Formal Specification of a Simple Operating System

Dissertation

zur Erlangung des Grades
Doktor der Ingenieurswissenschaften (Dr.-Ing.)
der Naturwissenschaftlich-Technischen Fakultäten
der Universität des Saarlandes

Sebastian Bogan

sebastian@wjpserver.cs.uni-sb.de

Saarbrücken, August 2008

Tag des Kolloquiums:	25. August 2008
Dekan:	Prof. Dr. Joachim Weickert
Vorsitzender des Prüfungsausschusses:	Prof. Dr.-Ing. Philipp Slusallek
1. Berichterstatter:	Prof. Dr. Wolfgang J. Paul
2. Berichterstatter:	Prof. Dr. Bernhard Beckert
akademischer Mitarbeiter:	Dr. Mark Hillebrand

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus anderen Quellen oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet. Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder ähnlicher Form in einem Verfahren zur Erlangung eines akademischen Grades vorgelegt.

Saarbrücken, im August 2008

There is no reason for any individual to have a computer in his home.
—Ken Olsen, founder of Digital Equipment Corporation, 1977

Danksagung

An dieser Stelle möchte ich all jenen danken, die zum Gelingen der vorliegenden Arbeit beigetragen haben.

Mein Dank gilt zunächst meinen Eltern, die mich während der gesamten Zeit meiner Ausbildung gefördert haben. Am Ende habe ich dank ihnen den Unterschied zwischen „R“ und „L“ doch noch begriffen.

Herrn Prof. Wolfgang Paul danke ich für die Möglichkeit zur Promotion, sowie für seine wissenschaftliche Betreuung.

Ganz besonders danke ich meiner Freundin Andrea Biehl, ohne deren grenzenlose Geduld und häufige Aufmunterungen diese Arbeit nicht zustande gekommen wäre.

Ein großes Dankeschön geht an meinen Freund und Kletterkollegen Sebastian Puhl für die zahllosen Stunden der „rauchfreien Compiler-Arbeit“.

Des Weiteren danke ich Irena Dotcheva für ihre unermüdliche Englischkorrektur.

Nicht zuletzt danke ich meinen Arbeitskollegen und Freunden am Lehrstuhl Paul für die gute Arbeitsatmosphäre und den einen oder anderen „ernsten Abend“.

This work was funded by the German Federal Ministry of Education and Research (BMBF) in the Verisoft project under grant 01 IS C38.

Abstract

Within the Verisoft project, we aim at the pervasive modeling, implementation, and verification of a complete computer system, from gate-level hardware to applications running on top of an operating system. As an adequate representative for such a system we choose a system for writing, signing, and sending emails.

The starting point of our work was a processor together with its assembly language, a compiler for a type safe C variant and a micro kernel. The goal of our work was to develop a (user-mode) operating system that bridges the gap between micro kernel and user applications. That is, formally specify and implement a system that, on the one hand, is built right on top of our micro kernel and, on the other hand, provides everything necessary for user applications such as an SMTP server, a signing server, and an email client. Furthermore, the design of this system should support its verification in a pervasive context.

Within this thesis, we present the formal specification of such an operating system. Along with this specification, we (i) discuss the current state-of-the-art in formal methods applied to operating-systems design, (ii) justify our approach and distinguish it from other people's work, (iii) detail our implementation- and verification stack, (iv) describe the realization of our operating system, and (v) outline the verification of this system.

Zusammenfassung

Innerhalb des Verisoft-Projekts streben wir die durchgängige Modellierung, Implementierung und Verifikation eines kompletten Computersystems, von der Hardware auf Gatterebene bis hin zu Benutzeranwendungen, an.

Ausgangspunkt unserer Arbeit war ein Prozessor inklusive Assembler Sprache, ein Compiler für eine typen-sichere C Variante und ein Mikrokern. Ziel unserer Arbeit war es, ein Betriebssystem (auf Benutzerebene) zu entwickeln, welches die Verbindung zwischen Mikrokern und Benutzeranwendungen herstellt. Das bedeutet, ein System formal zu spezifizieren und zu implementieren, welches auf der einen Seite direkt auf dem Mikrokern aufsetzt und auf der anderen Seite alle Voraussetzungen für Benutzeranwendungen wie einen SMTP Server, einen Signatur Server und ein E-Mail Programm erfüllt. Außerdem soll das Design dieses Systems seine durchgängige Verifikation unterstützen.

In dieser Arbeit präsentieren wir die formale Spezifikation eines solchen Systems. Ferner (i) diskutieren wir den aktuellen Stand im Bereich der formalen Methoden im Betriebssystemdesign, (ii) rechtfertigen unseren Ansatz und differenzieren ihn von dem anderer, (iii) stellen die unterschiedlichen Implementierungs- und Verifikations-Schichten unseres Projektes vor, (iv) beschreiben unsere Umsetzung des Systems und (v) skizzieren seine Verifikation.

Contents

List of Figures

List of Tables

Chapter

1

Introduction

Contents

In this chapter we describe the background of this work, discuss related work, and motivate our own work.

1.1 Background

There is an ever increasing amount of computer systems in our daily life. For example, we use computers to play games, listen to music, or watch videos. Besides using them for entertainment, we also use computers to shop, bank, and pay bills. Most of our communication and information relies on computers. In industry, production is likely to be planned and coordinated with the aid of computers. They are widely used in the medical- and service sector. Our stock markets are controlled by computers and we trust in them to operate airplanes and nuclear power plants. Finally, even weapons of mass destruction are controlled by computer systems. In fact, computers are so prevalent in our daily lives that should they be taken away, almost everything would shut down; our world would change dramatically.

Although computer systems play such an important and responsible role, only few of us really understand how they work. In many cases, we use them without asking questions. Fortunately, most of the time, these systems behave well and there is no reason to be sceptical. If, however, such a system fails,

severe consequences are possible. It is, therefore, not surprising that the demand for truly robust, safe, and secure systems constantly grows.

Computer systems are composed of many layers of hard- and software. In most cases, however, the user only perceives the top-most layer, i. e. the application layer. Past attempts to satisfy the desire for reliable systems often focused on this layer. By means of testing, people tried to establish a certain level of reliability. But only limited reliability can be achieved this way. This is because (i) unless tested exhaustively, there may be unexpected corner cases, and (ii) other system components (e. g. underlying system software) may behave differently than expected. Hence, in order to gain a truly reliable system, it does not suffice to test independent pieces, but the correctness of the entire system has to be proven. The Verisoft project [Ver07] aims at the pervasive modeling, implementation, and verification of a complete computer system, from gate-level hardware to applications running on top of an operating system.

The modeling and implementation of a user-mode operating system in a pervasive context is the topic of this work.

1.2 Related Work

There have been numerous attempts to increase confidence in system software by means of formal methods. Each of the different attempts may be assigned to one of the following categories: (i) the component approach, (ii) the parallel approach, (iii) the single-layer approach, or (iv) the pervasive approach. Below, we will discuss the relevant research results for each of these categories. Although our work belongs to the pervasive-approach category, results from the other categories are worthwhile looking at.

1.2.1 Component Approach

Projects in this category focus on selected system components. They choose some part of the implementation of a complex system, contrive some specification for it (neglecting the rest of the system), and then show some high level properties. From a distant point of view, one could say that projects in this category horizontally and vertically split a single system layer (Figure 1.1(a) on page 6).

All sorts of system components have been the subject of formal work. However, as the principal part of the work at hand deals with the formalization of a file system, a socket API [IEE04] and TCP / IP [Pos81b, Pos81a], as well as virtual terminals, we will only consider publications related to these components.

Hard-disc driver and file systems. In [BCT95], Bevier et al. describe how they specified a subset of the interface functions of the Synergy file system using

the specification language Z [Spi92]. This specification is provided on a very abstract level. It is intended as a programmers manual rather than a model for proving implementation correctness. The specification describes what happens if the preconditions for an interface function are met but ignores what happens if they are not. Many (other) details are ignored and the issue of concurrent file access is not treated at all. In order to validate their specification, Bevier et al. later (re-) implemented their Z specification using ACL2 [KMM00]. Definitions and theorems about the ACL2 model are presented in [BC96].

In 2004, Arkoudas et al. [AZKR04] established a simulation relation between the specification of a file system (which models the file system as an abstract map from file names to sequences of bytes) and its implementation (a model which assumes fixed-size disk blocks to store the contents of the files). They proved the correspondence between these two models but did not consider an actual implementation. They omitted details such as file permissions, dates, links, and multi-layered directories and they assumed an unbounded hard disk.

Yang et al. [YTEM06] used model checking to systematically test for file-system errors. They did not focus on a single file system, but developed a method to 'stress-test' different file-system implementations. Among other things, they tested for memory leaks and deadlocks.

In 2007, Joshi and Holzmann [JH07] suggested that "Building a Verifiable Filesystem" could be a suitable candidate for a so-called mini challenge. Their goal is to build a small file system for flash memory and at the same time produce as much as possible machine-readable documentation. This documentation should then be used for automatic verification. No results have been reported up to now.

