High Assurance System Software

Gerwin Klein **Ralf Huuck**

National ICT Australia Ltd. (NICTA)
Locked Bag 6016
The University of New South Wales
Sydney NSW 1466
Australia
Email: {gerwin.klein|ralf.huuck}@nicta.com.au

Abstract

This paper describes an approach to developing high assurance system software. We demonstrate how different formal methods can be applied in the development process by matching specific techniques and tools to the different levels of system requirements and how those techniques can complement each other.

Keywords: System software, kernel design, theorem proving, static analysis.

1 Introduction

System software is at the core of any application and the correctness of the system layer is crucial to their safety and security. This is even more important if the application itself is used in an embedded, safety critical environment. We believe that the convergence of two trends in system design will help to develop such high assurance software. Those trends are:

Small trusted computing base. The *trusted computing base* (TCB) comprises all system components that are essential to the safety and security of the system. In the operating systems area, micro-kernel designs show that the size of the trusted computing base can be drastically reduced. Micro-kernels provide the basic mechanisms for isolation and confinement of system components such that only a small part of the system has to be trusted to ensure safe and secure operation. Keeping the TCB small has the great advantage of focussing the design and implementation on a few highly reliable components while components outside the TCB require medium analysis effort since they can be updated or fixed much more easily without affecting the integrity of the system design as a whole.

Formal analysis. It is virtually impossible to guarantee correctness of a system, and in turn the absence of bugs by standard software engineering practice such as code review, systematic testing and good software design alone. The complexity of system software is typically too high to be manageable by informal reasoning or reasoning that is not tool supported. The formal methods community has developed various rigorous, mathematically sound techniques and tools that have matured enough within the last decades to allow the formal analysis of system software.

National ICT Australia is funded by the Australian Government's Department of Communications, Information Technology, and the Arts and the Australian Research Council through *Backing Australia's Ability* and the ICT Research Centre of Excellence programs.

In this paper, we present an approach that allows the design of high assurance system software by matching appropriate formal methods to the development process and to the various components of a system. Section 2 describes the concepts of trusted computing base and micro-kernels in more detail. Section 3 gives a short overview of the two methods we propose to apply: interactive theorem proving and static analysis. Section 4 reports on experience gathered in applying the former to L4, a modern high-performance micro-kernel.

2 Trusted Computing Base

The trusted computing base is a priori a somewhat vague concept. It comprises all system components that are essential to the safety and security of the system. This potentially encompasses the operating system, device drivers, the application itself, middleware etc. It also critically depends on what exactly the safety and security policies of the system are.

On one end of the spectrum, the TCB of traditionally developed software can be very large, and in effect encompass the entire system. A standard web server running a monolithic operating system like Windows or Linux, for instance, with the goals of preventing unauthorized access and delivering service up to a certain number of requests per second, potentially counts inconspicuous pieces of code like the sound card driver to its TCB. It runs in the privileged mode of the hardware and in principle has access to every part of the system. A bug in this driver can directly lead to a system crash and therefore to denial of service, or — with the right kind of bug and exploit — it could even allow arbitrary code to run on the machine and thereby circumvent access control.

It is clearly beneficial for the trustworthiness of the system to reduce the amount of code that needs to be trusted. This is one of the reasons well secured web servers run with a minimal set of software and services.

On the other end of the sprectrum is the approach known as proof-carrying code (PCC) (Necula 1997). Here, the application itself can be completely untrusted, but is accompanied by a formal proof of safety. The goal usually is a simple property like memory safety, i.e., the guarantee that the application only accesses memory that it has allocated itself. The TCB of a PCC system consists of a small proof checker that takes the (binary) application code and the proof that is delivered with it, and then checks

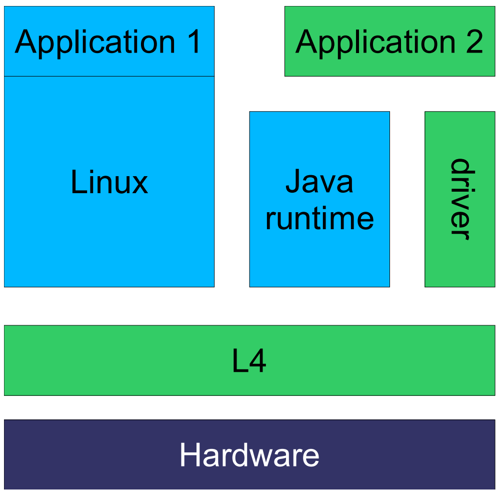

Figure 1: Example mobile phone architecture with the L4 micro-kernel

that this is indeed a valid proof of the desired safety property for this particular program. Of course, it still relies on the operating system to provide its service correctly to the application, but apart from that, the TCB in proof-carrying code systems is not only minimal, but also constant over a wide range of applications, and thus easier to validate and gain trust in. Unfortunately this approach is still in its infancy and has not gained a wide foothold yet in practice. It would require a standardized proof platform, vendors that ship proofs with their code, and extensions to more interesting safety and security policies. Although many of these proofs can be obtained completely automatically, the extension to other safety policies is still a very active area of research.

