abstract address spaces ADT, and to supply semantics for the cache and status bits in the PTE. Finally, we assume translation and protection granularity are identical which is not the case in general, for example sub-page permissions on the ARM architecture.

4.2 Concrete model

A simple way to implement the page table would be to use a linear array in physical memory, indexed with the virtual page number. This would have the advantage of a fast lookup time, which is desirable as the page table lookup operation is a major component of the TLB refill cost. Unfortunately, this is wasteful of physical memory and does not scale with larger address spaces. For example, consider a 32-bit virtual address space, with 4KB pages and a 24-bit physical address space with 4KB frames[2]. Assume each page table entry is a single word, 4 bytes. The frame number can easily be stored in the PTE, requiring only 12 bits. However, the array will require 2^{20} PTE locations, and hence require 2^{22} bytes of contiguous storage in physical memory, which may potentially only be used sparsely. In addition, this has poor support for superpages, with large superpages requiring massive duplication of PTEs, making insertion and PTE update operations costly.

Modern architectures and operating systems therefore use data structures that balance the requirement for fast traversal and memory use considerations. These include multi-level page tables, inverse page tables, hashed page tables [8] and guarded page tables [13]. L4Ka::Pistachio implements a multi-level hierarchical page table (MLPT). The page table format defined by the ARM hardware, a two-level page table, is an instance of this.

MLPTs are tree data structures where each node contains an array of a fixed, level dependent, size. Elements of these arrays are either invalid, leaves corresponding to PTEs in the abstract model, or pointers to the next level. They provide both storage for valid PTE heap entries and the mapping function from V to $PTEName$ for an address space N. Lookup proceeds by indexing the root table, with a base address equivalent to the address space name, with the most significant k_t bits of the virtual address, where k_i is the number of bits in the index field for level i and the page table has a maximum of $t + 1$ levels. Figure 13 shows a two-level page table with the root page table indexing occurring at level 1. Each entry of the table corresponds to a contiguous region of the address space. If the address space is of size 2^n then the array will have 2^{k_t} entries and

[2] Physical pages are also called frames.

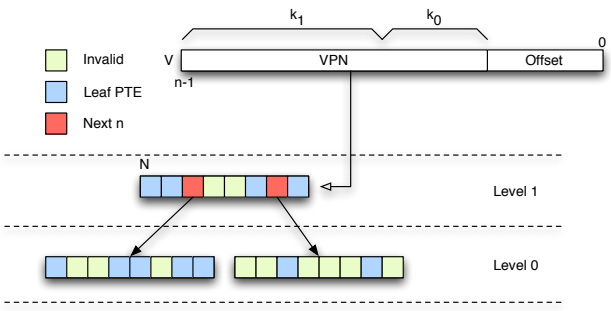


Fig. 13. Indexing during lookup and insertion

each entry will map a 2^{n-k_t} region. If either an invalid entry or leaf PTE is found at the index then a pointer to this is returned. Otherwise, if the entry points to a table at the next level then the algorithm recurses, with the next table, the $n - k_t$ least significant bits of the virtual address and the next index size, k_{t-1}, until either a valid PTE is found or the bottom level is reached and a pointer to the indexed entry is returned. If the returned pointer references a leaf PTE then a valid (in the sense of the abstract model) mapping exists for the virtual address.

Assuming no conflicts, insertion works similarly, with the exception that a new node of appropriate size is allocated and linked to when an invalid node is indexed at a level above the intended insertion point.

We describe the two larger operations, lookup and insertion, from the concrete model below. We utilise the verification environment of Schirmer [16] with custom pretty-printing to provide C-like syntax. Keywords, procedure names, and program variables referring to the current state are printed in typewriter font. Normal Isabelle functions and constants are unchanged. In the Hoare triple $\{\!|\sigma.\ P|\!\}\ s\ \{\!|\ Q\ ^{\sigma}n\ |\!\}$, the name σ is bound to the pre-state, and $^{\sigma}n$ refers to the program variable n in state σ.

The concrete state space has 3 components in its global state — a heap for nodes in the page table *pt-h*, a list of pointers allocated in the heap *v-pt* and the set of currently active address spaces *vN*. We model arrays as lists and hence the type of *pt-h* is given by

$$\textbf{types}\ \mathit{PageTableHeap} = \mathit{PTabName} \Rightarrow \mathit{PTE\ list}$$

where *PTabName* is the type of pointers to page table nodes and *PTE* is defined as the disjoint union:

$$\textbf{datatype}\ \mathit{PTE} = \mathit{Leaf}\ \mathit{PTE}_a \mid \mathit{Next}\ \mathit{PTabName} \mid \mathit{Invalid}$$

The type *PTEName* is now a pointer to an array entry and hence consists of two components — a pointer to the base of the array and an index of type *PTabOffset*.

$$\textbf{types}\ PTEName = PTabName \times PTabOffset$$

For convenience we introduce two abbreviations when working with *PTEName* pointers. The *PTEName* for a virtual address v at level l and node n is given by $\varphi\ n,v,l$. The *PTE* for a *PTEName* pointer in page table heap *pt-h* is written as ψ *pt-h p*.

The names of address spaces are now synonymous with the root node in the page table.

$$\textbf{types}\ N = PTabName$$

The source code for the page table lookup operation is given in Figure 15. The parameters are found in variables n and v. Variables with a `tmp_` prefix are local to this function. Various functions are called inside the body of this function. One such example is `ptab_index` which performs the indexing operation for a given page table level, shown in Figure 14.

```
procedures ptab-index(n,v,l|r2) =
r = v >> hw_pgshifts[w2n l] &
    (1 << (hw_pgshifts[w2n l + 1] - hw_pgshifts[w2n l]) - 1);
r2 = (n, r);
```

Fig. 14. Page table indexing code

The function `ptab_index` takes an address space name n, a virtual page v, and a level l in the page table. The result is returned in variable $r2$. The array `hw_pgshifts` represents *page-bits* from the abstract model. The predicates `pte_is_valid` and `pte_is_subtree` on *PTENames* indicate whether the dereferenced *PTE* has a flag different from *Invalid* or possesses a *Next* tag respectively. For *PTE*s of the form *Next n*, `pte_get_next` gives the pointer to the next level in the page table n. It is a precondition on all these functions that the supplied pointer p is valid in the current state s, i.e. $p \in set\ (v\text{-}pt\text{-}'\ s)$. We omit the source code of these functions for brevity, with the intention that the source code presented so far provides a sufficient idea of the level of abstraction and language in which these are expressed.

The invariant is necessary to discharge the proof obligations related to the Hoare triple used to show refinement. *wfpt* is a well-formedness predicate on the page table structure, with conditions expressing properties of the tree structure, the size of nodes at different levels, the height of the tree, etc. *table-level* is a relation between page tables nodes and level numbers. R is the abstraction function for the page table — a description of R, *pt-lookup-f* and *pt-lookup-g* is provided in Section 4.3.

Page table insertion source code is provided in Figure 16. Two additional functions are present here. `pte_make_subtree` creates a new node, allocating an array n in the heap of size appropriate for the given level and setting the *PTE* referenced by the supplied pointer to *Next* n. `pte_make_leaf` sets the *PTE* referenced by the supplied pointer to a *Leaf* value with no access rights. The invariant is similar to that for `pt_lookup`, however the final conjunct describes the changing page table structure as new levels are added.

It should be noted that there is not necessarily any overhead from structuring the code as a series of function calls, since small functions can be either inlined during code generation or flagged as inlineable to the compiler.

```
procedures pt-lookup(n,v|r5) =
{|σ. n ∈ vN ∧ wfpt (pt_h, v_pt, vN)|}
tmp_level = pt-top-level;
tmp_tab = n;
tmp_pte = ptab_index(tmp_tab,v,tmp_level);
tmp_valid = pte_is_valid(tmp_pte);
tmp_subtree = pte_is_subtree(tmp_pte);
while (!(tmp_level == 0) && tmp_valid && tmp_subtree)
/* INV: {|wfpt (pt_h, v_pt, vN) ∧ w2n tmp_level ≤ pt-top-level' ∧
        tmp_valid = pte-is-valid' pt_h tmp_pte ∧
        tmp_subtree = pte-is-subtree' pt_h tmp_pte ∧
        tmp_pte = φ tmp_tab,v,tmp_level ∧
        (tmp_tab, w2n tmp_level) ∈ table-level vN pt_h ∧
        pt-lookup-g tmp_tab v (w2n tmp_level) pt_h =
        pt-lookup-f σ n σ v σ pt-h ∧
        v = σ v ∧ pt_h = σ pt-h ∧ vN = σ vN ∧ v_pt = σ v-pt|}
*/
{
  tmp_level = tmp_level - 1;
  tmp_tab = pte_get_next(tmp_pte);
  tmp_pte = ptab_index(tmp_tab,v,tmp_level);
  tmp_valid = pte_is_valid(tmp_pte);
  tmp_subtree = pte_is_subtree(tmp_pte);
}
r5 = (tmp_pte, tmp_level);
{|r5 = pt-lookup-f σ n σ v σ pt-h ∧ R (pt_h, vN) = R (σ pt-h, σ vN) ∧
wfpt (pt_h, v_pt, vN)|}
```

Fig. 15. Page table lookup code

While quite low-level, this is in fact an abstraction of actual page table implementations. In reality, multiple page levels in this model may consist of a single page level at the hardware level, where duplication is used to achieve superpages. Also, PTEs are bitfields and link nodes are stored at a fixed, level dependent, offset from the PTE. Procedures such as `pte_get_next` and `pte_is_subtree`

```
procedures pt-insert(n,v,l|r6) =
⦃σ. n ∈ vN ∧ w2n l ≤ pt-top-level' ∧ wfpt (pt_h, v_pt, vN)⦄
tmp_level = pt-top-level;
tmp_tab = n;
tmp_pte = ptab_index(tmp_tab,v,tmp_level);
tmp_valid = pte_is_valid(tmp_pte);
tmp_subtree = pte_is_subtree(tmp_pte);
while (!(tmp_level == l) && (tmp_subtree || !tmp_valid))
/* INV: ⦃w2n tmp_level ≤ pt-top-level' ∧ wfpt (σ pt-h, σ v-pt, σ vN) ∧ v = σ v ∧
        tmp_subtree = pte-is-subtree' pt_h tmp_pte ∧ vN = σ vN ∧ σ n ∈ vN ∧
        tmp_valid = pte-is-valid' pt_h tmp_pte ∧ wfpt (pt_h, v_pt, vN) ∧
        ¬ w2n tmp_level < w2n l ∧
        (tmp_tab, w2n tmp_level) ∈ table-level vN pt_h ∧
        tmp_pte = φ tmp_tab,v,tmp_level ∧
        pt-lookup-g tmp_tab v (w2n tmp_level) pt_h =
        pt-lookup-f σ n σ v pt_h ∧
        l = σ l ∧ R (pt_h, vN) = R (σ pt-h, σ vN) ∧
        (∀ x l. if pt-lookup-f σ n σ v σ pt-h = (x, l) ∧ w2n tmp_level ≤ w2n l
              then ∃ y. pt-lookup-g tmp_tab v (w2n tmp_level) pt_h =
                        (y, tmp_level)
              else pt_h = σ pt-h ∧ v_pt = σ v-pt)⦄
*/
{
  tmp_level = tmp_level - 1;
  if (!tmp_valid) {
    pte_make_subtree(tmp_level,tmp_pte);
  }
  tmp_tab = pte_get_next(tmp_pte);
  tmp_pte = ptab_index(tmp_tab,v,tmp_level);
  tmp_subtree = pte_is_subtree(tmp_pte);
  tmp_valid = pte_is_valid(tmp_pte);
}
if (!tmp_subtree) {
  if (!tmp_valid) {
    pte_make_leaf(tmp_pte);
  }
  r6 = ⌊(tmp_pte, tmp_level)⌋;
} else {
  r6 = None;
}
⦃r6 = pt-insert_a-out σ n σ v σ l (R (pt_h, vN)) ∧
R (pt_h, vN) ∈ pt-insert_a σ n σ v σ l (R (σ pt-h, σ vN)) ∧ wfpt (pt_h, v_pt, vN)⦄
```

Fig. 16. Page table insertion code

constitute the underlying ADT, of which concrete models should correspond to architecture-specific implementations of page tables.

