A Road to a Formally Verified General-Purpose Operating System

Martin Děcký

Department of Distributed and Dependable Systems
Faculty of Mathematics and Physics, Charles University
Malostranské náměstí 25, Prague 1, 118 00, Czech Republic
martin.decky@d3s.mff.cuni.cz

Abstract. Methods of formal description and verification represent a viable way for achieving fundamentally bug-free software. However, in reality only a small subset of the existing operating systems were ever formally verified, despite the fact that an operating system is a critical part of almost any other software system. This paper points out several key design choices which should make the formal verification of an operating system easier and presents a work-in-progress and initial experiences with formal verification of HelenOS, a state-of-the-art microkernel-based operating system, which, however, was not designed specifically with formal verification in mind, as this is mostly prohibitive due to time and budget constrains.

The contribution of this paper is the shift of focus from attempts to use a single "silver-bullet" formal verification method which would be able to verify everything to a combination of multiple formalisms and techniques to successfully cover various aspects of the operating system. A reliable and dependable operating system is the emerging property of the combination, thanks to the suitable architecture of the operating system.

1 Introduction

Operating systems (OSes for short) have a somewhat special position among all software. OSes are usually designed to run on bare hardware. This means that they do not require any special assumptions on the environment except the assumptions on the properties and behavior of hardware. In many cases it is perfectly valid to consider the hardware as *idealized hardware* (zero mathematical probability of failure, perfect compliance with the specifications, etc.). This means that it is solely the OS that defines the environment for other software.

OSes represent the lowest software layer and provide services to essentially all other software. Considering the principle of recursion, the properties of an OS form the assumptions for the upper layers of software. Thus the dependability of end-user and enterprise software systems is always limited by the dependability of the OS.

Finally, OSes are non-trivial software on their own. The way they are generally designed and programmed (spanning both the kernel and user mode, manipulating execution contexts and concurrency, handling critical hardware-related operations) represent significant and interesting challenges for software analysis.

These are probably the most important reasons that led to several research initiatives in the recent years which target the creation of a formally verified OSes from scratch (e.g. [14]). Methods of formal description and verification provide fundamentally better guarantees of desirable properties than non-exhaustive engineering methods such as testing.

However, 98 %[1] of the market share of general-purpose OSes is taken by Windows, Mac OS X and Linux. These systems were clearly not designed with formal verification in mind from the very beginning. The situation on the embedded, real-time and special-purpose OSes market is probably different, but it is unlikely that the segmentation of the desktop and server OSes market is going to change very rapidly in the near future.

The architecture of these major desktop and server OSes is monolithic, which makes any attempts to do formal verification on them extremely challenging due to the large state space. Fortunately we can observe that aspects of several novel approaches from the OS research from the late 1980s and early 1990s (microkernel design, user space file system and device drivers, etc.) are slowly emerging in these originally purely monolithic implementations.

In this paper we show how specific design choices can markedly improve the feasibility of verification of an OS, even if the particular OS was not designed specifically with formal verification in mind. These design choices can be gradually introduced (and in fact some of them have already been introduced) to mainstream general-purpose OSes.

Our approach is not based on using a single "silver-bullet" formalism, methodology or tool, but on combining various engineering, semi-formal and formal approaches. While the lesser formal approaches give lesser guarantees, they can complement the formal approaches on their boundaries and increase the coverage of the set of all hypothetical interesting properties of the system.

We also demonstrate work-in-progress case study of an general-purpose research OS that was not created specifically with formal verification in mind from the very beginning, but that was designed according to state-of-the-art OS principles.

Structure of the Paper. In Section 2 we introduce the design choices and our case study in more detail. In Section 3 we discuss our approach of the combination of methods and tools. In Section 4 we present a preliminary evaluation of our efforts and propose the imminent next steps to take. Finally, in Section 5 we present the conclusion of the paper.

[1] 98 % of client computers connected to the Internet as of January 2010 [13].

2 Operating Systems Design

Two very common schemes of OS design are *monolithic design* and *microkernel design*. Without going into much detail of any specific implementation, we can define the monolithic design as a preference to put numerous aspects of the core OS functionality into the kernel, while microkernel design is a preference to keep the kernel small, with just a minimal set of features.

The features which are missing from the kernel in the microkernel design are implemented in user space, usually by means of libraries, servers (e.g. processes/tasks) and/or software components.

2.1 HelenOS

HelenOS is a general-purpose research OS which is being developed at Charles University in Prague. The source code is available under the BSD open source license and can be freely downloaded from the project web site [11]. The authors of the code base are both from the academia and from the open source community (several contributors are employed as Solaris kernel developers and many are freelance IT professionals).