Note, an additional discussion and comparison of our file-system formalization and the formalizations within the above mentioned related work can be found in § 4.3.2.12.

TCP / IP stack and socket API. In 1996, Smith [Smi96b, Smi96a] reports on the formal verification of TCP. In his work, he first of all provides an abstract specification for TCP / IP transport level protocols and then proves that different (more concrete) models of TCP satisfy this specification. He does not consider an actual TCP implementation.

In 2002, Smith et al. [SR02] specified the selective acknowledgment (SACK) [MMFR96] extension for the TCP standard. Based on this specification, they were able to verify that SACK does not violate TCP's safety properties, i. e. "no data is discarded, duplicated, or reordered because of the SACK mechanism that could not have been discarded, duplicated, or reordered with the standard acknowledgment mechanism". Other than the earlier work of Smith, this work only focuses on a particular algorithm for network congestion avoidance [Jac88]. They present proofs for a (retransmission) strategy but (again) do not consider a particular implementation.

The NETSEM project [Net08], headed by Peter Sewell, did a lot of work in the area of protocol- specification and validation. This project is an ongoing effort. So far, they have developed formal models of TCP, UDP [Pos80], and sockets at protocol level [BFN+05, BFN+06] and at service level [RNS08]. Their extensive specifications are formalized in higher order logic using the HOL automated proof system [Hol08]. They validated their protocol-level specification by comparing model traces with traces captured from the implementations in FreeBSD, Linux, and WinXP. In order to also validate their service-level specification, they specified an abstraction function and showed that traces of the protocol-level model have a counter part in the service-level model. The specifications they provide are very comprehensive and contain many details and special cases of the standard TCP, UDP, and socket implementations. However, they are not aiming at proving correctness of a particular implementation but try to establish a technique that allows us (i) to formalize complex protocols and (ii) to mechanically validate the achieved specifications. They suggest that their specifications should be used for conformance testing of new implementations.

Note, as for the file system, an additional discussion and comparison of our TCP- and socket formalization and the formalizations within the above mentioned related work can be found in § 4.3.4.16.

Serial interface driver and terminal I/O. In terms of a component-wise approach, we do not know of any formal work on serial interfaces.

All projects in this category achieve valuable results. Yet, they suffer from the fact that integrated components are treated as if they were isolated from the remaining system (although they are not). That means, correspondence between implementation and specification can not be proven. Thus, results can not be transferred to actual systems and are, therefore, of rather limited value.

1.2.2 Parallel Approach

Projects in the parallel-approach category try to protect security-sensitive data and, at the same time, provide complete functionality of common general-purpose operating systems. These projects 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. In a sense, the projects in this category horizontally split a system layer (Figure 1.1(b) on page 6).

The Perseus project [PRS+01] and the Nizza architecture [HHF+05] belong to this category. Both projects use an L4 micro kernel [L407] implementation as virtualization platform. On top of the micro kernel, they implement means that permit the running of legacy operating systems and secure applications

(e. g. a signature module) in parallel. Furthermore, they provide methods to exchange data between insecure and secure parts.

Certainly, both projects increase faith in system software but this faith is based on the correctness of the underlying micro kernel. For the Perseus project as well as the Nizza project, no formal proofs have been published so far.

1.2.3 Single-Layer Approach

Work in this third category applies formal methods to an entire system layer, usually the micro kernel layer (Figure 1.1(c) on the following page).

Recent candidates in this category are L4.verified [HEK+07, EKD+07], VFiasco [HTS02, HT05, Tew07b, Tew07a], and Eros [SW00]. All three projects have established semantics for C variants and have verified different properties on source-code level. But kernel implementations also contain hardware specific parts that are necessarily implemented in assembly language. To the best of our knowledge, (up to now) none of the above mentioned projects formally treats these hardware specific parts. As far as we can see, these parts are simply postulated to be correct and solely described by their semantic effects. Moreover, these projects rely on compiler correctness.

Within the Flint project [NYS07], a verification framework for assembly code was developed. Using this framework and a formalization of a subset of the x86 instruction set, they were able to formally prove the correctness for some context-switching code.

Robin [Tew07a] and Coyotos [SDN+04] are the successor projects of VFiasco and Eros, respectively. So far, none of them has published formal results.

Projects in this category achieve great results. They enrich the micro-kernel family and increase faith in them. Yet, they discard pervasive verification and instead focus on the verification of a single software layer.

1.2.4 Pervasive Approach

Projects in this last category aim at pervasive system verification. Formal methods span multiple layers of hard- and software. That is, several verified system layers are integrated into a stack. Here, integration means that the verification of one layer is based on the guarantees of the underlying layers (Figure 1.1(d) on the next page).

The community in this last category is small. An early but famous project in this category is the verified stack [Moo02] of Computational Logic Inc. (CLI). Just like the Verisoft project, this project aimed at pervasive verification of a complete system. They started from a hardware model and developed an assembler and a code generator for a simple high-level language. Using machine language, they implemented the simple operating system kernel KIT

Figure 1.1: Categories in System Verification. (a) the component approach, (b) the parallel approach, (c) the single-layer approach, and (d) the pervasive approach.

[Bev89]. KIT, however, was never really integrated into their stack; the modeled hardware lacked the necessary features (e. g. memory management, different modes of operation, or I/O interrupts). KIT featured services that are also found in today's micro kernels, namely memory virtualization, process isolation, a round-robin scheduler, message passing, and device-driver support. Yet, these services were still very limited compared to those found in modern micro kernels. For example, message passing was restricted to single words, and dynamic process creation was entirely impossible.

Inspired by the CLI stack, the ProCoS project [BHL+96] researched the development process. They focused on the theoretical background of pervasive system verification. Consequently, they neither focused on a particular system nor aimed at machine-checked proofs.

Similarly to ProCoS, FOCUS [BDD+92] provides formalisms for the specification and verification of distributed interactive systems. FOCUS aims at the modular development and implementation of such systems by refining requirement specifications down to concrete implementations. FOCUS itself is only a framework that provides the methods but, by using these methods, Spies [Spi98] was able to specify some key concepts of operating system kernels. In her work, she specifies, for example, process management and memory virtualization. However, her abstract specification does not correspond to a particular implementation.

Although the last two projects somehow belong to the pervasive approach category, they are also examples of a whole group of projects / languages that aim at providing formal methods. Other examples are TLA+ [Lam02], CSP [Hoa78], and Z [Spi92].

1.3 Motivation

Without a doubt, the last type of approach, the pervasive approach, is the most satisfactory one but also the most expensive one. Still, 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. As we will describe in Chapter 3, a processor together with its assembly language, a verified compiler for a type safe C variant and a micro kernel have already

emerged from this project. What is missing to achieve the ambitious goal 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.

The document at hand presents a formal specification of the user-mode operating system SOS, a system that satisfies the above-mentioned requirements and thereby fills the gap. In contrast to other projects, our specification is fully connected to specifications of lower system layers. As we will show in the following chapters, we are actually incorporating an entire system, consisting of a processor, external devices, and a micro kernel. This is new. We are the first to present a formal specification of an operating system that, on the one hand, reaches so far up and, on the other hand, is part of an integrated system stack. Still, this operating system is more than a toy. As we will describe in Chapter 3, our system has been used to successfully implement and run applications such as an SMTP server, a signing server, and an email client. Furthermore, our formal specification has been used to specify and partially prove properties of these user applications.

It is worth mentioning that the specification at hand has a corresponding Isablele / HOL [NPW02] specification [Bog08c] and a C0 [LPP05] implementation [Bog08a]. Thus, in a sense, this document is only one of the three corner stones of our work. (Figure 1.2).

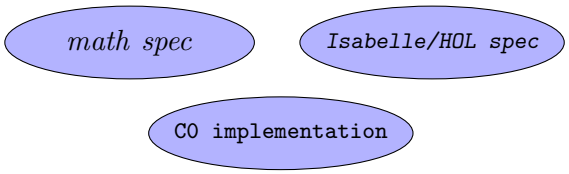

Figure 1.2: Cornerstones of This Work.

1.4 Document Organization

The remaining document is organized in the following way. In Chapter 2, we detail conventions followed in this document. Chapter 3 describes the foundation of our work and the implementation of the SOS. Chapter 4 is the key chapter of this work. There, we describe the SOS* model, i. e. the specification of SOS. In Chapter 5, we extend SOS*. There, we present the DSOS* model, i. e. the model of a distributed system containing a number of SOS* instances. In Chapter 6, we outline the verification obligations and sketch the proofs for the top level theorems. Finally, in Chapter 7, we summarize our work.

Chapter

2

Preliminaries

Contents

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

2.1 Typography

Within this document we describe the SOS from different perspectives. In order to avoid confusion between the different perspectives, we generally use an *italic* font for the mathematical specification, `monospace` for the C0 implementation, and *`slanted monospace`* for Isabelle / HOL related material. Names of constants, used in the mathematical specification, are set in SMALL UPPERCASE LETTERS.

