

Proving the correctness of client/server software

EYAD ALKASSAR, SEBASTIAN BOGAN and
WOLFGANG J PAUL

Computer Science Department, Saarland University, 66123 Saarbrücken, Germany
e-mail: {eyad, sebastian, wjp}@wjpserver.cs.uni-sb.de

Abstract. Remote procedure calls (RPCs) lie at the heart of any client/server software. Thus, formal specification and verification of RPC mechanisms is a prerequisite for the verification of any such software. In this paper, we present a mathematical specification of an RPC mechanism and we outline how to prove the correctness of an implementation — say written in C — of this mechanism at the code level. We define a formal model of user processes running concurrently under a simple operating system, which provides inter-process communication and portmapper system calls. A simple theory of non-interference permits us to use conventional sequential program analysis between system calls (within the concurrent model). An RPC mechanism is specified and the correctness proof for server implementations, using this mechanism, is outlined. To the best of our knowledge this is the first treatment of the correctness of an entire RPC mechanism at the code level.

Keywords. Software verification; remote procedure calling; operating system correctness; non-interference; Isabelle/HOL; Verisoft.

1. Introduction

1.1 *Background*

Contrary to common perception, the industrial computer systems that we are using in daily life are incredibly well engineered and incredibly well understood by the engineers who construct them. In spite of their overwhelming complexity they function as expected — or even as specified — for trillions of operations in a row. Surprises do occur, but relatively speaking they are incredibly rare. We notice them only because of the truly unimaginable speed of the computer systems: at 2 billion operations per second it only takes less than 10 minutes to perform a trillion operations. In contrast, a chair on which people sat down only a modest billion times, say 100 times per day, would be $10^9 / (100 \cdot 365) = 29397$ years old and would be considered to be practically indestructible.

The existence of very rare faults in our computer systems demonstrates, that our understanding of them is in the end only partial: very good but not perfect. This is more troublesome than it seems at first sight, because the statement, that a computer system works according to its specification is a mathematical statement; and our understanding of it, i.e. the explanation why the system works, can — at least in principle — be translated into a mathematical proof

of this very statement. But proofs are not considered partially true or false. A mathematical proof, in which most of its lines are true and only a single line is wrong, is considered false. Just like a husband who is true most of the time is considered untrue and a person who speaks the truth most of the time is considered a liar. From this puristic point of view we don't really understand our computer systems yet. And since a fault every few trillions of operations can lead to heavy financial losses or even to loss of life the puristic view is — at least for critical systems — the relevant one.

Hence it is highly desirable to proceed from 'very well understood' and 'functioning most of the time' to 'perfectly understood' and 'always functioning until the device breaks physically'. Present license agreements for software tend to include a statement, that it is impossible to construct bug free computer systems. We, the authors of this article, even admit that we have recently and repeatedly clicked buttons signaling our agreement to such license agreements, but we have done this against our own better knowledge. It is true that, except for very simple systems, one cannot test the absence of bugs by running them on all possible inputs. But there is an alternative way for literally measuring the absence of bugs: cast your explanation that the system works properly in a mathematical correctness proof, formulate this proof in the formal language of some computer aided verification (CAV) system, and then let the CAV system check that the proof is complete. This check does not require to run the system to be proven correct on any input. The answer to the question what a proof is, is very similar to the answer of the question what a JAVA or C program is. Checking whether a given text (proof or program) conforms to such a definition can be done automatically, like the syntax check which is routinely performed by any compiler. CAV system like compilers can signal the place where the syntax check failed. At this place we can fix the proof — or the system or both — and then try again to check for absence of bugs. We can iterate this until all bugs in the program and in the correctness proof are gone.

This method for constructing bug free systems is called 'formal verification'. In research laboratories it was demonstrated to work for very small systems as early as back in 1989 (Bevier *et al* 1989), for microprocessors with the complexity of high end embedded controllers in the 1990s and 2003 (Sawada & Hunt 1998, Beyer *et al* 2005), for optimizing compilers in 2006 (The VERIFIX Consortium 2000), for email clients and signature software in (Langenstein *et al* 2007a) and for small operating system kernels of industrial complexity in 2007 (Elphinstone *et al* 2007, Alkassar *et al* 2008b, In der Rieden & Tsyban 2008). The list could be extended. Already in the 90's industry used this technology to verify components of industrial systems (Moore 2003); in particular floating point units of standard processors have been formally verified to a considerable extent (understandably not all details are public) (Moore *et al* 1998, Jacobi *et al* 2005). The formal verification of the processor core of a future high end controller of Infineon is announced in (Bormann *et al* 2007). Current efforts to verify small industrial operating system kernels with 5000 to 10 000 lines of code (LOC) running on Power-PC's are reported at NICTA (Elphinstone *et al* 2007) and the Verisoft-XT project (The Verisoft Consortium 2003). Also in the Verisoft-XT project the formal verification of the Microsoft Hypervisor is attempted. This hypervisor has roughly 50 000 LOC and runs on high end processors from Intel or AMD. Thus formally verified state of the art industrial processors are a reality, and current joint research projects between industry and academia are aiming at the formal verification of entire industrial kernels and hypervisors. How much new insight is — or can possibly be — gained from this research? The opinions on this question could hardly diverge more.

The argument toward 'uninteresting' goes like this: *all* basic problems concerning the construction of kernels, hypervisors, operating systems etc. are well understood. Abstract versions

of the basic problems have been studied and continue to be studied in the open literature since decades. It is industry's job to combine the known solutions of these problems into systems, and industry is indeed very good at this (see the very beginning of this introduction). In order to construct an overall correctness proof one only has to combine the correctness arguments for the solutions of the basic problems into a single proof and enter it into a CAV system. This is very hard work, but it is mostly tedious and not exciting. It is not surprising at all that this works. This tends to be the 'distant view', i.e. the standpoint of researchers who are not involved in the formal verification of entire systems, presently — despite the name — the vast majority in the field of formal verification.

Occasionally however, just as one gets closer to something, things change from 'straight forward' to 'more exciting than one gambled for'. This is the case for the typical day trip into the mountains with mortal outcome. It is also presently the case for the formal verification of industrial software. This has several reasons which are much more easily explained than remedied. We illustrate these reasons both with hypothetical examples from mathematics and recent examples from system verification projects. Many more examples follow in the remainder of this paper.

- Since the introduction of the abstractions commonly studied in academia, industrial applications have progressed to more complex and more general situations. To illustrate this with an example from mathematics, imagine that industry would compute with real numbers while we would only have a complete mathematical foundation for rational numbers. In computer science until very recently, compiler correctness proofs were carried out with respect to big step semantics. But programs in real systems are interleaved; the natural way to describe their behaviour is small steps semantics. This requires additional arguments for compiler correctness proofs (Leinenbach *et al* 2005a).
- For many important models/abstractions it is hard to find soundness proofs. Imagine we would use the field axioms when we compute with real numbers, and that we would have a nice theory of fields, while in the mathematical literature we would not find a proof that the field axioms hold for the real numbers. In computer science there is an old and nice theory of clock synchronization (Welch & Lynch 1988, Lamport & Melliar-Smith 1985). The fact that the corresponding algorithms are implemented by hardware on interfaces connecting processors to real time buses is not as widely known as one would expect. Attempts to prove that a particular hardware correctly implements such an algorithm are very recent (Schmaltz 2006, Alkassar *et al* 2008a). Incidentally such proofs cannot be done in the usual digital hardware model; the usual model assumes a single clock domain; within a single clock domain there are no different clocks to be synchronized.
- For some system layers there are simply no established computational models yet. This is like computing areas under curves without having even agreed on the definition of Riemann integrals. This sounds like an exaggeration? There is no public formal model of the full system programming part of the x64 instruction set or of the power PC instruction set. The role of kernels and hypervisors is to virtualize smaller or larger portions of the underlying hardware. Without a full model of the instruction set of a processor, one cannot even fully specify what a kernel or a hypervisor on such a processor is supposed to do. Similarly, there is no commonly accepted semantics for many of the common operating system calls. Without such semantics one cannot possibly argue formally about software invoking such calls.

- Computational programming models are among the very basic definitions in computer science. Correctness theorems for layers of system hardware or software are usually simulation theorems between ‘adjacent’ system layers. Compilation provides for instance a translation between high level languages and assembler language. In the industrial case, some of these models happen to be *much* larger than is usual in ordinary mathematics or in theoretical computer science. A formalization of the x64 instruction set has to compress the (slightly incomplete) documentation of the manufacturer — containing about 1500 (AMD) resp. 3000 (Intel) pages — into a hopefully manageable mathematical model.

The above list ‘only’ concerns the correctness theory of single components of computer systems. If we wish to formally prove the correctness of entire computer systems, we have necessarily to solve an additional problem:

- we have to unite the correctness theories for the components of the system into a single mathematical theory for the entire system. For ordinary mathematics we must imagine a situation, where we have different theories, with different notations and here and there very slightly incompatible definitions for numbers, sequences, continuity, differentiability, integrability, etc. *Because* we have for many abstractions that we use no published soundness proofs we *are* for large parts of computer science presently in this situation. In recent years rapid progress toward the unification of theories of component correctness has been made. In (Knapp & Paul 2007) the authors argue, on paper, about the correctness of an OSEKtime (OSE 2001) like distributed real time operating system and all the underlying software and hardware components including processor, serial interface, clock synchronization circuitry, worst case execution time analysis, compiler, machine dependent parts of the operating system kernel, machine independent part of the kernel. While this is quite a large mathematical theory, it is very simple compared with the theory necessary to deal e.g. with hypervisors running on modern multi core processors.

For completeness we list two further problems arising in system verification:

- the construction of complex formal proofs is a large engineering project, very similar to the projects constructing very large software systems. Similar to software development systems one needs ‘system verification systems’, i.e. non-trivial software systems supporting dozens of verification engineers in the joint effort to construct large proofs. An example of such a system is described in (Hillebrand & Paul 2007).
- if we want to step-up from the verification of systems with 5000 LOC to systems of 50 000 LOC we do not wish to simply use 10 times more verification engineers; it is much more desirable to double the number of engineers and to quintuple the efficiency and the degree of automation of the CAV tools used. The good news here is, that this — for a change — is mainstream research. The bad news is that the present CAV systems have already been constructed by extremely competent people. Beating what they have constructed by a factor of 5 or more is a formidable challenge.

1.2 Contribution

Currently, one of the main obstacles in perfectly understanding how applications behave is a lack of understanding of how the environment in which they are running behaves. So, we need a formal model of an operating system, i.e. a formalization of the system call semantics

and a precise description of the possible interactions of user applications. Furthermore, as many modern software systems often rely on remote procedure calling (RPC), we also need to specify and verify this communication mechanism. The contribution of this work is to provide a formal environment to verify programs running concurrently under an operating system. For that we combine:

- a minimal, yet self-contained cross-section of the formal model of the Simple Operating System (SOS) (Bogan 2008b).
- a methodology for sequentially reasoning (most of the time) on concurrently running applications in SOS,
- a specification and verification outline of an RPC mechanism to implement client/server architectures in SOS.

To the best of our knowledge this is the first work on verifying an RPC mechanism against a faithful operating system model.

1.3 Related work

Even though nobody tried to put all pieces of the puzzle together, i.e. to describe a verified and accurate specification for an environment running client/server software, there are still important contributions in the individual areas. In the following, we discuss related work within each of the three areas, i.e. system software verification, reduction theory for concurrent program verification, and RPC, separately.

1.3a *System software verification:* There were and there are numerous attempts to increase confidence in system software by means of formal methods. The different approaches may be divided into four categories:

- (i) There are projects that focus on selected system components. They chose some part of the implementation of a complex system, set up some model for the implementation of the selected component (neglecting the remaining components), and then show some high level properties. There is for example work dealing with TCP/IP (Bishop *et al* 2005, Smith 1996), work focusing on filesystems (Bevier & Cohen 1996, Yang *et al* 2006, Joshi & Holzmann 2007, Arkoudas *et al* 2004, Grünbacher 2003), work analysing rights management (Sikkel & Stiernerling 1998), or work examining user interfaces (Beckert & Beuster 2004). There are many more attempts in this category with various degrees of formalism. Work in this category achieves valuable results. However, they all suffer from the fact that they only model parts of a complex system — parts that are not truly independent from the remaining system. Furthermore, if their model is not supported by a precise formalization of the underlying system layers and tools, correspondence between implementation and model cannot be proven.
- (ii) Projects such as the Perseus project (Pfitzmann *et al* 2001) or the Nizza architecture (Härtig *et al* 2005) do not choose a component from *within* a complex system but argue about a *parallel* system. The goal of these projects is to protect security-sensitive data and at the same time support complete functionality of common general purpose operating systems (OSs). Their approach is to reduce the system's trusted computing base by running legacy operating systems and security-sensitive services in separate partitions on top of a micro kernel. Projects in this category do not aim at verifying the kernel but concentrate on the security-sensitive services running on top of it. This multi-OS approach will help to increase security in select OS aspects. However, the security

of such a system relies on the micro kernel, i.e. any security property could be violated if the kernel can not be trusted.

- (iii) In the third category there are many recent projects that focus on entire micro kernels. Well-known examples in this category are L4.verified (Elphinstone *et al* 2007), VFiasco (Hohmuth & Tews 2005) and the derived Robin Project (Tews 2007), as well as Coyotos (Shapiro *et al* 2004). Projects in this category are dealing with executable software, yet they usually lack a sophisticated model of the underlying hardware and rely for example on compiler correctness.
- (iv) Finally, the strongest approach focuses on pervasive system verification. Other than the approach followed by projects in the previous category, this type of approach spans multiple layers of hard- and software. Pioneering work in this category was undertaken in the CLI stack (Bevier 1989).

Without a doubt, the last type of approach, the pervasive approach, is the most satisfactory one. However, it is also the most expensive one. Thus, so far no one has managed to integrate and formally verify a system stack up to the level of user application. The Verisoft project aims at exactly that. Up to now, the VAMP micro processor (Beyer 2005, Dalinger 2006, Tverdyshev 2008) together with its assembly language (Tsyban 2008), a verified compiler for the type safe C variant C0 (Petrova 2007, Leinenbach & Petrova 2008, Leinenbach 2008) and the VAMOS micro kernel (Dörrenbächer 2006, Daum 2008) have emerged from this project. What is missing to complete the stack is a (user-mode) operating system, i.e. a system that bridges the gap between micro kernel and user applications, provides a high-level interface to the attached devices, supports different users, and permits a fine-grained control of system resources. In order to fill the gap we designed and implemented the user-mode operating system SOS. Furthermore, we formally specified the SOS and thereby established a computational model for communicating user applications, called SOS*. In contrast to other projects, our specification is fully connected to specifications of lower system layers. As we will outline in the following sections, our model actually comprises an entire system, consisting of a processor, external devices, and a micro kernel.

1.3b *Reduction theory for concurrent program verification:* Reordering transitions to obtain atomic specifications has been well studied in the literature.

Aspects of the reordering technique, we use, appear under different names. It is simply labelled reduction (Cohen & Lamport 1998, Cohen 2000) (reducing the search space), partial order reduction (Peled 1998) (in the context of model checking), or linearization (Lipton 1975) (pretending atomicity of execution). Lipton proved safety properties (of pre/post-condition style) sequentially and propagated these to the implementation (Lipton 1975). Cohen & Lamport extended this to liveness and a more fine-grained analysis of the reordered parts of the sequence (Cohen & Lamport 1998, Cohen 2000).

1.3c *Remote procedure calling:* For a long time programmers struggled with the time consuming implementation of communication protocols for distributed systems. Birell & Nelson (1984) responded to this problem and proposed the first RPC mechanism.

They suggested to allow programs to call procedures located on other machines. When a process on machine A (client) calls a procedure on machine B (server), the calling process on A is suspended, and execution of the called procedure takes place on B. The caller eventually regains control, extracts the results of the procedure, and continues execution. Information can be transported from the caller to the callee in the parameters and can come back in

the procedure result. Neither message passing nor I/O at all is visible to the programmer. This method is known as remote procedure call. RPC became a widely-used technique that underlies many distributed operating systems (Tanenbaum & Renesse 1985).

A well-known implementation of RPC was provided by Sun Microsystems (Srinivasan 1995). The best example for an application using RPC is probably the network file system (NFS) (Shepler *et al* 2003).

The challenge of verifying a simple RPC memory system proposed 1994 by Broy and Lamport triggered a widespread response. The result was impressive: as many as 15 different solutions were published (Broy *et al* 1996). Their goal, however, was to compare different formalization techniques and proof methods on an abstract case-study for distributed computing — rather than providing programmers with a verified RPC mechanism. In contrast, we are aiming at demonstrating that verification of RPC in a real setting, running under a real operating system is feasible.

As far as we know, we are the first to present a formal specification of an operating system that, on the one hand, reaches up to the level of RPC and, on the other hand, is part of an integrated system stack. Still, this operating system is more than a toy. Our implementations have been used to successfully develop and run applications such as an SMTP- and signing server and a simple email client (Bogan 2007). Our specifications have been used to specify and partially prove properties of these user applications (Langenstein *et al* 2007b, Beuster *et al* 2006; 2007).

1.4 Outline

We start with the mathematical notation required to describe our theories. Then, we proceed, in Section 3, with a sketch of the Verisoft model stack. This stack starts with a hardware description and reaches via a micro kernel the application layer.