HelenOS uses a preemptive priority-feedback scheduler, it supports SMP hardware and it is designed to be highly portable. Currently it runs on 7 distinct hardware architectures, including the most common IA-32, x86-64 (AMD64), IA-64, SPARC v9 and PowerPC. It also runs on ARMv7 and MIPS, but currently only in simulators and not on physical hardware.

Although HelenOS is still far from being an everyday replacement for Linux or Windows due to the lack of end-user applications (whose development is extremely time-consuming, but unfortunately of no scientific value), the essential foundations such as file system support and TCP/IP networking are already in place.

HelenOS does not currently target embedded devices (although the ARMv7 port can be very easily modified to run on various embedded boards) and does not implement real-time features. Many development projects such as task snapshoting and migration, support for MMU-less platforms and performance monitoring are currently underway.

HelenOS can be briefly described as microkernel multiserver OS. However, the actual design guiding principles of the HelenOS are more elaborate:

Microkernel principle Every functionality of the OS that does not have to be necessary implemented in the kernel should be implemented in user space. This implies that subsystems such as the file system, device drivers (except those which are essential for the basic kernel functionality), naming and trading services, networking, human interface and similar features should be implemented in user space.

Full-fledged principle Features which need to be placed in kernel should be implemented by full-fledged algorithms and data structures. In contrast to

several other microkernel OSes, where the authors have deliberately chosen the most simplistic approach (static memory allocation, naïve algorithms, simple data structures), HelenOS microkernel tries to use the most advanced and suitable means available. It contains features such as AVL and B+ trees, hashing tables, SLAB memory allocator, multiple in-kernel synchronization primitives, fine-grained locking and so on.

Multiserver principle Subsystems in user space should be decomposed with the smallest reasonable granularity. Each unit of decomposition should be encapsulated in a separate task. The tasks represent software components with isolated address spaces. From the design point of view the kernel can be seen as a separate software component, too.

Split of mechanism and policy The kernel should only provide low-level mechanisms, while the high-level policies which are built upon these mechanisms should be defined in user space. This also implies that the policies should be easily replaceable while keeping the low-level mechanisms intact.

Encapsulation principle The communication between the tasks/components should be implemented only via a set of well-defined interfaces. In the user-to-user case the preferred communication mechanism is HelenOS IPC, which provides reasonable mix of abstraction and performance (RPC-like primitives combined with implicit memory sharing for large data transfers). In case of synchronous user-to-kernel communication the usual syscalls are used. HelenOS IPC is used again for asynchronous kernel-to-user communication.

Portability principle The design and implementation should always maintain a high level of platform neutrality and portability. Platform-specific code in the kernel, core libraries and tasks implementing device drivers should be clearly separated from the generic code (either by component decomposition or at least by internal compile-time APIs).

In Section 3 we argue that several of these design principles significantly improve the feasibility of formal verification of the entire system. On the other hand, other design principles induce new interesting challenges for formal description and verification.

The run-time architecture of HelenOS is inherently dynamic. The bindings between the components are not created at compile-time, but during bootstrap and can be modified to a large degree also during normal operation mode of the system (via human interaction and external events).

The design of the ubiquitous HelenOS IPC mechanism and the associated threading model present the possibility to significantly reduce the size of the state space which needs to be explored by formal verification tools, but at the same time it is quite hard to express these constrains in many formalisms. The IPC is inherently asynchronous with constant message buffers in the kernel and dynamic buffers in user space. It uses the notions of uni-directional bindings, mandatory pairing of requests and replies, binding establishment and abolishment handshakes, memory sharing and fast message forwarding.

For easier management of the asynchronous messages and the possibility to process multiple messages from different peers without the usual kernel threading

overhead, the core user space library manages the execution flow by so-called *fibrils*. A fibril is a user-space-managed thread with cooperative scheduling. A different fibril is scheduled every time the current fibril is about to be blocked while sending out IPC requests (because the kernel buffers of the addressee are full) or while waiting on an IPC reply. This allows different execution flows within the same thread to process multiple requests and replies. To safeguard proper sequencing of IPC messages and provide synchronization, special fibril-aware synchronization primitives (mutexes, condition variables, etc.) are available.

Because of the cooperative nature of fibrils, they might cause severe performance under-utilization in SMP configurations and system-wide bottlenecks. As multicore processors are more and more common nowadays, that would be a substantial design flaw. Therefore the fibrils can be also freely (and to some degree even automatically) combined with the usual kernel threads, which provide preemptive scheduling and true parallelism on SMP machines. Needless to say, this combination is also a grand challenge for the formal reasoning.

Incidentally, the *full-fledged principle* causes that the size of the HelenOS microkernel is considerably larger compared to other "scrupulous" microkernel implementations. The average footprint of the kernel on IA-32 ranges from 569 KiB when all logging messages, asserts, symbol resolution and the debugging kernel console are compiled in, down to 198 KiB for a non-debugging production build. But we do not believe that the raw size of the microkernel is a relevant quality criterion per se, without taking the actual feature set into account.