2.2 Mathematical Notations

Basic types and operations. The basic types used in this document are $\mathbb{N}_{32} = \{0, \ldots, 2^{32} - 1\}$, $\mathbb{N}_{32}^{+} = \mathbb{N}_{32} \setminus \{0\}$, $\mathbb{Z}_{32} = \{-2^{31}, \ldots, 2^{31} - 1\}$, $\mathbb{B} = \{\text{TRUE}, \text{FALSE}\}$. Additionally we use $\mathbb{N} = \{0, 1, 2, \ldots\}$ for the set of natural numbers. In many places we introduce types for a particular purpose. Such types may be easily recognized by their name extension $\_t$. We use, for example, $word\_t = \{0, \ldots, 2^{32} - 1\}$ and $byte\_t = \{0, \ldots, 2^{8} - 1\}$ as basic units for I/O operations.

For the aforementioned numeric types we take the operations: $-$ (subtraction), $+$ (addition), / (integer division), $*$ (multiplication), % (modulo), and Σ (sum) for granted.

For boolean types we assume that the basic logical operations: $\wedge$ (conjunction), $\vee$ (disjunction), and $\neg$ (negation) are predefined. We write $\exists$ for the existential quantifier and $\forall$ for the universal quantifier. Sometimes the type of a quantified variable is omitted, if it can be inferred otherwise. An implication is denoted by $\longrightarrow$. In rare cases, we use $\Longrightarrow$, $\Longleftrightarrow$, and $\equiv$ to express an implication, tautology, and equivalences.

Sets. We take the set operations: $A \cup B$ (set union), $A \cap B$ (set intersection), $A \times B$ (Cartesian product), and $A \setminus B$ (set difference) as well as $A \subset B$ (strict subset), $A \subseteq B$ (subset), and $a \in A$ (element of) for granted. Besides these operations, we denote set comprehension by $\{x \mid x \in A \wedge P(x)\}$ and the power set of A by $\mathcal{P}(A)$. We write $|A|$ for the cardinality of the set A and $\{\,\}$ for the empty set. We compute the smallest member of a set of numbers using the *min* operator, i. e. $min(A) = x \Longleftrightarrow x \in A \wedge \forall y \in A.\ x \leq y$. Likewise, the greatest member of a set of numbers is computed using the *max* operator, i. e. $max(A) = x \Longleftrightarrow x \in A \wedge \forall y \in A.\ x \geq y$. In rare cases we use the Hilbert Choice Operator ϵ to select an arbitrary element from a given non-empty set.

Tuples and Records. For small structured values we use n-tuples. The type of such an n-tuple $(x_0, x_1, \ldots, x_n)$ is $T_0 \times T_1 \times \cdots \times T_n$, if $x_0 \in T_0 \wedge x_1 \in T_1 \wedge \cdots \wedge x_n \in T_n$. We assume that elements of an n-tuple can be enumerated (starting from 0). Access to the i-th element of an n-tuple x is denoted by $x[i]$.

We often 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 $rec\_t = \{n_0 : T_0, \ldots, n_n : T_n\}$. 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 $[\![n_0 = v_0, \ldots, n_n = v_n]\!]$. 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. Record types may be nested. Updating a 'deep component' of a nested-record-type value would require to unfold the whole structure and update all enclosing records, from the actual component up to the outermost record. As this is very inconvenient, we write, for example, $x' = x[\![n_i.m_j := v_j']\!]$ as a shorthand for $x' = x[\![n_i := x.n_i[\![m_j := v_j']\!]]\!]$.

Abstract data types. Some aspects are best modeled using so-called abstract data types. Let $T_{0,0}, \ldots, T_{0,m_0}, T_{1,0}, \ldots, T_{1,m_1}, \ldots, T_{n,m_n}$ be some types

and $\mathrm{C}_0, \ldots, \mathrm{C}_n$ some constructor names. Then, we declare the abstract data type T as:

$$\begin{aligned} T = \quad & \mathrm{C}_0\ T_{0,0} \ldots T_{0,m_0} \\ | \ & \mathrm{C}_1\ T_{1,0} \ldots T_{1,m_1} \\ & \ldots \\ | \ & \mathrm{C}_n\ T_{n,0} \ldots T_{n,m_n}. \end{aligned}$$

Now, if, for example, $(x_{1,0}, \ldots, x_{1,m_1}) \in T_{1,0} \times \cdots \times T_{1,m_1}$, then $\mathrm{C}_1\ x_{1,0} \ldots x_{1,m_1}$ would be a value of T. There may be abstract data types only containing constructors, i. e. constructors without parameters. Thus, the simplest abstract data type is nothing but an enumeration. However, more advanced abstract data types may contain different constructors with different numbers of parameters. Furthermore, abstract data types may be nested or even recursively defined.

Compound expressions. We use a small set of common functional programming notations. We write **if** x **then** a **else** b for conditional expressions. The value of this 'if then else' expression is: a if x evaluates to TRUE and b otherwise. To prevent ambiguous conditional expressions the **else** part is mandatory. In several places we use:

$$\begin{cases} a & \textbf{if } x, \\ b & \textbf{else if } y, \\ c & \textbf{else} \end{cases}$$

for a more intuitive version of **if** x **then** a **else** (**if** y **then** b **else** c).

Besides 'if then else' expression we use abbreviations. We write **let** $x_0 = y_0; \ldots; x_n = y_n$ **in** $e(x_0, \ldots, x_n)$ as an abbreviation for $e(y_0, \ldots, y_n)$. Note that the substitution of an abbreviation by the corresponding right hand side may be non-trivial. This is because pattern matching can be used. We write, for example, **let** $(x_0, x_1) = (y_0, y_1)$ **in** $x_0 + x_1$ to simultaniously assign abbreviations for x_0 and x_1, which are later used separately. Furthermore, abbreviation may be nested. That is, within the same **let** block, an abbreviation introduced earlier is sometimes used on the right hand side of a later assignment. In **let** $x_0 = y_0; x_1 = x_0 + 1$ **in** $\ldots$, for example, the abbreviation x_0 is used to define x_1.

Functions. Usually, we first of all declare the type of a function and only then define it. We write, for example, $f \in \mathbb{Z} \to \mathbb{N}$ to declare a function f that maps values from the domain $\mathbb{Z}$ to values in the range $\mathbb{N}$. An appropriate definition could then be $f(x) =$ **if** $x \geq 0$ **then** x **else** $-x$. In many places, we extend the type of the range of a function by the uninterpreted constant ε, in order to avoid partial functions. Thus, we use, for example, $g \in \mathbb{Z} \to \mathbb{N} \cup \{\varepsilon\}$

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 $f'(x) = (\textbf{if } x = n_i \textbf{ then } v_i' \textbf{ else } f(x))$. Also, for function updates, we write $f' = f[\![n_i.m_j := v_j']\!]$ as a shorthand for $f' = f[\![n_i := f(n_i)[\![m_j := v_j']\!]]\!]$. Occasionally, we use λ-notation. We write $\lambda x \in T.\ f(x)$ to denote the anonymous function that maps any $x \in T$ to $f(x)$.

Lists. We use the concept of abstract data types to recursively define lists. Let T be a type. Then, we define the list type T^* as $T^* = [\,] \mid \text{CONS } T\ T^*$, i. e. a list is either the empty list $[\,]$ or the concatenation of a single element and a list. Thus, lists have the constructors $[\,]$ and CONS. As a shorthand for CONS x xs, we write $x\#xs$. Here, x is called the head of the list and xs its tail. We assume that the tail of a list is returned by list operator *tail*. Constructing lists manually is not very handy. Thus, we write $[a]$ as a shorthand for $a\#[\,]$. Furthermore, let $a_0, \ldots, a_n \in T$. Then, the list containing these elements is denoted by $[a_0, \ldots, a_n]$. Let l_1 and l_2 be two lists of the same type. Then, their concatenation is denoted by $l_1 \circ l_2$. The $\mathit{length} \in T^* \to \mathbb{N}$ operator returns the length of a list. It is recursively defined as:

$$\mathit{length}(l) = \begin{cases} 0 & \textbf{if } l = [\,], \\ 1 + \mathit{length}(\mathit{tail}(l)) & \textbf{else.} \end{cases}$$

The elements of a list l can be enumerated (starting from 0). The i-th element of l is denoted by $l[i]$. The sublist of l that contains the elements in the range i to j is denoted by $l[i:j]$, i. e. $l[i:j] = [l[i], \ldots, l[j]]$. The type of a list with fixed length $n \in \mathbb{N}$ is denoted by T^n, if the type of the individual elements is T. The $\mathit{map} \in (T_0 \to T_1) \times T_0^* \to T_1^*$ operator is used to lift a function to operate on a list of elements, i. e. $\mathit{map}(f, [x_0, \ldots, x_n]) = [f(x_0), \ldots, f(x_n)]$. The $\mathit{filter} \in (T \to \mathbb{B}) \times T^* \to T^*$ operator is used to remove all those elements from a list that do not satisfy a certain predicate:

$$\mathit{filter}(P, l) = \begin{cases} [\,] & \textbf{if } l = [\,], \\ \mathit{filter}(P, \mathit{tail}(l)) & \textbf{else if } \neg P(l[0]), \\ l[0]\#\mathit{filter}(P, \mathit{tail}(l)) & \textbf{else.} \end{cases}$$