In Section 4, we develop SOS*, a distributed model of computation, describing the interactions of the simple operating system SOS with its user processes. Thereby, user processes are modelled in a quite abstract way such that they can be instantiated to model user programs in different programming languages.

In order to be realistic, the model necessarily has to include inter process communication (IPC) and basic portmapper functionality. Furthermore, to be able to derive any useful termination properties, fairness of the scheduler has to be postulated on the model. In the SOS* model, timeouts are restricted to finite and infinite timeouts. A more detailed theoretical treatment of run time behaviour is possible, but this would require us to extend the model hierarchy below the SOS* model, via compiler and assembly language, all the way down to the register transfer level of the processor hardware (Knapp & Paul 2007). The reason for that is that cache hits and misses influence the run time of assembly instructions by orders of magnitude, yet they become already invisible at the assembly language level.

Although we are arguing about a distributed system, we want to use conventional verification technology — say verification condition generators for Hoare logic — to argue about (as large as possible) sequential portions of the distributed computation. This can be justified by the simple theory of non-interference from Section 5.

In § 6.1, we instantiate the user process model for the C like programming language C0. An obvious ingredient is a small step semantics for C0 without system calls. We outline how to extend such small steps semantics for system calls like the portmapper- and IPC calls. This permits us to show that a service can indeed be looked up after it has been registered at the portmapper (Lemma 4).

In § 6.2, we show how to specify signatures of services in terms of C0types.

Interface compilers generate from such signatures a library of external C0functions implementing the remote procedure call primitives `RPCsend` and `RPCrecv`. The implementation of these functions uses the (previously defined) IPC primitives. One has to be able to show termination, if timeouts are finite (Lemma 5). Furthermore, one has to show that matching `RPCsend` and `RPCrecv` calls either lead to a timeout or to a rendezvous situation and that, in the latter case, the data is correctly communicated (Lemma 6).

The RPC primitives `RPCsend` and `RPCrecv` as well as the portmapper calls suffice to implement a wide variety of protocols. We demonstrate this in § 6-5 for a simple example of an RPC client protocol.

A library, providing functions implementing the complete client protocol for calling remote procedures, can be generated by a compilation process for interfaces. For the functions generated by this library, one has to be able to show that they look up services and, in the case of success, eventually send calls to the server and receive answers (Lemma 7). Lemma 7, just like Lemma 6, is carefully phrased to reflect the fact that protocols may get stuck if some clients do not obey the required protocol.

Finally, in § 6-6, we implement, with the given primitives, a simple Math Server. This server registers its services at the portmapper and then services remote procedure calls. If all clients adhere to the protocol, then one can show that the protocol never gets stuck and that remote procedure calls with infinite timeouts eventually get answers (Lemma 8).

2. Mathematical notations

In this section, we review typographical and mathematical issues that are relevant to the unambiguous understanding of the work at hand.

2.1 Basic types

The basic types used in this document are $\mathbb{B} = \{\text{TRUE}, \text{FALSE}\}$, $\mathbb{N}_{32} = \{0, \dots, 2^{32} - 1\}$, $\mathbb{N}_{32}^+ = \mathbb{N}_{32} \setminus \{0\}$, and $\mathbb{Z}_{32} = \{-2^{31}, \dots, 2^{31} - 1\}$. Additionally we use $\mathbb{N} = \{0, 1, 2, \dots\}$ for the set of natural numbers. In some places we introduce types for a particular purpose. Such types may be easily recognized by their name extension `.t`. Sometimes we write $p(\dots)$ instead of $\forall x. p(x)$, and $p(?)$ instead of $\exists x. p(x)$.

2.2 Sets

The power set of A is denoted by $\text{pow}(A)$. In rare cases we use the Hilbert Choice Operator to select an arbitrary element from a given non-empty set. We denote the Hilbert Choice Operator by **SOME**.

2.3 Tuples and records

For small structured values we use n-tuples. The type of such an n-tuple (x_0, x_1, \dots, x_n) is $T_0 \times T_1 \times \dots \times T_n$, if $x_0 \in T_0 \wedge x_1 \in T_1 \wedge \dots \wedge x_n \in T_n$.

Sometimes, we have to deal with structured values consisting of many components. To effectively model these values, we use records. Essentially records are n-tuples with explicitly labeled components. To declare a record type `rec.t`, we write $\text{rec.t} = \{n_0 : T_0, \dots, n_n : T_n\}$. Here, n_0 through n_n are distinct identifiers and T_0 through T_n are the types of the corresponding components. When referring to individual components of a record, we use the dot notation.

We write, for example, $x.n_i$ to refer to the component n_i of the value x . To construct an instance of some record type, we write $[[n_0 = v_0, \dots, n_n = v_n]]$. Here, the values v_0 through v_n must match the types from the corresponding record declaration. In many places we only need to update individual components of record-type values. Instead of reconstructing the entire value, mostly copying values, we write, for example, $x' = x[[n_i := v'_i]]$. Here, x' is a copy of x where the component n_i is updated to have the value v'_i . All other components remain unchanged.

2.4 Functions

In many places, we extend the type of the range of a function by the uninterpreted constant ε , in order to avoid partial functions. So, we use, for example, $g \in \mathbb{Z} \rightarrow \mathbb{N} \cup \{\varepsilon\}$ to declare a function that would otherwise only be defined for a subset of \mathbb{Z} . As we use this ‘type extension’ frequently, we write T_ε as a shorthand for $T \cup \{\varepsilon\}$. Occasionally, we need to update a function definition only for particular domain elements. Just like the notation used for records, we write $f' = f[[n_i := v'_i]]$ to denote $f'(x) = (\text{if } x = n_i \text{ then } v'_i \text{ otherwise } f(x))$. The domain of a function f is denoted by $\text{dom}(f)$.

2.5 Sequences

For any type T and the natural numbers n , we denote by T^n the sequence consisting of n elements of type T . Furthermore, we denote the type of arbitrary (finite and infinite) sequences of elements of type T by T^* . We assume that elements of a sequence can be enumerated (starting from 0). Access to the i -th element of a sequence x is denoted by x^i .

3. The verisoft stack

Our system stack comprises many layers. In general, each of the implementation layers is accompanied by a model that formalizes the user visible behaviour of all implementation layers up to this layer. A model of one layer is derived from the model of the (next) lower layer by means of instantiation and abstraction. Within the Verisoft stack, the most important layers are (also see figure 1):

- ISA, a the programmer’s model for the VAMP processor, i.e. the instruction set architecture.

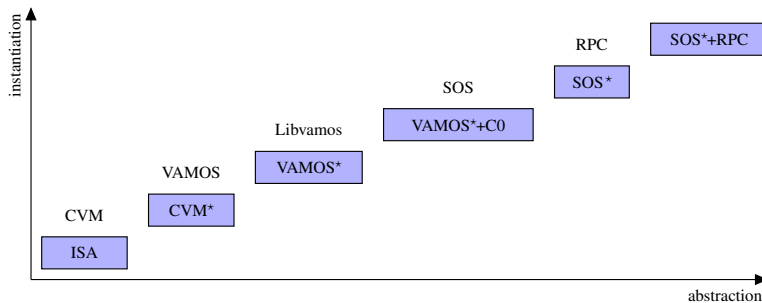


Figure 1. Verification stairs.

- CVM*, the model of communicating virtual machines (Rieden & Tsyban 2008). This model provides memory virtualization and switching between different threads of execution.
- VAMOS*, a model of the VAMOS micro kernel including Assembly user processes (Dörrenbaecher 2006). This model provides an abstract view on the application binary interface (ABI) of the VAMOS micro kernel.
- VAMOS*+C0, a model of the VAMOS micro kernel including user processes written in C0 (Daum 2008). This model generalizes VAMOS* in order to provide means to reason about user applications in C0 small-step semantics (rather than Assembly). Furthermore, VAMOS*+C0 formalizes the behaviour of the C0 interface/library Libsos.
- SOS*, a model of the entire operating including, for example, device drivers and the concept of distinct users (Bogan 2008a).
- SOS*+RPC, the SOS* model enhanced by RPC primitives (Shadrin 2006).

If all adjacent models are sewed together by proving simulation theorems, then it is guaranteed that properties can be transferred from the top-level abstraction down to the hardware level.

4. Operating system

4.1 Implementation

As in many other recent projects, we have split our operating system into a part running in system mode, i.e. the VAMOS micro kernel, and a part running in user mode, i.e. the SOS. In this subsection, we summarize the properties of the VAMOS micro kernel and briefly describe the SOS implementation.

4.1a *VAMOS*: The VAMOS micro kernel provides isolation of processes by means of virtual memory. Processes can be dynamically created and killed. They may communicate via synchronous inter-process communication (IPC). This communication can be controlled via permissions assigned to each pair of communication partners. IPC messages may have arbitrary size. VAMOS is equipped with a priority-based scheduler. Processes may register as user level device drivers. Processes that are registered as user level device drivers (for a certain device), receive interrupt notifications by means of IPC messages. Finally, the kernel maintains different privilege levels and thereby facilitates the implementation of user mode operating systems.

4.1b *SOS*: The operating system SOS is implemented on top of the VAMOS micro kernel. It is a process running in user mode. The kernel initially starts the SOS as the only process. It assigns the highest scheduling priority to the SOS and marks it as privileged. Therefore the SOS is enabled to register device drivers, to start and stop user processes, or to assign memory. In fact, most of the kernel calls provided by VAMOS require the calling process to be a privileged process. Unprivileged processes, which we will call *user applications*, are restricted to IPC and a small number of IPC related kernel calls. However, by means of *SOS calls*, user applications are provided with more versatile and more powerful calls. For example, the kernel call that allows to create a new process is substituted by an SOS call that allows to start a new application from an executable file. These SOS calls are transmitted to and answered by the SOS via kernel IPC calls.

Clearly, writing and verifying applications on Assembler level would be a tedious task. It is more likely that programmers will use something like the typesafe C variant C0(Leinenbach *et al* 2005b), which was specifically designed with verification in mind. In fact, the SOS itself is entirely written in C0.

In order to support user applications calling the SOS, a C0library is provided that wraps the necessary IPC calls and hides the difference between kernel calls and SOS calls. Figure 2 shows the user applications' limited access to the kernel. Furthermore, it shows how the SOS serves as intermediate layer between user applications and kernel. Note that, for performance reasons, a number of kernel calls remain accessible to user applications. For example, routing IPC-messages through the SOS would dramatically slow down IPC, double the number of context switches, and increase the SOS's latency and memory consumption.

Upon start the SOS process registers as device driver for the connected devices. Currently we support a hard disk, a terminal, and a network card. Because our kernel does not support shared memory, we implemented all device drivers as part of a single process; the SOS process. In terms of SOS calls, these drivers provide a POSIX-like (IEEE 2004) interface. Underneath the socket calls we have a TCP implementation, file IO calls access a FAT 32 file system, and terminal IO provides a number of different virtual terminals. Access to these resources is managed by the SOS based on the notion of different users. In fact, besides driving devices, this access control is one of the SOS's main purposes.

At the end of the initialization phase, the SOS starts a login shell on each of the virtual terminals and then listens to SOS calls. These login shells are the initial user applications. All further applications descent from one of these applications. It is important to know that all user applications are unprivileged and run with the same priority. If one application executes or forks a new one, it does so by calling the SOS. This way, the SOS keeps tight control and

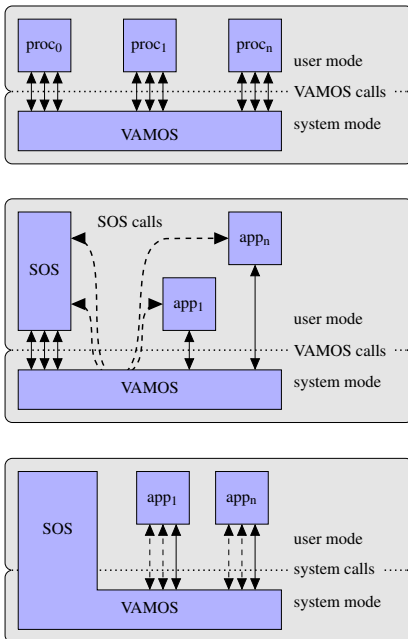


Figure 2. SOS Calls. (i) User processes solely rely on VAMOS calls. (ii) In the presence of the SOS, user applications are restricted to a few VAMOS calls but (via IPC) they may use SOS calls. (iii) User applications cannot see the difference between VAMOS calls and SOS calls. Thus, from the application point of view, the SOS process and the VAMOS micro kernel melt together. The resulting (single) operating system provides services in terms of system calls.

has insights on every process running on the system, i.e. it can establish its own view of the system.

Besides SOS calls, the SOS also receives interrupt notifications. As for SOS calls, these notifications are delivered to the SOS via IPC. The SOS receives this notification as soon as it has finished any current work. However, since it has the highest scheduling priority, none of the user applications will be scheduled until all interrupts are handled. Interrupt delivery via IPC results in a high latency but we achieve a system that can be easily abstracted to a sequential system.

The SOS provides 31 calls, allowing user applications to: (i) manage different users, (ii) interact with a file system, (iii) communicate with the outside world, (iv) handle applications, (v) locate and register remote procedure services, and (vi) interact with virtual terminals. Altogether the implementation of the SOS, including the device drivers and the C0library, consists of about 8000 lines of C0code.

4.2 Specification

In this section we describe SOS*, a model of a whole computer system. We start out with an overview of the main SOS* components and then present an exact definition of each of these components.

Note that the specification we present here is only an extract of our Isabelle/HOL formalization (Bogan 2008b). Many aspects (of the complete specification) are either left out or highly simplified. For example, within this document, we are not formalizing any of the SOS calls related to network communication, file access, or terminal I/O. Furthermore, for IPC calls, we do not consider the combined send-receive operation and neglect the IPC rights.

4.2a *Overview:* Formally, SOS* is defined as a transition system:

$$\text{SOS}^* = (S, \Delta, R, \dots).$$

Where,

- S is the set of possible configurations (the SOS* state space),
- $\Delta \subset S \times S$ is the transition relation, and
- R characterizes valid SOS* runs.

SOS* is intended to be used as a programming model for communicating user applications. In SOS*, we choose not to restrict the types of user applications that may be verified to a particular programming language. Instead, we incorporate user applications in the form of I/O automata (Lynch 1996):

$$\text{APP}^* = (S_p, \Sigma_p, \Omega_p, \delta_p, \omega_p, \dots).$$

Where:

- S_p is the set of possible configurations (the application state space),
- Σ_p is the input alphabet (system call results),
- Ω_p is the output alphabet (system calls),
- $\delta_p \in (S_p \times \Sigma_p \cup \{\varepsilon\}) \rightarrow S_p$ is the transition function, and
- $\omega_p \in (S_p) \rightarrow (\Omega_p \cup \{\varepsilon\})$ computes the application output for a given state.

Describing user applications as automata has the advantage that the abstraction can be easily instantiated by different machine types (e.g. Assembly- or C programs). A different machine type does not change the global transition system, as long as the new machine type complies with the (interface) alphabets Σ_p and Ω_p . For SOS^* that means that the alphabets Σ_p and Ω_p must be well defined. The remaining types and functions of APP^* , however, may be SOS^* parameters. Hence, we get the following (updated) definition of SOS^* :

$$SOS^*(S_p, \delta_p, \omega_p, \dots) = (S, \Delta, R, \Sigma_p, \Omega_p, \dots).$$

Note, although different machine types are supported, S_p , δ_p , and ω_p have to be fixed throughout a model run. That means all user applications share the same S_p , δ_p , and ω_p (which still might cover Assembly- and C programs). Hence, the subscript “p” (as in ω_p) is not an index but belongs to the function name.

After introducing the overall concept of SOS^* , we will now elaborate on each of its components.

4.2b State space: User applications perceive their environment, i.e. the underlying hardware and system software as well as other user applications, by means of system calls. In order to correctly formalize the behaviour of these calls, the SOS^* state space contains data structures that represent the user visible part of the environment. In this subsection we describe selected components of S . The complete SOS^* model has many more components. Here, we describe only those parts that are relevant for RPC.

(i) Users: SOS is a multi-user operating system. It allows different registered users to log in. A user is thereby referred to by his user id. In SOS , all registered users are stored in the user data base.

In SOS^* , we represent user ids by numbers. User ids have the type $uid.t \subset \mathbb{N}_{32}$. The state-space component udb contains all registered users:

$$udb : pow(uid.t).$$

(ii) User applications: The SOS supports communicating user applications. Previously, we already explained how user applications can be modelled as I/O automata. Now we describe how their representation is integrated into the SOS^* state space.

User applications are manifested in the SOS^* state space in three ways. They are represented by:

- a local state, i.e. the application’s internal configuration (e.g. the Assembly- or C configuration),
- a number of process-related data structures, i.e. bookkeeping data structures about the user process maintained by the kernel, and
- a number of application-related data structures, i.e. bookkeeping data structures maintained by the SOS .