To sum up, the choice of HelenOS as our case study is based on the fact that it was not designed in advance with formal verification in mind (some of the design principles might be beneficial, but others might be disadvantageous), but the design of HelenOS is also non-trivial and not obsolete.

2.2 The C Programming Language

A large majority of OSes is coded in the C programming language (HelenOS is no exception to this). The choice of C in the case of kernel is usually well-motivated, since the C language was designed specifically for implementing system software [10]: It is reasonably low-level in the sense that it allows to access the memory and other hardware resources with similar effectiveness as from assembler; It also requires almost no run-time support and it exports many features of the von Neumann hardware architecture to the programmer in a very straightforward, but still relatively portable way.

However, what is the biggest advantage of C in terms of run-time performance is also the biggest weakness for formal reasoning. The permissive memory access model of C, the lack of any reference safety enforcement, the weak type system and generally little semantic information in the code – all these properties do not allow to make many general assumptions about the code.

Programming languages which target controlled environments such as Java and C♯ are generally easier for formal reasoning because they provide a well-known set of primitives and language constructs for object ownership, threading

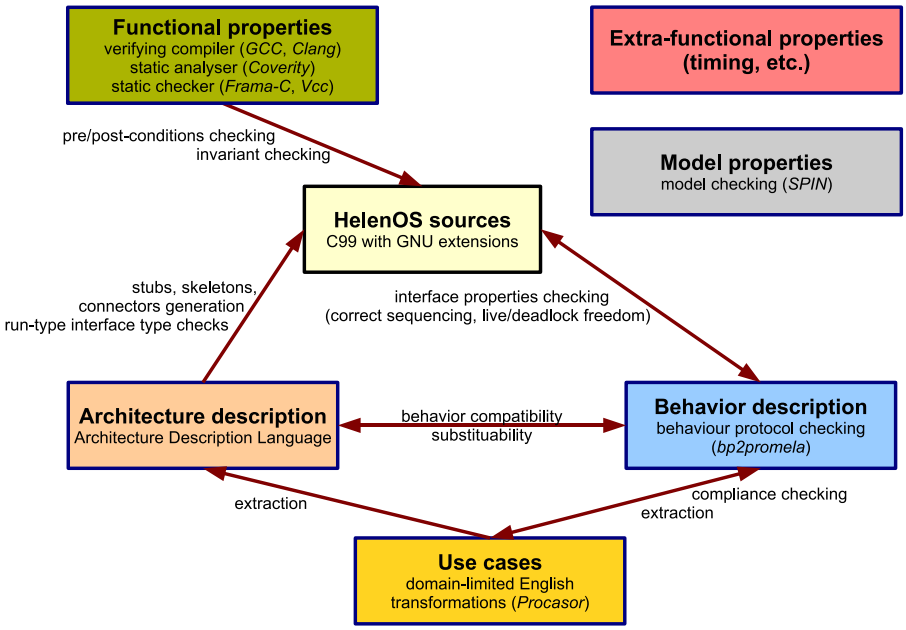

Fig. 1. Overview of the formalisms and tools proposed.

and synchronization. Many non-imperative programming languages can be even considered to be a form of "executable specification" and thus very suitable for formal reasoning. In C, almost everything is left to the programmer who is free to set the rules.

The reasons for frequent use of C in the user space of many established OSes (and HelenOS) is probably much more questionable. In the case of HelenOS, except for the core libraries and tasks (such as device drivers), C might be easily replaced by any high-level and perhaps even non-imperative programming language. The reasons for using C in this context are mostly historical.

However, as we have stated in Section 1, the way general-purpose OSes are implemented changes only slowly and therefore any propositions which require radical modification of the existing code base before committing to the formal verification are not realistic.

3 Analysis

In this section, we analyze the properties we would like to check in a general-purpose OS. Each set of properties usually requires a specific verification method, tool or toolchain.

Our approach will be mostly bottom-up, or, in other words, from the lower levels of abstraction to the higher levels of abstraction. If the verification fails on

a lower level, it usually does not make much sense to continue with the higher levels of abstraction until the issues are tackled. A structured overview of the formalisms, methods and tools can be seen on Figure 1.

Some of the proposed methods cannot be called "formal methods" in the rigorous understanding of the term. However, even methods which are based on semi-formal reasoning and non-exhaustive testing provide some limited guarantees in their specific context. A valued property of the formal methods is to preserve these limited guarantees even on the higher levels of abstraction, thus allowing the semi-formal methods to complement the big picture where the formal methods do not provide any feasible verification so far. This increases the coverage of the set of all hypothetical interesting properties of the system (although it is probably impossible to formally define this entire set).