The $\mathit{take} \in \mathbb{Z} \times T^* \to T^*$ operator returns the prefix of a list. Assuming $\mathit{take}(z, l)$, this prefix contains (at most) z elements. If $z \leq 0$, then the empty list is returned:

$$\mathit{take}(z, l) = \begin{cases} [\,] & \textbf{if } l = [\,] \vee z \leq 0, \\ l[0]\#\mathit{take}(z - 1, \mathit{tail}(l)) & \textbf{else.} \end{cases}$$

The $drop \in \mathbb{Z} \times T^* \to T^*$ operator returns the postfix of a list. Assuming $drop(z, l)$, the first z elements (at most) are dropped from the list. If $z \leq 0$, then the original list is returned:

$$drop(n, l) = \begin{cases} l & \textbf{if } l = [\,] \vee z \leq 0, \\ drop(z - 1, tail(l)) & \textbf{else.} \end{cases}$$

Chapter

3

Foundation

Contents

In this chapter we elaborate on the design and implementation of the SOS with respect to its environment, i. e. the underlying system layers and the applications running on top of it.

First, we introduce the micro processor VAMP, an appropriate Assembler, and the high level programming language C0. Based on these, we work our way through the different layers of the system stack. That is, we describe the model of communicating virtual machines, detail the micro kernel VAMOS, outline the implementation of the SOS, and finally discuss some applications.

3.1 VAMP

At the bottom of our system stack we have the Verified Architecture Microprocessor (VAMP). This processor is a pipelined 32-bit RISC processor based on the MIPS instruction set. Among other things, the VAMP

comes with a memory unit with a cache system, a Tomasulo [Tom67] out-of-order scheduler, fully IEEE 754 [IEE85] compliant floating point units, a fixed point unit, and precise interrupts [Bey05]. Numerous people (e. g. [BJK+03, BJK+05, JB05, DHP05, Hil05]) have worked on the design, implementation and correctness proofs of the individual parts of the VAMP. In the end, however, the overall proof, i. e. the correctness of the gate-level implementation of the entire VAMP with respect to the programmer's model of a step-by-step instruction execution, has been carried out by Sven Beyer [Bey05], using the theorem proving system PVS [ORS92], and Sergey Tverdyshev [Tve08] and Iakov Dalinger [Dal06], using the Isabelle/HOL proof assistant [NPW02]. The programmer's model of the processor is realized by the instruction set of the VAMP. This model is called the instruction set architecture (ISA). ISA is the basis for the following software layers.

3.2 Assembler

Clearly, we do not want to implement or, even worse, verify any software at machine-code level. Thus, for the Verisoft project, Mark Hillebrand et al. have implemented an assembler and Alexandra Tsyban [Tsy08] proved the correctness of the corresponding assembler model against ISA.

3.3 C0

The verification of operating-system code at assembler level would still be a very tedious and error prone task. Hence, a high-level language, including formal semantics and compiler-correctness theorems, is desirable. Having these in place, we are, on the one hand, able to (more) efficiently argue about the correctness of operating systems (OSs) and user applications and, on the other hand, verification results can be brought down to machine level. Then, on machine level, software verification results can be combined with hardware correctness into an overall system-correctness proof.

For the Verisoft project, Leinenbach et al. designed the high level programming language C0 and provided a formal small-step semantics. Together with Elena Petrova, they implemented a C0 compiler and proved its correctness [LPP05, Pet07, Lei08, LP08].

In order to reason on an even more abstract level, Norbert Schirmer has provided a verification environment for sequential imperative programming languages and an automatic verification condition generator [Sch05]. This environment is built on top of Isabelle/HOL. Via an intermediate C0 big-step layer, it allows us to reason about C0 programs using classical Hoare triples [Hoa69].

The higher layers of our system stack, e. g. the SOS and the applications, are implemented in C0. Their implementations are much influenced by the

abilities and limitations of C0. Below, we will, therefore, present a short summary about C0.

The syntax of C0 is similar to the one of standard C [ANS99]. Operational semantics, however, is similar to Pascal [ANS83]. Compared to C, the main language restrictions are the lack of pointer arithmetic, function pointers, pointers to local variables, and prefix and postfix arithmetic operations. Furthermore, the size of arrays (including the type of the individual elements) has to be statically defined, there is only one `return` statement (which has to be the last statement of a function body), and side effects in expressions are forbidden. C0 is a type-safe programming language. The basic types it provides are signed and unsigned integers (`int` and `unsigned int`), characters (`char`), and booleans (`bool`). Based on these basic types, structures, array types, and pointers can be constructed. Type casts are allowed for basic types. C0 inherits the following operators from standard C:

- `!`, `&&`, and `||` for logical expressions,
- `==`, `!=`, `>`, `<`, `>=`, and `<=` for (numerical) comparisons,
- `+`, `-`, `*`, and `/` for arithmetic expressions,
- `<<`, `>>`, `|`, `&`, and `^` for bitwise manipulations, and
- `&` and `*` for pointer manipulations.

Finally, C0 provides `while` loops, `if-then-else` conditionals, function calls, assignments, and basic means for dynamic memory allocation (`new`) and garbage collection.

3.4 CVM

As in many recent projects, we have split our OS into a part running in system mode, i. e. the micro kernel, and a part running in user mode, i. e. the user-mode OS. Furthermore, we have divided the micro kernel into a hardware-dependent part and a hardware-independent part [Tan01].

In our implementation, the hardware-dependent part contains portions of assembly code. It encapsulates all hardware-specific low-level functionality in so-called CVM primitives and takes care of page faults. Thus, it provides a framework for the hardware-independent part.

From a verification point of view, the hardware-dependent part provides an independent layer, i. e. the model of communicating virtual machines (CVM). Each of these virtual machines (VMs) is essentially an abstract processor with virtual memory. It is the context for a single thread of execution, i. e. a process. Thus, the two major tasks of the CVM layer are memory virtualization and switching between different threads of execution.

CVM primitive	Description
`cvm_reset`	removes memory and initializes registers of a VM
`cvm_clone`	duplicates a VM
`cvm_alloc`	increases memory of a VM
`cvm_free`	decreases memory of a VM
`cvm_copy`	copies data between VMs
`cvm_get_gpr`	reads VM registers
`cvm_set_gpr`	writes VM registers
`cvm_dev_io`	copies data between virtual mem. and a device
`cvm_set_mask`	sets the external interrupt mask
`cvm_load_os`	loads initial user process
`cvm_wait`	(idle) loops until there is a runable VM
`cvm_start`	start / switch to a VM

Table 3.1: CVM Primitives

In our formalization, we call the hardware-independent part the abstract kernel. By compiling and linking the implementations of the CVM primitives and some implementation of an abstract kernel (see Section 3.5) we obtain an executable micro kernel. In our formalization, we call this combination the concrete kernel.

The implementation of the abstract kernel uses the CVM primitives to manipulate the virtual machines running in user mode, i. e. the user processes. Among other things, CVM primitives allow us: to copy data between virtual machines, to modify their virtual memory size, and to access their general purpose registers. The set of the available CVM primitives is given in Table 3.1. Using these primitives, the hardware-independent part can be implemented in plain C0.

The verification of CVM is nearly finished. The overall CVM correctness is described in a paper-and-pencil style in [RT08] and in a formal way in [GHLP05, Rie08]. It comprises the following propositions: (i) the page fault handler correctly implements memory virtualization [ASS08], (ii) the CVM primitives establish the CVM model and their implementation is functionally correct [ST08], and (iii) the CVM model can be instantiated by arbitrary abstract kernels, written in C0, and arbitrary user processes, written in assembly.

3.5 VAMOS

In the Verisoft project, we have two abstract kernel implementations: OLOS [KP07] and VAMOS [Dör06]. OLOS's primary targets are automotive applications. OLOS will not be discussed in the work at hand. VAMOS, however, provides means for general purpose OSs. The SOS is built on top of VAMOS. Thus, the implementation of VAMOS and the corresponding model of the concrete kernel will be the subject of this section.[1]

3.5.1 Implementation

The main features of VAMOS are: (i) user processes can be created and killed, (ii) user processes may have different privileges, (iii) user processes are scheduled via a priority-based round-robin scheduler, (iv) user processes are strictly isolated by means of memory virtualization, (v) user processes may communicate via synchronous inter-process communication (IPC), and (vi) user processes may register as device drivers and interact with devices. These features can be controlled via so-called kernel calls or VAMOS calls.