(iii) Local state: From the kernel’s point of view, the SOS and the user applications are user processes. The maximum number of simultaneously running user processes is denoted by $MAXPROC \in \mathbb{N}_{32}^+$. The kernel uses process identifiers (PIDs) to refer to specific processes. In SOS^* , the set of all PIDs is represented by $pid.t = \{1, \dots, MAXPROC\}$. Here, the constant $OSPID \in pid.t$ denotes the PID of the SOS process. The local states of all user applications

are stored in the process data base pdb . This state-space component is a function that maps PIDs to process states. Currently unassigned PIDs are mapped to ε . Thus:

$$pdb : pid_t.$$

Where:

$$pdb_t = pid_t \setminus \{OSPID\} \rightarrow S_p \cup \{\varepsilon\}.$$

Note that although the SOS is a user process, it is not a user application and, therefore, invisible in SOS^* . Hence, $OSPID$ is excluded from the domain of pdb .

(iv) *Kernel data structures about user processes:* In SOS^* , there are a number of data structures that are, in the implementation, maintained by the kernel. In the complete model, these kernel data structures are combined in the (SOS^*) state space component kds . Here, we discuss only one of these kernel data structures, i.e. the handle data base.

User processes exclusively identify each other using so-called handles. Handles are local names for PIDs. On a per-process basis, the kernel maintains the mapping between *handles* and *PIDs*. This mapping is called the handle data base. As we will explain later, unless a process has a handle for another process, there is no way for the earlier to approach the latter. Thus, the indirection through handles provides for better control of information flow. In SOS^* , the handle data base is represented by the kds component hdb :

$$hdb : pid_t \times hn_t \rightarrow pid_t_\varepsilon.$$

Here, hn_t , with $hn_t \subset \mathbb{N}_{32}$, represents the set of possible handles. Currently unassigned handles are mapped to ε . There are a number of special handles. For the work at hand, only the pseudo handle identifying no process is relevant. In SOS^* this handle is denoted by $HN-NONE \in hn_t$.

As mentioned earlier, the complete SOS^* model contains more kernel data structures. There, handles are, for example, accompanied by communication rights. These rights provide additional means to fine tune the inter-process communication. For the work at hand, however, the handle data base is the only relevant kernel data structure. Thus, here, the state space component kds has only one component:

$$kds : kds_t.$$

Where:

$$kds_t = \{hdb : pid_t \times hn_t \rightarrow pid_t_\varepsilon, \dots\}.$$

(v) *SOS Data structures about user applications:* The SOS keeps track of all user processes and adds rights management and access control based on users. It thereby establishes the concept of user applications. For each user application, the SOS maintains a certain amount of information. It stores, for example, which user started a particular application and whether a certain application has access to the screen. In SOS^* , the information about a single user application is represented by an instance of type app_t . Among other things, this type contains the field *owner* representing the user owning the application:

$$app_t = \{owner : uid_t, \dots\}.$$

Again, in the complete SOS* model, app_t contains more fields. The information about all user applications is assembled in the application data base. In SOS*, this data base is represented by the function adb , which maps handles to the associated information. Currently unassigned application handles are mapped to ε :

$$adb : hn_t \rightarrow app_t_\varepsilon.$$

(vi) *Portmapper*: The SOS provides infrastructure for so-called RPCs. RPCs allow one application, the client, to take advantage of some service provided by another application, the server. Here, a service is specified by an interface name and a procedure name. At compile time, clients know the names of the services they intend to call. However, the location of this service, i.e. the handle of the providing application, is unknown at that time. Hence, we need a runtime mapping of service names to service providers. This mapping is called *portmapper data base*.¹

In SOS*, a service name is represented by the type $service_t$. Here, a service name is the combination of the interface id $iid_t \subset \mathbb{N}_{32}$ and a procedure id $prcid_t \subset \mathbb{N}_{32}$, i.e. $service_t = iid_t \times prcid_t$.

Based on service names, the portmapper data base comprises:

- *serv* a mapping between interface ids and the handles of the providing servers,
- *reg* a set of registered services, and
- *known*, the set of known services.

In SOS*, the portmapper data base is represented by the state-space component $pmdb$:

$$pmdb : pmdb_t.$$

Where:

$$pmdb_t = \{ serv : iid_t \rightarrow hn_t_\varepsilon, reg : pow(service_t), known : pow(service_t) \}.$$

Note that we need the component *known* because a portmapper usually only supports a set of well-known services. In addition, note that (supported) interfaces that are not served, are mapped to ε . Finally, servers can only register for one interface. Such an interface, however, may contain several procedures serving different purposes.

(vii) *Summing up*: Now, collecting all pieces of the (reduced) SOS* state space, S is defined as follows:

$$S = \{ \begin{array}{l} udb : pow(uid_t), \\ pdb : pdb_t, kds : kds_t, adb : hn_t \rightarrow app_t_\varepsilon, \\ pmdb : pmdb_t, \\ \dots \end{array} \}.$$

¹Currently, our portmapper implementation only supports local inquiries and instead of mapping services to IP addresses and port numbers, it maps services to handles. This could be easily changed but for now this simplified version suffices to serve our needs.

4.2c *Alphabets:* In SOS^* , we choose to represent system calls and system-call results in a machine independent form, i.e. the alphabet Ω_p and Σ_p . In this section, we describe all those elements of Ω_p and Σ_p that are relevant in the context of this paper.

The alphabet Ω_p contains the abstract representations of the available system calls. In the work at hand, Ω_p only contains:

- representations for the portmapper calls to register, look up, and unregister services (REG, LUP, and UNREG),
- representations for sending and receiving IPC messages (SND and RCV), and
- representations for undefined SOS- and kernel calls (UNDEF-SC and UNDEF-KC).

Each of these calls is represented by a constructor indicating the type of system call and (possibly) a number of parameters:

$$\Omega_p = \{ \begin{array}{l} \text{REG } id_i \ id_p, \text{ LUP } id_i \ id_p, \text{ UNREG } id_i \ id_p, \\ \text{UNDEF-SC}, \text{ UNDEF-KC}, \\ \text{SND } h \ m \ t, \text{ RCV } h \ b \ t, \\ \dots \\ | \quad id_i \in iid_t_\varepsilon \wedge id_p \in prcid_t_\varepsilon \\ \quad \wedge h \in hn_t_\varepsilon \wedge t \in \{\text{FINITE}, \text{INFINITE}\}, m \in \text{byte_}t^* \cup \{\varepsilon\}, b \in \mathbb{N}_{32} \end{array} \}.$$

Note that we denote all finite timeouts by FINITE and infinite timeouts by INFINITE.² Further note, we use $m \in \text{byte_}t^* \cup \{\varepsilon\}$ and $b \in \mathbb{N}_{32}$ as abstract representations for messages and buffers, respectively. We choose this representation to be universal enough to support arbitrary process abstraction and at the same time describe the essence of messages and buffers, i.e. a sequence of bytes and a container of a certain size. Finally, note that it may be that a process calls the system but passes along parameter values that do not match the required type (e.g. $id_i \notin iid_t$). In SOS^* , we represent all of these values through the symbol ε . Hence, most of the parameter types are unions of the required type and ε .

The counterpart to Ω_p is Σ_p . The alphabet Σ_p contains the abstract representations of the available system-call results. In the work at hand, Σ_p only contains:

- the result messages for a successful portmapper look-up and a successful IPC-receive operation (SUCC-LUP and SUCC-RCV) and
- some general purpose success and error messages (SUCC, ERR, and TIMEOUT).

Again, each of these results is represented by a constructor indicating the type of system-call result and (possibly) an additional parameter:

$$\Sigma_p = \{ \begin{array}{l} \text{ERR}, \text{ TIMEOUT}, \text{ SUCC}, \text{ SUCC-LUP } h, \text{ SUCC-RCV } h \ m, \dots \\ | \quad h \in hn_t \wedge m \in \text{byte_}t^* \end{array} \},$$

²For a number of reasons that are detailed in (Daum 2008), the scheduler is no longer visible in SOS^* . Along with that, there is no precise notion of time. Therefore, we are not able to model timeout situation more precisely than ‘a certain system call might timeout’ or ‘a certain system call cannot timeout’.

Now, no matter which process abstraction was chosen, one can define a function $\omega_p \in S_p \rightarrow \Omega_p \cup \{\varepsilon\}$ that maps a process's state to:

- $x \in \Omega_p$, if a particular process wants to call the system, or
- ε , if the processes wants to perform a local computation, i.e. a computation that does not involve calling the system.

If an SOS call was received and treated by the SOS, then the results must be returned to the appropriate application. For returning these results, we simply assume that the local transition function δ_p is defined to take $\Sigma_p \cup \{\varepsilon\}$ as (process-external) inputs, i.e. $\delta_p \in S_p \times \Sigma_p \cup \{\varepsilon\} \rightarrow S_p$. Using δ_p , we can define how the (global) SOS * state is changed when a system-call result is returned. As this is a quite common task, we define the function res as follows:

$$res \in S \times pid.t \times \Sigma_p \rightarrow S$$

$$res(s, p, r) = s[[pdb(p) := \delta_p(s.pdb(p), r)]].$$

4.2d Transition relation: In this section we define the (reduced) SOS * transition relation. Again, we only define those transitions that are relevant in order to argue about RPC.

(i) Auxiliaries: Resolving handles to PIDs and vice versa will be done quite often. Thus, as a short hand, we define the functions $ph2p$ and $pp2h$.

The call $ph2p(s, p, hn)$ inspects the handle data base of p and returns the PID, hn is pointing to. If, for p , the handle hn is not assigned, then ε is returned:

$$ph2p \in S \times pid.t \times hn.t \rightarrow pid.t_\varepsilon$$

$$ph2p(s, p, hn) = s.kds.hdb(p)(hn).$$

The call $pp2h(s, p_1, p_2)$ returns the handle the process p_1 may use to refer to p_2 . If p_2 is unknown to p_1 , then ε is returned:

$$pp2h \in S \times pid.t \times pid.t \rightarrow hn.t_\varepsilon$$

$$pp2h(s, p_1, p_2) =$$

$$\begin{cases} \text{SOME}\{x \mid s.kds.hdb(p_1)(x)\} & \text{if } \exists x \in pid.t. s.kds.hdb(p_1)(x) = p_2, \\ \varepsilon & \text{otherwise.} \end{cases}$$

Note, for $pp2h(s, p_1, p_2)$ to be deterministic, $s.hdb$ needs to be an injective function. Thus, we require that (for each process) there are no two (different) handles pointing to the same process.

For resolving handles and PIDs from the perspective of the SOS, we additionally provide the following short hands $h2p(s, hn) = ph2p(s, OSPID, hn)$ and $p2h(s, p) = pp2h(s, OSPID, p)$.

(ii) Transitions: After the preparatory work, we can now specify the individual transitions.

(iii) Register a service: The SOS implementation provides a system call that allows user applications to register as server providing a certain service. In SOS *, this call is represented by `REG id_i id_p` , where id_i and id_p are the interface- and procedure id, respectively.

In order to successfully register a service, a number of preconditions must be met. The predicate $vrega?$ is satisfied, if these preconditions are met. For $vrega?(s, p, id_i, id_p)$ to hold,

(id_i, id_p) must be a service known to the portmapper, the calling process p is not yet registered to serve a different interface, and the interface id_i has not been registered by a different process, i.e. $s.pmdb.serv(id_i) \in \{p2h(s, p), \varepsilon\}$. That is:

$$\begin{aligned} vrega? &\in S \times pid_t \times iid_t_\varepsilon \times prcid_t_\varepsilon \rightarrow \mathbb{B} \\ vrega?(s, p, id_i, id_p) &\equiv (id_i, id_p) \in s.pmdb.known \\ &\wedge \forall x \neq id_i. s.pmdb.serv(x) \neq p2h(s, p) \\ &\wedge s.pmdb.serv(id_i) \in \{p2h(s, p), \varepsilon\}. \end{aligned}$$

Now, if there exists a process p that wants to register a service, i.e. $\omega_p(s.pdb(p)) = \text{REG } id_i id_p$, but one or more of the preconditions are not met, then an error message is returned to the caller. That is, the next SOS* state is computed by applying ERR to p 's local state:

$$\begin{aligned} \Delta \supset \{ (s, res(s, p, \text{ERR})) \mid \\ \exists id_i, id_p. \omega_p(s.pdb(p)) = \text{REG } id_i id_p \wedge \neg vrega?(s, p, id_i, id_p) \}. \end{aligned}$$

If there exists a process p that wants to register a service and all preconditions are met, then it is ensured that p is registered for this services, i.e. $s'.pmdb.serv(id_i) = p2h(s, p)$ and $s'.pmdb.reg = s.pmdb.reg \cup \{(id_i, id_p)\}$. Furthermore, a success message is returned to the calling process.

$$\begin{aligned} \Delta \supset \{ (s, res(s', p, \text{SUCC})) \mid \\ \exists id_i, id_p. \omega_p(s.pdb(p)) = \text{REG } id_i id_p \wedge vrega?(s, p, id_i, id_p) \\ \wedge s' = s \left[\left[\begin{array}{l} pmdb.serv(id_i) := p2h(s, p), \\ pmdb.reg := s.pmdb.reg \cup \{(id_i, id_p)\} \end{array} \right] \right] \}. \end{aligned}$$

(iv) *Lookup a service:* The SOS implementation provides a system call that allows user applications to lookup the handle of the server providing a certain service. In SOS*, this call is represented by LUP $id_i id_p$, where id_i and id_p are the interface- and procedure id, respectively.

If there exists a process p that wants to lookup a service, i.e. $\omega_p(s.pdb(p)) = \text{LUP } id_i id_p$, but the service (id_i, id_p) is not registered (either because the service does not exist at all or because the service is not yet registered), then an error message is returned:

$$\begin{aligned} \Delta \supset \{ (s, res(s, p, \text{ERR})) \mid \\ \exists id_i, id_p. \omega_p(s.pdb(p)) = \text{LUP } id_i id_p \wedge (id_i, id_p) \notin s.pmdb.reg \}. \end{aligned}$$

If there exists a process p that wants to lookup a service and there exists a process p_s that previously registered for this service, then a success message, including the handle to the providing server, is returned. That is, the handle data base is updated to reflect p 's right to communicate with p_s , i.e. $s'.kds.hdb(p)(hn_s) = p_s$. Furthermore, SUCC-LUP hn_s is applied

to p 's local state:

$$\begin{aligned} \Delta \supset \{ (s, res(s', p, SUCC-LUP \ hn_s)) \mid \\ \exists id_i, id_p, p_s. \quad & \omega_p(s.pdb(p)) = LUP \ id_i \ id_p \wedge (id_i, id_p) \in s.pmdb.reg \\ & \wedge p_s = h2p(s, s.pmdb.serv(id_i)) \\ & \wedge hn_s = \begin{cases} \min\{x \mid ph2p(s, p, x) = \varepsilon\} & \text{if } pp2h(s, p, p_s) = \varepsilon \\ pp2h(s, p, p_s) & \text{otherwise} \end{cases} \\ & \wedge s' = s[[s.kds.hdb(p)(hn_s) := p_s]] \} \end{aligned}$$

Note, because we require $|pid_t| \leq |hn_t|$, assigning handles for, so far, unknown processes never fails. Because of this, we can be sure that either there exists already a mapping that points to the PID of the service provider, i.e. $pp2h(s, p, p_s)$, or there exists a (so far) unused handle, i.e. $\exists x. ph2p(s, p, x) = \varepsilon$, that may be used to establish a new mapping.

(v) *Unregister a service:* The SOS implementation provides a system call that allows a user applications to unregister a service it provides. In SOS^* , this call is represented by $UNREG \ id_i \ id_p$, where id_i and id_p are the interface- and procedure id, respectively.