Please note that the titles of the following sections do not follow any particular established taxonomy. We have simply chosen the names to be intuitively descriptive.

3.1 C Language Compiler and Continuous Integration Tool

The initial levels of abstraction do not go far from the C source code and common engineering approaches. First, we would certainly like to know whether our code base is compliant with the programming language specification and passes only the basic semantic checks (proper number and types of arguments passed to functions, etc.). It is perhaps not very surprising that these decisions can be made by any plain C compiler. However, with the current implementation of HelenOS even this is not quite trivial.

Besides the requirement to support 7 hardware platforms, the system's compile-time configuration can be also affected by approximately 65 configuration options, most of which are booleans, the rest are enumerated types.

These configuration options are bound by logical propositions in conjunctive or disjunctive normal forms and while some options are freely configurable, the value of others gets inferred by the build system of HelenOS. The overall number of distinct configurations in which HelenOS can be compiled is at least one order of magnitude larger than the plain number of supported hardware platforms.

Various configuration options affect conditional compilation and linking. The programmers are used to make sure that the source code compiles and links fine with respect to the most common and obvious configurations, but the unforeseen interaction of the less common configuration options might cause linking or even compilation errors.

A straightforward solution is to generate all distinct configurations, starting from the open variables and inferring the others. This can be part of the continuous integration process which would try to compile and link the sources in all distinct configurations.

If we want to be really pedantic, we should also make sure that we run all higher level verification methods on all configurations generated by this step.

That would certainly require to multiply the time required by the verification methods at least by the number of the distinct configurations. Constraining the set of configurations to just the most representative ones is perhaps a reasonable compromise to make the verification realistic.

3.2 Regression and Unit Tests

Running regression and unit tests which are part of HelenOS code base in the continuous integration process is fairly easy. The only complication lies in the technicalities: We need to setup an automated network of physical machines and simulators which can run the appropriate compilation outputs for the specific platforms. We need to be able to reboot them remotely and distribute the boot images to them. And last but not least, we need to be able to gather the results from them.

Testing is always non-exhaustive, thus the guarantees provided by tests are strictly limited to the use cases and contexts which are being explicitly tested. However, it is arguably easier to express many common use cases in the primary programming language than in some different formalism. As we follow the bottom-up approach, filtering out the most obvious bugs by testing can save us a lot of valuable time which would be otherwise waisted by a futile verification by more formal (and more time-consuming) methods.

3.3 Instrumentation

Instrumentation tools for detecting memory leaks, performance bottlenecks and soft-deadlocks are also not usually considered to be formal verification tools (since it is hard to define exact formal properties which are being verified by the non-exhaustive nature of these tools). They are also rarely utilized on regular basis as part of the continuous integration process. But again, it might be helpful to just mention them in the big picture.

If some regression or unit tests fail, they sometimes do not give sufficient information to tell immediately what is the root cause of the issue. In that case running the faulting tests on manually or automatically instrumented executable code might provide more data and point more directly to the actual bug.

3.4 Verifying C Language Compiler

C language compilers are traditionally also not considered to be formal verification tools. Many people just say that C compilers are good at generating executable code, but do not care much about the semantics of the source code (on the other hand, formal verification tools usually do not generate any executable code at all). However, with recent development in the compiler domain, the old paradigms are shifting.

As the optimization passes and general maturity of the compilers improve over time, the compilers try to extract and use more and more semantic information from the source code. The C language is quite poor on explicit semantic

information, but the verifying compilers try to rely on vendor-specific language extensions and on the fact that some semantic information can be added to the source code without changing the resulting executable code.

The checks done by the verifying compilers cannot result in fatal errors in the usual cases (they are just warnings). Firstly, the compilers still need to successfully compile a well-formed C source code compliant to some older standard (e.g. C89) even when it is not up with the current quality expectations. Old legacy source code should still pass the compilation as it did decades ago.

Secondly, the checks run by the verifying compilers are usually not based on abstract interpretation. They are mostly realized as abstract syntax tree transformations much in the line with the supporting routines of the compilation process (data and control flow graph analysis, dead code elimination, register allocation, etc.) and the evaluation function is basically the matching of antipatterns of common programming bugs.

The checks are usually conservative. The verifying compilers identify code constructs which are suspicious, which might arise out of programmer's bad intuition and so on, but even these code snippets cannot be tagged as definitive bugs (since the programmer can be simply in a position where he/she really wants to do something very strange, but nevertheless legitimate). It is upon the programmer to examine the root cause of the compiler warning, tell whether it is really a bug or just a false positive and fix the issue by either amending some additional semantic information (e.g. adding an explicit typecast or a vendor-specific language extension) or rewriting the code.