4.3 Refinement

We can define an ADT for the operations in the above model and show refinement using the abstraction relation *r*. *pt-lookup-f* is a functional implementation of page table lookup as described in the previous section.

```
pt-lookup-g n v l pt-h =
(let p = φ n,v,n2w l
 in case ψ pt-h p of
    Next n' ⇒ if l ≠ 0 then pt-lookup-g n' v (l − 1) pt-h else (p, w-0)
    | - ⇒ (p, n2w l))
```

```
pt-lookup-f n v pt-h ≡ pt-lookup-g n v pt-top-level' pt-h
```

The function R maps from concrete page tables to the abstract page table function. This hides the type of nodes other than *Leaf* by returning *None* if the pointer returned by page table lookup does not reference a *Leaf*.

```
R c ≡
let (pt-h, N) = c
in λ(n, v).
     if n ∉ N then None
     else let (p, l) = pt-lookup-f n v pt-h
          in if ψ pt-h p ≠ Invalid then ⌊(p, l)⌋ else None
```

The same set of valid address spaces should be in both concrete and abstract models. Valid *Leaf PTE*s appear at the same location in the abstract heap.

```
r ≡
{(a, c) | set (N a) = vN-' c ∧
 (∀ p. case ψ pt-h-' c p of
       Leaf pte ⇒ pt-h-' c,v-pt-' c ⊢ p ⟶ Heap a p = pte | - ⇒ True) ∧
 PageTable a = R (pt-h-' c, vN-' c)}
```

The conditions in the Hoare triple specifications for the source code ensure that well-formedness holds and is preserved, that the abstraction relation holds on the concrete and abstract states pre- and post-operation, and that the expected values are returned. Using the soundness result of the Hoare logic [16], we get that the concrete implementations on the semantic level correctly simulate the abstract model of page tables.

5 Conclusion

We have presented some important aspects of the refinement process in verifying the virtual memory subsystem of the L4 microkernel. We have shown an abstract model of address spaces together with the operations on them that the kernel API offers. We have taken the memory lookup operation of this model, and

described its implementation in the kernel using a page table data structure. We have further refined this abstract view of the page table data structure towards an implementation in the C programming language.

While we have not yet completely reached the level of C source code as it is accepted by standard C compilers, it is already apparent that this final step is within reach.

Further work in this direction includes enhancements to the Hoare-logic verification environment, such as the ability to directly use concrete C-syntax within Isabelle, as well as using this verified implementation of page tables as a drop-in replacement for the current L4Ka::Pistachio implementation. Our final goal is a verified, high performance implementation of L4. Since our verified implementation is very close in terms of code and data structures being used to the existing one, we do not expect any decrease in performance.

Acknowledgements We thank Espen Skoglund for providing a clean and generic interface of the page table data structure in L4Ka::Pistachio that was nicely amenable to verification.

References

1. ARM Limited. *ARM Architecture Reference Manual*, Jun 2000.
2. William R. Bevier. Kit: A study in operating system verification. *IEEE Transactions on Software Engineering*, 15(11):1382–1396, 1989.
3. William R. Bevier and Lawrence M. Smith. A mathematical model of the Mach kernel. Technical Report 102, Computational Logic, Inc., Dec 1994.
4. Thierry Cattel. Modelization and verification of a multiprocessor realtime OS kernel. In *Proceedings of FORTE '94, Bern, Switzerland*, October 1994.
5. Willem-Paul de Roever and Kai Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*. Number 47 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1998.
6. Gregory Duval and Jacques Julliand. Modelling and verification of the RUBIS μ-kernel with SPIN. In *SPIN95 Workshop Proceedings*, 1995.
7. Michael Hohmuth, Hendrik Tews, and Shane G. Stephens. Applying source-code verification to a microkernel — the VFiasco project. Technical Report TUD-FI02-03-März, TU Dresden, 2002.
8. Jerry Huck and Jim Hays. Architectural support for translation table management in large address space machines. In *Proc. 20th ISCA*, pages 39–50. ACM, 1993.
9. Gerwin Klein and Harvey Tuch. Towards verified virtual memory in L4. In *TPHOLS'04 emerging trends*, Park City, Utah, Sep 2004.
10. *L4 eXperimental Kernel Reference Manual Version X.2*, 2004.
11. L4Ka Team. L4Ka::Pistachio kernel. `http://l4ka.org/projects/pistachio/`.
12. Jochen Liedtke. On μ-kernel construction. In *Proc. 15th SOSP*, pages 237–250, Copper Mountain, CO, USA, Dec 1995.
13. Jochen Liedtke. *On the Realization Of Huge Sparsely-Occupied and Fine-Grained Address Spaces*. Oldenbourg, Munich, Germany, 1996.
14. Wolfgang Naraschewski and Markus Wenzel. Object-oriented verification based on record subtyping in higher-order logic. In Jim Grundy and Malcom Newey, editors, *Proc. Theorem Proving in Higher Order Logics: TPHOLs '98*, volume 1479 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

15. P. G. Neumann, R. S. Boyer, R. J. Feiertag, K. N. Levitt, and L. Robinson. A provably secure operating system: The system, its applications, and proofs. Technical Report CSL-116, Computer Science Laboratory, SRI International, 1980.
16. Norbert Schirmer. A verification environment for sequential imperative programs in Isabelle/HOL. In Gerwin Klein, editor, *Proc. NICTA workshop on OS verification 2004, Technical Report 0401005T-1*, Sydney, Australia, Oct 2004.
17. J. S. Shapiro and S. Weber. Verifying operating system security. Technical Report MS-CIS-97-26, Distributed Systems Laboratory, University of Pennsylvania, 1997.
18. Madhusudhan Talluri, Shing Kong, Mark D. Hill, and David A. Patterson. Tradeoffs in supporting two page sizes. In *Proc. 19th ISCA*. ACM, 1992.
19. P. Tullmann, J. Turner, J. McCorquodale, J. Lepreau, A. Chitturi, and G. Back. Formal methods: a practical tool for OS implementors. In *Proceedings of the Sixth Workshop on Hot Topics in Operating Systems*, pages 20–25, 1997.
20. Bruce J. Walker, Richard A. Kemmerer, and Gerald J. Popek. Specification and verification of the UCLA Unix security kernel. *Communications of the ACM*, 23(2):118–131, February 1980.

A Verification Environment for Sequential Imperative Programs in Isabelle/HOL *

Norbert Schirmer

Technische Universität München, Institut für Informatik
http://www4.in.tum.de/~schirmer

Abstract. We develop a general language model for sequential imperative programs together with a Hoare logic. We instantiate the framework with common programming language constructs and integrate it into Isabelle/HOL, to gain a usable and sound verification environment.

1 Introduction

The main goal of this work is to develop a suitable programming language model and proof calculus, to support program verification in the interactive theorem prover Isabelle/HOL. The model should be lightweight so that program verification can be carried out on the abstraction level of the programming language. The design of a framework for program verification in an expressive logic like HOL is driven by two main goals. On the one hand we want to derive the proof calculus in HOL, so that we can guarantee soundness of the calculus with respect to the programming language semantics. On the other hand we want to apply the proof calculus to verify programs. During program verification we focus on one single program for which we want to derive some properties. But for a general soundness proof of the calculus we have to regard the whole programming language, not just a single program. The tradeoff can be illustrated by a simple program that only concerns two local variables: `i` and `b` of type `int` and `bool`, respectively. It is desirable to obtain verification conditions in terms of logical variables i and b of type *int* and *bool* in HOL. But of course we do not want to fix the state space of the general proof calculus only to programs on these two variables. One solution is to model local variables as a function, mapping variable names to values: $name \Rightarrow value$. But in this model all variables are represented by the same logical type, namely *value*. If the programming language supports more than one type for variables we can define the type *value* as datatype, e.g.: $value = Int\ int \mid Bool\ bool \mid \ldots$. We tag values with their programming language type. The drawback of this approach is that we have to explicitely deal with programming language typing in assertions and proofs. If for example a program adds the constant `2` to the variable `i` we have to know that the current environment holds a value of the kind $Int\ i$ and not $Bool\ b$ to properly reason about the addition. This knowledge has to stem from a typing

* This research is funded by the project Verisoft (http://www.verisoft.de)

constraint. We somehow always have to deal with type safety of the programming language during program verification.

The main contribution of this work is to present a programming language model that operates on a polymorphic state space, but still can handle local and global variables throughout procedure calls. By this we can achieve both desired goals. We can once and for all develop a sound proof calculus as well as later on tailor the state space to fit to the current program verification task. Moreover the model is expressive enough to handle abrupt termination, runtime faults and dynamic procedure calls. Finally we instantiate the framework with a state space representation that allows us to match programming language typing with logical typing. So type inference will take care of basic type safety issues, which simplifies the assertions and proof obligations. Parts of the frame condition for procedure specifications can be naturally expressed in this state space representation and can already be handled during verification condition generation.

We start with a brief introduction to Isabelle/HOL in Section 2; in Section 3 we introduce the syntax and semantics of the basic programming language model; Section 4 describes a Hoare logic for partial correctness; Section 5 a Hoare logic for total correctness; Section 6 will instantiate the framework with common language features and sketch the integration into Isabelle; Section 7 concludes.

Related Work The tradition of embedding a programming language in HOL goes back to the work of Gordon [11], where a while language with variables ranging over natural numbers is introduced. A polymorphic state space was already used by Harrison in his formalisation of Dijkstra [4] and by Prensa to verify parallel programs [18]. Still procedures are not present. Homeier [5] introduces procedures, but the variables are again limited to numbers. Later on detailed semantics for Java [16, 6] and C [15] where embedded in a theorem prover. But verification of even simple programs suffers from the complex models.