Initial / privileged process. When VAMOS boots, it launches an initial user process. This process has to set up the user-mode OS. As a privileged user process, it is allowed to bring up new user processes and kill existing ones, it is able to control the memory available to user processes, it may change scheduling parameters, it can alter the registration of device drivers, and it is able to add other user processes to the set of privileged processes. In contrast, a non-privileged user process is basically only allowed to perform IPC operations. Besides being privileged, the initial user process has the highest scheduling priority. That means, as long as it does not assign this priority to another user process, it can be sure to be scheduled next, as soon as it is ready.

Synchronous IPC. VAMOS supports synchronous IPC. Messages of (almost) arbitrary size may be exchanged via a send and a receive operation. Furthermore, an operation for a combined send and receive operation, i. e. a request, is provided.

Handles. The kernel implements a capability-like security concept for IPC. While the kernel identifies user processes by unique process identifiers (PIDs), user processes refer to each other via process-local aliases, so-called handles. The kernel maintains the mapping between handles and PIDs in the handle data base. This indirection permits authentic process identification. When a user process dies, all handles to it are invalidated and all handle owners receive a death notification.

[1] Below, if it is clear from the context, we will (also) refer to the concrete kernel as VAMOS or simply as kernel.

IPC rights. Together with handles, the kernel maintains IPC rights. These rights encode whether a certain user process has the right to send a message to a certain other user process, if it can only request something, or if it is not allowed to know the other user process at all. Additionally, a user process can control whether a finite timeout may be used and whether the IPC rights are valid for more than one successful IPC operation. The kernel maintains IPC rights in the rights data base.

Device driver. If a user process is registered as a device driver for a particular device, then it can read from and write to the corresponding device registers. Furthermore, it will be notified about interrupts from that device. There is a one-to-one mapping between devices and interrupt numbers and only one device driver may be registered for a certain device. Interrupts for devices without a registered device driver are lost.

Kernel notifications. Besides pure message delivery, the IPC mechanism is used to control and propagate updates of the handle data base and the rights data base. Furthermore, it is used to synchronously deliver kernel notifications, i. e. interrupt- and death notifications.

Kernel calls. A user process may call the kernel using the trap instruction while passing along the appropriate kernel-call number. The trap instruction triggers an exception, which causes the system to switch to system mode. In system mode the interrupt service routine is called. This routine saves the caller's context and executes an appropriate interrupt handler. The latter evaluates the trap instruction and passes control to the kernel-call dispatcher. The kernel-call dispatcher passes the call to the appropriate kernel-call handler. The kernel-call handler, finally, reads the kernel-call arguments from the caller's registers and services the call. At some point, the kernel call will be processed and the call chain reverses. In the end, the caller's context is restored and the mode switched back to user mode. Upon return from the kernel call, the caller can find the results in its registers. Note that in our implementation, the kernel will not be interrupted. Thus, there is no nested interrupt handling. Now, the mapping between the contents of the caller's registers and the effect of a trap instruction constitute the kernel's binary interface. A formal specification of this interface is established by the model VAMOS* (see § 3.5.2).

3.5.2 Model

The model of the VAMOS micro kernel including assembly user processes is called VAMOS*.[2] Formal verification of VAMOS*, i. e. the proof of correspon-

[2]Currently, there is no VAMOS* documentation publicly accessible. The descriptions here are based on the Verisoft-internal Technical Report #38

dence between VAMOS⋆, on the one hand, and the CVM model instantiated by the implementation of the abstract kernel, on the other hand, is still work in progress (Figure 3.3 on page 27). The main goal for the VAMOS⋆ verification is proving functional correctness of the individual kernel-call handlers, the scheduler implementation, and the delivery of kernel notifications. This verification relies on the CVM model and the semantics of the CVM primitives.

3.6 Libvamos

High-level programming languages usually do not allow the direct manipulation of registers. This is also true for C0. Thus in order to permit user processes, implemented in C0, to use kernel calls, we provide a library that hides the necessary assembly code. In our case this library is called Libvamos.

3.6.1 Implementation

The implementation of the Libvamos library is straightforward. In general, for each of the main cases considered by the kernel call dispatcher, the library implements a C0 function, a so-called kernel-call wrapper, that: (i) copies arguments to registers, (ii) calls the trap instruction, and (iii) upon return (of the trap instruction) extracts values from registers and passes them back as results. A complete list of the available kernel-call wrappers is presented in Table 3.2 on the next page.[3] A formal specification of Libvamos is established by the model VAMOS⋆+C0 (see § 3.6.2).

3.6.2 Model

VAMOS⋆+C0 is the model of the VAMOS micro kernel including C0 user processes, i. e. user processes written in C0 using the functions provided by the library Libvamos. As for the VAMOS⋆, the verification of VAMOS⋆+C0 is work in progress. The main goal of this work is to justify the new process abstraction of C0 user processes with respect to corresponding assembly user processes (Figure 3.3 on page 27). The challenge is that the different process abstractions do not have the same granularity; a C0 step usually consists of many assembly steps. Thus, a C0 user process could be scheduled away during the execution of a C0 statement. It is, therefore, necessary to abstract from the concrete scheduler and show that scheduling events can be shifted. Abstracting the scheduler leads to nondeterministic execution of user processes.

[3] Below, if it is clear from the context, we will simply write “kernel call” or “VAMOS call” as a shorthand for “kernel-call wrapper”.

VAMOS Call	Description
`process_create`	creates a new user process from a memory image
`process_clone`	duplicates a process
`process_kill`	kills a process
`set_privileged`	adds a process to the set of privileged processes
`chg_sched_params`	changes scheduling parameters
`memory_add`	increases the amount of virtual memory for a process
`memory_free`	decreases the amount of virtual memory for a process
`ipc_send`	sends an IPC message
`ipc_receive`	receives an IPC message
`ipc_request`	sends an IPC message and waits for a reply
`change_rights`	manipulates IPC rights
`read_kernel_info`	asks for kernel information
`change_driver`	(un-)registers a process as handler for a set of devices
`enable_interrupts`	(re-)enables interrupts
`dev_read`	reads from a device
`dev_write`	writes to a device

Table 3.2: VAMOS Calls

However, fairness between user processes should be preserved.[4] Thus, it is also necessary to show that the VAMOS scheduler ensures fairness between user processes and that this fairness is, indeed, preserved by VAMOS*+C0. Finally, the functional correctness of the individual kernel-call wrappers needs to be shown.[5]

3.7 SOS

One goal of the Verisoft project is a pervasively verified system for writing, signing, and sending emails. Analyzing these applications, we derived the following requirements for our user-mode operating system SOS: (i) the OS

[4]Note that our scheduler does not ensure that each of the user processes gets exactly the same amount of computing time. Depending on the type of kernel calls used, a user process might be scheduled longer than other user processes.

[5]Currently, there is no VAMOS*+C0 documentation publicly accessible. The descriptions here are based on the Verisoft-internal Technical Report #67

must support different users, (ii) concurrently running user applications need file-system- and network access, (iii) users should be able to interact with the system by means of a keyboard and a screen, (iv) users should be able to dynamically start and stop applications, and (v) user applications should be able to use remote procedure calls. None of these services is directly supported by the kernel. Instead, the SOS must provide the appropriate calls, i. e. the SOS calls.

3.7.1 Implementation

Earlier we said that when VAMOS boots, it launches an initial user process. This user process, the SOS, will be the one providing the SOS calls.[6] As the SOS launches, it registers itself as a device driver, starts an initial user application, and then serves incoming requests. Note that all user processes, except for the SOS, are called (user) applications.

SOS server. The SOS is implemented as a server. It waits for an IPC request, tries to interpret the IPC message as an SOS call, and then dispatches the call to an appropriate SOS-call handler. This handler processes the call and returns a result to the calling application. This result is returned by means of IPC-send and thus completes the caller's request. As the handler returns, the SOS turns over by waiting for another SOS call (Figure 3.1 on the next page).

Users and login shells. The initial application, started by the SOS, is a login shell. Like every other user application, the login shell is a non-privileged user process. It is not part of the SOS implementation. It is, however, the starting point for all other user applications. Internally, the SOS associates a user with each application. In the case of the login shell, this is the super user. Usually, login shells implement some sort of user authentication and upon successful authentication they pass control to a user-specific application. This user-specific application will be owned by the particular user and in most cases, it is some sort of general-purpose shell. For our system, we have implemented a login shell which authenticates users based on a password file, which is only accessible by the super user. Among other things, our general-purpose shell allows the logged-in user to interactively start further applications.

Resource management. As said earlier, user applications are non-privileged user processes. Hence, except for IPC, they are restricted to SOS calls. Thus, the SOS keeps tight control over every application running in the system.

[6] Below, if it is clear from the context, we will write "process" as a shorthand for "user process".