A practical approach to reducing the TCB and thus gaining a high degree of trustworthiness and reliability has its origin in standard good software engineering practice: modularization, minimality, and loose coupling of components. Micro-kernel designs take these principles to the base of the system, the OS layer. The first generation of micro-kernels like Mach (Rashid, Julin, Orr, Sanzi, Baron, Forin, Golub & Jones 1989) were not able to deliver on the promises of minimality, and, for an OS almost more importantly, performance. Modern micro-kernels like L4 (L4Ka Team 2001) have overcome these problems and are in industrial use for a wide range of platforms. The design goals of modern micro-kernels are minimality, performance, and the ability to give strong separation and confinement guarantees such that for instance multiple full operating system personalities (e.g. Linux) can run on top of them independently. Separation means that failure of one process can not be observed by other processes and confinement refers to the ability of controlling communication between processes.

Figure 1 shows an example system architecture using the L4 micro-kernel for a (hypothetical) mobile phone. The phone is running Linux with a graphical front-end as user-interface, provides a native Java runtime for downloaded applications and games, and a separate module for handling the real-time and security sensitive communication tasks. The modules are loosely coupled, they communicate by synchronous IPC (inter-process communication) only.

The TCB of this architecture for a safety policy that focuses on the availability and security of communication encompasses the micro-kernel and the communications module only. The TCB for a policy that allows untrusted code to be downloaded onto the phone consists of the micro-kernel and the Java runtime environment only. The TCB for smooth operation of the user interface is again independent of communications and untrusted programs. This shows that the micro-kernel design not only has benefits for the reliability of the operating system:

- it reduces the TCB of applications, and
- it enables the analysis of applications to proceed in isolation.

Both help to make the problem of establishing the trustworthiness of a system manageable and they bring the size of the problem down to a level that might be amenable to more rigorous techniques than mere testing.

3 Methods

The last section showed a technique to bring the TCB down to a manageable level. This sections gives an overview of two methods that show promise of significant impact on the problem of establishing that a reduced size TCB is indeed reliable.

3.1 Theorem Proving

The reduction in size, compared to traditional approaches, already goes a long way towards making the TCB more trustworthy. Standard methods for establishing the trustworthiness of software, such as testing and code review (while they inherently cannot guarantee absence of faults) work better on a smaller code base. However, they cannot provide confidence in full functional correctness, nor can they give hard security guarantees.

The only real solution to establishing trustworthiness is formal verification, *proving* the implementation correct. This has, until recently, been considered an intractable proposition — the OS layer alone was already too large and complex. Owing to the combination of improvements in formal methods and the trend towards micro-kernel designs and hence smaller TCBs, full formal verification of at least parts of the system now seems to be within reach.

Even with these two trends combined, formal verification still requires an high initial investment and experts, both in the application domain as well as in the verification method and logic that is being applied. We therefore propose to use full formal verification where the impact is highest and benefit greatest. In micro-kernel based designs this clearly is the kernel itself. As figure 1 indicated, it necessarily is part of every TCB of a system, and it stays constant over different systems.

Formal verification is about producing a strict mathematical proof of the correctness of a system. From the formal methods point of view, this means that a formal *model* of the *system* behaves in a manner that is consistent with a formal *specification* of the *requirements*. This leaves a significant semantic gap between the formal verification and the user's view of correctness. The user views the system as "correct" if the behavior of its object code on the target hardware is consistent with the user's interpretation of the (usually informally specified) behavior. Bridging this semantic gap is called formalization.

For the purpose of verifying the functional correctness of a kernel, the specification is a formalization of its applications programmer interface (API), and, ideally, the model is the code that is being executed. At present there are two main verification techniques in use for establishing a correspondence between them: model checking and theorem proving.

Model checking works on a model of the system that is typically reduced to what is relevant to the specific properties of interest. The model checker then exhaustively explores the model's reachable state space to determine whether the properties hold. This approach is only feasible for systems with a moderately-sized state space, which implies dramatic simplification. As a consequence, model checking is unsuitable for establishing a kernel's full compliance with its API. Instead it is typically used to establish very specific safety or liveness properties.

The *theorem proving* approach involves describing the intended properties of the system and its model in a formal logic, and then deriving a mathematical proof showing that the model satisfies these properties. The size of the state space is not a problem, as mathematical proofs can deal with large or even infinite state spaces. This makes theorem proving applicable to more complex models and full functional correctness.

Contrary to model checking, theorem proving is usually not an automatic procedure, but requires human interaction. While modern theorem provers remove some of the tedium from the proof process by providing rewriting, decision procedures, automated search tactics, etc, it is ultimately the user who guides the proof, provides the structure, or comes up with a suitably strong induction statement. While this is often seen as a drawback of theorem proving, we consider it its greatest strength: it ensures that verification does not only tell you *that* a system is correct, but also *why* it is correct.

Proof-based OS verification has been tried in the past (Neumann, Boyer, Feiertag, Levitt & Robinson 1980, Walker, Kemmerer & Popek 1980). The rudimentary tools available at the time meant that the proofs had to end at the design level; full implementation verification was not feasible. The verification of Kit (Bevier 1989) down to object code demonstrated the feasibility of this approach to kernel verification, although on a system that is far simpler than any real-life OS kernel in use in secure systems today.