The Why tool [3] implements a program logics for annotated functional programs (with references) and produces verification conditions for an external theorem prover. It can handle uninterpreted parts of annotations that are only meaningful to the external theorem prover. With this approach it is possible to map imperative languages like Java to the tool by representing the heap in a reference variable. Splitting up verification condition generation and their proofs to different tools is also followed in [10, 17].

2 Preliminary Notes on Isabelle/HOL

Isabelle is a generic logical framework which allows one to encode different object logics. In this article we are only concerned with Isabelle/HOL [14], an encoding of higher order logic augmented with facilities for defining data types, records, inductive sets as well as primitive and total general recursive functions.

The syntax of Isabelle is reminiscent of ML, so we will not go into detail here. There are the usual type constructors $T_1 \times T_2$ for product and $T_1 \Rightarrow T_2$ for function space. The syntax $[\![P;\ Q]\!] \Longrightarrow\ R$ should be read as an inference rule with the two premises P and Q and the conclusion R. Logically it is just a shorthand for $P \Longrightarrow Q \Longrightarrow\ R$. There are actually two implications $\longrightarrow$ and $\Longrightarrow$. The two mean the same thing except that $\longrightarrow$ is HOL's "real" implication, whereas $\Longrightarrow$ comes from Isabelle's meta-logic and expresses inference rules. Thus $\Longrightarrow$ cannot appear inside a HOL formula. For the purpose of this paper the two may be identified. Similarly, we use $\bigwedge$ for the universal quantifier in the meta logic.

To emulate partial functions the polymorphic option type is frequently used:

datatype *'a option* = *None* | *Some 'a*

Here *'a* is a type variable, *None* stands for the undefined value and *Some x* for a defined value *x*. A partial function from type T_1 to type T_2 can be modelled as $T_1 \Rightarrow (T_2\ option)$.

There is also a destructor for the constructor *Some*, the function *the*:: *'a option* ⇒ *'a*. It is defined by the sole equation *the* (*Some x*) = *x* and is total in the sense that *the None* is a legal, but indefinite value.

Appending two lists is written as *xs* @ *ys* and "consing" as *x* # *xs*.

3 Programming Language Model

3.1 Abstract Syntax

The basic model of the programming language is quite general. We want to be able to represent a sequential imperative programming language with mutually recursive procedures, local and global variables and heap. Abrupt termination like `break`, `continue`, `return` or exceptions should also be expressible in the model. Moreover we support a dynamic procedure call, which allows us to represent procedure pointers or dynamic method invocation.

We only fix the statements of the programming language. Expressions are ordinary HOL-expressions, therefore they do not have any side effects. Nevertheless we want to be able to express faults during expression evaluation, like division by *0* or dereferencing a *Null* pointer. We introduce *guards* in the language, which check for those runtime faults.

The state space of the programming language and also the representation of procedure names is polymorphic. The canonical type variable for the state space is *'s* and for procedure names *'p*. The programming language is defined by a **datatype** (*'s*, *'p*) *com* with the following constructors:

Skip: Do nothing.
Basic f: Basic commands like assignment.
Seq c_1 c_2: Sequential composition, also written as $c_1;c_2$.
Cond b c_1 c_2: Conditional statement.

Guard g c: Guarded command, also written as $g \mapsto c$.
While g b c: Loop.
Call init p return result: Static procedure call.
DynCall init p return result: Dynamic procedure call.
Throw: Initiate abrupt termination.
Catch c_1 c_2: Handle abrupt termination.

3.2 Semantic

State Space Representation Although the semantics is defined for polymorphic state spaces we introduce the state space representation which we will use later on to give some illustrative examples. We represent the state space as a **record** [14, 12] in Isabelle/HOL. This idea goes back to Wenzel [19]. A simple state space with three local variables B, N and M can be modelled with the following record definition:

record *vars* = *B*::*bool* *N*::*int* *M*::*int*

Records of type *vars* have three fields, named B, N and M of type *bool* resp. *int*. An example instance of such a record is $(\!|B = True, N = 42, M = 3|\!)$. For each field there is a *selector* function of the same name, e.g. $N\ (\!|B = True, N = 42, M = 3|\!) = 42$. The *update* operation is functional. For example, $v(\!|N := 0|\!)$ is a record where component N is *0* and whose B and M component are copied from v. Selections of updated components can be simplified automatically e.g. $N\ (r(\!|N := 43|\!)) = 43$. The representation of the state space as record has the advantage that the typing of variables can be expressed by means of typing in the logic. Therefore basic type safety requirements are already ensured by type inference.

Operational Semantics We give an operational (big step) semantics for the programming language, written as $\Gamma \vdash s\ -c\rightarrow\ t$. Starting in state s, execution of command c leads to the final state t. Γ is the procedure environment, which maps procedure names to procedure bodies. The states s and t are not just plain state spaces of type *'s*, but extended states of type *'s xstate* which allow us to identify runtime faults, stuck calculations and abrupt termination. During normal execution such an extended state has the form *Normal s*, during abrupt termination the form *Abrupt s*, runtime faults are captured by the (extended) state *Fault* and stuck calculation by the (extended) state *Stuck*. The execution relation is defined inductively.

Basic commands The command *Basic f* just applies the function f to the current state. An example of a basic operation may be an assignment `N = 2`. This can be represented as *Basic* $(\lambda s.\ s(\!|N{:=}2|\!))$ in our language model. We can also model field assignment or memory allocation as basic commands.

$\Gamma \vdash Normal\ s\ -Skip\rightarrow\ Normal\ s$ $\qquad$ $\Gamma \vdash Normal\ s\ -Basic\ f\rightarrow\ Normal\ (f\ s)$

Composition Sequential composition combines the execution of the two commands.

$$\frac{\Gamma\vdash Normal\ s\ -c_1\rightarrow\ s' \qquad \Gamma\vdash s'\ -c_2\rightarrow\ t}{\Gamma\vdash Normal\ s\ -Seq\ c_1\ c_2\rightarrow\ t}$$

Conditional The conditional statement executes the first or the second command depending on the branching condition b. We represent boolean expressions as state sets.

$$\frac{s\in b \qquad \Gamma\vdash Normal\ s\ -c_1\rightarrow\ t}{\Gamma\vdash Normal\ s\ -Cond\ b\ c_1\ c_2\rightarrow\ t} \qquad \frac{s\notin b \qquad \Gamma\vdash Normal\ s\ -c_2\rightarrow\ t}{\Gamma\vdash Normal\ s\ -Cond\ b\ c_1\ c_2\rightarrow\ t}$$

Guards The guarded command is used to model runtime faults which may occur during expression evaluation. The guard g is a boolean expression that checks for possible faults in the expressions of command c and only executes c if the test is passed. Otherwise the fault will be signalled. Once a fault has occurred we cannot leave the error state *Fault* anymore.

$$\frac{\begin{array}{c}s\in g\\ \Gamma\vdash Normal\ s\ -c\rightarrow\ t\end{array}}{\Gamma\vdash Normal\ s\ -Guard\ g\ c\rightarrow\ t} \qquad \frac{s\notin g}{\Gamma\vdash Normal\ s\ -Guard\ g\ c\rightarrow\ Fault}$$

$$\Gamma\vdash Fault\ -c\rightarrow\ Fault$$

Loop If the guard g for the condition b fails the while loop will end up in the state *Fault*. If the guard and the loop condition hold, first the loop body c is executed, followed by the recursive execution of the while loop. If the guard holds, but the loop condition does not, we exit the loop.

$$\frac{s\notin g}{\Gamma\vdash Normal\ s\ -While\ g\ b\ c\rightarrow\ Fault}$$

$$\frac{\begin{array}{c}s\in g \qquad s\in b\\ \Gamma\vdash Normal\ s\ -c\rightarrow\ s'\\ \Gamma\vdash s'\ -While\ g\ b\ c\rightarrow\ t\end{array}}{\Gamma\vdash Normal\ s\ -While\ g\ b\ c\rightarrow\ t} \qquad \frac{s\in g \qquad s\notin b}{\Gamma\vdash Normal\ s\ -While\ g\ b\ c\rightarrow\ Normal\ s}$$

Abrupt termination The *Throw* statement transforms a *Normal* state to an *Abrupt* state. For *Abrupt* states execution is skipped. A *Catch* c_1 c_2 statement will handle an *Abrupt* final state of c_1 by continuing execution of c_2 in a *Normal* state. Otherwise execution of c_2 is skipped.

$$\Gamma\vdash Normal\ s\ -Throw\rightarrow\ Abrupt\ s \qquad \Gamma\vdash Abrupt\ s\ -c\rightarrow\ Abrupt\ s$$

$$\frac{\begin{array}{c}\Gamma\vdash Normal\ s\ -c_1\rightarrow\ Abrupt\ s'\\ \Gamma\vdash Normal\ s'\ -c_2\rightarrow\ t\end{array}}{\Gamma\vdash Normal\ s\ -Catch\ c_1\ c_2\rightarrow\ t} \qquad \frac{\Gamma\vdash Normal\ s\ -c_1\rightarrow\ t \qquad \neg\ isAbr\ t}{\Gamma\vdash Normal\ s\ -Catch\ c_1\ c_2\rightarrow\ t}$$

$$
\begin{array}{l}
isAbr\ (Normal\ s) = False \\
isAbr\ (Abrupt\ s) = True \\
isAbr\ Fault = False \\
isAbr\ Stuck = False
\end{array}
$$

Procedure call To execute a procedure call *Call init p return result* we first pass the parameters by applying *init* to the starting state *s*. Then we execute the procedure body that is given by a lookup in the procedure environment $\Gamma\ p$. If this lookup fails $\Gamma\ p = None$ execution gets *Stuck*. For *Stuck* states execution is skipped. If we find a procedure body in the environment the further execution depends on the kind of state, resulting from the body:

$$\frac{\Gamma\ p = None}{\Gamma \vdash Normal\ s\ -Call\ init\ p\ return\ result\rightarrow\ Stuck}$$

$$\Gamma \vdash Stuck\ -c\rightarrow\ Stuck$$

$$\frac{\Gamma\ p = Some\ bdy \qquad \Gamma \vdash Normal\ (init\ s)\ -bdy\rightarrow\ Fault}{\Gamma \vdash Normal\ s\ -Call\ init\ p\ return\ result\rightarrow\ Fault}$$

$$\frac{\Gamma\ p = Some\ bdy \qquad \Gamma \vdash Normal\ (init\ s)\ -bdy\rightarrow\ Stuck}{\Gamma \vdash Normal\ s\ -Call\ init\ p\ return\ result\rightarrow\ Stuck}$$

$$\frac{\Gamma\ p = Some\ bdy \qquad \Gamma \vdash Normal\ (init\ s)\ -bdy\rightarrow\ Normal\ t}{\Gamma \vdash Normal\ s\ -Call\ init\ p\ return\ result\rightarrow\ Normal\ (result\ s\ t)}$$

$$\frac{\Gamma\ p = Some\ bdy \qquad \Gamma \vdash Normal\ (init\ s)\ -bdy\rightarrow\ Abrupt\ t}{\Gamma \vdash Normal\ s\ -Call\ init\ p\ return\ result\rightarrow\ Abrupt\ (return\ s\ t)}$$

If execution of the body fails or gets stuck, the whole call fails or gets stuck. If execution of the body ends up in a *Normal* state *t*, the outcome of the call is given by *result s t*. If execution of the body ends up in an *Abrupt* state *t*, the outcome of the call is given by *return s t*. The function *return* passes back the global variables (and heap components), and *result* additionally assigns results to local variables of the caller.