In order to successfully unregister a service, two preconditions must be met. The predicate $vunrega?$ is satisfied, if these preconditions are met. For $vunrega?(s, p, id_i, id_p)$ to hold, p must be the process that serves the interface id_i and (id_i, id_p) must be registered. That is:

$$vunrega? \in S \times pid_t \times iid_t_\varepsilon \times prcid_t_\varepsilon \rightarrow \mathbb{B}$$

$$vunrega?(s, p, id_i, id_p) \equiv s.pmdb.serv(id_i) = p2h(s, p) \wedge (id_i, id_p) \in s.pmdb.reg.$$

If there exists a process p that wants to unregister a service, i.e. $\omega_p(s.pdb(p)) = UNREG \ id_i \ id_p$, but one or more of the preconditions are not met, then an error message is returned to the caller:

$$\begin{aligned} \Delta \supset \{ (s, res(s, p, ERR)) \mid \\ \exists id_i, id_p. \omega_p(s.pdb(p)) = UNREG \ id_i \ id_p \wedge \neg vunrega?(s, p, id_i, id_p) \}. \end{aligned}$$

If there exists a process p (serving the interface id_s) that wants to unregister one of the services it is registered for, then this service is removed from the set of registered services, i.e. $s'.pmdb.reg = s.pmdb.reg \setminus \{(id_i, id_p)\}$, and a success message returned:

$$\begin{aligned} \Delta \supset \{ (s, res(s', p, SUCC)) \mid \\ \exists id_i, id_p. \quad & \omega_p(s.pdb(p)) = UNREG \ id_i \ id_p \wedge vunrega?(s, p, id_i, id_p) \\ & \wedge \exists x \neq id_p. (id_i, x) \in s.pmdb.reg \\ & \wedge s' = s[[pmdb.reg := s.pmdb.reg \setminus \{(id_i, id_p)\}]] \} \end{aligned}$$

If there exists a process p (serving the interface id_s) that wants to unregister the last service it is registered for, then this service is removed from the set of registered services. Furthermore, unlike in the previous case, p is removed from the mapping indicating it as server providing services from the interface id_i , i.e. $s'.pmdb.serv(id_i) = \varepsilon$. Finally a success

message is returned:

$$\Delta \supset \{ (s, \text{res}(s', p, \text{SUCC})) \mid \\ \exists id_i, id_p. \quad \omega_p(s.pdb(p)) = \text{UNREG } id_i id_p \wedge \text{vunrega?}(s, p, id_i, id_p) \\ \wedge \nexists x \neq id_p. (id_i, x) \in s.pmdb.reg \\ \wedge s' = s \left[\left[\begin{array}{l} pmdb.reg := s.pmdb.reg \setminus \{(id_i, id_p)\}, \\ pmdb.serv(id_i) := \varepsilon \end{array} \right] \right] \\ \}.$$

(vi) *Undefined SOS calls:* It may be, that a user application calls the SOS but the call is unknown to the SOS. In such a case the SOS does not reply to the caller. In SOS* such calls are represented by UNDEF-SC. If there exists a process p , whose output is UNDEF-SC, then handling this call does not change the (visible) state:

$$\Delta \supset \{ (s, s) \mid \exists p. \omega_p(s.pdb(p)) = \text{UNDEF-SC} \}.$$

(vii) *Undefined kernel calls:* Similarly to an undefined SOS call, it could also happen that a user application calls the kernel but the call is unknown to the kernel. In such a case the kernel returns an error message. In SOS* such calls are represented by UNDEF-KC. If there exists a process p , whose output is UNDEF-KC, then handling this call results in an error message:

$$\Delta \supset \{ (s, \text{res}(s, p, \text{ERR})) \mid \exists p. \omega_p(s.pdb(p)) = \text{UNDEF-KC} \}.$$

(viii) *IPC- send and receive:* The kernel implementation provides system calls that allow user applications to communicate between each other. That is, user applications may send and receive messages. In SOS*, the send call is represented by $\text{SND } h_r m t_s$, where h_r is the handle of the receiving application, m is the message to send, and t_s is the send timeout. Analogously, the receive call is represented by $\text{RCV } h_s b t_r$, where h_s is the handle of the sending application, b is the buffer where to place the message, and t_r is the receive timeout. It may be that an application wants to receive a message from any application (other than a particular one). Such an open-receive call differs from a regular receive call in the specified handle. If $h_s = \text{HN-NONE}$, then the calling application wants to do an open receive. The opposite operation, i.e. sending a broadcast, is currently not supported in SOS.

In order to successfully send or receive messages, the arguments supplied with the calls must match the required types. That means, for a (potentially) successful send operation the receiver handle h_r must point to a process known to the caller, i.e. $h_r \notin \{\varepsilon, \text{HN-NONE}\} \wedge \text{ph2p}(s, p, h_r) \neq \varepsilon$, and the message must be well formed, i.e. $m \neq \varepsilon$:³

$$\text{vsnda?} \in S \times \text{pid.t} \times \text{hn.t}_\varepsilon \times \text{byte.t}^* \cup \{\varepsilon\} \rightarrow \mathbb{B}$$

$$\text{vsnda?}(s, p_s, h_r, m) \equiv h_r \notin \{\varepsilon, \text{HN-NONE}\} \wedge \text{ph2p}(s, p_s, h_r) \neq \varepsilon \wedge m \neq \varepsilon.$$

Similarly, for a (potentially) successful receive operation the sender handle h_s must be HN-NONE or it must point to a process known to the caller, i.e. $h_s \neq \varepsilon \wedge \text{ph2p}(s, p, h_s) \neq \varepsilon$,

³The well-formedness of a messages also includes things like memory safety. That means, if, for example, a message is not entirely in the memory region of the calling process, then, in SOS*, this message is represented by ε .

and the buffer must be well formed, i.e. $b \neq \varepsilon$ ⁴

$$vrcva? \in S \times pid.t \times hn.t_\varepsilon \times \mathbb{N}_{32} \rightarrow \mathbb{B}$$

$$vrcva?(s, p_r, h_s, b) \equiv (h_s = \text{HN-NONE} \vee (h_s \neq \varepsilon \wedge ph2p(s, p, h_s) \neq \varepsilon)) \wedge b \neq \varepsilon.$$

Now, if there exists a process p_s that wants to send a message, i.e. $\omega_p(s.pdb(p_s)) = \text{SND } h_r m t_s$, but one or more of the preconditions are not met, then an error message is returned to the caller:

$$\begin{aligned} \Delta \supset \{ (s, res(s, p_s, \text{ERR})) \mid \exists h_r, m, t_s. \omega_p(s.pdb(p_s)) \\ = \text{SND } h_r m t_s \wedge \neg vsnda?(s, p_s, h_r, m) \}. \end{aligned}$$

If there exists a process p_r that wants to receive a message, i.e. $\omega_p(s.pdb(p_r)) = \text{RCV } h_s b t_r$, but one or more of the preconditions are not met, then an error message is returned to the caller:

$$\begin{aligned} \Delta \supset \{ (s, res(s, p_r, \text{ERR})) \mid \exists h_s, b, t_r. \omega_p(s.pdb(p_r)) \\ = \text{RCV } h_s b t_r \wedge \neg vrcva?(s, p_r, h_s, b) \}. \end{aligned}$$

In our implementation, messages are synchronously exchanged. That is, a pending send operation is completed by a matching receive operation and vice versa. Whether there exists a rendezvous situations is inspected by the predicate $rv?$. $rv?(s, p_s, h_r, p_r, h_s)$ is satisfied, if both parties want to communicate with each other. That is, each of the specified handles matches the PIDof the opposite site or the handle specified by the sender matches the PIDof the receiver and the receiver uses an open receive call ($h_s = \text{HN-NONE}$):

$$rv? \in S \times pid.t \times hn.t \times pid.t \times hn.t \rightarrow \mathbb{B}$$

$$rv?(s, p_s, h_r, p_r, h_s) \equiv p_r = ph2p(s, p_s, h_r) \wedge (p_s = ph2p(s, p_r, h_s) \vee h_s = \text{HN-NONE}).$$

If there exists a process p_r that wants to receive a message with a finite timeout, then it may be that this operation fails because of a timeout. That is, if the caller satisfied the preconditions $vrcva?$ but, up to now, the call could not be answered and currently there is no rendezvous situation, then the receive operation fails, returning a timeout error. This does not mean that any receive call with a finite timeout fails, but, unless an infinite timeout was specified, it could always be that such a call returns because of a timeout:

$$\begin{aligned} \Delta \supset \{ (s, res(s, p_r, \text{TIMEOUT})) \mid \\ \exists h_s, b. \quad \omega_p(s.pdb(p_r)) = \text{RCV } h_s b \text{ FINITE} \\ \wedge vrcva?(s, p_r, h_s, b) \\ \wedge (\forall p_s, h_r, m, t_s. \omega_p(s.pdb(p_s)) = \text{SND } h_r m t_s \wedge \neg rv?(s, p_s, h_r, p_r, h_s)) \}. \end{aligned}$$

Similarly, if there exists a process p_s that wants to send a message with a finite timeout, then it may be that this operation fails because of a timeout:

$$\begin{aligned} \Delta \supset \{ (s, res(s, p_s, \text{TIMEOUT})) \mid \\ \exists h_r, m. \quad \omega_p(s.pdb(p_s)) = \text{SND } h_r m \text{ FINITE} \\ \wedge vsnda?(s, p_s, h_r, m) \\ \wedge (\forall p_r, h_s, b, t_r. \omega_p(s.pdb(p_r)) = \text{RCV } h_s b t_r \wedge \neg rv?(s, p_s, h_r, p_r, h_s)) \}. \end{aligned}$$

⁴Just like for well-formedness of messages, the well-formedness of buffers also include things like memory safety. Again, if, for example, a buffer is not entirely in the memory region of the calling process, then, in SOS*, this buffer is represented by ε .

If all goes well and there is a rendezvous situation, it may still be that the buffer b , provided by the receiver, is too small to fit the message m . In this case, the senders call fails with an error message. The receivers call, however, remains pending — waiting for a timeout or another matching send operation:

$$\begin{aligned} \Delta \supset & \{ (s, \text{res}(s, p_s, \text{ERR})) \mid \\ & \exists p_r, h_s, h_r, b, m, t_s, t_r. \\ & \omega_p(s.pdb(p_s)) = \text{SND } h_r m t_s \wedge \omega_p(s.pdb(p_r)) = \text{RCV } h_s b t_r \\ & \wedge \text{vsnda?}(s, p_s, h_r, m) \wedge \text{vrcva?}(s, p_r, h_s, b) \\ & \wedge \text{rv?}(s, p_s, h_r, p_r, h_s) \\ & \wedge \text{len}(m) > b \}. \end{aligned}$$

Finally, if all goes well, there is a rendezvous situation, and the message fits into the receive buffer, then both parties receive a success message. That is, **SUCC** is applied to the senders (local) state and **SUCC-RCV** $h_n m$ is applied to the receivers (local) state. The latter includes the actual message m and the (possibly new) handle h_n for the sender.

$$\begin{aligned} \Delta \supset & \{ (s, \text{res}(\text{res}(s', p_r, \text{SUCC-RCV } h_n m), p_s, \text{SUCC})) \mid \\ & \exists h_s, h_r, b, t_s, t_r. \\ & \omega_p(s.pdb(p_s)) = \text{SND } h_r m t_s \wedge \omega_p(s.pdb(p_r)) = \text{RCV } h_s b t_r \\ & \wedge \text{vsnda?}(s, p_s, h_r, m) \wedge \text{vrcva?}(s, p_r, h_s, b) \\ & \wedge \text{rv?}(s, p_s, h_r, p_r, h_s) \\ & \wedge \text{len}(m) \leq b \\ & \wedge h_n = \begin{cases} \min\{x \mid \text{ph2p}(s, p_r, x) = \varepsilon\} & \text{if } \text{pp2h}(s, p_r, p_s) = \varepsilon \\ \text{pp2h}(s, p_r, p_s) & \text{otherwise} \end{cases} \\ & \wedge s' = s[[s.kds.hdb(p_r)(h_n) := p_s]]. \end{aligned}$$

Note, this definition does not *explicitly* model the actual copy operation. This is because such a definition would very much depend on the particular process abstraction. However, just like ω_p computes the (generic) process output, we assume that δ_p updates the state according to the (generic) input. That means, for example, we assume $\text{res}(s', p_r, \text{SUCC-RCV } h_n m)$ updates the state of process p_r in such a way that the message m is copied to the place specified within the receive call (also see § 4.2c).

(ix) *Local computation*: Last but not least, there are **SOS*** transitions that represent local computations of user applications. If there exists an application p , such that its output is equal to ε , then this application may do a local step:

$$\Delta \supset \{(s, s') \mid \exists p. \omega_p(s.pdb(p)) = \varepsilon \wedge s' = s[[pdb(p) := \delta_p(s.pdb(p), \varepsilon)]]\}.$$

4.2e *Runs*: The model **SOS*** exhibits properties that can not be expressed *solely* by means of transition relation and state space. These properties are formalized by describing valid sequences of transitions, so-called runs.

We define a run to be an infinite sequence $r \in S^*$ such that $\forall n \in \mathbb{N}. (r^n, r^{n+1}) \in \Delta$. We define a (valid) **SOS*** run to be a run r such that $R(r)$, i.e. a sequence of states that is ‘covered’ by the transition relation, *and* that satisfies the predicate R (to be defined below).

Fairness between user applications is an important property. However, in our abstraction, the concrete scheduler is invisible. Thus, fairness can no longer be inferred by studying the

scheduler and the interrupt handling mechanism. Here, we use runs to explicitly state this property.

Intuitively, fairness may be expressed by claiming that all applications eventually get to do something, and thereby change their state. In SOS^* this is unfortunately not true as an application might wait infinitely long for another application to match its IPC operation. Thus, an application p might not progress, if it uses an infinite IPC call (in state s):

$$\begin{aligned} \text{possibly-no-progress} &\in S_p \times \text{pid.t} \rightarrow \mathbb{B} \\ \text{possibly-no-progress}(s, p) &\equiv \\ &(\exists h, m. \omega_p(s.pdb(p)) = \text{SND } h \ m \ \text{INFINITE} \wedge \text{vsnda?}(s, p, h, m)) \\ &\vee (\exists h, b. \omega_p(s.pdb(p)) = \text{RCV } h \ b \ \text{INFINITE} \wedge \text{vrcva?}(s, p, h, b)). \end{aligned}$$

In order to formalize fairness between user applications, we also need to characterize progress. In SOS^* , we can simply assume that a process p has progressed, between s and s' , if s' could be the result of applying δ_p , with some input i , to p in s :

$$\begin{aligned} \text{progress} &\in S_p \times \text{pid.t} \times S_p \rightarrow \mathbb{B} \\ \text{progress}(s, p, s') &\equiv \exists i. s' = s[[\text{pdb.}(p) := \delta_p(s.pdb(p), i)]]. \end{aligned}$$

Being able to identify processes that might not progress and processes that have progressed, we can state fairness between user applications as predicates that must be satisfied by all valid SOS^* runs.

The predicate $\text{app-fairness-finite}(r)$ is satisfied, if each finite operation (of each process) is finally served.

$$\begin{aligned} \text{app-fairness-finite} &\in S^* \rightarrow \mathbb{B} \\ \text{app-fairness-finite}(r) &\equiv \\ &\forall p, i. r^i.pdb(p) \neq \varepsilon \wedge \neg \text{possibly-no-progress}(r^i, p) \\ &\implies \\ &\exists j \geq i. \text{progress}(r^j, p, r^{j+1}). \end{aligned}$$

The predicate $\text{app-fairness-infinite}(r)$ is satisfied, if each process that infinitely often *could* make progress infinitely often *does* make progresses.

$$\begin{aligned} \text{app-fairness-infinite} &\in S^* \rightarrow \mathbb{B} \\ \text{app-fairness-infinite}(r) &\equiv \\ &(\forall p. (\forall i. \exists j \geq i. r^j.pdb(p) \neq \varepsilon \wedge \text{progress}(r^j, p, r^{j+1})) \\ &\implies \\ &(\forall k. \exists l \geq k. \text{progress}(r^l, p, r^{l+1}))). \end{aligned}$$

Now, R is simply the conjunction of all predicates defined over SOS^* runs. For now, this is only $\text{app-fairness-finite}$ and $\text{app-fairness-infinite}$. Thus:

$$\begin{aligned} R &\in (S \times \Sigma)^* \rightarrow \mathbb{B} \\ R(r) &\equiv \text{app-fairness-finite}(r) \wedge \text{app-fairness-infinite}(r). \end{aligned}$$

5. Reasoning about applications in SOS^*

Non-determinism in a model is often a source of difficulty when it comes to verification. This is simply because it implies a larger search space than in a purely sequential model. In many

cases non-deterministic models are still desirable as they allow to easily hide unnecessary details. In SOS^* , for example, there are two such hidden details:

- (i) The concrete scheduler is no longer visible. Instead, we chose non-deterministically which process to schedule next. This is commonly known as concurrency.
- (ii) There is no notion of time in SOS^* . However, the implementation might cancel an IPC operation because of a timeout. Thus, in order to represent such situations, we terminate IPC operations non-deterministically and return an appropriate timeout error.

Now, when verifying a property of SOS^* , we have to argue on all (fair) runs of the system. As this is a challenging task, we aim at reducing the search space. In this section, we present a set of general theorems which reduce non-determinism caused by concurrency and hence reduce the number of runs to analyse for correctness proofs.

5.1 Key observation

The key observation for concurrency is that certain application steps may be reordered. For example, if a certain file remains unchanged, then the order of two concurrent read operations (on this file) is irrelevant in terms of the overall execution. Thus, we may arrange these non-interfering operations in any order we like. In general, such reordering is sound, if the steps do not interfere with each other. Applying reordering repeatedly, we can separate parts of execution traces of two applications, in which both are communicating only with each other. This may be useful, when verifying a communication protocol. During the RPC data transfer protocol, for example, neither the server nor the client communicates with other processes. Another argument for reordering is that complex operations may be grouped together to atomic steps. For example, a server may be considered to respond to requests immediately, while, in the implementation, there are many steps necessary to compute the response.

5.2 Non-interference

For simplicity, in the remainder of this section, we assume that the mapping from handles to process ids is the identity. That means, we assume that the sets pid_t and hn_t are equal and that for all handles h and process ids p it holds:

$$\text{ph2p}(s, p, h) \neq \epsilon \implies \text{ph2p}(s, p, h) = h.$$

On the one hand, this simplification does not affect the concept of rights associated with handles. Furthermore, the kernel still maintains the information on which process knows which other processes. On the other hand, an implementation that uses such a simple mapping function may expose some system properties. For example, the number of currently running processes or the up-time of the system could be inferred. One could extend the theory presented here to a general mapping from handles to PIDs; formulae just get slightly larger.

Before we can express non-interference, we need to introduce some notations.