Although this level of abstraction is coarse-grained and conservative, it can be called semi-formal, since the properties which are being verified can be actually defined quite exactly and they are reasonably general. They do not deal with single traces of methods, runs and use cases like tests, but they deal with all possible contexts in which the code can run.

3.5 Static Analyzer

Static analyzers try to go deeper than verifying compilers. Besides detecting common antipatterns of bugs, they also use techniques such as abstract interpretation to check for more complex properties.

Most commercial static analyzers come with a predefined set of properties which cannot be easily changed. They are coupled with the commonly used semantics of the environment and generate domain-specific models of the software based not only on the syntax of the source code, but also based on the assumptions derived from the memory access model, allocation and deallocation rules, tracking of references and tracking of concurrency locks.

The biggest advantage of static analyzers is that they can be easily included in the development or continuous integration process as an additional automated step, very similar to the verifying compilers. No manual definition of code-specific properties is needed and false positives can be relatively easily eliminated by amending some explicit additional information to the source code within the boundaries of the programming language.

The authors of static analyzers claim large quantities of bugs detected or prevented [1], but static analyzers are still relatively limited by the kind of bugs they are designed to detect. They are usually good at pointing out common issues with security implications (specific types of buffer and stack overruns, usage of well-known functions in an unsafe way, clear cases of forgotten deallocation of resources and release of locks, etc.). Unfortunately, many static analyzers only analyze a single-threaded control flow and are thus unable to detect concurrency issues such as deadlocks.

3.6 Static Verifier

There is one key difference between a static analyzer and a static verifier: Static verifiers allow the user to specify one's own properties, in terms of preconditions, postconditions and invariants in the code. Many static verifiers also target true multithreaded usage patterns and have the capability to check proper locking order, hand-over-hand locking and even liveliness.

In the context of an OS kernel and core libraries two kinds of properties are common:

Consistency constrains These properties define the correct way how data is supposed to be manipulated by some related set of subroutines. Checking for these properties ensures that data structures and internal states will not get corrupt due to bugs in the functions and methods which are designed to manipulate them.

Interface enforcements These properties define the correct semantics by which a set of subroutines should be used by the rest of the code. Checking for these properties ensures that some API is always used by the rest of the code in a specified way and all reported exceptions are handled by the client code.

3.7 Model Checker

On the first sight it does not seem to be reasonable to consider general model checkers as relevant independent tools for formal verification of an existing OS. While many different tools use model checkers as their backends, verifying a complete model of the entire system created by hand seems to be infeasible both in the sense of time required for the model creation and resources required by the checker to finish the exhaustive traversal of the model's state space.

Nevertheless, model checkers on their own can still serve a good job verifying abstract properties of key algorithms without dealing with the technical details of the implementation. Various properties of synchronization algorithms, data structures and communication protocols can be verified in the most generic conditions by model checkers, answering the question whether they are designed properly in theory.

If the implementation of these algorithms and protocols do not behave correctly, we can be sure that the root cause is in the non-compliance between the design and implementation and not a fundamental flaw of the design itself.

3.8 Architecture and Behavior Checker

All previously mentioned verification methods were targeting internal properties of the OS components. If we are moving to a higher-level abstraction in order to specify correct interaction of the encapsulated components in terms of interface compatibility and communication, we can utilize *Behavior Protocols* [2] or some other formalism describing correct interaction between software components.

To gain the knowledge about the architecture of the whole OS in terms of software component composition and bindings, we can use *Architecture Description Language* [12] as the specification of the architecture of the system. This language has the possibility to capture interface types (with method signatures), primitive components (in terms of provided and required interfaces), composite components (an architectural compositions of primitive components) and the bindings between the respective interfaces of the components.

It is extremely important to define the right role of the behavior and architecture description. A flawed approach would be to reverse-engineer this description from the source code (either manually or via some sophisticated tool) and then verify the compliance between the description and the implementation. However, different directions can give more interesting results:

Description as specification Behavior and architecture description created independently on the source code serves the role of specification. This has the following primary goals of formal verification:

- **Horizontal compliance** Also called *compatibility*. The goal is to check whether the specifications of components that are bound together are semantically compatible. All required interfaces need to be bound to provided interfaces and the communication between the components cannot lead to *no activity* (a deadlock), *bad activity* (a livelock) or other communication and synchronization errors.
- **Vertical compliance** Also called *substituability*. The goal is to check whether it is possible to replace a set of primitive components that are nested inside a composite component by the composite component itself. In other words, this compliance can answer the question whether the architecture description of the system is sound with respect to the hierarchical composition of the components.
- **Specification and implementation compliance** Using various means of generating artificial environments for an isolated component a checker is able to partially answer the question whether the implementation of the component is an instantiation of the component specification.