The *return*/*result* functions get both the initial state *s* before the procedure call and the final state *t* after execution of the body. It is the purpose of *return* to restore the local variables of the caller and update the global variables. The *result* function will additionally assign the result to the caller. If the body terminates abruptly we apply the *return* function, thus the global state will be propagated to the caller but no result will be assigned. This is the expected semantics of an exception. Note that we can also store a description of the raised exception in a global variable so that a *Catch* can peek at it, to decide whether to handle the exception or to re-raise it.

As an example for a procedure call, consider a function definition `int fac(int n)` and a call to this function `m = fac(m)`. When we do not regard global variables, we can model this call by: *Call* $(\lambda s.\ s(\!|N := M\ s|\!))$ *fac* $(\lambda s\ t.\ s)$ $(\lambda s\ t.\ s(\!|M := N\ t|\!))$. The state space of the programming language is flat. We do not explicitly model a stack. Locality of variables and parameters is maintained by the *return* and *result* functions. The body of *fac* expects the input to be stored

in the formal parameter N. But we call the function with the actual parameter M. So the *init* function $\lambda s.\ s(\!|N := M\ s|\!)$ copies the content of M to component N. The trivial *return* function $\lambda s\ t.\ s$ gives back the initial state. State s is the initial state of the caller, and t is the state after executing the body. By this all local variables of the caller are restored. Consider that the body of *fac* internally holds the result of the factorial calculation in its local variable N. Then the *result* function has to copy the content of N to component M because of the assignment `m = fac(m)`. This is implemented by the *result* function $\lambda s\ t.\ s(\!|M := N\ t|\!)$. The *result* function just takes the initial state and performs the necessary update by peeking on the actual state. These ideas can be extended to global variables. Consider B to be a global variable. We just have to adapt the *return*/*result* function, so that they will copy B back to the caller: *return* = $(\lambda s\ t.\ s(\!|B := B\ t|\!))$, *result* = $(\lambda s\ t.\ s(\!|B := B\ t,\ M := N\ t|\!))$.

In contrast to the static procedure call, a dynamic procedure call first calculates the procedure from the actual state. The rest is handled by the ordinary procedure call rules.

$$\frac{\Gamma\vdash Normal\ s\ -Call\ init\ (p\ s)\ return\ result\rightarrow\ t}{\Gamma\vdash Normal\ s\ -DynCall\ init\ p\ return\ result\rightarrow\ t}$$

Termination To characterise terminating programs we introduce the inductively defined judgement $\Gamma\vdash c\downarrow s$ expressing that in procedure environment Γ program c will terminate when it is started in state s. The rules should be self-explanatory:

Basic commands

$$\Gamma\vdash Skip\downarrow Normal\ s \qquad\qquad \Gamma\vdash Basic\ f\downarrow Normal\ s$$

Composition

$$\frac{\Gamma\vdash c_1\downarrow Normal\ s \qquad \forall s'.\ \Gamma\vdash Normal\ s\ -c_1\rightarrow\ s'\longrightarrow\Gamma\vdash c_2\downarrow s'}{\Gamma\vdash Seq\ c_1\ c_2\downarrow Normal\ s}$$

Conditional

$$\frac{s\in b \qquad \Gamma\vdash c_1\downarrow Normal\ s}{\Gamma\vdash Cond\ b\ c_1\ c_2\downarrow Normal\ s} \qquad\qquad \frac{s\notin b \qquad \Gamma\vdash c_2\downarrow Normal\ s}{\Gamma\vdash Cond\ b\ c_1\ c_2\downarrow Normal\ s}$$

Guards

$$\frac{s\in g \qquad \Gamma\vdash c\downarrow Normal\ s}{\Gamma\vdash Guard\ g\ c\downarrow Normal\ s} \qquad \frac{s\notin g}{\Gamma\vdash Guard\ g\ c\downarrow Normal\ s} \qquad \Gamma\vdash c\downarrow Fault$$

Loop

$$\frac{s\notin g}{\Gamma\vdash While\ g\ b\ c\downarrow Normal\ s} \qquad\qquad \frac{s\in g \qquad s\notin b}{\Gamma\vdash While\ g\ b\ c\downarrow Normal\ s}$$

$$\frac{\begin{array}{c}s\in g \qquad s\in b\\ \Gamma\vdash c\downarrow Normal\ s \qquad \forall s'.\ \Gamma\vdash Normal\ s\ -c\rightarrow\ s'\longrightarrow\Gamma\vdash While\ g\ b\ c\downarrow s'\end{array}}{\Gamma\vdash While\ g\ b\ c\downarrow Normal\ s}$$

Abrupt termination

$$\Gamma\vdash \mathit{Throw} \downarrow \mathit{Normal}\ s \qquad\qquad \Gamma\vdash c \downarrow \mathit{Abrupt}\ s$$

$$\frac{\Gamma\vdash c_1 \downarrow \mathit{Normal}\ s \quad \forall s'.\ \Gamma\vdash \mathit{Normal}\ s\ -c_1\rightarrow\ \mathit{Abrupt}\ s' \longrightarrow \Gamma\vdash c_2 \downarrow \mathit{Normal}\ s'}{\Gamma\vdash \mathit{Catch}\ c_1\ c_2 \downarrow \mathit{Normal}\ s}$$

Procedure call

$$\frac{\Gamma\ p = \mathit{Some}\ \mathit{bdy} \qquad \Gamma\vdash \mathit{bdy} \downarrow \mathit{Normal}\ (\mathit{init}\ s)}{\Gamma\vdash \mathit{Call}\ \mathit{init}\ p\ \mathit{return}\ \mathit{result} \downarrow \mathit{Normal}\ s}$$

$$\frac{\Gamma\ p = \mathit{None}}{\Gamma\vdash \mathit{Call}\ \mathit{init}\ p\ \mathit{return}\ \mathit{result} \downarrow \mathit{Normal}\ s} \qquad \Gamma\vdash c \downarrow \mathit{Stuck}$$

$$\frac{\Gamma\vdash \mathit{Call}\ \mathit{init}\ (p\ s)\ \mathit{return}\ \mathit{result} \downarrow \mathit{Normal}\ s}{\Gamma\vdash \mathit{DynCall}\ \mathit{init}\ p\ \mathit{return}\ \mathit{result} \downarrow \mathit{Normal}\ s}$$

4 Hoare Logic for Partial Correctness

The first question concerning a Hoare logic is how to represent the assertions. The model of the imperative programming language is quite general. The state space is polymorphic. So the variables and their types are not fixed until we regard a program to verify. Therefore the assertion language is not fixed either. An assertion on states of type *'s* is a set of states: *'s set*.

We first define a Hoare logic for partial correctness. The judgement is of the general form $\Gamma,\Theta\vdash P\ c\ Q,A$ where P is the precondition, c the program, Q the postcondition for normal termination, A the postcondition for abrupt termination, Γ the procedure environment and Θ is a set of Hoare quadruples that we may assume. Θ is used to handle recursive procedures as we will see later on. The approach to split up the postcondition for normal and abrupt termination is also followed by [3, 7].

The semantics of these judgements is given by the notion of validity:

$$\Gamma\models P\ c\ Q,A \equiv$$
$$\forall s\ t.\ \Gamma\vdash s\ -c\rightarrow\ t \longrightarrow s \in \mathit{Normal}\ `\ P \longrightarrow t \in \mathit{Normal}\ `\ Q \cup \mathit{Abrupt}\ `\ A$$

Given an execution of command c which takes us from the starting state s to the final state t, if s is a *Normal* state for which the precondition P holds, then the final state t will either be a *Normal* state for which the postcondition Q holds, or an *Abrupt* state for which postcondition A holds. The *Fault* and *Stuck* states are no valid outcomes. This extends the traditional partial correctness interpretation to abrupt termination and the additional constraint that no runtime fault may occur. The assertions P, Q and A are of type *'s set*, whereas s and t are of type *'s xstate*. We do not have to deal with the extended state in assertions, which makes them easier. The operator ` is the set image (like *map* for lists). So $s \in \mathit{Normal}\ `\ P$ can be rephrased by the set comprehension $\{\mathit{Normal}\ s.\ s \in P\}$.

When designing the Hoare logic we should always keep soundness and completeness in mind, which we have both proven:

– **theorem** *soundness*: $\Gamma,\{\}\vdash P\ c\ Q,A \longrightarrow \Gamma\models P\ c\ Q,A$
  We can only derive valid Hoare quadruples out of the empty context.
– **theorem** *completeness*: $\Gamma\models P\ c\ Q,A \longrightarrow \Gamma,\{\}\vdash P\ c\ Q,A$
  We can derive every valid Hoare quadruple out of the empty context.

The Hoare logic is defined inductively. The rules are syntax directed, and most of them are defined in a weakest precondition style. This makes it easy to automate rule application in a verification condition generator. Handling abrupt termination is surprisingly simple. The postcondition for abrupt termination is left unmodified by most of the rules. Only if we actually encounter a *Throw* it has to be a consequence of the precondition. This means that the proof rules do not complicate the verification of programs where abrupt termination is not present.