In the following we write $s \xrightarrow{\text{pid}, o} s'$, if there is a valid SOS^* transition from s to s' in which a system call $o \in \Omega_p$ of user process $\text{pid} \in \text{pid}_t$ is processed.

$$\begin{aligned} s \xrightarrow{\text{pid}, o} s' \equiv & \quad \Delta(s, s') \\ & \wedge \omega_p(s.\text{pdb}(\text{pid})) = o \\ & \wedge \exists i. \delta_p(s.\text{pdb}(\text{pid}), i) = s'.\text{pdb}(\text{pid}). \end{aligned}$$

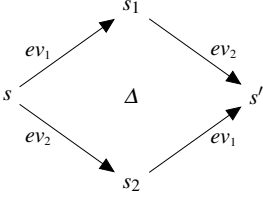


Figure 3. Swapping two non-interfering system call events.

In case of an IPC-*rendezvous* situation, it may be that two (different) processes, i.e. the sender and the receiver, progress simultaneously. This is because, in the successful case, both of them receive a result message within a single SOS* step. This case cannot be expressed in the form $s \xrightarrow{pid,o} s'$. Thus, we introduce the more general idea of *system call events*. A system call event $ev \in pow(pid_t \times \Omega'_p)$ is a set of at most two system calls:

$$s \xrightarrow{ev} s' \equiv \Delta(s, s') \wedge \forall (pid, o) \in ev. s \xrightarrow{pid,o} s'.$$

Now, we call two system call events ev_1, ev_2 *non-interfering*, denoted by $ev_1 \diamond ev_2$, if their execution can be swapped without modifying the result state (figure 3):

$$\begin{aligned} ev_1 \diamond ev_2 &\equiv (\forall s, s'. (\exists s_1. s \xrightarrow{ev_1} s_1 \wedge s_1 \xrightarrow{ev_2} s') \\ &\quad \Leftrightarrow \\ &\quad (\exists s_2. s \xrightarrow{ev_2} s_2 \wedge s_2 \xrightarrow{ev_1} s')). \end{aligned}$$

We extend the non-interference relation to sets of system call events:

$$Ev_1 \diamond Ev_2 \equiv \forall ev_1 \in Ev_1, ev_2 \in Ev_2. ev_1 \diamond ev_2,$$

and define the set of processes involved in an event by:

$$ap(ev) = \{p \mid \exists o. (p, o) \in ev\}.$$

Now, the following lemma defines the set of non-interfering system calls.

Lemma 1 (Non-interfering system calls). *Non-interfering system call events should not share involved processes: $ap(ev_1) \cap ap(ev_2) = \emptyset$. In the following we omit repeating this condition for each pair of system call events.*

Local transitions do neither interfere with local transitions nor with system calls of other processes:

$$ev \diamond \{(p, \epsilon)\}.$$

Portmapper lookup requests do not interfere with each other, but with portmapper register calls for the same procedure. Only portmapper register calls to different interfaces are non-interfering:

$$\begin{aligned} &\{(p_1, LUP \ id_i \ id_p)\} \diamond \{(p_2, LUP \ id'_i \ id'_p)\} \\ &(id_i = id'_i \implies id_p \neq id'_p) \implies \{(p_1, LUP \ id_i \ id_p)\} \diamond \{(p_2, REG \ id'_i \ id'_p)\} \\ &id_i \neq id'_i \implies \{(p_1, REG \ id_i \ id_p)\} \diamond \{(p_2, REG \ id'_i \ id'_p)\}. \end{aligned}$$

IPC system call events are non-interfering, if they do not match with each other, i.e. they do not lead to a rendezvous situation:

$$\begin{aligned} & (p_s, \text{SND } h_r \ m \ t_s) \in ev_1 \\ & \wedge (p_r, \text{RCV } h_s \ b \ t_r) \in ev_2 \\ & \wedge h_r \neq p_r \wedge h_s \neq p_s \\ & \implies \\ & ev_1 \diamond ev_2. \end{aligned}$$

The lemma above only defines the subset of the \diamond -relation relevant for the verification of RPC. Moreover, the \diamond -relation works solely on system call events and process identifiers, ignoring the current state. By extending the relation to states, one would, on the one hand, generate more complex non-interference assumptions, but, on the other hand, possibly identify more non-interfering actions. For example, in a file-system, a read operation and a write operation may be non-interfering due to missing rights of the writing process. These rights are not visible from the event but are encoded in the current state of the system.

5.3 Local reasoning

Applications which do not invoke system calls at all during a computation — i.e. a finite run — can also be verified in isolation. That means we can, for instance, prove the correctness of such code, which neither involves communication with other processes nor with the SOS, sequentially in traditional Hoare logic. Note that the considered properties over the code can be either state- or termination properties, i.e. the lemmas can not be applied to simplify the verification of arbitrary temporal claims over the system.

For stating the lemma corresponding to the idea formulated above, we need some more notation. We denote the execution of i consecutive and local transitions of an application by δ_i :

$$\delta_i^i(x) = \begin{cases} x & \text{if } i = 0 \\ \delta_i^{i-1}(\delta_p(x, \epsilon)) & \text{otherwise.} \end{cases}$$

Furthermore, we call an execution of an application *isolated until step i* , if no system call is invoked:

$$isolated?(x, i) \equiv \forall k < i. \omega_p(\delta_i^k(x)) = \epsilon.$$

Now, we can generalize each isolated computation to an arbitrary one.

Lemma 2 (Isolated computations). *The result of an isolated computation is neither influenced by the SOS nor by other processes:*

$$isolated?(x, i) \implies \forall r \in R, k \geq 0. r^k.pdb(p) = x \implies \exists l \geq k. r^l.pdb(p) = \delta_i^i(x).$$

Proof idea. The proof is based on the fairness of execution and on the fact that the SOS accesses an application only in response to a system call invoked by that application.

The following Lemma 3 gives us a tool to separate the verification of properties over two non-interfering sets of applications. In short, two sets of applications are non-interfering, if the system calls invoked by applications of the first set do not interfere with those invoked by applications of the second set.

First, we define, for a given run r and step number i , the set of system-call events, invoked between r^0 and r^i :

$$out(r, i) = \{(pid, o) \mid \exists j < i, pid. \omega_p(r^j).pdb(pid) = o\}.$$

We define properties in a pre/post condition form. We formulate these properties over a restricted set of processes, i.e. we assume that the initial configuration only contains processes that are in a given set. Now, given some set X of process identifiers, a set Ev of system call events, and predicates P, Q , we call P, Q valid over X and Ev , if for all valid runs, with initial states satisfying P , we will finally reach a state satisfying Q . Furthermore, processes may only invoke system call events in Ev :

$$\begin{aligned} X, Ev \vdash P, Q \equiv \\ \forall r \in R. P(r^0) \wedge \forall q \notin X. r^0.pdb(q) = \epsilon \implies \exists i. Q(r^i) \wedge out(r, i) \subseteq Ev. \end{aligned}$$

After the (above) preparatory work, we can now formulate a lemma which states that the correctness of computations, that do not interfere with each other, can be proven separately.

Lemma 3 (Separability of non-interfering computations).

$$\begin{aligned} & X, Ev_x \vdash P_x, Q_x \\ & \wedge Y, Ev_y \vdash P_y, Q_y \\ & \wedge Ev_x \diamond Ev_y \\ & \implies \\ & (X \cup Y), (Ev_x \cup Ev_y) \vdash P_x, Q_x \\ & \wedge (X \cup Y), (Ev_x \cup Ev_y) \vdash P_y, Q_y. \end{aligned}$$

The lemma above is extremely helpful for proving correctness of a communication protocol between two processes. Suppose the processes p and q only communicate with each other, i.e. they only invoke IPC send and IPC receive operations in which the sender and receiver handles are either p or q . Then we can generalize any correctness proof of this communication to arbitrary valid runs with arbitrary other processes. This is expressed in the following corollary, where the set $Ev_{p,q}$ denotes the above mentioned restriction to system call events:

COROLLARY 1 (Communication protocols).

$$\forall X \subseteq pid.t. \{p, q\}, Ev_{p,q} \vdash P, Q \implies \{p, q\} \cup X, \Omega_p \vdash P, Q.$$

5.4 Reordering

In the ideal case, reasoning on programs should be kept sequential and local as much as possible. Using Lemma 1, local steps of a process can always be grouped together. Furthermore, we can pretend that control is only switched in case of system calls.

This idea is expressed in the Reordering Theorem, claiming for each computation the existence of an equivalent and pure computation, in which local steps of different processes are (almost) never interleaved. Before stating the lemma, we first formalize the notion of *pure* computations. We say a computation is *pure* if control between processes is only switched due to a system call invocation or in case the currently executed process performs no more steps in the considered computation.

The latter case is expressed by the predicate $free?$, formally defined over a range $[s, e]$ of step numbers in a given run r :

$$free?(r, s, e, p) \equiv \forall s \leq i < e. r^i \xrightarrow{q, o} r^{i+1} \implies p \neq q.$$

Now, purity of a computation ranging from step s to step e can be formally defined by:

$$\begin{aligned} pure?(r, s, e) &\equiv \\ \forall s \leq i < e. r^i &\xrightarrow{p_1, o} r^{i+1} \xrightarrow{p_2, o'} r^{i+2} \wedge p_1 \neq p_2 \\ &\implies \\ \omega_p(r^{i+1}.pdb(p_1)) &\neq \epsilon \vee free?(r, i, e, p_1). \end{aligned}$$

Using the definition above our reordering theorem reads as follows:

Theorem 1 (Reordering). For a given run r and step numbers s and e we can always find a corresponding pure computation on a run r_p evaluating to the same state:

$$\forall r \in R, s, e. \exists r_p. pure?(r_p, s, e) \wedge r^e = r_p^e.$$

Proof sketch. The proof is done by induction on i . In the induction step we shift the local transition $i + 1$ to its appropriate block of isolated computations. This is done by a repeated application of swapping, which is possible for local transitions due to their non-interleaving nature.

The application of the reordering theorem leads to a significant reduction of the number of runs to analyse for proving a property over the whole system to be correct. Note that our reordering maintains liveness- and fairness properties, as always only a finite sequence is considered.

There is an intuitive way to think of pure computations: Consider system calls as function calls, where the functions are the processes from whose computation the result of the system call depends. For example, if process p invokes a receive from process q then we switch to the execution of process q until this one returns control again with a send call to p .

5.5 Summing up

The presented theorems can be combined to the following proof strategy for communicating applications: By Lemma 3 we can restrict our analysis to runs where only applications of interest take steps. Using the Reordering Theorem we can assume that those runs are all *pure*, i.e. isolated computations of applications are grouped together. Finally, properties over those isolated chunks are verified according to Lemma 2 in traditional Hoare logic.

6. Correctness of remote procedure calls

So far we have considered a model with concurrently executed processes running in the environment of the Simple Operating System. These processes communicate via IPC with each other and with the SOS.

Next, we introduce the programming language C0 and present, for applications written in C0, a more powerful communication mechanism than IPC: Remote Procedure Calls. RPC enables a process — the client — to invoke some service (i.e. a function) on a remote process — the server. From the client's point of view the invocation of the service should look similar to a local function call.

At compile time clients must know the names and the signatures of the services they intend to call. However, the location of the service (i.e. the name of the providing server) is then not necessarily known: During runtime the location of the service might change. Thus, we need a runtime mapping of service names to their locations. This mapping is provided by the SOS as described in Section 4.2: Servers register and clients look up services through invoking special SOS system calls.

After looking up the server handle, the client has to send the parameters of the call and, later on, receive the result. For each parameter and argument type we need to implement a pair of communication functions sending and receiving the data. For most types those functions can be implemented by single IPC calls. The only exception are pointer types, such as lists. Since IPC only supports copying data, which lies consecutively in the memory, for data structures that have to be traced over a chain of pointers, we need a more sophisticated mechanism.

Having these new communication functions, we define the protocol that the client must obey when requesting a service.

Next, we construct a simple example server in C0, identify the correctness criteria, and prove that it holds for our simple server. In short, the correctness criteria states that if the client strictly obeys the protocol it will eventually receive the result of the invoked call. The correctness statement and proof for the example server can be easily transferred and generalized to other server implementations.

6.1 The programming language C0

Our RPC mechanism is implemented as a set of libraries for the programming language C0. Here, we only sketch the main concepts of C0 needed to describe our RPC mechanism. A detailed description of syntax and semantics can be found in (Leinenbach 2008).

C0 has been developed for and is extensively used within the Verisoft project. In short, C0 is Pascal with C syntax, i.e. its type-system is sound and it supports references but no pointer-arithmetic. Syntax and semantics of C0 are fully formalized in Isabelle/HOL. The C0 semantics stack covers all common layers: from a low-level small-step semantics, via a big-step semantics up to a Hoare logic as an effective means of program verification (an instantiation of Schirmer's Hoare logic, see (Schirmer 2005)). Moreover, a C0 compiler has been implemented and formally verified (Leinenbach 2007). In the following, we instantiate the generic SOS* model, with C0 applications. Since applications are interleaved, only small step semantics is appropriate.

6.1a *Syntax and types:* In the remainder of this paper we assume that the following types are given:

- var_t denoting the set of variable identifiers in C0.
- $value_t$ denoting the set of values in C0.
- $expr_t$ denoting the set of C0 expressions.
- $stmt_t$ denoting the set of C0 statements.

All common imperative statements as skip, assignments, sequential composition, if-then-else constructs, loops and function calls are supported. We denote a function call, which assigns to variable $x \in var_t$ the result of the invocation of the function with name f_n and the parameters $arg_1, \dots, arg_n \in expr_t$ by $x = f_n(arg_1, \dots, arg_n)$.

C0 supports the basic types integer, unsigned integer, character, boolean and the composed types structures and arrays. Furthermore, pointer types are supported. Casts between different pointer types are not allowed.

6.1b *Configuration*: C0configurations are records of type $S_{c0} = \{mf : memconf.t, prog : stmt.t, pt : lib.t\}$ consisting of the three components:

- The memory configuration $mf \in memconf.t$ storing the values of all stack and heap variables. If some configuration c is given in which some expression e occurs, then we denote the value of e in configuration c by $va(c, e) \in value.t$.
- The program rest $pr \in stmt.t$. With $head(c.prog)$ and $tail(c.prog)$ we denote the head statement and the tail of the program rest, respectively.
- The procedure table $pt \in lib.t$, which is a set of pairs consisting of the function name and a function descriptor: $lib.t = pow(f-names \times f-desc.t)$. A function descriptor consists of a signature, a set of local variables and a body. In the following we will refer to sets of function descriptors simply as libraries.

The predicate $isLinked? \in S_{c0} \times lib.t \rightarrow \mathbb{B}$ indicates if the program was linked to the given library: $isLinked?(c, lib) \equiv lib \subseteq c.pt$.⁵

6.1c *Semantics*: In this paragraph we instantiate the generic process model of SOS* by the C0small-step semantics. For that the state space is instantiated to $S_{c0} \subset S_p$. Furthermore, we define the transition function $\delta_{c0} \subset \delta_p$ and the output function $\omega_{c0} \subset \omega_p$.

(i) *Transition function*: The type of the transition function δ_{c0} is:

$$\delta_{c0} \in S_{c0} \times \Sigma_p \rightarrow S_{c0}.$$

An exhaustive definition of its local part, i.e. of $\delta_{c0}(c, \epsilon)$, can be found in (Leinenbach 2008). The non-local parts involve invocations of system calls. In the program rest system call invocations appear as ordinary function calls. However, they are not part of the procedure table. Rather, the semantical effects of system calls on a C0application is specified as one atomic step in δ_{c0} . For example, a portmapper lookup call has the following effect on a C0application:

$$c.prog = x = sc_pm_lkp(iid, precid) \wedge \delta_{c0}(c, SUCC-LUP h_s) = c' \implies va(c', x) = h_s.$$

IPC send and IPC receive system calls are slightly more complicated. Since C0forbids type casts, we have to implement a pair of IPC functions for each type of data that should be sent or received. For simple types, IPC functions can be easily implemented and they can be assumed by the C0programmer. Simple types cover all C0types, in which no nested pointer types occur. The type restriction results from the fact that the basic IPC mechanism provided by the kernel only supports copying consecutive data chunks. Hence, types requiring pointer chasing can not be interpreted at this level.

(ii) *Output function*: If the head statement of the program rest is not a system call, the output function ω_{c0} returns ϵ . Otherwise, a corresponding output message is constructed. For example, a configuration c with $head(c.prog) = x = sc_ipc_send_int(h_{exp}, m_{exp}, to_{exp})$ generates the output:

$$\omega_{c0}(c) = \text{SND } t-hn(va(c, h_{exp})) \ t-m(va(c, m_{exp})) \ t-to(va(c, to_{exp})).$$

The functions $t-hn$, $t-m$ and $t-to$ map C0 values to their corresponding interpretation in the SOS* abstraction. For example $t(va(c, t_{exp}))$ maps any timeout value which is not equal to

⁵Indeed linking is a much more complicated process, as types and global variables have to be defined unambiguously.

the integer constant INFINITE to the constant FINITE. In the following, for simplicity, we will omit mentioning these translations, i.e. we write $va(\dots)$ instead of $t\text{-}x(va(\dots))$.

6.1d *Doubly-linked lists in C0*: Lists are provided to the C0 programmer by means of a generic library. For a given type T this library includes a type defining a doubly-linked list over elements of T . Furthermore, operations on lists are provided, such as creation, and element insertion and deletion. These operations have been formally verified against abstract lists (Starostin 2006) (which are basically sequences on T).