Since the early attempts at kernel verification there have been dramatic improvements in the power of available theorem proving tools. Proof assistants like ACL2, Coq, PVS, HOL and Isabelle have been used in a number of successful verifications, ranging from mathematics and logics to microprocessors (Brock, Hunt, Jr. & Kaufmann 1994), compilers (Berghofer & Strecker 2003), and full programming platforms like JavaCard (Ver 2005*a*).

We therefore decided about a year ago to attempt a verification of a real kernel. We are among several current efforts with this goal, notably VFiasco (Hohmuth, Tews & Stephens 2002), VeriSoft (Ver 2005*b*) and Coyotos (Shapiro, Doerrie, Northup, Sridhar & Miller 2004). We target the L4 microkernel in our work as it is one of the smallest and best performing general-purpose kernels, it is deployed industrially and its design and implementation is well understood in our lab.

The main challenges for a project like this are the following.

Size We are attempting to formally prove the functional correctness of a program measuring in the order of 10.000 lines of C++ and assembler code. This is very small for an operating system, but beyond what has been done before in formal verification of implementation correctness.

Complexity Following the philosophy that only those parts of the system that strictly require privileged hardware access are part of the microkernel, we are left with a program that is cut down to only the essential concepts of interprocess communication (IPC), threads, and virtual memory management. These three concepts are heavily intertwined and hard to modularize. This means that also the verification of the kernel is hard to separate into independent parts. Therefore the full conceptual complexity of the kernel is visible to most parts of the verification process.

Level of Abstraction The two main techniques for managing complexity are modularization and abstraction. This is true for software engineering and even more so for software verification. Unfortunately and contrary to applications, the internals of micro-kernels are not easily modularized and they work on a very low level of abstraction, partly with hand-optimized assembler code, hardware architecture dictated data structures, and direct hardware access. One major challenge is to work on a higher level of abstraction as long as possible and only use lower levels of abstraction in isolated places where necessary.

Requirements Specification The starting point of a verification project is capturing the intended behavior of the system in a formal specification. Ideally the system interface is well documented, and the API specified in precise natural language that only needs to be translated into a suitable formalism. Reality, of course, is different. Although L4 comes with extensive documentation that for an operating system is of high quality, it tends to focus on the syntactic parts of the API and is vague on the semantics, the intended behavior. It describes for instance in very precise detail, how each bit in each argument of a system call is decoded, but it does not describe at the same level of precision, how the call affects the state of the system. This is not surprising and not specific to L4. The semantics of these system calls is on the one hand intuitively clear to kernel developers and experienced users and on the other hand difficult to describe precisely. This is exactly the task of the formal specification: capturing the understanding of what for instance 'sending an IPC' means. A specification like this should be as abstract as possible and as concrete as necessary. At the API level there should be no need to describe the hardware dictated layout of page tables, for instance. It should be enough to say that there is a data structure that can deliver a physical address for each virtual address, possibly together with some constraints on the size and structure of addresses. There exists a multitude of different formal languages that are more or less suited for this task. For this verification project, we chose higher-order logic (HOL) as the description language. It is well supported by the main theorem proving tool we use (Isabelle (Nipkow, Paulson & Wenzel 2002)), it is expressive enough to conveniently describe kernel behavior in a operational manner that is easy to understand, and it can be used as a typed, functional programming language, which makes it more accessible to programmers than a purely mathematical formalism like set theory.

Requirements Analysis Writing down a formal specification of the intended behavior is a good first step, but in itself does not guarantee anything about the actual system apart from the fact that some thought has been given to how it ought to work and what correct operation means. The next step is to analyze the formal description and to derive some (formal) properties about it.

Examples of this are 'with the exported virtual memory management operations one virtual address can never be associated with two physical addresses at the same time'. This can be phrased as a theorem and be formally proven. This activity is one the most effective ones to find inconsistencies, specification errors and errors in the intended behavior of the system at a very early stage in the development process — possibly before any code has been written, even before any concrete design has been finalized.

System Model As mentioned above, the kernel model is ideally the kernel executing on the hardware. In reality it is preferable to take advantage of the abstraction provided by the programming language in which the kernel is implemented, so the model becomes the kernel's source-level implementation. This introduces a reliance on the correctness of the compiler and linker (in addition to the hardware, boot-loader and firmware), but verification of the compiler becomes an orthogonal issue, and is an active area of research which has recently achieved some success (Berghofer & Strecker 2003). The challenge for the system model is again to capture not the syntax, but the semantics of the program. While the complete formal semantics of systems languages is an active area of research (Norrish 1998, Hohmuth et al. 2002), a complete semantics is not required. For our purpose it suffices to have a semantics for the language subset that is actually used in the implementation. In isolated places we even change the implementation to remain in a safe subset of C++. Such changes are acceptable as long as they have no significant performance impact.