Description as abstraction Generating the behavior and architecture description from the source code by means of abstract interpretation can serve the purpose of verifying various properties of the implementation such as invariants, preconditions and postconditions. This is similar to static verification, but on the level of component interfaces.

Unfortunately, most of the behavior and architecture formalisms are static, which is not quite suitable for the dynamic nature of most OSes. This limitation

can be circumvented by considering a relevant snapshot of the dynamic run-time architecture. This snapshot fixed in time is equivalent to a statically defined architecture.

The key features of software systems with respect to behavior and architecture checkers are the granularity of the individual primitive components, the level of isolation and complexity of the communication mechanism between them. Large monolithic OSes created in semantic-poor C present a severe challenge because the isolation of the individual components is vague and the communication between them is basically unlimited (function calls, shared resources, etc.).

OSes with explicit component architecture and fine-grained components (such as microkernel multiserver systems) make the feasibility of the verification much easier, since the degrees of freedom (and thus the state space) is limited.

Horizontal and vertical compliance checking can be done exhaustively. This is a fundamental property which allows the reasoning about the dependability of the entire component-based OS. Assuming that the lower-level verification methods (described in Sections 3.1 to 3.7) prove some specific properties of the primitive components, we can be sure that the composition of the primitive components into composite components and ultimately into the whole OS does not break these properties.

The feasibility of many lower-level verification methods from Sections 3.1 to 3.7 depends largely on the size and complexity of the code under verification. If the entire OS is decomposed into primitive components with a fine granularity, it is more likely that the individual primitive components can be verified against a large number of properties. Thanks to the recursive component composition we can then be sure that these properties also hold for the entire system.

The compliance between the behavior specification and the actual behavior of the implementation is, unfortunately, the missing link in the chain. This compliance cannot be easily verified in an exhaustive manner. If there is a discrepancy between the specified and the actual behavior of the components, we cannot conclude anything about the properties holding in the entire system.

However, there is one way how to improve the situation: *code generation*. If we generate implementation from the specification, the compliance between them is axiomatic. If we are able to generate enough code from the specification to run into the hand-written "business code" where we check for the lower-level properties, the conclusions about the component composition are going to hold.

3.9 Behavior Description Generator

To conclude our path towards higher abstractions we can utilize tools that can generate the behavior descriptions from *textual use cases* written in a domain-constrained English. These generated artifacts can be then compared (e.g. via vertical compliance checking) with the formal specification. Although the comparison might not provide clean-cut results, it can still be helpful to confront the more-or-less informal user expectations on the system with the exact formal description.

3.10 Summary

So far, we have proposed a compact combination of engineering, semi-formal and formal methods which start at the level of C programming language, offer the possibility to check for the presence of various common antipatterns, check for generic algorithm-related properties, consistency constrains, interface enforcements and conclude with a framework to make these properties hold even in the case of a large OS composed from many components of compliant behavior.

We do not claim that there are no missing pieces in the big picture or that the semi-formal verifications might provide more guarantees in this setup. However, state-of-the-art OS design guidelines can push further the boundaries of practical feasibility of the presented methods. The limited guarantees of the low-level methods hold even in the composition and the high-level formal methods do not have to deal with unlimited degrees of freedom of the primitive component implementation.

We have spoken only about the functional properties. In general, we cannot apply the same formalisms and methods on extra-functional properties (e.g. timing properties, performance properties, etc.). And although it probably does make a good sense to reason about component composition for the extra-functional properties, the exact relation might be different compared to the functional properties.

The extra-functional properties need to be tackled by our future work.

4 Evaluation

This section copies the structure of the previous Section 3 and adds HelenOS-specific evaluation of the the proposed formalisms and tools. As this is still largely a work-in-progress, in many cases just the initial observations can be made.

The choice of the specific methods, tools and formalisms in this initial phase is mostly motivated by their perceived commonality and author's claims about fitness for the given purpose. An important part of further evaluation would certainly be to compare multiple particular approaches, tools and formalisms to find the optimal combination.

4.1 Verifying C Language Compiler and Continuous Integration Tool

The primary C compiler used by HelenOS is *GNU GCC 4.4.3* (all platforms) [3] and *Clang 2.6.0* (IA-32) [4]. We have taken some effort to support also *ICC* and *Sun Studio* C compilers, but the compatibility with these compilers in not guaranteed.

The whole code base is compiled with the `-Wall` and `-Wextra` compilation options. These options turn on most of the verification checks of the compilers. The compilers trip on common bug antipatterns such as implicit typecasting

of pointer types, comparison of signed and unsigned integer values (danger of unchecked overflows), the usage of uninitialized variables, the presence of unused local variables, NULL-pointer dereferencing (determined by conservative local control flow analysis), functions with non-void return typed that do not return any value and so on. We treat all compilation warnings as fatal errors (`-Werror`), thus the code base must pass without any warnings.