6.2 Signatures of services

The portmapper, maintained in the SOS, is not aware of the services' signatures. However, at compile time, the client must know the signatures and names of the services it intends to call. The signature of a service is given by the type of the input parameter and the type of the result:⁶

$$service_sig_t = \{arg : rpc_type_t, res : rpc_type_t\}.$$

Here, rpc_type_t denotes a subset of all possible C0 type descriptors. The formal definition of this subset is given in (Shadrin 2006). In short, rpc_type_t contains all simple and structured types. Pointer types are only allowed as part of the pre-defined doubly-linked lists.

Services are organized in so-called interfaces. The signature of an interface consists of a name and a mapping from procedure ids to corresponding signatures:

$$ifc_t = \{iid : iid_t, procs : procid_t \rightarrow service_sig_t\}.$$

In the remainder of this paper we use the terms interface signatures and interfaces interchangeably.

6.3 Portmapper correctness

The next lemma is a property of the interaction of different portmapper calls. This lemma states that, after a server has registered a service, any client that is looking up this service will finally receive the correct handle. Thus, it expresses the programmer's point of view of interacting portmapper calls.

Lemma 4. (Correctness of interacting portmapper register- and lookup calls)

Suppose there exists a step l at which a server with PID p_s wants to register the service (id_i, id_p) . Further suppose that the server will not try to unregister the service, nor any other process will try to register the same service at any step during the run:

$$\begin{aligned} & \forall r \in R, id_i, id_p, p_s, l. \\ & \quad \omega_{c0}(r^l.pdb(p_s)) = \text{REG } id_i \ id_p \\ & \quad \wedge (\forall k, p'_s. \\ & \quad \quad \omega_{c0}(r^l.pdb(p'_s)) = \text{REG } id_i \ id_p \implies p'_s = p_s \\ & \quad \quad \wedge \neg \omega_{c0}(r^l.pdb(p_s)) = \text{UNREG } id_i \ id_p) \\ & \implies \end{aligned}$$

⁶In the context of RPC, the formalism describing the service signatures is called Interface Definition Language (IDL).

Then, finally, a step in the run r is reached, after which the following holds: whenever some process with pid p_c looks up the service, it will receive a success message with the handle to the server p_s .

$$\begin{aligned} & (\exists k > l. \forall j_1 > k. \\ & \omega_{c0}(r^{j_1}.pdb(p_c)) = \text{LUP } id_i id_p \\ & \implies (\exists j_2 > j_1. \delta_{c0}(r^{j_2}.pdb(p_c), \text{SUCC-LUP } pp2h(p_c, p_s)) = r^{j_2+1}.pdb(p_c))). \end{aligned}$$

Proof sketch. By applying Lemma 3, we can pretend that the lookup call is executed immediately after the execution of the register call. We only have to ensure that no system calls interfering with any of both portmapper calls is executed meanwhile. This follows from the assumptions of Lemma 4.

6.4 Sending and receiving data structures

As described before, the IPC mechanism, provided in VAMOS does not fulfill the requirements for RPC. Therefore, C0applications are supported with a library containing the implementation of RPC send and RPC receive functions. In the following we call these functions, RPC primitives.

These RPC primitives depend on the types of the data structure to be transmitted. Hence, we need a library generator, that produces, for a given interface signature $itfc$, the C0implementation of functions for sending and receiving RPC messages.

The implementation is simple. For the sending side, non-list data structures are sent via a single IPC operation. List data structures are chased and sent element by element (recursively) via RPC. The receiver reconstructs the list, by chaining these elements together. Termination of both, the sending part and the receiving part, follows from the well-formedness of the list implementation.

This mechanism of packing and unpacking is an implementation detail and should not be visible to programmers of RPC servers and especially not to programmers of clients. Therefore, later on, we provide the programmers with a set of properties over the execution of those primitives, abstracting from their concrete implementation.

6.4a The interface compiler: The interface compiler takes as input an interface signature and generates a C0library that contains the implementation of the corresponding RPC functions.

The function $genRPCprim$ abstracts the implementation of the SOS interface compiler, i.e. it represents the semantics of the interface compiler. Since we are only interested in properties over the primitives, both the concrete code implementing $genRPCprim$ and the specification defining the output of $genRPCprim$, is omitted:

$$\begin{aligned} & genRPCprim \in itfc.t \rightarrow lib.t \\ & genRPCprim(itfc) \equiv librpc_{itfc}, \end{aligned}$$

where $librpc_{itfc}$ contains for each type $T \in \{p.arg, p.res \mid p \in range(itfc.procs)\}$ occurring in the signature of one of the procedures of $itfc$, two functions:

1. $RPCsend_T(h_r, arg, to)$ This primitive sends the argument $arg \in T$ to the application with handle h_r . The value to specifies some timeout value.⁷ The function returns a value

⁷Note that in the implementation, in contrast to SOS*, arbitrary integer values (except the constant INFINITE) can be specified for a finite timeout.

of type $rpcerr_t$, which indicates if the send operation was successful, a timeout occurred, or the specified handle is not valid:

$$rpcerr_t = \{SUCC, TIMEOUT, INVALID\}.$$

2. $RPCrecv_T(h_s, to)$ This primitive is called to receive an argument of type T from the application with the handle h_s . This call returns a value, which is a structure of type $rpcrcv_T_t$ containing two components. The first component indicates if the operation was successful, and the second one holds the received data:

$$rpcrcv_T_t = \{status : rpcerr_t, data : T\}.$$

These RPC primitives hide the details of sending and receiving parameters and results. This, for example, allows us to extend RPC to invocation of procedures via the Internet, without changing the signature of the RPC primitives. Even the specification would remain the same.

6.4b *Predicates signalling RPC primitives:* In order to specify the behaviour of RPC primitives, we need to introduce a number of auxiliary predicates and functions that are used to describe the current state in the execution of the RPC primitives.

The predicate $beforeS?(c, T, resv, h_r, arg, to)$ evaluates to true if:

- The head of the program rest of c is an RPC send primitive, where T is the type of the parameter arg to be transferred, h_r is the handle of the receiver, and to is some timeout value.
- The C0machine c was linked with an RPC library of an interface containing the argument type T .
- The result of the primitive is assigned to the C0variable $resv$.

Formally we get:

$$\begin{aligned} beforeS? &\in S_{c0} \times rpc_type_t \times var_t \times hn_t \times expr_t \times \mathbb{N} \rightarrow bool \\ beforeS?(c, T, resv, h_r, arg, to) &\equiv \\ \exists ifc \in ifc_t, h_{expr}, to_{expr} \in expr_t. & \\ (RPCsend_T, ?) \in librpc_{ifc} \wedge isLinked?(c, librpc_{ifc}) \wedge & \\ head(c.prog) = resv = RPCsend_T(h_{expr}, arg, to_{expr}) \wedge & \\ va(c, h_{expr}) = h_r \wedge & \\ va(c, to_{expr}) = to. & \end{aligned}$$

The predicate $beforeR?(c, T, resv, h_s, to)$ evaluates to true, if:

- The head of the program rest of c is a receive RPC primitive, where T is the type of the parameter to be received, h_s is the handle of the sender and to is the timeout.
- The result of the primitive is assigned to the C0variable $resv$.

Formally we get:

$$\begin{aligned} beforeR? &\in S_{c0} \times rpc_type_t \times var_t \times hn_t \times \mathbb{N} \rightarrow bool \\ beforeR?(c, T, resv, h_s, to) &\equiv \end{aligned}$$

$$\begin{aligned}
&\exists \text{ itfc} \in \text{itfc}_t, h_{\text{expr}}, to_{\text{expr}} \in \text{expr}_t. \\
&(\text{RPCrecv_T}, ?) \in \text{librpc}_{\text{itfc}} \wedge \text{isLinked?}(c, \text{librpc}_{\text{itfc}}) \wedge \\
&\text{head}(c.\text{prog}) = \text{resv} = \text{RPCrecv_T}(h_{\text{expr}}, to_{\text{expr}}) \wedge \\
&\text{va}(c, h_{\text{expr}}) = h_s \wedge \\
&\text{va}(c, to_{\text{expr}}) = to.
\end{aligned}$$

For a given C0machine c and a function name fn , the predicate duringFC? indicates that c is currently executing a call of the function fn :

$$\text{duringFC?} \in S_{c_0} \times \text{fname}_t \rightarrow \mathbb{B}.$$

A formal description of duringFC? is omitted.

The function finishedFC indicates the time when the execution of a function is finished. For a given run r , step i , process id pid and a function name fn , it returns the first step, after i , in which the execution of function fn has finished:

$$\begin{aligned}
&\text{finishedFC} \in R \times \mathbb{N} \times \text{pid}_t \times \text{fname}_t \rightarrow \mathbb{N} \\
&\text{finishedFC}(r, i, p, fn) = \\
&\min\{j \mid j \geq i. \text{duringFC?}(r^{j-1}.\text{pdb}(p), fn) \wedge \neg \text{duringFC?}(r^j.\text{pdb}(p), fn)\}
\end{aligned}$$

Finally, we need one predicate that compares two states of a single C0machine. The predicate $\text{changed?}(c, c', v)$ evaluates to true, if in state c a function was called and in c' the same function returned. Furthermore, the result of the function call evaluates to the value v .⁸

$$\begin{aligned}
&\text{changed?} \in S_{c_0} \times S_{c_0} \times \text{value}_t \rightarrow \mathbb{B} \\
&\text{changed?}(c, c', v) \equiv \\
&\exists x, \text{fn}. c.\text{prog} = x = \text{fn}(\dots) \wedge c'.\text{prog} = \text{tail}(c.\text{prog}) \wedge \text{va}(c', x) = v.
\end{aligned}$$

6.4c Properties of RPC primitives: Instead of defining a new model, the semantics of the RPC primitives is given through a small set of theorems, describing *relevant parts* of their behaviour. *Relevant parts* includes all properties which are needed for proving correctness of programs that use the RPC libraries. Similar to the correctness statement of the portmapper calls, the following lemmas should provide the programmer's point of view of the primitives. Thus, they are to be understood as specifications of the interface compiler's output. Given the concrete definition of idgenRPCprim , they can be discharged. Lemma 5 states that RPC primitives always terminate if a finite timeout was specified. Lemma 6 states that if two applications try to communicate via RPC send or RPC receive, then, finally, either the message will be transferred correctly or a timeout occurs. The lemmas are not proved for a concrete interface compiler. Rather the proof methodology is outlined.

⁸Only functions with no side-effects are considered.

Lemma 5. (Termination of RPC function calls)

This lemma states that every invocation of either the RPC send or RPC receive function terminates (possibly unsuccessfully, for example with a timeout error), in case the timeout value is finite:

1. $\forall r \in R, i \in \mathbb{N}, sn \in pid_t, to \in \mathbb{N}.$
 $beforeS?(r^i.pdb(sn), T, \dots, to) \wedge to \neq \text{INFINITE} \implies$
 $\exists j > i. j = finishedFC(r, i, sn, \text{RPCsend_T})$
2. $\forall r \in R, i \in \mathbb{N}, receiver \in pid_t, to \in \mathbb{N}.$
 $beforeR?(r^i.pdb(rc), T, \dots, to) \wedge to \neq \text{INFINITE} \implies$
 $\exists j > i. j = finishedFC(r, i, rc, \text{RPCrecv_T}).$

Proof method. The C0 statements appearing in the implementation of the send- and receive primitives are either ordinary or IPC communication statements with finite timeout. By the fairness property of runs, we can deduce that each statement is finally executed (since at each configuration the C0 machine can make progress). The only remaining (general) source of non-termination is a type T defining an infinite data structure to process. However, such a type can only be defined by using pointers — and the only pointer structures allowed are well-formed and finite lists.

Lemma 6. Correctness of RPC Communication Primitives

Suppose, there is a sender sn and a receiver rc which want to communicate with each other using RPC primitives. The applications with ids sn and rc invoke the RPC send and RPC receive functions at steps i and j , respectively. Furthermore, suppose, the type of the transferred parameter arg is T .

$$\begin{aligned} &\forall r \in R, i, j \in \mathbb{N}, sn, rc \in pid_t, T \in type_t, \\ &res_s, res_r \in var_t, arg_s \in expr_t, to_s, to_r \in \mathbb{N}. \\ &beforeS?(r^i.pdb(sn), T, res_s, pp2h(r^i, sn, rc), arg_s, to_s) \wedge \\ &beforeR?(r^j.pdb(rc), T, res_r, pp2h(r^j, rc, sn), to_r) \wedge \\ &\implies \end{aligned}$$

Then there exist steps i' and j' in the run, at which the send and receive function calls have returned.

$$\begin{aligned} &(\exists i', j' \in \mathbb{N}. \\ &i' = finishedFC(r, i, sn, \text{RPCsend_T}) \wedge \\ &j' = finishedFC(r, j, rc, \text{RPCrecv_T}) \wedge \end{aligned}$$

and if furthermore the following two conditions hold:

- If the sender is waiting for the receiver, then the receiver will not try to receive messages (via IPC) from the sender until it invokes RPC receive.:

$$\begin{aligned} &(i < j \implies \forall k \in \mathbb{N}. i \leq k \leq j \implies \\ &\omega_{c0}(r^k.pdb(rc)) \neq \text{RCV } pp2h(r^k, rc, sn) \dots \wedge \\ &\omega_{c0}(r^k.pdb(rc)) \neq \text{RCV HN_NONE } \dots) \wedge \end{aligned}$$

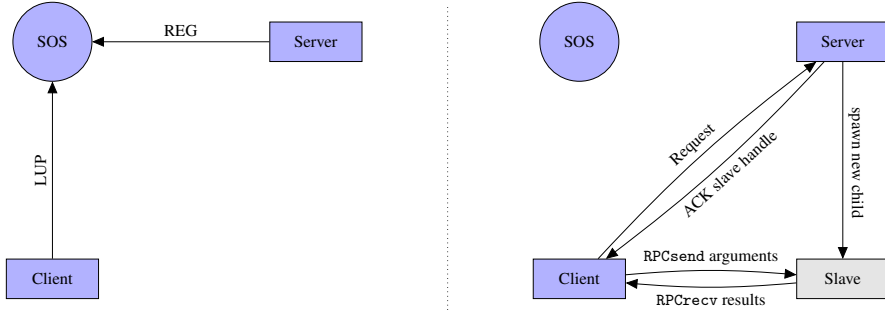


Figure 4. A sample execution of the RPC protocol — *left*: service register and lookup, *right*: server spawns child to handle client request.

- If the receiver is waiting for the sender, the sender will not try to send messages (via IPC) to the receiver until it invokes RPC send:

$$(j < i \Rightarrow \forall k \in \mathbb{N}. j \leq k \leq i \Rightarrow \omega_{co}(r^k.pdb(sn)) \neq \text{SND } pp2h(r^k, s, r) \dots) \implies$$

Then either the receiver will have received the correct data (sent by the sender) and a success message is reported to sender and receiver, or, in case the timeout was not set to infinite, a timeout could have occurred:

$$(changed?(r^i.pdb(sn), r^{i'}.pdb(sn), \text{SUCC}) \wedge changed?(r^j.pdb(rc), r^{j'}.pdb(rc), (status = \text{SUCC}, data = va(r^i.pdb(sn), arg_s))) \vee$$

$$(to_s \neq \text{INFINITE} \Rightarrow changed?(r^i.pdb(sn), r^{i'}.pdb(s), \text{TIMEOUT}) \wedge to_r \neq \text{INFINITE} \Rightarrow changed?(r^i.pdb(rc), r^{i'}.pdb(r), [[status = \text{TIMEOUT}, data = ?]])) \vee$$

Proof method. By using Lemma 3 we can prove the correctness of Lemma 6 ignoring all applications except the sender and the receiver. Applying the Reorder Theorem we can assume that control between sender and receiver is only switched at IPC-call borders. So, we can verify code segments between the invocation of two consecutive IPC operations purely sequentially in Hoare logic.

6.5 RPC client protocol

Having the new primitives of communication, we can define the protocol that the client must obey when requesting a service. Under a protocol we understand the sequence of messages that are sent and received by the client. This protocol should *not* depend on the concrete server architecture, but only on the service name. Figure 4 depicts the interaction of a client and a server.

The RPC client protocol is fairly simple. It consists of the following five steps:

- Request the location of the service via a portmapper call.

- Send the request to the server via IPC. This request contains the id of the desired (remote) procedure.
- Wait for an answer of the server. The answer contains a constant denoting whether the request was successful or not and the handle of the application processing the call (either the server itself or a newly spawned slave application dedicated for the request).
- Send, via RPC, the parameter of the service to the application referenced by the received handle.
- Wait, via RPC, for the result of the call.