Basic Commands The rule for *Basic f* commands is a variation of the classical assignment rule. If the postcondition is Q, then the precondition is the set of all states that will lead into Q after applying f.

$$\Gamma,\Theta\vdash Q\ \mathit{Skip}\ Q,A \qquad\qquad \Gamma,\Theta\vdash\{s.\ f\ s\in Q\}\ \mathit{Basic}\ f\ Q,A$$

Composition and Conditional The rule for sequential composition and the conditional are almost standard. In case of sequential composition the postcondition for abrupt termination has to hold in either statement independently, in contrast to the intermediate assertion R for normal termination. This is simply because in case of abrupt termination of the first statement the second one will be skipped.

$$\frac{\Gamma,\Theta\vdash P\ c_1\ R,A \qquad \Gamma,\Theta\vdash R\ c_2\ Q,A}{\Gamma,\Theta\vdash P\ \mathit{Seq}\ c_1\ c_2\ Q,A} \qquad\qquad \frac{\Gamma,\Theta\vdash(P\cap b)\ c_1\ Q,A \quad \Gamma,\Theta\vdash(P\cap -\ b)\ c_2\ Q,A}{\Gamma,\Theta\vdash P\ \mathit{Cond}\ b\ c_1\ c_2\ Q,A}$$

Guards To prove a guarded command correct, we have to show that both the precondition P of the statement c and the guard g hold. This ensures that no runtime fault occurs.

$$\frac{\Gamma,\Theta\vdash P\ c\ Q,A}{\Gamma,\Theta\vdash(g\cap P)\ \mathit{Guard}\ g\ c\ Q,A}$$

Loop The rule for the while loop is also almost the traditional invariant rule. But we also have to ensure that the guard g for the conditional b always holds. Otherwise a runtime fault could occur. The verification condition generator will use a derived rule which takes an invariant annotation into account.

$$\frac{\Gamma,\Theta\vdash(g\cap P\cap b)\ c\ (g\cap P),A}{\Gamma,\Theta\vdash(g\cap P)\ \mathit{While}\ g\ b\ c\ (g\cap P\cap -\ b),A}$$

Abrupt Termination In case of a *Throw* the abrupt postcondition has to stem from the precondition. The rule for *Catch* is dual to sequential composition. Here the postcondition for normal termination can be derived independently. The intermediate assertion R is the precondition for the second statement and the postcondition for abrupt termination of the first statement.

$$\Gamma,\Theta\vdash A\ \mathit{Throw}\ Q,A \qquad \frac{\Gamma,\Theta\vdash P\ c_1\ Q,R \qquad \Gamma,\Theta\vdash R\ c_2\ Q,A}{\Gamma,\Theta\vdash P\ \mathit{Catch}\ c_1\ c_2\ Q,A}$$

Consequence We have a quite general form of the consequence rule. The traditional rules like precondition strengthening or postcondition weakening can easily be derived from it.

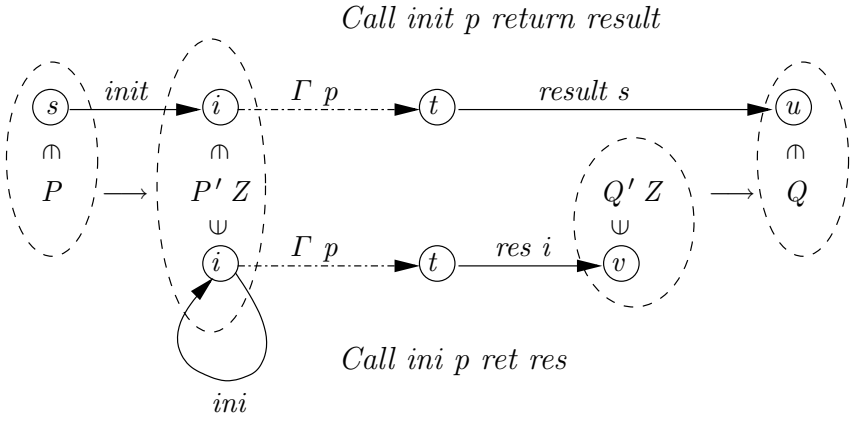
$$\frac{\begin{array}{c}\forall Z.\ \Gamma,\Theta\vdash(P'\ Z)\ c\ (Q'\ Z),(A'\ Z)\\ \forall s.\ s\in P\longrightarrow(\exists Z.\ s\in P'\ Z\wedge Q'\ Z\subseteq Q\wedge A'\ Z\subseteq A)\end{array}}{\Gamma,\Theta\vdash P\ c\ Q,A}$$

The consequence rule can be used to adapt a given specification $\Gamma,\Theta\vdash(P'\ Z)\ c\ (Q'\ Z),(A'\ Z)$ about command c to $\Gamma,\Theta\vdash P\ c\ Q,A$. The auxiliary variable Z can be used to transport state information from the pre state to the post state. This is a crucial tool to deal with procedure specifications, where the postcondition is defined by means of the pre state. For completeness issues it is sufficient that Z has the type $'s$ of the state space. Z can be used to fix the pre-state logically. That is why the given specification must be valid for all Z and the side-condition allows us to select a specific Z dependent on the current state s. A more detailed discussion of consequence rules and auxiliary variables can be found in [8,13,16].

Procedure Call If we encounter a procedure call during verification condition generation, like $\Gamma,\Theta\vdash P\ \mathit{Call\ init\ p\ return\ result}\ Q,A$, we do not look inside the procedure body, but instead use a specification $\Gamma,\Theta\vdash(P'\ Z)\ \mathit{Call\ ini\ p\ ret\ res}\ (Q'\ Z),(A'\ Z)$ of the procedure. We then adapt the specification to the actual calling context mainly by a variant of the consequence rule, where we also take parameter and result passing into account.

$$\frac{\begin{array}{c}\forall Z.\ \Gamma,\Theta\vdash(P'\ Z)\ \mathit{Call\ ini\ p\ ret\ res}\ (Q'\ Z),(A'\ Z)\\ \forall s.\ \mathit{ini}\ (\mathit{init}\ s)=\mathit{init}\ s\\ P\subseteq\{s.\ (\exists Z.\ \mathit{init}\ s\in P'\ Z\wedge\\ \qquad(\forall t.\ \mathit{res}\ (\mathit{init}\ s)\ t\in Q'\ Z\longrightarrow\mathit{result}\ s\ t\in Q)\wedge\\ \qquad(\forall t.\ \mathit{ret}\ (\mathit{init}\ s)\ t\in A'\ Z\longrightarrow\mathit{return}\ s\ t\in A))\}\end{array}}{\Gamma,\Theta\vdash P\ \mathit{Call\ init\ p\ return\ result}\ Q,A}$$

The central idea of this rule is to simulate the actual call *Call init p return result* in state s, with a call of the specification *Call ini p ret res* in state *init s*. The following figure shows the sequence of intermediate states for normal termination of both executions. On the top the actual call, on the bottom the call of the specification:



We start in state s for which the precondition P holds. To be able to make use of the procedure specification we have to find a suitable instance of the auxiliary variable Z so that the precondition of the specification holds: $init\ s \in P'\ Z$. Let t be the state immediately after execution of the procedure body, before returning to the caller and passing results. We know from the procedure specification that when exiting the procedure according to *res* the postcondition will hold: $res\ (init\ s)\ t \in Q'\ Z$. From this we have to conclude that exiting the procedure according to the actual function *result* will lead us to a state in Q. For abrupt termination the analogous simulation idea applies.

The side-condition $\forall\, s.\ ini\ (init\ s) = init\ s$ is no real burden. A procedure specification will be given with the canonical procedure parameters, according to the procedure declaration. So *ini* will be the identity. Also in actual program verification the formal procedure exit protocol defined by *ret* and *res* and the actual protocol *return* and *result* will be closely related to each other. Just because these protocols are modelled so generically here it might seem tedious at first sight. But *ret* and *return* will both copy back global variables to the caller (so they will actually be the same), and *res* will just store the result of the procedure at the formal result parameter in the callers local variables, whereas *result* will store it to the actual one. In the record implementation, the verification condition generator can use simplification of the record updates and selections encoded in the functions and the assertions, to achieve the expected adaption of the specification to the actual calling context.

Procedure Implementation To verify a procedure implementation against its specification we also need a rule that descends into the procedure body. The Hoare logic can deal with (mutually) recursive procedures. The basic idea of a Hoare rule for recursive procedures is simple. We prove that the procedure body respects the specification, under the assumption that recursive calls to the procedure will meet the specification. To model this assumption the context Θ comes in. If a procedure specification is in this context, we can immediately derive this specification within the Hoare logic.

$$\frac{(P,\ c,\ Q,\ A) \in \Theta}{\Gamma,\Theta \vdash P\ c\ Q,A}$$

To handle a set *Procs* of mutually recursive procedures we enrich the context by all the procedure specifications, while we prove their bodies.

$$\frac{\begin{array}{l}\Theta' = \Theta \cup (\bigcup p \in Procs.\\ \qquad \bigcup Z.\{(P\ p\ Z, Call\ (Init\ p)\ p\ (Ret\ p)\ (Res\ p), Q\ p\ Z, A\ p\ Z)\})\\ \qquad \forall\, p \in Procs.\ \forall\, Z.\ \Gamma,\Theta' \vdash (P\ p\ Z)\ the\ (\Gamma\ p)\ (Q'\ p\ Z), (A'\ p\ Z)\\ \forall\, p \in Procs.\ \forall\, Z\ s\ t.\ s \in P\ p\ Z \longrightarrow\\ \qquad (t \in Q'\ p\ Z \longrightarrow Res\ p\ s\ t \in Q\ p\ Z) \wedge (t \in A'\ p\ Z \longrightarrow Ret\ p\ s\ t \in A\ p\ Z)\\ \qquad\qquad \forall\, p \in Procs.\ Init\ p = (\lambda s.\ s) \qquad Procs \subseteq dom\ \Gamma\end{array}}{\forall\, Z.\ \forall\, p \in Procs.\ \Gamma,\Theta \vdash (P\ p\ Z)\ Call\ (Init\ p)\ p\ (Ret\ p)\ (Res\ p)\ (Q\ p\ Z), (A\ p\ Z)}$$

Since we deal with the set *Procs* of procedures we also have to give the pre- and postconditions and the parameter and return/result passing protocols for all these procedures. We use the functions P, Q, A, *Init*, *Ret* and *Res* which map procedure names to the desired entities. Z plays the role of an auxiliary (or logical) variable. It usually fixes (parts of) the pre state, so that we can refer to it in the post state. In the Hoare rule for procedure specifications, which we have described before, we had the freedom to pick a particular Z so that $s \in P \longrightarrow init\ s \in P'\ Z$ holds. Since we have the freedom there, we now have to prove the procedure bodies for all possible Z. Think of Z as the pre state. We prove that the specification holds for all pre states (that satisfy the precondition). When we later on use the specification to prove a procedure call we instantiate Z to the actual state to adapt the specification to the current calling context. Whereas the postconditions for the procedures are given by $Q\ p\ Z$ and $A\ p\ Z$ we prove different postconditions for the procedure bodies, namely $Q'\ p\ Z$ and $A'\ p\ Z$. This stems from the fact that the final state of a procedure body is not the final state of the corresponding procedure call, since exiting the procedure lies in-between them. We take the intermediate assertions $Q'\ p\ Z$ and $A'\ p\ Z$ to describe the final state of the procedure body. The big side-condition then links $Q'\ p\ Z$ and $Q\ p\ Z$ or $A'\ p\ Z$ and $A\ p\ Z$ together. It is worth noticing that in most cases Q' and Q or A' and A will actually be the same, since a proper postcondition will only talk about global variables or result variables of the final state and their relation to the initial state and not about local variables. Keep in mind that we verify a procedure specification here. So the *init* functions will not pass any parameters ($Init\ p = (\lambda s.\ s)$), and the return/result functions will only assign to global variables or formal result parameters. So global variables and result variables at the end of the procedure body will be the same as after exiting the procedure. Finally, with $Procs \subseteq dom\ \Gamma$, we make sure that the calculation will not get stuck.