We also turn on several more specific and strict checks. These checks helped to discover several latent bugs in the source code:

`-Wfloat-equal` Check for exact equality comparison between floating point values. The usage of equal comparator on floats is usually misguided due to the inherent computational errors of floats.

`-Wcast-align` Check for code which casts pointers to a type with a stricter alignment requirement. On many RISC-based platforms this can cause run-time unaligned access exceptions.

`-Wconversion` Check for code where the implicit type conversion (e.g. from float to integer, between signed and unsigned integers or between integers with different number of bits) can cause the actual value to change.

To enhance the semantic information in the source code, we use GCC-specific language extensions to annotate some particular kernel and core library routines:

`__attribute__((noreturn))` Functions marked in this way never finish from the point of view of the current sequential execution flow. The most common case are the routines which restore previously saved execution context.

`__attribute__((returns_twice))` Functions marked in this way may return multiple times from the point of view of the current sequential execution flow. This is the case of routines which save the current execution context (first the function returns as usual, but the function can eventually "return again" when the context is being restored).

The use of these extensions has pointed out to several hard-to-debug bugs on the IA-64 platform.

The automated continuous integration building system is currently work-in-progress. Thus, we do not test all possible configurations of HelenOS with each changeset yet. Currently only a representative set of 14 configurations (at least one for each supported platform) is tested by hand by the developers before committing any non-trivial changeset.

From occasional tests of other configurations by hand and the frequency of compilation, linkage and even run-time problems we conclude that the automated testing of all feasible configurations will be very beneficial.

4.2 Regression and Unit Tests

As already stated in the previous section, the continuous integration building system has not been finished yet. Therefore regression and unit tests are executed occasionally by hand, which is time consuming and prone to human omissions. An automated approach is definitively going to be very helpful.

4.3 Instrumentation

We are in the process of implementing our own code instrumentation framework which is motivated mainly by the need to support MMU-less architectures in the future. But this framework might be also very helpful in detecting memory and generic resource leaks. We have not tried *Valgrind* [17] or any similar existing tool because of the estimated complexity to adopt it for the usage in HelenOS.

4.4 Static Analyzer

The integration of various static analyzers into the HelenOS continuous integration process is underway. For the initial evaluation we have used *Stanse* [16] and *Clang Analyzer* [5]. Both of them showed to be moderately helpful to point out instances of unreachable dead code, use of language constructs which have ambiguous semantics in C and one case of possible NULL-pointer dereference.

The open framework of Clang seems to be very promising for implementing domain-specific checks (and at the same time it is also a very promising compiler framework). Our mid-term goal is to incorporate some of the features of Stanse and VCC (see Section 4.5) into Clang Analyzer.

HelenOS was also scanned by *Coverity* [7] in 2006 when no errors were detected. However, since that time the code base has not been analyzed by Coverity.

4.5 Static Verifier

We have started to extend the source code of HelenOS kernel with annotations understood by *Frama-C* [9] and *VCC* [18]. Initially we have targeted simple kernel data structures (doubly-linked circular lists) and basic locking operations. Currently we are evaluating the initial experiences and we are trying to identify the most suitable methodology, but we expect quite promising results.

As the VCC is based on the Microsoft C++ Compiler, which does not support many GCC extensions, we have been faced with the requirement to preprocess the source code to be syntactically accepted by VCC. This turned out to be feasible.

4.6 Model Checker

We are in the process of creating models of kernel wait queues (basic HelenOS kernel synchronization primitive) and futexes (basic user space thread synchronization primitive) using *Promela* and verify several formal properties (deadlock freedom, fairness) in *Spin* [15]. As both the Promela language and the Spin model checker are mature and commonly used tools for such purposes, we expect no major problems with this approach. Because both synchronization primitives are relatively complex, utilizing a model checker should provide a much more trustworthy proof of the required properties than "paper and pencil".

The initial choice of Spin is motivated by its suitability to model threads, their interaction and verify properties related to race conditions and deadlocks (which is the common sources of OS-related bugs). Other modeling formalisms might be more suitable for different goals.

4.7 Architecture and Behavior Checker

We have created an architecture description in ADL language derived from *SOFA ADL* [12] for the majority of the HelenOS components and created the Behavior Protocol specification of these components. Both descriptions were created independently, not by reverse-engineering the existing source code. The architecture is a snapshot of the dynamic architecture just after a successful bootstrap of HelenOS.

Both the architecture and behavior description is readily available as part of the source code repository of HelenOS, including tools which can preprocess the Behavior Protocols according to the architecture description and create an output suitable for *bp2promela* checker [2].