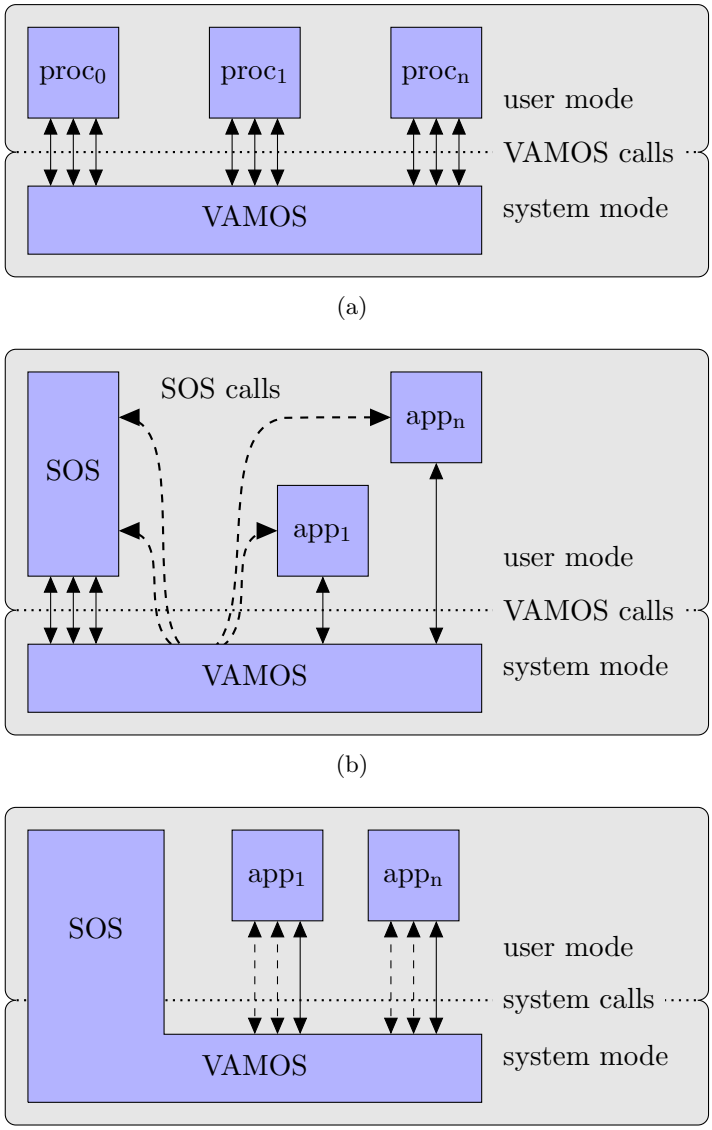

Figure 3.1: SOS Calls. a) User processes solely rely on VAMOS calls. b) In the presence of the SOS, user applications are restricted to a few VAMOS calls but (via IPC) they may use SOS calls. c) User applications cannot see the difference between VAMOS calls and SOS calls. Thus, from the application 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.

Together with the knowledge about the user owning a particular application, the SOS is able to enforce a strict resource management. That is, it can, for example, control the number of processes and files as well as socket- and terminal access on a per application and per user basis.

Device drivers. Device drivers account for the major part of the SOS implementation. These drivers provide access to a hard disk, a keyboard and a screen, and (partially) to a network card. The front end to these drivers are SOS calls, i. e. file system calls, terminal calls, and socket calls. Internally, the drivers are implemented in different layers, each providing a different level of abstraction (see Figure 3.2 on the following page).

For the hard disk, we have implemented a low-level hard-disk driver providing word-based access to the hard-disk contents. Based on this, we have implemented a driver providing a file-based hard-disk access.[7] Finally, the topmost layer adds user-based access-control lists [Grü03] to the individual files.

For the keyboard and the screen, we have implemented a serial device driver that hides the communication with the UART chip [Uar07]. This driver provides a queue of keyboard inputs and an array representing the contents of the screen. Above this low-level driver, there is a layer that multiplexes the input and output onto multiple virtual terminals.

For network access, there is an implementation of the TCP layer.[8] Based on an emulated IP layer, this layer provides reliable data exchange. Finally, on top of the TCP layer, we have implemented a socket interface [IEE04].

In our implementation, all the drivers are part of the SOS server. If the SOS receives an interrupt notification, it is treated just like an SOS call, i. e. the notification is passed to the appropriated handler. From there, it will be passed down to the lower driver layers. On its way back up, each layer processes the results from its adjacent layer. Once the interrupt notification is handled, the SOS goes back to receive further SOS calls and interrupt notifications.

A single process. Keeping the whole implementation of the SOS, including drivers, in a single process is not very efficient (in terms of interrupt- and SOS-call latency). However, from a verification point of view, this is a lot easier than arguing about a distributed implementation. In order to reduce the interrupt- and SOS-call latency, we run the SOS with the highest priority. That means, the SOS is scheduled next, if there is an interrupt or an SOS call. While treating one of these, the SOS will not be interrupted by a user application. Both, interrupts and SOS calls, are delivered to and treated by the SOS synchronously.

[7] The low-level hard-disk driver and the driver providing a file-based hard-disk access were implemented by Mark Hillebrand.

[8] The TCP implementations were provided by Ulan Degenbaev and Jérôme Creci.

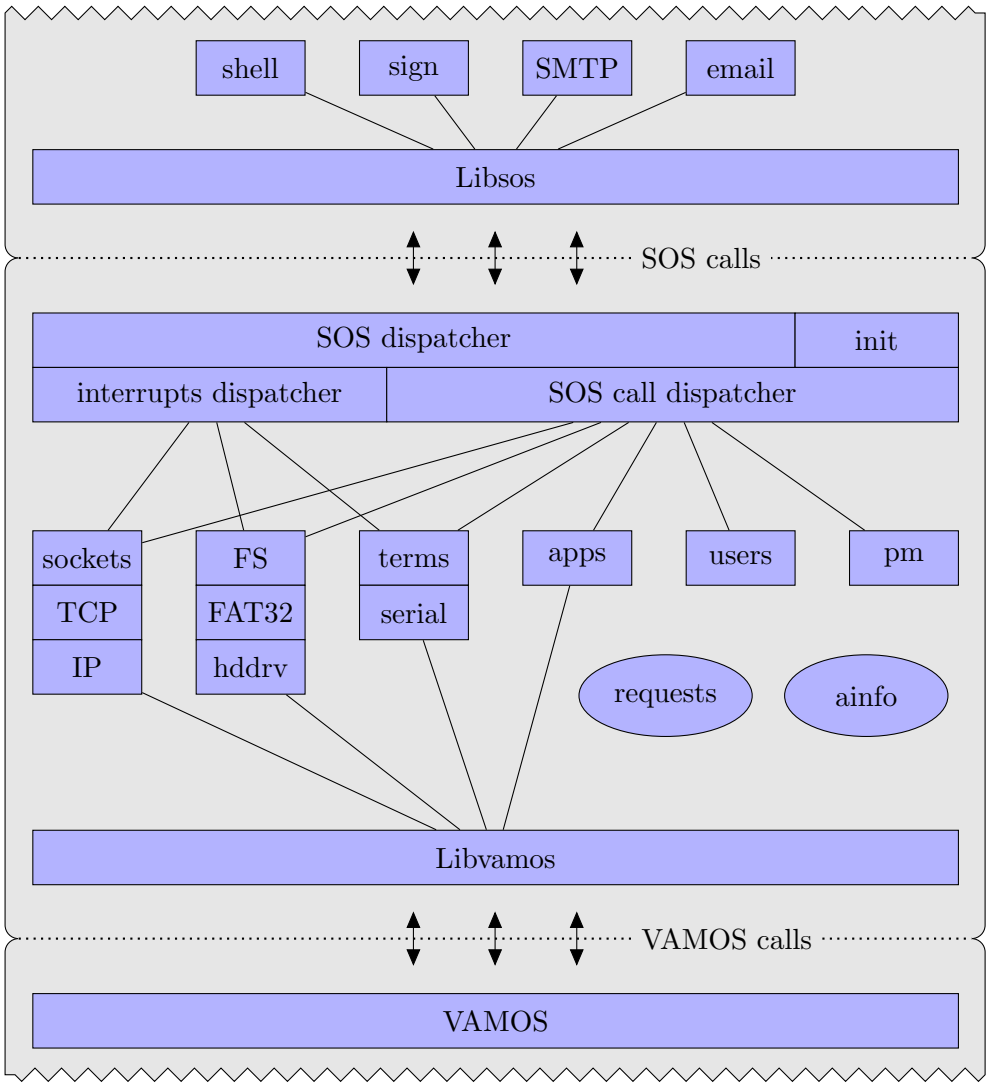

Figure 3.2: Overview of the Implementation. User applications use Libsos calls that wrap the actual SOS calls. The SOS server is implemented in a single user process. The main part of the SOS implementation are the device drivers. These drivers are implemented in different layers. The SOS implementation uses Libvamos calls to call the VAMOS micro kernel.