Dynamic Procedure Call The rule for dynamic procedure call is a slight generalisation of the rule for static procedure call. Since the selected procedure

depends on the state, we have the liberty to select a suitable specification dependent on the state.

$$\frac{\begin{array}{c}\forall\, s{\in}P.\ \forall\, Z.\ \Gamma,\Theta{\vdash}(P'\ s\ Z)\ \mathit{Call\ ini}\ (p\ s)\ \mathit{ret\ res}\ (Q'\ s\ Z),(A'\ s\ Z) \\ \forall\, s.\ \mathit{ini}\ (\mathit{init}\ s) = \mathit{init}\ s \\ P \subseteq \{s.\ (\exists\, Z.\ \mathit{init}\ s \in P'\ s\ Z\ \wedge \\ \qquad (\forall\, t.\ \mathit{res}\ (\mathit{init}\ s)\ t \in Q'\ s\ Z \longrightarrow \mathit{result}\ s\ t \in Q)\ \wedge \\ \qquad (\forall\, t.\ \mathit{ret}\ (\mathit{init}\ s)\ t \in A'\ s\ Z \longrightarrow \mathit{return}\ s\ t \in A))\}\end{array}}{\Gamma,\Theta{\vdash}P\ \mathit{DynCall\ init\ p\ return\ result}\ Q,A}$$

5 Hoare Logic for Total Correctness

The Hoare logic for total correctness ensures both partial correctness and termination. The judgement is written as $\Gamma,\Theta{\vdash_t}P\ c\ Q,A$. The intended semantics of this judgement is described by the notion of validity.

$$\Gamma{\models_t}\ P\ c\ Q,A \equiv \Gamma{\models}\ P\ c\ Q,A \wedge (\forall\, s{\in}\mathit{Normal}\ `\ P.\ \Gamma{\vdash}c \downarrow s)$$

Validity for total correctness directly captures the informal description given above. The quadruple must be valid in the sense of partial correctness, and the program c has to terminate for all *Normal* states that satisfy the precondition P.

Again we have proven both soundness and completeness of the Hoare logic.

- **theorem** *soundness*: $\Gamma,\{\}{\vdash_t}P\ c\ Q,A \longrightarrow \Gamma{\models_t}\ P\ c\ Q,A$
  We can only derive valid Hoare quadruples out of the empty context.
- **theorem** *completeness*: $\Gamma{\models_t}\ P\ c\ Q,A \longrightarrow \Gamma,\{\}{\vdash_t}P\ c\ Q,A$
  We can derive every valid Hoare quadruple out of the empty context.

Most of the Hoare logic rules for total correctness are structurally equivalent to their partial correctness counterparts. We will only focus on those interesting rules with an impact on termination, namely loops and recursion. The basic idea is to justify termination by a well-founded relation on the state space.

Loop We have to supply a well-founded relation r on the state space, which decreases by evaluation of the loop body. Formally this is expressed by first fixing the pre-state with the singleton set $\{\tau\}$. In the postcondition for *Normal* termination of the loop body we end up in a state s and have to show that this state is "smaller" as τ according to the relation: $(s,\ \tau) \in r$. For *Abrupt* termination we do not have to take any care, since it will exit the loop anyway.

$$\frac{wf\ r \qquad \forall\, \tau.\ \Gamma,\Theta{\vdash_t}(\{\tau\} \cap g \cap P \cap b)\ c\ (\{s.\ (s,\ \tau) \in r\} \cap g \cap P),A}{\Gamma,\Theta{\vdash_t}(g \cap P)\ \mathit{While}\ g\ b\ c\ (g \cap P \cap -\ b),A}$$

Procedure Implementation In contrast to partial correctness we now only assume "smaller" recursive procedure calls correct while verifying the procedure bodies. Here "smaller" again is in the sense of a well-founded relation r. To be able to handle mutually recursive procedures the relation r not only relates state spaces but also takes the procedure names into account. We fix the pre-state of the procedure p with the singleton set $\{\tau\}$. For every call to a procedure q in a state s which is "smaller" than the initial call of p in state τ according to the relation $(((s,q),(\tau,p)) \in r)$, we can safely assume the specification of q while verifying the body of p.

$$\frac{\begin{array}{c} wf\ r \\ \Theta' = \lambda\tau.\ \Theta \cup (\bigcup q{\in}Procs.\ \bigcup Z. \\ \{(P\ q\ Z \cap \{s.\ ((s,q),(\tau,p)) \in r\}, Call\ (Init\ q)\ q\ (Ret\ q)\ (Res\ q), Q\ q\ Z, A\ q\ Z)\}) \\ \forall p{\in}Procs.\ \forall \tau\ Z.\ \Gamma,\Theta' \tau \vdash_t (\{\tau\} \cap P\ p\ Z)\ the\ (\Gamma\ p)\ (Q'\ p\ Z), (A'\ p\ Z) \\ \forall p{\in}Procs.\ \forall Z\ s\ t.\ s \in P\ p\ Z \longrightarrow \\ (t \in Q'\ p\ Z \longrightarrow Res\ p\ s\ t \in Q\ p\ Z) \wedge (t \in A'\ p\ Z \longrightarrow Ret\ p\ s\ t \in A\ p\ Z) \\ \forall p{\in}Procs.\ Init\ p = (\lambda s.\ s) \qquad Procs \subseteq dom\ \Gamma \end{array}}{\forall Z.\ \forall p{\in}Procs.\ \Gamma,\Theta \vdash_t (P\ p\ Z)\ Call\ (Init\ p)\ p\ (Res\ p)\ (Ret\ p)\ (Q\ p\ Z), (A\ p\ Z)}$$

6 Utilising the Framework

In this section we will sketch the integration of the Hoare logics in Isabelle/HOL and how we can express and deal with typical programming language constructs in our framework. Our purpose is to give an impression of how program verification "feels" like in our verification environment. The main tool is a verification condition generator that is implemented as tactic called *vcg*. The Hoare logic rules are defined in a weakest precondition style, so that we can almost take them as they are. We derive variants of the Hoare rules where all assertions in the conclusions are plain variables so that they are applicable to every context. We get the following format: $\dfrac{P \subseteq WP\ \ldots}{\Gamma,\Theta \vdash P\ c\ Q,A}$. The $\ldots$ may be recursive Hoare quadruples or side-conditions which somehow lead to the weakest precondition WP. If we recursively apply rules of this format until the program c is completely processed, then we have calculated the weakest precondition WP and are left with the verification condition $P \subseteq WP$. The set inclusion is then transformed to an implication. Then we can split the state records so that the record representation will not show up in the resulting verification condition. This leads to quite comprehensible proof obligations that closely resemble the specifications. Moreover we supply some concrete syntax for programs. The mapping to the abstract syntax should be obvious. As a shorthand an empty set Θ can be omitted and writing a Hoare triple instead of the quadruples is an abbreviation for an empty postcondition for abrupt termination.

Basics If we refer to components (variables) of the state-space of the program we always mark these with ´(in assertions and also in the program itself). Assertions are ordinary Isabelle/HOL sets. As we usually want to refer to the state space in the assertions, we provide special brackets $\{\!|\ldots|\!\}$ for them. Internally, an assertion of the from $\{\!|\acute{I} \leq 3|\!\}$ gets expanded to $\{s.\ I\ s \leq 3\}$ in ordinary set comprehension notation of Isabelle.

Although our assertions work semantically on the state space, stepping through verification condition generation "feels" like the expected syntactic substitutions of traditional Hoare logic. This is achieved by light simplification on the assertions calculated by the Hoare rules.

lemma $\Gamma\vdash \{\!|\acute{M} = a \wedge \acute{N} = b|\!\}$
$\acute{I} := \acute{M};\ \acute{M} := \acute{N};\ \acute{N} := \acute{I}$
$\{\!|\acute{M} = b \wedge \acute{N} = a|\!\}$
apply *vcg-step*

1. $\Gamma\vdash\{\!|\acute{M} = a \wedge \acute{N} = b|\!\}\ \acute{I} := \acute{M};\ \acute{M} := \acute{N}\ \{\!|\acute{M} = b \wedge \acute{I} = a|\!\}$

apply *vcg-step*

1. $\Gamma\vdash\{\!|\acute{M} = a \wedge \acute{N} = b|\!\}\ \acute{I} := \acute{M}\ \{\!|\acute{N} = b \wedge \acute{I} = a|\!\}$

apply *vcg-step*

1. $\{\!|\acute{M} = a \wedge \acute{N} = b|\!\} \subseteq \{\!|\acute{N} = b \wedge \acute{M} = a|\!\}$

apply *vcg-step*

1. $\bigwedge M\ N.\ N = N \wedge M = M$

by *simp*

Loops The following example calculates multiplication by an iterated addition. The user annotates the loop with an invariant.

lemma $\Gamma\vdash \{\!|\acute{M} = 0 \wedge \acute{S} = 0|\!\}$
WHILE $\acute{M} \neq a$ ***INV*** $\{\!|\acute{S} = \acute{M} * b|\!\}$
DO $\acute{S} := \acute{S} + b;\ \acute{M} := \acute{M} + 1$ ***OD***
$\{\!|\acute{S} = a * b|\!\}$
apply *vcg*

1. $\bigwedge M\ S.\ [\![M = 0;\ S = 0]\!] \Longrightarrow S = M * b$
2. $\bigwedge M\ S.\ [\![S = M * b;\ M \neq a]\!] \Longrightarrow S + b = (M + 1) * b$
3. $\bigwedge M\ S.\ [\![S = M * b;\ \neg\ M \neq a]\!] \Longrightarrow S = a * b$

The verification condition generator gives us three proof obligations, stemming from the path from the precondition to the invariant, from the invariant together with the loop condition through the loop body to the invariant, and finally from the invariant together with the negated loop condition to the postcondition.

For total correctness the user also has to supply the variant, which in our case is a well-founded relation. We make use of the infrastructure for well-founded recursion that is already present in Isabelle/HOL [14]. In the example the distance of the loop variable M to a decreases in every iteration. This is expressed by the measure function $a - \acute{M}$ on the state-space.

lemma $\Gamma\vdash_t \{\!|\acute{M} = 0 \wedge \acute{S} = 0|\!\}$
WHILE $\acute{M} \neq a$ ***INV***$\{\!|\acute{S} = \acute{M} * b \wedge \acute{M} \leq a|\!\}$ ***VAR*** *MEASURE* $a - \acute{M}$
DO $\acute{S} := \acute{S} + b;\ \acute{M} := \acute{M} + 1$ ***OD***
$\{\!|\acute{S} = a * b|\!\}$
apply *vcg*

1. $\bigwedge M\ S.\ [\![M = 0;\ S = 0]\!] \Longrightarrow S = M * b \wedge M \leq a$
2. $\bigwedge M\ S.\ [\![S = M * b;\ M \leq a;\ M \neq a]\!]$
   $\Longrightarrow a - (M + 1) < a - M \wedge S + b = (M + 1) * b \wedge M + 1 \leq a$
3. $\bigwedge M\ S.\ [\![S = M * b;\ M \leq a;\ \neg\ M \neq a]\!] \Longrightarrow S = a * b$

The variant annotation results in the proof obligation $a - (M + 1) < a - M$ after verification condition generation.