Proof Methodology The main activity of the verification is coming up with a formal, machine-checked proof that the model of the system implementation correctly implements the specification; that it either exhibits exactly the same or only a subset of the behaviors described by the specification. Ideally this should be done in such a way that all safety theorems that were shown in the formal requirements analysis phase automatically hold for the implementation as well. This is not easily possible for all kinds of properties. Information flow related properties for instance might hold on the specification, but not on all functionally correct implementations. However, the majority of safety-related properties are preserved by this process. The problem of establishing that the system model implements its specification is by no means easy, but relatively well understood (Morgan 1990, de Roever & Engelhardt 1998). The challenge lies in mechanizing the known theory in a theorem proving tool and applying it to large-scale problems.

Maintenance One of the major open problems is code maintenance: how to deal with change once the micro-kernel is verified. In principle every small change invalidates the correctness proof that took a major project to accomplish. Reality is not quite that bleak. Only the process of coming up with the proof requires human interaction, the process of checking if an existing proof still works is automatic. This means it is easy to determine which sub-proofs are affected by a change and only these need to be fixed. Depending on the nature of the change, this can either be a small lemma about one function that is only used locally, or it could be a main theorem that has to be restated completely, affecting a whole kernel subsystem. We expect the former kind to be something like local performance optimizations that might occur relatively often, the latter major conceptual changes in the way the micro-kernel works which happen comparatively rarely. At least the most common kind of change in normal programs — bug fixes — are unlikely to happen in a formally verified program.

One concern that is often voiced for formal verification is *How do you know that your proof and your theorem prover are correct?* This really is the question *Can I trust formal verification?* It is voiced, because formal verification is difficult and hence not a widely used and experienced technique. Its limitations are not widely known which makes it hard to trust completely. The answer to this concern is twofold.

On the social side, we hope that projects like this one, aiming to formally verify software in wide-spread industrial use, show how reliable the resulting code is and thus give an empirically and psychologically more accessible reason for trust rather than theoretical results and case studies with limited practical use.

On the technical side, the soundness problem has had significant research exposure and has lead to modern theorem provers that can guarantee correctness of formal proofs to a very high degree by architectural design, independence of particular compilers and machine architectures, and independent, small proof checkers. Some provers like Isabelle allow proofs to be written in human-readable form that can again be independently checked by human experts. The proof and the theorem prover are the least likely sources of errors in the process. Of somewhat more concern are *Does the formal specification say what the author thinks it says?* and *Is the system model really what is being executed?* The former is reasonably easy to answer when the specification language is expressive enough and fits the problem. The latter is the larger gap. As mentioned above, modeling C++ semantics correctly is not an easy task, and working on the source code level requires trusting the compiler, linker, and hardware. Compared to the absolute assurance that the formal proof gives this seems like a big gap, but trusting their understanding of the language, trusting the compiler and hardware it is of course something that programmers do routinely. This is the part that for now still needs to be validated by traditional means, but it is orders of magnitude easier than the original problem of validating an operating systems kernel.

This current situation does not necessarily need to continue. Verified compilers are an active research area as is verified hardware. It is entirely possible that in the medium to long term the required level of trust can be pushed down to the hardware manufacturing process only.

3.2 Static Analysis

Static analysis (Muchnick 1997, Nielson, Nielson & Hankin 1999) is a general term comprising a number of analysis techniques which can be applied at compile-time, i.e., prior to the execution of the actual code. In fact, some of these techniques can be applied in even earlier design stages when the code does not yet compile and, therefore, is not executable. Moreover, static analysis typically refers to techniques which can be executed fully automatically, i.e., there is no interaction from the user needed during the analysis process. However, for some analysis methods the user might be required to annotate his code appropriately and, of course, the user has to be able to interpret the analysis results, in particular when the analysis reveals any errors.

The drawback of any automatic software analysis technique is that almost all properties are generally undecidable as the problem can be reduced to the halting problem (Turing 1936-1937). To make them nonetheless usable in practice, decidable approximations are computed. These can be either *over-* or *under-approximations.*

With regard to safety properties over-approximations consider abstract programs which exhibit more behavior than the actual concrete program. If an abstract program still satisfies a given safety property, although it exhibits more behavior, then the concrete program will do so as well (since it has less behavior that can violate that property). However, if the abstract program does violate the given property it does not necessarily mean that the concrete program does violate it as well, since the violation might just be in the over-approximated part of the program behavior which is not in the concrete program. In this case we have a *false alarm* or *false positive.* It is a major research challenge in the area of automatic software analysis to minimize the number of false alarms.

Under-approximations on the other hand consider program approximations which exhibit less behavior than the original program. In this case any violation of a safety property in the approximated program is certainly a violation in the original program while the absence of a violation does not guarantee that the program has no harmful (i.e., property violating) behavior. Again, it is a major research challenge to keep the gap between the actual behavior and the approximated one as small as possible.

Under-approximations are well suited to exploit bugs in programs while over-approximations are used to establish correctness. Most static analysis approaches pursue over-approximations.

In the remainder of this section we illustrate two well-known static analysis techniques, data flow analysis and abstract interpretation, and their application to software analysis.

3.2.1 Data Flow Analysis

Data flow analysis (Aho, Sethi & Ullman 1986) is a flow sensitive method to derive information related to the flow of data along control paths, more precisely, for every program point information that summarizes some property of all the possible dynamic instances of that point are computed. It does distinguish between when and how a particular instance is reached. Data flow analysis originates from compiler construction and was born out of the urge to develop efficient and compact code. For instance, a program containing assignments to a variable which is not used later, i.e., containing dead code, is non-optimal, this code fragment can be eliminated and the resulting program code is more compact.