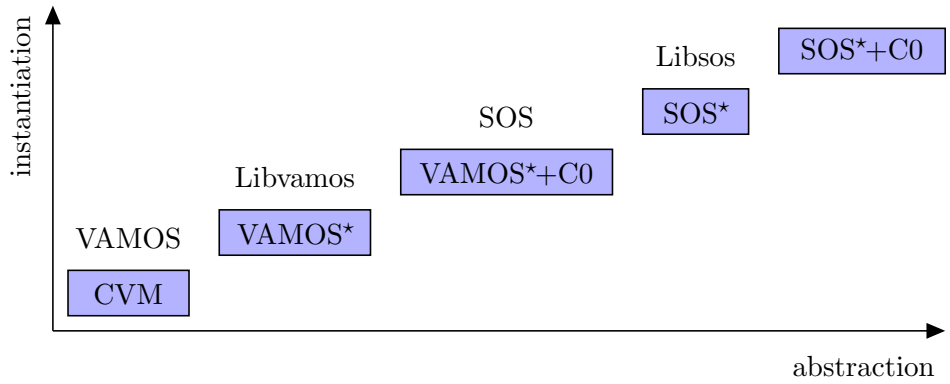

Figure 3.3: Verification Stairs.

Blocking requests. In some cases, an SOS call cannot be answered directly. For example, the lock for a file can only be granted, if it is available. If not, the request will be saved and answered as soon as another application releases it. Because the SOS only permits IPC requests with infinite timeouts, we can be sure that the original caller waits for the answer.

3.7.2 Model

The formal specification of the SOS implementation is established by the model SOS*, which is a computational model for communicating user applications. It is an abstraction of the VAMOS*+C0 model instantiated by the SOS implementation (Figure 3.3). In a sense, it subsumes all lower system layers and provides a coherent framework for user applications. SOS* hides as much as possible of the underlying hard- and software and provides a formal specification. SOS* is the main topic of the work at hand and will be treated in detail in Chapter 4.

3.8 Libsos

Analogously to Libvamos, we have implemented a C0 library for the SOS calls. This library is called Libsos. It relieves the application programmer from the burden of manually constructing IPC messages (that can be successfully interpreted by the SOS). The functions provided by Libsos are called SOS-call wrappers.[9] A complete list of these wrappers is provided in Table 3.3 on the following page. A detailed description (from a programmer's point of view) is part of the publicly accessible SOS implementation [Bog08a].[10]

[9] Below, if it is clear from the context, we will write "SOS call" as a shorthand for "SOS-call wrapper".

[10] The Libsos documentation is also available as Verisoft-internal Technical Report #13.

SOS Call	Description
sc_user_add	adds a user
sc_user_del	removes a user
sc_file_creat	creates a file
sc_file_truncate	reduces the size of a file
sc_file_unlink	removes a file
sc_file_info	retrieves information about a file
sc_file_write	writes to a file
sc_file_seek	changes the current position within a file
sc_file_read	reads from a file
sc_file_lock	locks a file for exclusive access
sc_file_unlock	unlocks a file
sc_file_chmod	changes permissions for a file
sc_file_chown	changes the owner of a file
sc_term_write	writes to the screen
sc_term_seek	changes the cursor position
sc_term_info	retrieves information about the screen
sc_term_read	reads from the keyboard
sc_socket_open	opens and binds a socket
sc_socket_listen	listens on a socket
sc_socket_connect	connects to a remote site
sc_socket_accept	accepts an incoming connection
sc_socket_read	reads from a socket
sc_socket_write	writes to a socket
sc_socket_close	closes a connection
sc_app_exec	executes a file
sc_app_fork	forks an application
sc_app_wait	waits for an application to terminate
sc_app_exit	exits an application
sc_pm_reg	registers a service
sc_pm_lookup	finds a service
sc_pm_unreg	unregisters a service

Table 3.3: SOS Calls

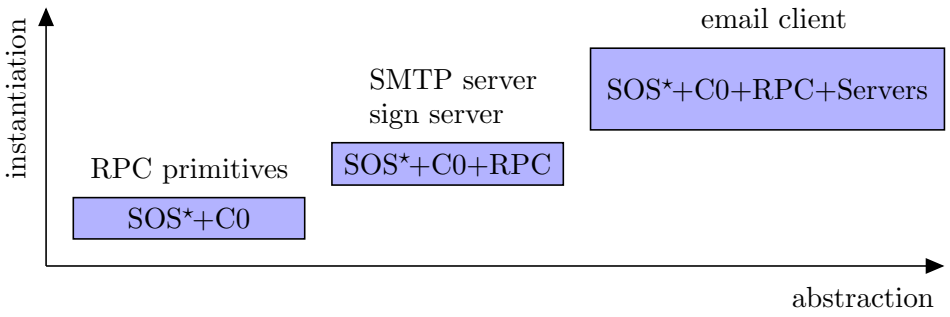

Figure 3.4: Extended Verification Stairs.

Other than for Libvamos, the verification of Libsos is straight forward (see Section 6.3). There would be no need for an additional layer in our verification stack. However, in order to follow the scheme of instantiation and abstraction, we call this layer SOS*+C0 (Figure 3.3 on page 27).

3.9 Applications

There are a number of user applications running on top of the SOS. Most notably: Remote Procedure Calls, an SMTP server as well as a signature server, and an email client.

In [Sha06], Andrey Shadrin describes his implementation of an interface compiler that provides primitives for an easier application of Remote Procedure Calls (RPCs). Using SOS calls, so-called RPC primitives simplify the task of locating service providers and transferring large and / or dynamic data structures between user applications.

Although not yet publicly available, a complete and formal description of RPC was presented in the Verisoft-internal Technical Report #68. Using the SOS specification, Eyad Alkassar formalized the semantics of the RPC primitives and the RPC protocol. Following the scheme of abstraction and instantiation, he instantiated SOS*+C0 with the implementation of the RPC primitives and abstracted that to the SOS*+C0+RPC model (Figure 3.4). Furthermore, as an example, he used this model to prove the correctness of an RPC server that provides a basic (mathematical) service [Alk08].

A large SOS application, namely an SMTP server, was implemented and verified by Langenstein et al. [LNRS07]. Their implementation comprises three modules: (i) a module that serves local requests (from an email client), (ii) a module that sends mail to the outside world, and (iii) a module that receives mail from the outside world. Using the entire spectrum of SOS calls, their SMTP server fully supports the SMTP standard [Pos82] as well as the standard email formats [Cro82].

The SMTP server has been specified and formally verified [LNRS07] in VSE

[HLS+96]. For that, the set of system calls and system call results (Σ_p and Ω_p, defined in § 4.3.8) were translated to VSE. In combination with an identifier for a sender and a receiver, these system calls and system call results form so-called events. Now, the operational behavior of the SMTP server was specified in terms of (valid) sequences of such events, so called histories. Langenstein et al. proved that their implementation obeys the specified set of histories. That is, they proved that their implementation meets their specification.[11]

Another application that runs on top of the SOS is an RSA-signature server. This RPC server provides two services: (i) it receives a text message and a private key and signs the message or (ii) it receives a signed text message and a public key and verifies the message.

As the SMTP server, the signature server has been specified and (partially) verified using VSE. Other than the SMTP server, the signature server has only few interaction with the operating system. In fact, only for receiving RPC calls and returning the corresponding results, SOS calls are necessary. Otherwise the signature server (only) performs local computations. Thus the correctness proof of the signature server is largely a proof of the C0 implementation of the RSA algorithm. In terms of interaction with the SOS, the correctness proof should be similar to the one provided for the example RPC server. Up to now, the VSE theories and proofs are not publicly available.

Finally, on top of the SMTP server and the signature server, Beuster et al. have implemented an email application [BB04, BHW06, BBuMW07]. Using the calls provided by the SOS, the SMTP server and the signature server, the email-client allows different users to compose, sign, and send as well as receive, verify, and read emails. The formal specification of the email client is provided in Isabelle / HOL. Using a reduced SOS* state and axiomatizing the inputs and outputs of the SMTP server and the signature server (in our terminology SOS*+C0+RPC+Servers), Beuster et al. were able to completely verify the C0 implementation of the email client [BHW06]. Their implementation and verification results are publicly available in [BBuMW07].

3.10 Running the Whole System

In order to comfortably test the SOS implementation we use the `dlxsim` simulator, which simulates the VAMP micro processor and provides basic means for debugging. Using QEMU [Qmu07], we are able to emulate a hard disk and the UART chip. Furthermore, combining QEMU and TUN / TAP [Tun07] we are able to provide access to the network card of the host. Figure 3.5 on the next page shows the SOS running the email client. A demonstration of two SOS instances communicating with each other via the Internet, sending and receiving signed emails, was given at the German Verification Day '07 [Bog07].