As the resulting complexity of the description is larger than any of the previously published case studies on Behavior Protocols (compare to [6]), our current work-in-progress is to optimize and fine-tune the bp2promela checker to process the input.

We have not started to generate code from the architecture description so far because of time constrains. However, we believe that this is a very promising way to go and provide reasonable guarantees about the compliance between the specification and the implementation.

4.8 Behavior Description Generator

We have not tackled the issue of behavior description generation yet, although tools such as *Procasor* [8] are readily available. We do not consider it our priority at this time.

5 Conclusion

In this paper we propose a complex combination of various verification methods and tools to achieve the verification of an entire general-purpose operating system. The proposed approach generally follows a bottom-up route, starting with low-level checks using state-of-the-art verifying C language compilers, following by static analyzers and static verifiers. In specific contexts regression and unit tests, code instrumentation and model checkers for the sake of verification of key algorithms are utilized.

Thanks to the properties of state-of-the-art microkernel multiserver operating system design (e.g. software component encapsulation and composition, fine-grained isolated components), we demonstrate that it should be feasible to successfully verify larger and more complex operating systems than in the case of monolithic designs. We use formal component architecture and behavior description for the closure. The final goal – a formally verified operating system – is the emerging property of the combination of the various methods.

The contribution of this paper is the shift of focus from attempts to use a single "silver-bullet" method for formal verification of an operating system to

a combination of multiple methods supported by a suitable architecture of the operating system. The main benefit is a much larger coverage of the set of all hypothetical properties.

We also argue that the approach should be suitable for the mainstream general-purpose operating systems in the near future, because they are gradually incorporating more microkernel-based features and fine-grained software components.

Although the evaluation of the proposed approach on HelenOS is still work-in-progress, the preliminary results and estimates are promising.

Acknowledgments. The author would like to express his gratitude to all contributors of the HelenOS project. Without their vision and dedication the work on this paper would be almost impossible

This work was partially supported by the Ministry of Education of the Czech Republic (grant MSM0021620838).

References

1. Bessey A., Block K., Chelf B., Chou A., Fulton B., Hallem S., Henri-Gros C., Kamsky A., McPeak S., Engler D.: *A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World*, Communications of the ACM, Vol. 53 No. 2, pp 66-75, 2010
2. Kofron J.: *Checking Software Component Behavior Using Behavior Protocols and Spin*, in proceedings of Applied Computing 2007, Seoul, Korea, pp. 1513-1517, 2007
3. GCC, the GNU Compiler Collection, `http://gcc.gnu.org/`
4. Clang: a C language family frontend for LLVM, `http://clang.llvm.org/`
5. Clang Static Analyzer, `http://clang-analyzer.llvm.org/`
6. Bulej L., Bures T., Coupaye T., Decky M., Jezek P., Parizek P., Plasil F., Poch T., Rivierre N., Sery O., Tuma P.: *CoCoME in Fractal*, chapter in The Common Component Modeling Example: Comparing Software Component Models, Springer-Verlag, LNCS 5153, 2008
7. Coverity, `http://scan.coverity.com/`
8. Mencl V.: *Deriving Behavior Specifications from Textual Use Cases*, in Proceedings of Workshop on Intelligent Technologies for Software Engineering (WITSE04, Sep 21, 2004, part of ASE 2004), Linz, Austria, pp. 331 - 341, Oesterreichische Computer Gesellschaft, 2004
9. Frama-C, `http://frama-c.cea.fr/`
10. Lawlis P. K: *Guidelines for Choosing a Computer Language: Support for the Visionary Organization*, Ada Information Clearinghouse, `http://archive.adaic.com/docs/reports/lawlis/k.htm`, 1998
11. HelenOS Project, `http://www.helenos.org/`
12. Oplustil T.: *Inheritance in Architecture Description Languages*, reviewed section of Proceedings of the Week of Doctoral Students 2003 conference (WDS 2003), Charles University, Prague, Czech Republic, 2003, pp. 124 - 131, 2003
13. Operating System Market Share, `http://marketshare.hitslink.com/report-.aspx?qprid=8&qptimeframe=M&qpsp=132`, retrieved on 2010-02-28

14. Klein G., Elphinstone K., Heiser G., Andronick J., Cock D., Derrin P., Elkaduwe D., Engelhardt K., Kolanski R., Norrish M., Sewell T., Tuch H., Winwood S.: *seL4: Formal verification of an OS kernel*, Proceedings of the 22nd ACM Symposium on Operating Systems Principles, Big Sky, MT, USA, 2009
15. Spin, `http://spinroot.com/`
16. Stanse: Static Analysis Framework for C code, `http://stanse.fi.muni.cz/`
17. Valgrind, `http://valgrind.org/`
18. VCC, `http://vcc.codeplex.com/`