We will illustrate the data flow problem by an example. Consider the contrived C++ code in Figure 2. A variable `tbn` is initialized by one, a spinlock is acquired and afterwards depending on the value of `tbn` an error message is thrown or some `updateState` function will be called repeatedly. Only in the later case will the spinlock be freed again. We assume that both the `err` and the `updateState` function terminate and do not abort the program and that `updateState` does not modify `tbn`.

Data flow analysis works on the control graph of a program. It is custom to first partition the program into *basic blocks* which are maximal sequences of assignments which are executed sequentially. In particular they do not contain loops or branching.

```
int contrived_fun()
 {
    int tbn = 1;

    spinlock.lock();

    if tbn > 10
    {
      err("not expected!");
    }
    else
    {
      while (tbn <= 10)
      {
        updateState(tbn);
        tbn += 1;
      }
     spinlock.unlock();
   }
 return 0;
}
```

Figure 2: Example Code Fragment

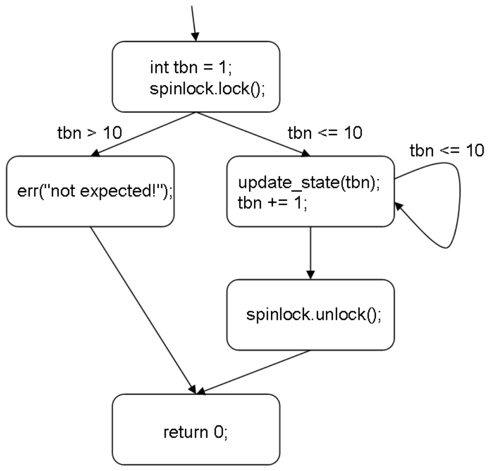

Figure 3: Control Graph

The resulting control graph for the example of Figure 2 where the nodes are basic blocks is shown in Figure 3.

Assume we like to figure out if every `lock` operation is eventually followed by an `unlock` operation. Any violation of this property might indicate a flaw in the resource management. We can state this as a data flow problem as follows: Determine for all program nodes q if a lock has been acquired on any path to q and not yet released. This means in particular, if an unreleased lock reaches the node containing the `return` statement our desired property will be violated.

Computing solutions. Data flow problems consist of a *property space* and a *flow function.* The property space characterizes the information of interest, e.g., the lock acquired but not released or in the case of several locks, the set of locks which haven't been released yet. The flow function characterizes the transfer of information by computing the effect of a basic block on a property in the property space. For instance a block might generate a lock that has to be released (added to the set of locks) or might release one (removed from the set of locks). In general, its effect (output) depends on the output of its predecessor blocks.

The ideal solution to a data flow problem is to take all the paths in a program which will actually be executed, apply the flow function to any block along the path and take the *meet* of all the path results. The meet can be either the union or disjunction of the information obtained, i.e., property computed, depending on the problem. In our example we have to take the union since we are interested in any path that might not release a lock.

Unfortunately, it is undecidable to statically determine exactly the set of paths that will be executed. An approximation to this is to take the set of all paths in a program. Certainly, this is an over-approximation, since the program semantics is not taken into account and, i.e., more possible executions are considered than there might actually be. However, loops introduce an infinite set of paths as every loop can be traversed arbitrarily often. This makes it infeasible to effectively compute the meet over all paths.

A feasible solution is to concentrate on edges rather than paths in the control graph. For every node the effect of all incoming edges (in a forward analysis) is computed leading to an effect for the outgoing edge. This is repeated all over until a fixed point in the data flow is reached. As long as the property space can be described as a lattice of finite height (cf. (Nielson et al. 1999)), this procedure will terminate. Moreover, the result is a solution to the data flow problem which is a further over-approximation of the solution taking all paths into account.

Depending on the problem the propagation of information is done from incoming to outgoing edges (forward analysis) or vice versa (backward analysis). Moreover, there are sophisticated heuristics to determine the fixed point without unnecessary recomputations.

All the solutions which are over-approximations of the actual paths executed in the program of Figure 2 imply that at the end of the program the lock might have been acquired but not released. The reason for this is, of course, that data flow analysis does not take the program semantics into account, which would rule out the "then" branch of the program.

3.2.2 Abstract Interpretation

Abstract interpretation (Cousot 1978, Cousot & Cousot 1979) is a very general framework that can be applied to various fields: to syntax in order to compare grammars, to semantics that helps to design semantic hierarchies, to typing, to model checking and program transformations in order to provide suitable means of abstractions. In this paper we present the idea of abstract interpretation for program analysis.

As seen in the previous section, data flow analysis might result in coarse over-approximations, since it takes the whole control flow graph into account, no matter whether some branches are semantically impossible to take. Abstract interpretation can sometimes remedy this problem by adding semantic information to the analysis, making it more precise.

Abstract interpretation is about relating two structures, the *concrete domain* and the *abstract domain,* and defining for every operation on the concrete domain a matching operation on the abstract domain. This enables the execution of programs purely on the abstract level, which leads to over-approximated but decidable program behavior.