Abrupt Termination We can implement breaking out of a loop by a ***THROW*** inside the loop body and enclosing the loop into a ***TRY*−*CATCH*** block.

lemma $\Gamma\vdash \{\!|\acute{I} \leq 3|\!\}$
TRY WHILE *True* ***INV*** $\{\!|\acute{I}\leq 10|\!\}$
DO IF $\acute{I} < 10$ ***THEN*** $\acute{I} := \acute{I} + 1$ ***ELSE THROW FI OD***
CATCH SKIP YRT
$\{\!|\acute{I} = 10|\!\}$,{}
apply *vcg*

1. $\bigwedge I.\ I \leq 3 \Longrightarrow I \leq 10$
2. $\bigwedge I.\ [\![I \leq 10;\ True]\!]$
   $\Longrightarrow (I < 10 \longrightarrow I + 1 \leq 10) \wedge (\neg\ I < 10 \longrightarrow I = 10)$
3. $\bigwedge I.\ [\![I \leq 10;\ \neg\ True]\!] \Longrightarrow I = 10$

The first subgoal stems from the path from the precondition to the invariant. The second one from the loop body. We can assume the invariant and the loop condition and have to show that the invariant is preserved when we execute the ***THEN*** branch, and that the ***ELSE*** branch will imply the assertion for abrupt termination, which will be $\{\!|\acute{I} = 10|\!\}$ according to the rule for *Catch*. The third subgoal expresses that normal termination of the while loop has to imply the postcondition. But the loop will never terminate normally and so the third subgoal will trivially hold. All subgoals are quite simple and can be proven automatically.

To model a `continue` we can use the same idea and put a ***TRY−CATCH*** around the loop body. Or for `return` we can put the procedure body into a ***TRY−CATCH***. To distinguish the kind of abrupt termination we can add a ghost variable *Abr* to the state space and store this information before the ***THROW***. For example `break` can be translated to $'Abr := ''Break''$; ***THROW***, and the matching ***CATCH*** will peek for this variable to decide whether it is responsible or not: ***IF*** $'Abr = ''Break''$ ***THEN SKIP ELSE THROW FI***. This idea can immediately be extended to exceptions. We just have to make sure to use a global variable to store the kind of exception, so that it will properly pass procedure boundaries.

Procedures We provide the command **procedures**, to declare, define and specify a procedure.

procedures *Fac* $(N|R) =$
IF $'N = 0$ ***THEN*** $'R := 1$
ELSE $'R :=$ ***CALL*** $Fac('N - 1);\ 'R := 'N * 'R$
FI

Fac-spec: $\forall n.\ \Gamma \vdash \{\!|\ 'N = n\ |\!\}\ 'R :=$ ***CALL*** $Fac('N)\ \{\!|\ 'R = fac\ n\ |\!\}$

A procedure is given by the signature of the procedure followed by the procedure body and named specifications. The signature consists of the name of the procedure and a list of parameters. The parameters in front of the pipe | are value parameters and behind the pipe are the result parameters. Value parameters model call by value semantics. The value of a result parameter at the end of the procedure is passed back to the caller.

The procedure specifications are ordinary Hoare quadruples. The precondition here fixes the current value $'N$ to the logical variable n. Universal quantification of n enables us to adapt the specification to an actual parameter. The specification will be used in the rule for procedure call when we come upon a call to *Fac*. Thus n plays the role of the auxiliary variable Z.

The procedures command provides convenient syntax for procedure calls (that creates the proper *init*, *return* and *result* functions on the fly), defines a constant for the procedure body (named *Fac-body*) and creates two *locales*. The purpose of locales is to set up logical contexts to support modular reasoning [1].

One locale is named like the specification, in our case *Fac-spec*. This locale contains the procedure specification. The second locale is named *Fac-impl* and contains the assumption $\Gamma\ ''Fac'' = Some\ Fac\text{-}body$, which expresses that the procedure is defined in the current context. The purpose of these locales is to give us easy means to setup the context in which we will prove programs correct.

By including the locale *Fac-spec*, the following lemma assumes that the specification of the factorial holds. The *vcg* will make use the specification to handle the procedure call. The lemma also illustrates locality of *I*.

lemma includes *Fac-spec* **shows**
$\Gamma \vdash \{\!|\ 'M = 3 \wedge\ 'I = 2\ |\!\}\ 'R :=$ ***CALL*** $Fac\ ('M)\ \{\!|\ 'R = 6 \wedge\ 'I = 2\ |\!\}$

apply *vcg*

1. $\bigwedge I\ M.\ [\![M = 3;\ I = 2]\!] \Longrightarrow fac\ M = 6 \wedge I = 2$

To verify the procedure body we use the rule for recursive procedures. We extend the context with the procedure specification. In this extended context the specification will hold by the assumption rule. We then verify the procedure body by using *vcg*, which will use the assumption to handle the recursive call.

lemma includes *Fac-impl* **shows**
$\forall n.\ \Gamma \vdash \{\!|\ 'N = n\ |\!\}\ \textbf{CALL}\ Fac('N, 'R)\ \{\!|\ 'R = fac\ n\ |\!\}$
apply (*hoare-rule CallRec1-SamePost*)

1. $\forall n.\ \Gamma, (\bigcup n\ \{(\{\!|\ 'N = n\ |\!\},\ 'R := \textbf{CALL}\ Fac('N),\ \{\!|\ 'R = fac\ n\ |\!\},\ \{\})\})$
$\vdash \{\!|\ 'N = n\ |\!\}$
$\textbf{IF}\ 'N = 0\ \textbf{THEN}\ 'R := 1$
$\textbf{ELSE}\ 'R := \textbf{CALL}\ Fac('N - 1);\ 'R := 'N * 'R\ \textbf{FI}$
$\{\!|\ 'R = fac\ n\ |\!\}$

apply *vcg*

1. $\bigwedge N.\ (N = 0 \longrightarrow 1 = fac\ N) \wedge (N \neq 0 \longrightarrow N * fac\ (N - 1) = fac\ N)$

The rule *CallRec1-SamePost* is a specialised version of the general rule for recursion, tailored for one (mutual recursive) procedure, and where the intermediate assertions for the procedure body and the actual postcondition are the same. The method *hoare-rule* applies a single rule and solves the canonical side-conditions concerning the parameter passing and returning protocols. Moreover it expands the procedure body.

For total correctness the user supplies a well-founded relation. For the factorial the input parameter N decreases in the recursive call. This is expressed by the measure function $\lambda(s,p).\ {}^{s}N$. The relation can depend on both the state-space s and the procedure name p. The latter is useful to handle mutual recursion. The prefix superscript in ${}^{s}N$ is a shorthand for record selection $N\ s$ and is used to refer to state components of a named state.

lemma includes *Fac-impl* **shows**
$\forall n.\ \Gamma \vdash_t \{\!|\ 'N = n\ |\!\}\ 'R := \textbf{CALL}\ Fac('N)\ \{\!|\ 'R = fac\ n\ |\!\}$
apply (*hoare-rule CallRec1-SamePost*$_t$ [**where** *r=measure* $(\lambda(s,p).\ {}^{s}N)$])

$$
\begin{array}{l}
1.\ \forall\tau\ n.\ \Gamma,(\bigcup n\ \{(\{\!|\,\acute{N} = n\,|\!\} \cap \{\!|\,\acute{N} < {}^{\tau}N\,|\!\},\ \acute{R} := \textbf{\textit{CALL}}\ Fac(\acute{N}),\\
\qquad\qquad\qquad \{\!|\,\acute{R} = fac\ n\,|\!\},\ \{\})\})\\
\qquad \vdash_t (\{\tau\} \cap \{\!|\,\acute{N} = n\,|\!\})\\
\qquad\quad \textbf{\textit{IF}}\ \acute{N} = 0\ \textbf{\textit{THEN}}\ \acute{R} := 1\\
\qquad\quad \textbf{\textit{ELSE}}\ \acute{R} := \textbf{\textit{CALL}}\ Fac(\acute{N} - 1);\ \acute{R} := \acute{N} * \acute{R}\ \textbf{\textit{FI}}\\
\qquad\quad \{\!|\,\acute{R} = fac\ n\,|\!\}
\end{array}
$$

We may only assume the specification for "smaller" states $\{\!|\,\acute{N} < {}^{\tau}N\,|\!\}$, where state τ gets fixed in the precondition.

apply *vcg*

$$
\begin{array}{l}
1.\ \bigwedge N.\ (N = 0 \longrightarrow 1 = fac\ N) \wedge\\
\qquad (N \neq 0 \longrightarrow N - 1 < N \wedge N * fac\ (N - 1) = fac\ N)
\end{array}
$$

The measure function results in the proof obligation $N - 1 < N$ in the verification condition.

Heap The heap can contain structured values like `structs` in C or records in Pascal. Our model of the heap follows Bornat [2]. We have one heap variable f of type *ref* $\Rightarrow$ *value* for each component f of type *value* of the `struct`.

A typical structure to represent a linked list in the heap is `struct {int cont; list *next} list`. The structure contains two components, `cont` and `next`. So we will also get two heap variables, *cont* of type *ref* $\Rightarrow$ *int* and *next* of type *ref* $\Rightarrow$ *ref* in our state space record:

record *list-vars* =
next::*ref* $\Rightarrow$ *ref*
cont::*ref* $\Rightarrow$ *ref*
p::*ref*
q::*ref*
r::*ref*

In this state space *next* and *cont* are global variables and p and q are local ones. This is given to Isabelle by the **globals** command.

globals *list-vars* = *next* **and** *cont*

The only effect of this command is that the *return*/*result* functions that are created by the syntax translations will actually pass all global variables back to the caller. References *ref* are isomorphic to the natural numbers and contain *Null*.

The approach to specify procedures on lists basically follows [9]. From the pointer structure in the heap we (relationally) abstract to HOL lists of references. Then we can specify further properties on the level of HOL lists, rather then on the heap:

$List\ x\ h\ [] = (x = Null)$
$List\ x\ h\ (p\ \#\ ps) = (x = p \wedge x \neq Null \wedge List\ (h\ x)\ h\ ps)$

The list of references is obtained from the heap h by starting with the reference x, following the references in h up to *Null*.