6.5a *CallService library for C0 applications:* In order to release the programmer from manually implementing each step of the protocol, we provide him with a library compiler:

$$\begin{aligned} \text{genCSprim} &\in \text{itfc}_t \rightarrow \text{lib}_t \\ \text{genCSprim}(\text{itfc}) &= \text{libService}_{\text{itfc}}. \end{aligned}$$

This compiler takes as input an interface signature and returns a C0library containing the definition of the function `callService_iid_prcid`, for each service $(iid, prcid)$ in the given interface. Now, this function joins the individual steps of the above described client protocol into a single function call. The function `callService_iid_prcid` takes as parameters the argument of the service and a timeout value. Given the return type T of the service $(iid, prcid)$, `callService_iid_prcid` returns a value of type csrcv_T :

$$\begin{aligned} \text{csrcv}_T &= \{res : \text{cserr}_t, data : T\}, \text{ where} \\ \text{cserr}_t &= \{\text{SUCC}, \text{NOTREG}, \text{DENIED}\}. \end{aligned}$$

6.5b *Formal description of the client protocol:* For defining the RPC client protocol formally, we need three more predicates. The predicates *beforeCS?*, *request?* and *recvAnsw?* are used to characterize different aspects of the first three steps of the client protocol.

The predicate *beforeCS?* $(c, iid, prcid, arg, to, resv)$ evaluates to true if:

- The head of the program *rest* of c is the function call `callService_iid_prcid`, where $(iid, prcid)$ denotes the name of the service to invoke, *arg* denotes the argument to be passed to the server, and *to* is the timeout value used during communication with the server.
- The result of the function call is assigned to the C0variable *resv*.

Thus, we get the definition:

$$\begin{aligned} \text{beforeCS?} &\in S_{c_0} \times \text{iid}_t \times \text{prcid}_t \times \text{expr}_t \times \mathbb{N} \times \text{var}_t \rightarrow \mathbb{B} \\ \text{beforeCS?}(c, iid, prcid, arg, to, resv) &\equiv \\ &\exists \text{itfc} \in \text{itfc}_t. \text{itfc.iid} = iid \wedge \text{prcid} \in \text{dom}(\text{itfc.procs}) \wedge \\ &\text{isLinked?}(c, \text{librpc}_{\text{itfc}}) \wedge \text{isLinked?}(c, \text{libservice}_{\text{itfc}}) \\ &\text{head}(c.\text{prog}) = \text{resv} = \text{callService_iid_prcid}(arg, to) \end{aligned}$$

The predicate *request?* $(c, h_s, prcid)$ evaluates to true if:

- The C0 application c is sending (via IPC) a request to invoke procedure *prcid* on the server with handle h_s .

- Directly after sending the request, the application will be waiting (via IPC) for an answer. The corresponding receiving buffer b must be large enough to store the answer of type integer (four bytes).

Thus, we get the definition:

$$\begin{aligned}
request? &\in S_{c0} \times hn_t \times prcid_t \rightarrow \mathbb{B} \\
request?(c, h_s, prcid) &\equiv \exists msg \in byte_t^*, b \geq 4. \\
\omega_{c0}(c) &= SND \ h_s \ msg \ INFINITE \ \wedge \\
byte2prcid(msg) &= prcid \ \wedge \\
\omega_{c0}(\delta_p(c, SUCC)) &= RCV \ h_s \ b \ INFINITE .
\end{aligned}$$

Here, the function $byte2prcid \in byte_t^* \rightarrow prcid_t$ translates a byte sequence to a procedure id.

The predicate $requestAnsw?(c, c', res, h_s)$ evaluates to true if:

- The C0application in state c' has successfully returned from an IPC receive call in state c .
- The message res and the handle h_s were received. In case the IPC message was an RPC request, res should encode whether the request was acknowledged or not, i.e. its value is either ACK and NACK .ACK and NACK are constants of type \mathbb{Z}_{32} . If an ACK message was received, h_s denotes the handle of the application that will serve the request.

Thus, we get the definition:

$$\begin{aligned}
requestAnsw? &\in S_{c0} \times S_{c0} \times \mathbb{Z}_{32} \times hn_t \rightarrow \mathbb{B} \\
requestAnsw?(c, c', res, h_s) &\equiv \\
\exists msg \in byte_t^*. \delta_p(c, SUCC-RCV \ h_s \ msg) &= c' \ \wedge \\
byte2ack(msg) &= res.
\end{aligned}$$

Here, the function $byte2ack \in byte_t^* \rightarrow \{ACK, NACK\}$ translates a byte sequence to a server response.

After defining these auxiliary predicates we can now specify the intended behaviour of the *callService* library. The code of any C0 library claiming to implement the client protocol, must fulfill Lemma 7.

In the following lemma we use the abbreviations: $T_{arg} = itfc.procs(prcid).arg$ and $T_{res} = itfc.procs(prcid).res$.

Lemma 7. (behaviour of callService)

$$\forall r \in R, i_0 \in \mathbb{N}, iid \in iid_t, prcid \in prcid_t, c \in pid_t, to \in \mathbb{N}, arg \in expr_t.$$

1. Assume that the next statement to execute of the client process c is an invocation of a service $(iid, prcid)$. Then, finally, the client will be requesting the location of the service from the SOS.

$$\begin{aligned}
beforeCS?(r^{i_0}.pdb(c), iid, prcid, arg, INFINITE, res_{cs}) \\
\implies (\exists i_1 > i_0. \omega_{c0}(r^{i_1}.pdb(c)) = LUP \ iid \ prcid \ \wedge .
\end{aligned}$$

2. If the portmapping request was successful and the service location h_s was returned, the client will send an RPC request to the server:

$$\begin{aligned} & (\forall i_2. i_2 = \min\{j > i_1 \mid \text{progress}(r^{j-1}, c, r^j)\} \wedge \\ & \delta_{c0}(r^{i_2-1}.pdb(c), \text{SUCC-LUP } h_s) = r^{i_2}.pdb(c) \\ & \implies (\exists i_3 > i_2. \text{request?}(r^{i_3}.pdb(c), h_s, prcid) \wedge . \end{aligned}$$

3. If the answer to the request was positive, the handle h_s of the process that will serve the call is received. The client will next try to send the arguments of the call via RPC to the process h_s :

$$\begin{aligned} & (\forall i_4 > i_3. i_4 = \min\{j \mid \text{requestAnsw?}(r^{i_3}.pdb(c), r^j.pdb(c), b, h_s)\} \wedge b = \text{ACK} \\ & \implies (\exists i_5 > i_4. \text{res_send. beforeS?}(r^{i_5}.pdb(c), T_{arg}, \text{res_send}, h_s, \text{INFINITE}) \wedge . \end{aligned}$$

4. If the RPC sending of the argument was successful, the client will wait (via RPC) for the result of the call.

$$\begin{aligned} & (\forall i_6 > i_5. i_6 = \text{finishedFC}(r, i_6, c, \text{RPCsend_T}_{arg}) \wedge \\ & \text{va}(r^{i_5}.pdb(c), \text{res_send}) = \text{SUCC} \\ & \implies (\exists i_7 > i_6. \text{res_rcv. beforeR?}(r^{i_7}.pdb(c), T_{res}, \text{res_rcv}, h_s, \text{INFINITE}) \wedge . \end{aligned}$$

5. If the client successfully receives the result, then `callService_iid_prcid` will terminate and return the success message as well as the received data.

$$\begin{aligned} & (\forall i_8 > i_7. i_8 = \text{finishedFC}(r, i_8, c, \text{RPCrcv_T}_{res}) \\ & \wedge \text{va}(r^{i_8}.pdb(c), \text{res_rcv.res}) = \text{SUCC} \\ & \implies (\exists i_9 > i_8. i_9 = \text{finishedFC}(r, i_9, c, \text{callService_iid_prcid}) \wedge \\ & \text{changed?}(r^{i_9}.pdb(c), r^{i_9}.pdb(c), \\ & \quad [[\text{res} = \text{SUCC}, \text{data} = \text{va}(\text{res_rcv.data})]])))))))). \end{aligned}$$

Proof method. The lemma has not been proven for a concrete library generator. We rather outline the proof methodology, here. By applying Lemma 2 we can generalize properties proved locally to properties that also hold in the interleaved setting of SOS*. That means, the correctness of a given `callService` library can be proven almost completely sequentially (except when waiting for results from the SOS or other applications) and in traditional Hoare logic.

For example, for step 1 of Lemma 7, it suffices to prove the following, local property over the library code: $\text{beforeCS?}(c, iid, prcid, arg, \text{INFINITE}, \text{res}_{cs}) \implies \exists i. \omega_{c0}(\delta_i^j(c)) = \text{LUP } iid \text{ prcid}$.

6.6 Building a server and proving its correctness

Now we have everything we need for building arbitrary servers and specify their correctness: We can register services, send and receive messages of dynamic length and we know how a *correct* client should behave. In the next section, we give the implementation of a simple server and prove its correctness. Under correctness we understand, that whenever the client obeys the above described protocol, it will eventually receive the result of the invoked call.

Even so, one could implement the same interface of the Math Server through many different architectures — for instance, a simple architecture might handle one request after the other, whereas a more advanced architecture might spawn a new slave process for each request — all architectures have to fulfill the same correctness criteria. This criteria represents the client's view on the server.

6.6a *An example: MathServer:* The *MathServer* implements the interface *mathitfc* with only one procedure. This procedure takes two integers and returns their product. Hence, the parameter type of the service is given through:

$$rpcmul_t = \{x : \mathbb{Z}_{32}, y : \mathbb{Z}_{32}\}.$$

The interface id is `MATH` and the procedure id is `MUL`. *mathitfc* is formally described through:

$$mathitfc = [[iid = MATH, procs = (\lambda x. \mathbf{if} \ x = MUL \ \mathbf{then} \ (arg = rpcmul_t, res = \mathbb{Z}_{32}))]].$$

If we program a server, we first have to link it with the three libraries `Libvamos`, `Libsos` and `librpc`. The library `Libvamos` provides us with the functions for sending requests and receiving answers via IPC: `sc_ipc_send_int` and `sc_ipc_rcv_rpcreq_t`.⁹ The library `Libsos` provides us with the portmapping call `sc_pm_reg`, and `librpcmathitfc` contains the RPC send and RPC receive functions needed to transmit arguments and results of the `MUL` procedure: `RPCrcv_rpcmul_t` and `RPCsend_int`.

The implementation of *MathServer* is pretty simple. Listing ?? gives the code of the main function of *MathServer*. First the service is registered in the SOS. Then the main loop is entered. There, the server first waits for incoming requests via IPC. If a request to execute service `MUL` is received, the server sends an acknowledgment and waits for an RPC message containing the argument of the call. If, finally, the argument arrives, the result is calculated and sent to the client via RPC.

6.6b *Correctness of math server:* The correctness theorem assumes that all applications requesting the `MUL` service obey the RPC client protocol. This condition seems to be too strict. The reason for that condition, is that, meanwhile the server waits for the arguments of a remote call, it will block — possibly infinitely long. Thus, it must *trust* that the client eventually will send or receive the data.

Alternatively, the server could specify a finite timeout when waiting for the client. This solution is cumbersome when it comes to verification: It could happen that the client, although obeying the protocol, is simply *too slow* for sending the data. In practice one could easily try to estimate a robust timeout value for the server. However, proving that this value will never lead to a timeout is much harder, since it depends on the implementation of the hardware, the kernel, etc (for a detailed worst-case execution time analysis, covering a complete system stack, refer to (Knapp & Paul 2007)).

Yet, if we know that *all* clients during a run will obey the protocol, then we can set the timeout values of the server to infinite and hereby prove correctness *and* termination for all calls.

The above mentioned condition is formulated in the predicate *goodclients*. It is satisfied if, during the whole run, every client will only communicate with servers, in the context of a

⁹`Libvamos` contains `C0` macros, for generating IPC functions of the desired type.

```

(0)  dummy = sc_pm_reg (MATH, MUL);
(1)  while (true) {
(2)      // open receive for service requests
(3)      dummy = sc_ipc_rcv_rpcreq_t (HN_NONE, request, client, INFINITE);
(4)      // dispatching
(5)      if (*request.prcd == MUL) {
(6)          // sending acknowledgement message
(7)          dummy = sc_ipc_send_int (*client, ACK, INFINITE);
(8)          // receiving arguments of the client with infinite timeout
(9)          clientmsg = RPCrcv_rpcmul_t (*client, INFINITE);
(10)         // computing the result of the requested service
(11)         res = clientmsg.data.x * clientmsg.data.y;
(12)         // sending the result to the client
(13)         dummy = RPCsend_int (*client, res, INFINITE);
(14)     }
(15)     // in case of a call to a procedure other than MUL send denial
(16)     else {
(17)         dummy = sc_ipc_send_int (*client, NACK, 10);
(18)     }
(19) }

```

Figure 5. Implementation of Math Server — variables *client* and *request* are pointers to integers and * denotes the dereferencing operator.

callService function call. With other words, it is true, if all clients communicating with the server obey the RPC client protocol.

$$goodClients? \in R \times iid.t \times prcid.t \rightarrow \mathbb{B}$$

$$goodClients?(r, iid, prcid) \equiv$$

$$\forall i \in \mathbb{N}, s, c \in pid.t.$$

$$\begin{aligned}
 & r^i.pmdb.serv(iid) = pp2h(r^i, OSPID, s) \wedge (iid, prcid) \in r^i.pmdb.reg \wedge \\
 & \omega_{c0}(r^i.pdb(c)) = SND \ pp2h(r^i, c, s) \dots \\
 & \implies duringFC?(r^i.pdb(c), callService_iid_prcid).
 \end{aligned}$$

Now, we can express the correctness of the our Math Server. It states, that if all clients obey the client protocol, finally the MUL service will be registered and any client requesting it will finally receive the correct answer:

Lemma 8. (Correctness of math Server)

If, during a run, the *MathServer* application is started, all possible clients are *good* and no other application tries to register the interface *mathitfc*:

$$\forall r \in R, s, c \in pid.t, i \in \mathbb{N}.$$

$$\begin{aligned}
 & r^i.pdb(s) = MathServer \wedge goodClients?(r, MATH, MUL) \wedge \\
 & (\forall k \in \mathbb{N}, s' \in pid.t. \omega_{c0}(r^k.pdb(s')) = REG \ MATH \ ? \implies s' = s)
 \end{aligned}$$

$$\implies,$$

then a step *k* in the run is reached, after which the following holds: if some application calls the service (MATH, MUL) with integers *x* and *y* as parameters, then, finally, this call will return with the correct result, i.e. the product of *x* and *y*:

$$(\exists k > i. \forall j > k, arg \in expr.t.$$

$$beforeCS?(r^j.pdb(c), MATH, MUL, arg, INFINITE, \dots) \implies$$

$$(\exists j' > j. j' = finishedFC(r, j, c, callService_MATH_MUL) \wedge$$

$$changed?(r^j.pdb(c), r^{j'}.pdb(c),$$

$$[[res = SUCC, data = va(r^j.pdb(c), arg.x) * va(r^{j'}.pdb(c), arg.y)]])).$$

Proof. Since in SOS* a fair scheduling is assumed, eventually the first line in Listing ?? is executed. At that point all assumptions of Lemma 4 are satisfied and we conclude that finally (at step k) the service is registered in the portmapping component of the SOS. On the client side, it follows from Lemma 7 part 1 that the client will request the SOS for the address of the service:

$$\begin{aligned} & \dots \text{beforeCS?}(r^j.\text{pdb}(c), \text{MATH}, \text{MUL}, \text{arg}, \text{INFINITE}, \dots) \\ & \implies \exists j_1 > j. \omega_{c0}(r^{j_1}.\text{pdb}(c)) = \text{LUP MATH MUL} . \end{aligned}$$

From Lemma 7 follows that the service look up of the client will be successful, and from Lemma 7 part 2 follows that the client is finally sending a request to the server via IPC:

$$\begin{aligned} & \dots \text{beforeCS?}(r^j.\text{pdb}(c), \text{MATH}, \text{MUL}, \text{arg}, \text{INFINITE}, \dots) \\ & \implies \exists j_2 > j. \text{request}(r^{j_2}.\text{pdb}(c), s, \text{MUL}) . \end{aligned}$$

Now, (at step j_2) the server can be in two different states. Either it is waiting in an open receive for requests (line 3), or the server is already processing some request (lines 5–18). In the second case we have to show, that the server will finally finish the processing and return to the beginning of the main loop. For that we only have to prove the termination of each statement of the loop.

Since the timeout values of the RPC send and receive statements are set to infinite, termination does not follow directly¹⁰. By using Lemma 6 and because we know that the client is obeying the client protocol, we prove that each RPC send primitive on server side is matched by an RPC receive primitive on client side (where the timeout is set to infinite) and conclude rendezvous and termination. Similarly we prove termination of the RPC receive primitive.