We illustrate this by an example. Consider the program of Figure 2. To come up with a more precise data flow analysis result, it would be of interest to detect that the "then" branch of the program can actually never be taken. This, however, requires that

```
int contrived_fun()                     value of x
 {
    int tbn = 1;                        [1,1]

    spinlock.lock();                    [1,1]

    if tbn > 10                         [1,1]
    {
      err("not expected!");             [11,+∞]
    }
    else
    {
      while (tbn <= 10)                 [1,10]
      {
        updateState(tbn);               [1,10]
        tbn += 1;                       [1,11]
      }
     spinlock.unlock();                 [1,11]
   }
 return 0;                              [1,11]
}
```

Figure 4: Abstract Interpretation Result

we track the values of the tbn variable. In general, it is impossible to track variable behavior precisely without executing the full program. In particular, it is impossible to determine for each program location the set of all possible values a variable can take.

Assume that sets of integers constitute the concrete domain. We define the abstract domain by intervals. Precisely, for every the set we define an interval by taking the lowest and highest number as boundaries. The means the set $\{1, 2, 3\}$ is represented by $[1, 3]$, $\{1, 2, 5\}$ by $[1, 5]$, and $\{1, 2, 3, \ldots\}$ by $[1, +\infty]$. The abstract representation might lead to over-approximation, but allows also concise representation of infinite sets. Next, we introduce for any concrete program operation such as $+$, $*$, etc. an abstract operation on intervals, e.g., $[a, b] +_I [c, d] := [a + c, b + d]$.

We can now execute the program on an abstract level, observing the range of integer variables. This alone does not guarantee termination, but it comes in handy to simulate programs for sets of inputs simultaneously. Termination, however, can be enforced by *acceleration techniques* (Cousot 1981) speeding up the convergence of the analysis. These accelerations provide a safe approximation of the program behavior, however, they often come with an additional loss of precision, i.e., can lead to further over-approximation. E.g., after unfolding loops a few times and observing the change of some variable it is always safe to approximate its range by $[-\infty, +\infty]$, although it might not be very precise. Nonetheless, for the example above a sophisticated abstract interpretation framework can compute the results as shown in Figure 4.

A quick analysis[1] reveals that the "then" branch of the program will never be taken. This leads to a smaller control graph that has to be taken into account for the data flow analysis. In turn, the data flow analysis will reveal that indeed every lock operation will be followed by an unlock operation. Remember, since we are dealing with over-approximations, this result actually verifies the property for all possible program executions.

[1]This analysis can be part of the abstract interpretation by extending the framework to Boolean expressions.

4 Application

In the following we report on first experiences at applying the two formal methods described in section 3 to the L4 micro-kernel.

4.1 Theorem Proving

As the initial formal verification of an operating system kernel clearly is a high-risk project, we first embarked on a pilot project in the form of a constructive feasibility study. Its aim was three-fold: (i) to formalize the L4 API, (ii) to gain experience by going though a full verification cycle of a small portion of actual kernel code, and (iii) to develop a project plan for a verification of the full kernel. An informal aim was to explore and bridge the culture gap between kernel programmers and theorists, groups which have been known to eye each other with suspicion.

The formalization of the API was performed using the B Method (Abrial 1996), independently of the Isabelle development in the rest of the project, as there existed a significant amount of experience with this approach among our student population. While L4 has a detailed and mature informal specification of its API (L4Ka Team 2001), it contains the usual problems of natural language specifications: incompleteness, ambiguity, and, at points, inconsistency. Furthermore it was at times necessary to extract the intended and expected kernel behavior from the designers themselves and, occasionally, the source code.

This part of the project was done by a final-year undergraduate student. The result was a formal API specification, covering a large part of the system, describing in particular the IPC and threads subsystems of L4. The remaining subsystem (virtual memory) was formalized separately in the verification part of the project described below. The B specification consists of about 1000 lines of code.

The full verification was performed on the most complex subsystem, the one dealing with mapping of pages between address spaces and the revocation of such mappings. We formalized a significant part of this API section and verified a subset of its functionality, corresponding to approximately 5% of the kernel source code. Its implementation consists of the page tables, the *mapping database* (used to keep track of mappings for revocation purposes), and the code for lookup and manipulation of those data structures.

We use higher-order logic and the theorem prover Isabelle as our tool set to describe the behavior of the kernel at an abstract level in the form of an operational specification. This description is then *refined* inside the prover into a program written in a standard, imperative, C-like language. This means we did not verify existing kernel code as such, but wrote a new program inside the theorem prover that happens to look mostly like the original.

The abstract description is at the level of a reference manual and relatively easy to understand. At that level, address translation for instance is modeled as an abstract function from virtual to physical addresses, explicitly not mentioning that this lookup is implemented by complex page table data structures in the real system. This is the level we use for analyzing the behavior of the system and for proving additional simple safety properties, such as the requirement that the same virtual address can never be translated to two different physical addresses. At the end of the refinement process stands a formally verified imperative program — the kernel implementation in full detail. A purely syntactic translation then transforms this program into ANSI C. This last stage is soundness critical and not verified. However, the translation is very simple and only consists of about 300 lines of ML code. This is small enough to be confidently validated manually. Besides a number of small safety properties, the main property, we are verifying, is implementation correctness: the kernel implementation behaves as the abstract model prescribes. A detailed description of this process can be found elsewhere (Klein & Tuch 2004, Tuch & Klein 2004).