We define in place list reversal. The list pointed to by p in the beginning is Ps. In the end q points to the reversed list $rev\ Ps$. The notation $r{\rightarrow}f$ mimics the field selection syntax of C and is translated to ordinary function application for field lookup and function update for field assignment.

lemma $\Gamma{\vdash}\{\!|List\ \acute{}p\ \acute{}next\ Ps|\!\}$
$\acute{}q := Null;$
WHILE $\acute{}p \neq Null$
INV $\{\!|\exists Ps'\ Qs'.\ List\ \acute{}p\ \acute{}next\ Ps' \wedge List\ \acute{}q\ \acute{}next\ Qs' \wedge$
$set\ Ps' \cap set\ Qs' = \{\} \wedge rev\ Ps'\ @\ Qs' = rev\ Ps|\!\}$
DO $\acute{}r := \acute{}p;\ \acute{}p := \acute{}p{\rightarrow}\acute{}next;\ \acute{}r{\rightarrow}\acute{}next := \acute{}q;\ \acute{}q := \acute{}r$ ***OD***
$\{\!|List\ \acute{}q\ \acute{}next\ (rev\ Ps)|\!\}$
by (*vcg*,*fastsimp*+)

In the loop, pointer p sequentially steps through the list Ps and q accumulates the reversed list. Therefore the desired outcome $rev\ Ps$ can be obtained by appending the the reversed list pointed to by p and the list pointed to by q. This is expressed by $rev\ Ps'\ @\ Qs' = rev\ Ps$ in the invariant. Separation of the two lists Ps' and Qs' is captured by the empty intersection of references: $set\ Ps' \cap set\ Qs' = \{\}$.

The specification of list reversal above, does not capture the information about the parts of the heap that do not change. But this information is crucial to properly use the specification in different contexts. We encapsulate this code fragment in a procedure and give the following, more detailed specification.

procedures $Rev(p|q) =$
$\acute{}q := Null;$
WHILE $\acute{}p \neq Null$
DO $\acute{}r := \acute{}p;\ \acute{}p := \acute{}p{\rightarrow}\acute{}next;\ \acute{}r{\rightarrow}\acute{}next := \acute{}q;\ \acute{}q := \acute{}r$ ***OD***

Rev-spec:
$\forall \sigma\ Ps.\ \Gamma{\vdash}\ \{\!|\sigma.\ List\ \acute{}p\ \acute{}next\ Ps|\!\}\ \acute{}q :=$ ***CALL*** $Rev(\acute{}p)$
$\{\!|List\ \acute{}q\ \acute{}next\ (rev\ Ps) \wedge (\forall p.\ p \notin set\ Ps \longrightarrow (\acute{}next\ p = {}^{\sigma}next\ p))|\!\}$

Rev-modifies:
$\forall \sigma.\ \Gamma{\vdash}\{\sigma\}\ \acute{}q :=$ ***CALL*** $Rev(\acute{}p)\ \{t.\ t$ *may-only-modify* σ *in* $[next,q]\}$

We have given two specifications this time. The first one captures the functional behaviour and additionally expresses that all parts of the *next* heap not contained in Ps, will stay the same (σ denotes the pre-state). The second one is a modifies clause that lists all the state components that may be changed by the procedure. Therefore we know that the *cont* parts will not be changed. The assertion t *may-only-modify* σ *in* $[next, p]$ abbreviates the following relation between the

final state t and the initial state σ: $\exists\, next\ p.\ t{=}\sigma(\!|next{:=}next, p\ {:=}p|\!)$. This modifies clause can be exploited during verification condition generation. We derive that we can reduce the *result* function in the call to *Rev*, which copies the global components *next* and *cont* back, to one that only copies *next* back. So *cont* will actually behave like a local variable in the resulting proof obligation. This is an effective way to express separation of different pointer structures in the heap and can be handled completely automatic during verification condition generation. For example, reversing a list will only modify the *next* heap but not some *left* and *right* heaps of a tree structure. Moreover the modifies clause itself can be verified automatically. The following example illustrates the effect of the modifies clause.

lemma includes *Rev-spec* + *Rev-modifies* **shows**
$\Gamma \vdash \{\!|\, `cont{=}c \wedge List\ `p\ `next\ Ps \,|\!\}\ `p := \textbf{\textit{CALL}}\ Rev(`p)$
$\quad \{\!|\, `cont{=}c \wedge List\ `p\ `next\ (rev\ Ps) \,|\!\}$
apply *vcg*

1. $\bigwedge next\ cont\ p.$
$\quad List\ p\ next\ Ps \Longrightarrow$
$\quad \forall\, nexta\ q.$
$\quad\quad List\ q\ nexta\ (rev\ Ps) \wedge (\forall\, p.\ p \notin set\ Ps \longrightarrow nexta\ p = next\ p) \longrightarrow$
$\quad\quad cont = cont \wedge List\ q\ nexta\ (rev\ Ps)$

The impact of the modifies clause shows up in the verification condition. The content heap results in the same variable before and after the procedure call ($cont = cont$), whereas the next heap is described by *next* in the beginning and by *nexta* in the end. The specification of *Rev* relates both next heap states.

Memory Management To model allocation and deallocation we need some bookkeeping of allocated references. This can be achieved by an auxiliary ghost variable *alloc* in the state space. A good candidate is a list of allocated references. A list is per se finite, so that we can always get a new reference. By the length of the list we can also handle space limitations. Allocation of memory means to append a new reference to the allocation list. Deallocation of memory means to remove a reference from the allocation list. To guard against dangling pointers we can regard the allocation list: $\{\!|\, `p{\neq}Null \wedge `p \in set\ `alloc \,|\!\}{\mapsto}\ `p{\rightarrow}`cont := 2.$

The use of guards is a flexible mechanism to adapt the model to the kind of language we are looking at. If it is type safe like Java and there is no explicit deallocation by the user, we can remove some guards. If the `new` instruction of the programming language does not initialise the allocated memory we can add another ghost variable to watch for initialised memory through guards.

7 Conclusion

We have presented a flexible, sound and complete Hoare calculus for sequential imperative programs with mutually recursive procedures and dynamic procedure call. We have elaborated how to model various kinds of abrupt termination like `break`, `continue`, `return` and exceptions, how to deal with global variables, heap and memory management issues. The polymorphic state space of the programming language allows us to choose the adequate representation for the current verification task. Depending on the context we can for example decide, whether it is preferable to model certain variables as unbounded integers in HOL or as bit-vectors, without changing the program representation or logics. Guards make it possible to customise the runtime faults we are interested in. The usage of records as state space representation gives us a natural way to express typing of program variables and yields comprehensible verification conditions. Moreover in combination with the modifies clause we can lift separation of heap components, which are directly expressible in the split heap model, to the level of procedures. Crucial parts of the frame problem can then already be handled during verification condition generation. The calculus is developed, verified and integrated in the theorem prover Isabelle and the resulting verification environment is seamless fitting into the infrastructure of Isabelle/HOL.

References

1. C. Ballarin. Locales and locale expressions in Isabelle/Isar. In S. Berardi, M. Coppo, and F. Damiani, editors, *Types for Proofs and Programs: International Workshop, TYPES 2003, Torino, Italy, April 30–May 4, 2003, Selected Papers*, number 3085 in Lect. Notes in Comp. Sci., pages 34–50. Springer-Verlag, 2004.
2. R. Bornat. Proving pointer programs in Hoare Logic. In R. Backhouse and J. Oliveira, editors, *Mathematics of Program Construction (MPC 2000)*, volume 1837 of *Lect. Notes in Comp. Sci.*, pages 102–126. Springer-Verlag, 2000.
3. J.-C. Filliâtre. Why: a multi-language multi-prover verification tool. Research Report 1366, LRI, Université Paris Sud, March 2003.
4. J. Harrison. Formalizing Dijkstra. In J. Grundy and M. Newey, editors, *Theorem Proving in Higher Order Logics: 11th International Conference, TPHOLs'98*, volume 1497 of *Lect. Notes in Comp. Sci.*, pages 171–188, Canberra, Australia, 1998. Springer-Verlag.
5. P. V. Homeier. *Trustworthy Tools for Trustworthy Programs: A Mechanically Verified Verification Condition Generator for the Total Correctness of Procedures*. PhD thesis, Department of Computer Science, University of California, Los Angeles, 1995.
6. M. Huisman. *Java program verification in higher order logic with PVS and Isabelle*. PhD thesis, University of Nijmegen, 2000.
7. B. Jacobs. Weakest precondition reasoning for Java programs with JML annotations. *Journal of Logic and Algebraic Programming*, 58:61–88, 2004.
8. T. Kleymann. Hoare Logic and auxiliary variables. *Formal Aspects of Computing*, 11(5):541–566, 1999.

9. F. Mehta and T. Nipkow. Proving pointer programs in higher-order logic. In F. Baader, editor, *Automated Deduction — CADE-19*, volume 2741 of *Lect. Notes in Comp. Sci.*, pages 121–135. Springer-Verlag, 2003.
10. J. Meyer and A. Poetzsch-Heffter. An architecture for interactive program provers. In S. Graf and M. Schwartzbach, editors, *TACAS00, Tools and Algorithms for the Construction and Analysis of Systems*, volume 1785 of *Lect. Notes in Comp. Sci.*, pages 63–77. Springer-Verlag, 2000.
11. M.J.C. Gordon. Mechanizing programming logics in higher-order logic. In G.M. Birtwistle and P.A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automatic Theorem Proving (Proceedings of the Workshop on Hardware Verification)*, pages 387–439, Banff, Canada, 1988. Springer, Berlin.
12. W. Naraschewski and M. Wenzel. Object-oriented verification based on record subtyping in higher-order logic. In *Theorem Proving in Higher Order Logics: 11th International Conference, TPHOLs'98*, volume 1479 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 1998.
13. T. Nipkow. Hoare logics in Isabelle/HOL. In H. Schwichtenberg and R. Steinbrüggen, editors, *Proof and System-Reliability*, pages 341–367. Kluwer, 2002.
14. T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 2002. http://www.in.tum.de/~nipkow/LNCS2283/.
15. M. Norrish. *C formalised in HOL*. PhD thesis, University of Cambridge, 1998.
16. D. v. Oheimb. *Analyzing Java in Isabelle/HOL: Formalization, Type Safety and Hoare Logic*. PhD thesis, Technische Universität München, 2001.
17. C. Pierik and F. S. de Boer. Computer-aided specification and verification of annotated object-oriented programs. In *Formal Methods for Open Object-Based Distributed Systems 2003*, volume 2884 of *Lect. Notes in Comp. Sci.*, pages 163–177. Springer-Verlag, 2002.
18. L. Prensa Nieto. *Verification of Parallel Programs with the Owicki-Gries and Rely-Guarantee Methods in Isabelle/HOL*. PhD thesis, Technische Universität München, 2002.
19. M. Wenzel. Miscellaneous Isabelle/Isar examples for higher order logic. Isabelle/Isar proof document, 2001.