Hence, the server will always be finally waiting for requests. By the fairness property *app-fairness-infinite* over the run r we conclude that the server finally will also receive the request of client c . Since the requested procedure is MUL, the server will send an acknowledgment (line 7). From Lemma 7 part 3 it follows that finally the client will send the service parameters to the server:

$$\begin{aligned} & \dots \text{beforeCS?}(r^j.\text{pdb}(c), \text{MATH}, \text{MUL}, \text{arg}, \text{INFINITE}, \dots) \\ & \implies \exists j_3 > j. \text{beforeS?}(r^{j_3}.\text{pdb}(c), \text{arg}, ?, \text{pp2h}(r^{j_3}, c, s), \text{INFINITE}) . \end{aligned}$$

Next, the server will try to receive the parameters via RPC (line 9). Using Lemma 6 one can easily prove that the parameters of the client are correctly transmitted to the server. Hence, from Lemma 7 part 4 it follows that, the client will wait, via RPC, for the result:

$$\begin{aligned} & \dots \text{beforeCS?}(r^j.\text{pdb}(c), \text{MATH}, \text{MUL}, \text{arg}, \text{INFINITE}, \dots) \\ & \implies \exists j_4 > j. \text{beforeR?}(r^{j_4}.\text{pdb}(c), \mathbb{Z}_{32}, ?, \text{pp2h}(r^{j_4}, c, s), \text{INFINITE}) . \end{aligned}$$

On server side, next, the service (line 11), i.e. the computation of the product of the received integers, is executed. Since we assume the correctness of the C0 multiplication algorithm, it follows that the server will finally send via RPC the correctly computed product to the client (line 13). Using again Lemma 7 and Lemma 7 part 5 we conclude that the client will finally

¹⁰In case timeouts were set to a finite value, termination would follow from Lemma 5.

receive the result and return from the call with a success message:

$$\begin{aligned} & \dots \text{beforeCS?}(r^j.\text{pdb}(c), \text{MATH}, \text{MUL}, \text{arg}, \text{INFINITE}, \dots) \\ \implies & \exists j' > j. j' = \text{finishedFC}(r, j_4, c, \text{callService_MATH_MUL}) \wedge \\ & \text{changed?}(r^j.\text{pdb}(c), r^{j'}.\text{pdb}(c'), \\ & \quad [[\text{res} = \text{SUCC}, \text{data} = \text{va}(r^j.\text{pdb}(c), \text{arg}).x * \text{va}(r^{j'}.\text{pdb}(c), \text{arg}).y]]) \end{aligned}$$

q.e.d

The proof technique presented here, obviously applies also to server implementations other than the simple Math Server discussed in this Section.

7. Conclusion

This paper contains, to the best of our knowledge, for the first time the outline of a paper and pencil proof of a realistic client server mechanisms *at the code level*. Clearly, we used arguments and abstractions *known in one formalism or another* from previous work. As explained in the introduction, the contribution of this work is, however, to put all these concepts into a single mathematical theory which:

- *specifies the correctness* of systems as they are *without simplifications*, i.e. *at the code level*, and
- *justifies all abstractions* (used on the way) by proving that they are implemented correctly by lower system layers.

We, first of all, presented a subset of the specification of a simple operating system that supports communicating user applications. Then, we showed how the correctness of code, running in this concurrent setting, can be proven (almost) sequentially in traditional Hoare logic. For this we identified classes of non-interfering system calls. Finally, we presented the specification and outlined the verification of an RPC mechanism. We skipped over the not so simple theory of linking, required to show that functions linked to a C0 program behave — under certain conditions — in the desired way. Such a theory will be presented in (Rieden 2008).

The complete formalization of the simple operating system, including a file system, TCP sockets, and a sophisticated rights management, is available at (Bogan 2008b). On top of this, using the presented RPC mechanism, an email system, covering an SMTP server and an email client, has been constructed and partially verified. Thus, the work at hand is not only an abstract case-study, but it is applied to a concrete software system.

Because the implementation of client/server mechanisms involves many different concepts, the uniform theory developed here can not possibly be small. Indeed, even in a long paper like this, we had to skip over various details.

Basically, the way to mechanize the verification of RPC mechanisms, along the lines of the theory outlined here is a manageable task: (i) The C0 small-step semantics, the SOS * model and required low-level models of the kernel are fully formalized in Isabelle/HOL, (ii) a sound Hoare logic with a verification condition generator has been developed and is available, (iii) this paper provides a detailed verification plan and the required verification methodology for concurrency, (iv) the methodology for verifying library generators has been developed (while verifying the compiler), (v) a similar reordering theory has been formally verified in the context of Verisoft, and (vi) the invariants on the concrete code are not overly complex.

However, the mechanization of the proofs outlined here is still open due to much more practical obstacles: Integrating the huge amount of specifications, models and proofs emerge as a highly non-trivial and time-consuming engineering task. This covers, among other things, a social process, in which the work of dozens of researchers, located at many different places, has to be combined to one uniform and formal Isabelle/HOL theory. The problems and challenges related to this process, are not specific to the verification of RPC, but has been experienced and to a large extend solved in other sub-projects of Verisoft.

The author Eyad Alkassar acknowledges the support given by the German Research Foundation (DFG) within the program ‘Performance Guarantees for Computer Systems’. Sebastian Bogan would like to acknowledge the support given by the German Federal Ministry of Education and Research (BMBF) in the framework of the Verisoft project.

References

- Alkassar E, Böhm P, Knapp S 2008a Formal correctness of a gate-level automotive bus controller implementation. In B Kleinjohann, L Kleinjohann, W Wolf, eds., *6th IFIP Working Conference on Distributed and Parallel Embedded Systems (DIPES08)*, 57–68. Springer
- Alkassar E, Schirmer N, Starostin A 2008b Formal pervasive verification of a paging mechanism. In C R Ramakrishnan, J Rehof, eds., *14th Intl Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS08)*, volume 4963 of *LNCS*, 109–123. Springer
- Arkoudas K, Zee K, Kuncak V, Rinard M 2004 Verifying a file system implementation. In *Sixth International Conference on Formal Engineering Methods*, volume 3308 of *LNCS* 373–390
- Beckert B, Beuster G 2004 Formal specification of security-relevant properties of user interfaces. In *Proceedings, 3rd International Workshop on Critical Systems Development with UML, Lisbon, Portugal*, Munich, Germany. TU Munich Technical Report TUM-I0415
- Beuster G, Borner T, Markus Wagner P B 2007 Code-level verification of an email client. <http://www.verisoft.de/.rsrc/VerisoftRepository/vemail-trunk-r15868.tar.gz>
- Beuster G, Henrich N, Wagner M 2006 Real world verification — experiences from the Verisoft email client. In *Proceedings of the Workshop on Empirical Successfully Computerized Reasoning (ESCoR 2006)*
- Bevier W R 1989 Kit and the short stack. *Journal of Automated Reasoning*, 5(4): 519–530
- Bevier W R, Cohen R 1996 An executable model of the Synergy file system. Technical Report 121, Computational Logic Inc
- Bevier W R, Hunt Jr. W A, Moore J S, Young W D 1989 An approach to systems verification. *Journal of Automated Reasoning* 5(4): 411–428. ISSN 0168-7433
- Beyer S 2005 *Putting it all together - Formal Verification of the VAMP*. PhD thesis, Saarland University, Saarbrücken
- Beyer S, Jacobi C, Kröning D, Leinenbach D C, Paul W J 2005 Putting it all together - formal verification of the VAMP. *International Journal on Software Tools for Technology Transfer*
- Birrell A D, Nelson B J 1984 Implementing remote procedure calls. *ACM Trans. Comput. Syst.* 2(1): 39–59. ISSN 0734-2071
- Bishop S, Fairbairn M, Norrish M, Sewell P, Smith M, Wansbrough K 2005 TCP, UDP, and sockets: Rigorous and experimentally-validated behavioural specification: Volume 3: Overview. Technical report, University of Cambridge
- Bogan S 2007 Academic System — Demo 2007 Talk given at the 3rd German Verification Day. http://www-wjp.cs.uni-sb.de/leute/private_homepages/sebastian/Demo2.mov
- Bogan S 2008a *Formal Specification of a Simple Operating System*. PhD thesis, Saarland University, Saarbrücken

- Bogan S 2008b Isabelle/HOL specification of the SOS.
<http://www.verisoft.de/.rsrc/VerisoftRepository/sos-trunk-r22974.tar.gz>
- Bormann J, Beyer S, Maggiore A, Blackmore T 2007 Complete formal verification of tricore2 and other processors. In *Design and Verification conference (DVCon 2007)*
- Broy M, Merz S, Spies K 1996 The rpc-memory case study: A synopsis. In *Formal Systems Specification, The RPC-Memory Specification Case Study (the book grow out of a Dagstuhl Seminar, September 1994)*, pages 5–20, London, UK. Springer-Verlag. ISBN 3-540-61984-4
- Cohen E 2000 Separation and reduction. In *MPC'00*, pages 45–59. Springer. ISBN 3-540-67727-5
- Cohen E, Lamport L 1998 Reduction in TLA. In *CONCUR '98*, pages 317–331, London, UK. Springer. ISBN 3-540-64896-8
- Dalinger I 2006 *Formal Verification of a Processor with Memory Management Units*. PhD thesis, Saarland University
- Daum M 2008 Modelling user programs on top of a microkernel. In Troubitsyna E, editor, *Proceedings of Doctoral Symposium held in conjunction with Formal Methods 2008*, volume 48 of *General Publications*. Turku centre for computer science. http://www.wjp.cs.uni-sb.de/publikationen/Daum_FM08ds-.pdf
- Dörrenbächer J 2006 VAMOS microkernel: Formal models and verification 2006 Talk given at the International Workshop on Systems Software Verification, Australia, August 7–8, 2006. <http://www.cse.unsw.edu.au/formalmethods/events/svws-06/VAMOS-Microkernel.pdf>
- Elphinstone K, Klein G, Derrin P, Roscoe T, Heiser G 2007 Towards a practical, verified kernel. In *11th Workshop on Hot Topics in Operating Systems*, page 6, San Diego, CA, USA
- Grünbacher A 2003 POSIX Access Control Lists on Linux. In *USENIX Annual Technical Conference, FREENIX Track 259–272*
- Härtig H, Hohmuth M, Feske N, Helmuth C, Lackorzynski A, Mehnert F, Peter M 2005 The Nizza secure-system architecture. In *Proceedings of the 1st International Conference on Collaborative Computing: Networking, Applications and Worksharing*
- Hillebrand M A, Paul W J 2007 On the architecture of system verification environments. In *Haifa Verification Conference 2007, October 23–25, 2007, Haifa, Israel*, LNCS. Springer
- Hohmuth M, Tews H 2005 The VFiasco approach for a verified operating system. In *2nd ECOOP Workshop on Programm Languages and Operating Systems*
- IEEE 2004 IEEE std. 1003-1, 2004 edition. The Open Group Technical Standard. Base specifications, issue 6. Includes IEEE std 1003-1-2001, IEEE std 1003-1-2001/cor 1-2002 and IEEE std 1003-1-2001/cor 2-2004. Shell and utilities, 2004
- Jacobi C, Weber K, Paruthi V, Baumgartner J 2005 Automatic formal verification of fused-multiply-add fpus. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 1298–1303, Washington, DC, USA. IEEE Computer Society. ISBN 0-7695-2288-2
- Joshi R, Holzmann G J 2007 A mini challenge: Build a verifiable filesystem. *Formal Aspects of Computing* 19(2): 269–272
- Knapp S, Paul W J 2007 Realistic Worst Case Execution Time Analysis in the Context of Pervasive System Verification. In *Program Analysis and Compilation, Theory and Practice: Essays Dedicated to Reinhard Wilhelm*, volume 4444 of LNCS
- Lamport L, Melliar-Smith P M 1985 Synchronizing clocks in the presence of faults. *J. ACM* 32(1): 52–78. ISSN 0004-5411
- Langenstein B, Nonnengart A, Rock G, Stephan W 2007a A history-based verification of distributed applications. In Beckert B, editor, *Proceedings, 4th International Verification Workshop (VERIFY), Bremen, Germany*. CEUR-WS Workshop Proceedings
- Langenstein B, Nonnengart A, Rock G, Stephan W 2007b Verification of distributed applications. In F Saglietti, N Oster, eds., *Computer Safety, Reliability, and Security, 26th International Conference, SAFECOMP 2007, Nuremberg, Germany, September 18–21, 2007*, volume 4680 of LNCS 315–328. Springer
- Leinenbach D C 2008 *Compiler Verification in the Context of Pervasive System Verification*. PhD thesis, Saarland University, Saarbrücken

- Leinenbach D C, Paul W J, Petrova E 2005a Towards the formal verification of a C0 compiler: Code generation and implementation correctness. In Aichernig B and Beckert B, editors, *3rd International Conference on Software Engineering and Formal Methods (SEFM 2005)*, 5–9 September 2005, Koblenz, Germany 2–11
- Leinenbach D C, Paul W J, Petrova E 2005b Towards the formal verification of a C0 compiler: Code generation and implementation correctness. In *3rd International Conference on Software Engineering and Formal Methods*, 5–9 September 2005, Koblenz, Germany
- Leinenbach D C, Petrova E 2008 Pervasive compiler verification — From verified programs to verified systems. In *3rd International Workshop on Systems Software Verification*
- Lipton R J 1975 Reduction: A method of proving properties of parallel programs. *Commun. ACM* 18(12): 717–721. ISSN 0001-0782
- Lynch N A 1996 *Distributed Algorithms*. Morgan Kaufmann 1996 ISBN 1-55860-348-4
- Moore J S, Lynch T, Kaufmann M 1998 A mechanically checked proof of the amd5k86 floating point division program. In *10th Anniversary Colloquium of UNU/IIST*, volume 47(9). IEEE Transactions on Computers
- Moore J S 2003 A grand challenge proposal for formal methods: A verified stack. In B K Aichernig, T S E Maibaum, eds., *10th Anniversary Colloquium of UNU/IIST*, volume 2757 of LNCS, pages 161–172. Springer. ISBN 3-540-20527-6
- OSEK/VDX time-triggered operating system*. OSEK group 2001.
<http://www.osek-vdx.org/mirror/ttos10.pdf>
- Peled D 1998 Ten years of partial order reduction. In *CAV '98: Proceedings of the 10th International Conference on Computer Aided Verification*, pages 17–28, London, UK. Springer-Verlag. ISBN 3-540-64608-6
- Petrova E 2007 *Verification of the C0 Compiler Implementation on the Source Code Level*. PhD thesis, Saarland University, Saarbrücken
- Pfitzmann B, Riordan J, Stübke C, Waidner M, Weber A 2001 The PERSEUS system architecture. In D Fox, M Köhntopp, A Pfitzmann, eds., *VIS 2001, Sicherheit in komplexen IT-Infrastrukturen*, pages 1–18. Vieweg
- Rieden T I d, Tsyban A 2008 Cvm - a verified framework for microkernel programmers. In *3rd intl Workshop on Systems Software Verification (SSV08)*. Elsevier Science B.V
- Rieden T 2008 *CVM — A Formally Verified Framework for Microkernel Programmers*. PhD thesis, Saarland University, Saarbrücken. to appear
- Rieden T, Tsyban A 2008 CVM - A verified framework for microkernel programmers. In *3rd International Workshop on Systems Software Verification*
- Sawada J, Hunt W A 1998 Processor Verification with Precise Exceptions and Speculative Execution. In A J Hu, M Y Vardi, eds., *CAV'98* 135–146. Springer. ISBN 3-540-64608-6
- Schirmer N W 2005 *Verification of Sequential Imperative Programs in Isabelle/HOL*. PhD thesis, Technical University of Munich
- Schmaltz J 2006 A Formal Model of Lower System Layers. In *FMCAD'06*, pages 191–192, Los Alamitos, CA, USA. IEEE Computer Society. ISBN 0-7695-2707-8
- Shadrin A 2006 Design and implementation of the portmapper and RPC primitives in the context of the SOS. Master's thesis, Saarland University, Saarbrücken
- Shapiro J S, Doerrie M S, Northup E, Sridhar S, Miller M S 2004 Towards a verified, general-purpose operating system kernel. In G Klein, ed., *Proc. NICTA FM Workshop on OS Verification. Technical Report 0401005T-1*, pages 1–19. National ICT Australia
- Shepler S, Callaghan B, Robinson D, Thurlow R, Beame C, Eisler M, Noveck D, 2003. RFC 3530: Network file system (nfs) version 4 protocol, 2003
- Sikkel K, Stiemerling O 1998 User-oriented authorization in collaborative environments, 1998
- Smith M A 1996 *Formal Verification of TCP and TTCP*. PhD thesis, Massachusetts Institute of Technology
- Srinivasan R 1995 RFC 1831: RPC: Remote procedure call protocol specification version 2 1995.
<ftp://ftp.internic.net/rfc/rfc1831.txt>

- Starostin A 2006 Formal verification of a c-library for strings. Master's thesis, Saarland University. <http://www-wjp.cs.uni-sb.de/publikationen/St06.pdf>
- Tanenbaum A S, Renesse R V 1985 Distributed operating systems. *ACM Comput. Surv.* 17(4): 419–470. ISSN 0360-0300
- Tews H 2007 Micro hypervisor verification: Possible approaches and relevant properties. <http://robin.tudos.org/publications/hyperveri.pdf>
- The VERIFIX Consortium 2000 The VERIFIX Project. <http://www.info.uni-karlsruhe.de/verifix/>
- The Verisoft Consortium 2003 The Verisoft Project. <http://www.verisoft.de/>
- Tsyban A 2008 *Formal Verification of a Framework for Microkernel Programmers*. PhD thesis, Saarland University, Saarbrücken. to appear
- Tverdyshev S 2008 *Formal Verification of Gate-Level Computer Systems*. PhD thesis, Saarland University, Saarbrücken. to appear
- Welch J L, Lynch N 1988 A new fault-tolerant algorithm for clock synchronization. *Information and Communication* 77(1): 1–36
- Yang J, Twohey P, Engler D, Musuvathi M 2006 Using model checking to find serious file system errors. *ACM Transactions on Computer Systems* 24(4): 393–423. ISSN 0734-2071