We found Isabelle suitable for the task. It is mature enough for use in large-scale projects and well-documented, with a reasonably easy-to-use interface. Being actively developed as an open source tool, we are able to extend it and (working with the developers) to fix problems should they arise.

However, we are convinced that some important requirements must be met for such a project to have a chance of success. It is essential that some of the participants have significant experience with formal methods and a good understanding of what is feasible and what is not, and how best to approach it. On the other hand, it is essential that some of the participants have a good understanding of the kernel's design and implementation, the trade-offs underlying various design decisions, and the factors that determine the kernel's performance. It must be possible to change the implementation if needed, and that requires a good understanding of changes that can be done without undermining performance.

The investment for the virtual memory part of the pilot project was about 1.5 person years. All specifications and proofs together run to about 14,000 lines of proof scripts. This is significantly more than the effort invested in the virtual memory subsystem in the first place, but it includes exploration of alternatives, determining the right methodology, formalizing and proving correct a general refinement technique, as well as documentation and publications.

We estimate that the full verification of L4 will take about 20 person years, including verification tool development. This number must be seen in relation to the cost of developing the kernel in the first place, and the potential benefits of verification. The present kernel (L4Ka Team 2001) was written by a three-person team over a period of 8–12 months, with significant improvements since. Furthermore, for most of the developers it was the third in a series of similar kernels they had written, which meant that when starting they had a considerable amount of experience. A realistic estimate of the cost of developing a high-performance implementation of L4 is probably at least 5–10 person years.

Under these circumstances, the full verification no longer seems prohibitive, and we argue that it is, in fact, highly desirable. The kernel is the lowest and most critical part of any software stack, and any assurances on system behavior are built on sand as long as the kernel is not shown to behave as expected. Furthermore, formal verification puts pressure on kernel designers to simplify their systems, which has obvious benefits for maintainability and robustness even when not yet formally verified.

4.2 Static Analysis

We recently engaged in a project to apply static analysis techniques to system software, in particular the L4 micro-kernel. While there are few conclusions available right now, there are some interesting aspects of new challenges created by the unique architecture of micro-kernel software.

Properties. Traditionally, static analysis is concerned with properties such as buffer overflows, unreachable code, range violations, or division by zero. This properties are, however, of less importance for a micro-kernel such as L4. The simple reason is that there are hardly any arithmetic operations, buffers, arrays etc. that might cause problems. On the other hand, there a numerous interactions with the hardware, bit manipulations and excessive pointer arithmetic. This provides new challenges to static software analysis but also opens new areas of research.

Mixed code/parsing. The L4 micro-kernel is written in most parts in C++. However, critical parts implementing the interaction with the underlying hardware are implemented in assembly and sometimes inlined. Having to deal with a mixture of programming languages is a challenge. Moreover, there are few existing approaches that deal with C++ at all. Even parsing C++ is not easy. We therefore follow an approach of using gcc as a frontend. The advantages are that we do not have to write our own parser, are using the same compiler that produces the object code and do not have to worry about parser updates.

Specification Language. Ideally, a static analysis tool runs on some piece of code and returns for a given set of properties either error-traces or a correctness statement. While there are several generic properties to be satisfied for the L4 kernel, most of them are only expected to hold under certain circumstances and for certain program parts. This means, there are many exceptions to the rule, which might raise unnecessarily many false alarms in a generic analysis environment. It is therefore desirable to have an annotation and/or specification language that allows to clearly mark which program parts should be subject to a certain analysis. Moreover, a simple specification language such as a state machine is desirable to be more flexible and specify simple rules easily. Approaches such as (Engler, Chelf, Chou & Hallem 2000) appear to be promising.

A pilot project has been running for 2 months now helping to establish an understanding of the unique nature of system code in contrast to application code. The pilot phase is expected to go for one year (2 person years). The goal is to clearly identify research challenges, evaluate existing tools, and build an experimental tool for L4 micro-kernel analysis. The primary techniques the team is looking at are the ones outlined in Section 3.2.

5 Conclusions

Theorem proving and static analysis are two different approaches to support the development of high assurance software.

The flexibility and power of interactive theorem proving helps to create software of the highest degree of trustworthiness. The effort, however, is considerable. Not only is scalability a serious issue, but also the expertise of the people involved in the process is demanding. As shown in Section 4, only the latest developments of micro-kernels towards a small TCB brings interactive theorem proving into the realms of practical, full-fledged verification.

The strength of static analysis is that it is automatic, works on the source code, and is scalable. The downside are approximated results and rather generic properties. As we pointed out, the unique nature of system code requires tailored properties and algorithms and/or a specification language that matches the requirements of kernel developers. Such a static analysis tool could keep up with frequent kernel redesigns during experimental phases and would seamlessly integrate into the development process.

The ultimate goal is to combine the theorem proving and static analysis approach. This would include properties which are automatically verified during a static analysis phase as assumptions into the theorem proving framework. Such a transfer of information would save the tedious effort of proving many generic properties. Furthermore, properties which are shown to be incorrect during a static analysis phase don't even have to be attempted to be proven correct. In the other direction, theorems shown to be valid in the proving environment can ideally be used to annotate the source code in such a way that it guides the static analysis process.

We expect that the combination of both approaches will deliver highly trustworthy system software in acceptable time.

References

Abrial, J.-R. (1996), *The B Book: Assigning Programs to Meanings*, Cambridge University Press.

Aho, A. V., Sethi, R. & Ullman, J. D. (1986), *Compilers: Principles, Techniques and Tools*, Addison-Wesley.

Berghofer, S. & Strecker, M. (2003), Extracting a formally verified, fully executable compiler from a proof assistant, *in* 'Proc. COCV'03', Electronic Notes in Theoretical Computer Science, pp. 33–50.

Bevier, W. R. (1989), 'Kit: A study in operating system verification', *IEEE Transactions on Software Engineering* **15**(11), 1382–1396.

Brock, B. C., Hunt, Jr., W. A. & Kaufmann, M. (1994), The FM9001 microprocessor proof, Technical Report 86, Computational Logic, Inc.

Cousot, P. (1978), Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique de programmes, PhD thesis, Université scientifique et médicale de Grenoble, France.

Cousot, P. (1981), Semantic foundations of program analysis, *in* S. Muchnick & N. Jones, eds, 'Program Flow Analysis: Theory and Applications', Prentice-Hall, Inc., Englewood Cliffs, New Jersey, chapter 10, pp. 303–342.

Cousot, P. & Cousot, R. (1979), Systematic design of program analysis frameworks, *in* 'Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages', ACM Press, New York, NY, San Antonio, Texas, pp. 269–282.

de Roever, W.-P. & Engelhardt, K. (1998), *Data Refinement: Model-Oriented Proof Methods and their Comparison*, number 47 *in* 'Cambridge Tracts in Theoretical Computer Science', Cambridge University Press.

Engler, D., Chelf, B., Chou, A. & Hallem, S. (2000), Checking system rules using system-specific, programmer-written compiler extensions, *in* 'Proceedings of the Fourth Symposium on Operating Systems Design and Implementation, San Diego, CA'.

Hohmuth, M., Tews, H. & Stephens, S. G. (2002), Applying source-code verification to a microkernel — the VFiasco project, Technical Report TUD-FI02-03-März, TU Dresden.

Klein, G. & Tuch, H. (2004), Towards verified virtual memory in L4, *in* K. Slind, ed., 'TPHOLs Emerging Trends '04', Park City, Utah, USA.

L4Ka Team (2001), *L4 eX perimental Kernel Reference Manual Version X.2*, University of Karlsruhe. `http://l4ka.org/projects/version4/l4-x2.pdf`.

Morgan, C. (1990), *Programming from Specifications*, Prentice Hall.

Muchnick, S. (1997), *Advanced Compiler Design and Implementation*, Morgan Kaufmann Publishers.

Necula, G. C. (1997), Proof-carrying code, *in* 'Proc. POPL'97, 24th ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages', ACM Press, pp. 106–119.

Neumann, P. G., Boyer, R. S., Feiertag, R. J., Levitt, K. N. & Robinson, L. (1980), A provably secure operating system: The system, its applications, and proofs, Technical Report CSL-116, SRI International.

Nielson, F., Nielson, H. R. & Hankin, C. L. (1999), *Principles of Program Analysis*, Springer.

Nipkow, T., Paulson, L. & Wenzel, M. (2002), *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, Vol. 2283 of *LNCS*, Springer.

Norrish, M. (1998), C formalised in HOL, PhD thesis, Computer Laboratory, University of Cambridge.

Rashid, R., Julin, D., Orr, D., Sanzi, R., Baron, R., Forin, A., Golub, D. & Jones, M. (1989), Mach: A system software kernel, *in* 'Proceedings of the 34th Computer Society International Conference COMPCON 89'.

Shapiro, J., Doerrie, M. S., Northup, E., Sridhar, S. & Miller, M. (2004), Towards a verified, general-purpose operating system kernel, *in* G. Klein, ed., 'Proc. NICTA FM Workshop on OS Verification', Technical Report 0401005T-1, National ICT Australia, pp. 1–19.

Tuch, H. & Klein, G. (2004), Verifying the L4 virtual memory subsystem, *in* G. Klein, ed., 'Proc. NICTA FM Workshop on OS Verification', Technical Report 0401005T-1, National ICT Australia, pp. 73–97.

Turing, A. (1936-1937), 'On Computable Numbers, with an Application to the Entscheidungsproblem', *Proc. LMS, Series 2* **42**, 230–265.

Ver (2005*a*), 'VerifiCard project', `http://verificard.org`.

Ver (2005*b*), 'VeriSoft project', `http://www.verisoft.de`.

Walker, B. J., Kemmerer, R. A. & Popek, G. J. (1980), 'Specification and verification of the UCLA Unix security kernel', *Communications of the ACM* **23**(2), 118–131.