[11] Further documentation about the implementation and verification of the SMTP server

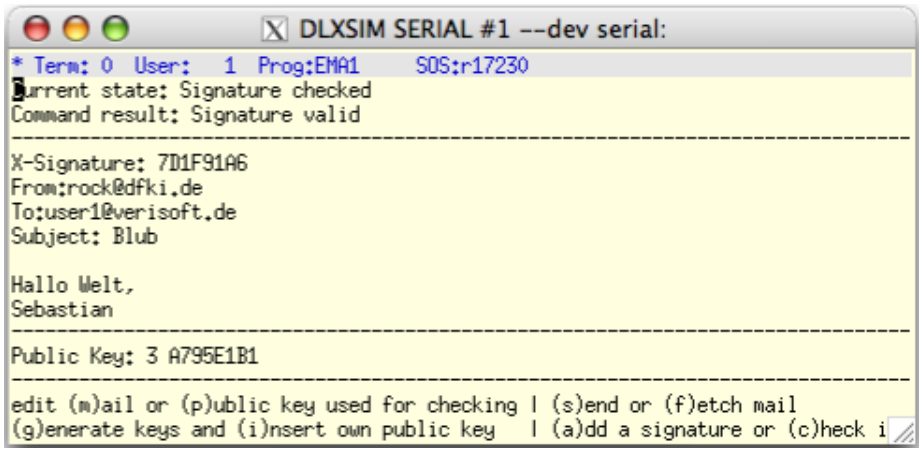

Figure 3.5: SOS Running the Email Client. Screenshot of the SOS (running on top of the `dlxsim`). Here, the email client occupies the first virtual terminal, displays a new email with a verified signature and waits for user input.

is provided in the Verisoft-internal Technical Reports #75, #76, and #77.

Chapter

4

SOS*

Contents

In this chapter 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.

4.1 Overview

Formally, SOS* is defined as a transition system:

$$\text{SOS}^\star = (\mathcal{S}, \mathcal{S}^0, \Sigma, \Omega, \Delta, \mathcal{R}).$$

Where,

- $\mathcal{S}$ is the set of possible configurations (the SOS* state space),
- $\mathcal{S}^0 \subset \mathcal{S}$ is the set of initial configurations,
- Σ is the set of inputs from the environment (external inputs),
- Ω is the set of outputs to the environment (external outputs),
- $\Delta \subset \mathcal{S} \times \Sigma_\varepsilon \times \mathcal{S} \times \Omega^*$ is the transition relation, and

- $\mathcal{R}$ 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 follow the approach used in VAMOS*+C0 and incorporate user applications in the form of self-contained I/O automata:

$$\text{APP}^\star = (\mathcal{S}_\text{p}, \Sigma_\text{p}, \Omega_\text{p}, \delta_\text{p}, \omega_\text{p}, \textit{vm-size}, \textit{interpret}).$$

Where,

- $\mathcal{S}_\text{p}$ 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),
- $\delta_\text{p} \in \mathcal{S}_\text{p} \times \Sigma_\text{p} \cup \{\varepsilon\} \to \mathcal{S}_\text{p}$ is the transition function,
- $\omega_\text{p} \in \mathcal{S}_\text{p} \to \Omega_\text{p} \cup \{\varepsilon\}$ computes the output for a given state,
- $\textit{vm-size} \in \mathcal{S}_\text{p} \to \mathbb{N}_{32}$ computes the size of the occupied virtual memory for a given state, and
- $\textit{interpret} \in \textit{word\_t}^* \to \mathcal{S}_\text{p}$ maps memory contents to an instance of the the application state space.

Describing user applications as self-contained automata has the advantage that the abstraction can be easily instantiated by different machine types. Adding a new 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.

Now, 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 arbitrary but fixed. Thus, these components can be SOS* parameters. Hence, we get the following updated definition of SOS*:

$$\text{SOS}^\star(\mathcal{S}_\text{p}, \delta_\text{p}, \omega_\text{p}, \textit{vm-size}, \textit{interpret}) = (\mathcal{S}, \mathcal{S}^0, \Sigma, \Omega, \Delta, \mathcal{R}, \Sigma_\text{p}, \Omega_\text{p}).$$

This last definition of SOS* is the final one. Below, we will describe each of its components in detail.

4.2 State Space

4.2.1 Users

SOS is a multi-user operating system. It allows different registered users to log in. A user is thereby referred to by their 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:[12]

$$udb : \mathcal{P}(uid\_t).$$

The system administrator, or super user, is a user with special privileges. This user is the only one allowed to perform certain administrative tasks. Registering new users, for example, can only be done by the super user. In SOS*, the user id of the super user is denoted by SU, where $\text{SU} \in uid\_t$.

4.2.2 Hard Disk and File System

Our SOS implementation supports a hard disk. The hard disk is presented to user applications in terms of a file system. Here, each file is owned by a particular user. Different types of file operations (e. g. read, write, or execute) are offered. In order to perform one of these operations, the particular file must be locked and the owner of the calling application must have the appropriate permissions.

In SOS*, the type *file_t* is used to represent a single file. This record type contains the fields *owner*, for the owner of the file; *pos*, for the current position within the file; *con*, for the contents of the file; and *perm*, for the permissions associated with the file. Here, the function *perm* maps different types of file operations, for example locking a file or retrieving information about a file, to sets of user ids. The type $fop\_t = \{\text{LCK}, \text{WRT}, \text{READ}, \text{CHMOD}, \text{EXEC}\}$ encodes the set of all types of file operations. Hence, *perm* stores what can be done and by whom it can be done. Before an application can access a file, it must obtain the lock for it. Such a lock guarantees exclusive file access. In SOS*, file locks are stored along with each file. For that, *file_t* contains the field *lock*. The type of *lock* is a list of handles. The formal representation of handles will be discussed in § 4.2.5. For now, note that these application identifiers are of type *hn_t*. In SOS*, the head of *lock* represents the application that currently owns the file lock and its tail represents lock requests from other applications.

[12] For technical reasons, identifiers in this mathematical specification, in the Isabelle / HOL specification [Bog08c], and in the implementation [Bog08a] have slightly different names. In order to allow easier navigation within one the later ones, Appendix A.1 provides the most important translations.

An empty list indicates that the file is currently not locked:

$$\mathit{file\_t} = \left\{ \begin{array}{ll} \mathit{owner} & : \mathit{uid\_t}, \\ \mathit{pos} & : \mathbb{N}_{32}, \\ \mathit{con} & : \mathit{word\_t}^*, \\ \mathit{perm} & : \mathit{fop\_t} \to \mathcal{P}(\mathit{uid\_t}), \\ \mathit{lock} & : \mathit{hn\_t}^* \end{array} \right\}.$$

The file data base is represented by the state-space component *fdb*. It is a function that maps file names to individual files. A file name, or file id, is thereby of type $\mathit{fid\_t} \subset \mathbb{N}_{32}$. Currently unassigned file ids are mapped to ε:

$$\mathit{fdb} : \mathit{fid\_t} \to \mathit{file\_t}_{\varepsilon}.$$

Because of the limited hard-disks space, we need to expose some more details of the file-system implementation.

File-system operations are based on words. The easiest way to store a file on the hard disk would be to store it as a continuous region of words. This, however, would result in either great fragmentation or much work in the case of changing a file's contents. A more advanced approach is used by the FAT32 file system [Mic07]. Here, the hard disk is divided up into identically sized clusters, i. e. small blocks of continuous space. Each file may occupy one or more of these clusters. Thus, a file is represented by a chain of clusters. However, these clusters are not necessarily stored adjacent to one another. Instead, the file allocation table (FAT) contains an entry for each cluster on the hard disk and provides the necessary links. Furthermore, it identifies reserved, bad, and unused clusters. As the cluster size is defined by the time the hard disk is formated, the FAT has a fixed size. Along with the FAT, directory tables are used to provide mappings between file names and starting clusters. These directory tables are special files on the hard disk. Other than the FAT, the size of directory tables changes as new files are added or existing ones delete. The root directory table (RDT) is the global entry point. It has a well-known starting cluster and, thus, allows us to find files in the root directory. These files may be standard files or directory tables. The latter are used to describe subdirectories.

Our file system implementation is based on the FAT32 file system standard. Other than in this standard, we only support a single directory level, i. e. none of the files in the root directory is a directory table. Furthermore, we do not compress the RDT, i. e. if a file is deleted, then the RDT file does not shrink. Instead, we only invalidate the corresponding directory entry and reuse it upon creating a new file. Together with some other restrictions, this simplification provides for a comparablely easy implementation and a fairly abstract specification.

In our specification, the number of words per cluster is denoted by $\mathrm{WPC} \in \mathbb{N}_{32}^{+}$. Knowing that, we can always compute the number of clusters occupied

by a file of a certain size, using the function *ocl*:

$$\begin{aligned} \mathit{ocl} &\in \mathbb{N} \to \mathbb{N} \\ \mathit{ocl}(\mathit{size}) &= \lceil \mathit{size}/\mathrm{WPC} \rceil. \end{aligned}$$

If the contents of a file is modified, then the available hard-disk space may change. In SOS*, we keep track of the available hard-disk space in terms of free clusters. The number of free clusters is maintained in the state-space component *free-clusters*:

$$\mathit{free\text{-}clusters} : \mathbb{N}.$$

In SOS*, the FAT is not visible. This is because it has a fixed size and the partitioning of the file contents into separate cluster is not visible.

For the RDT, this approach is not possible. This is because the size of the RDT file changes and thereby influences the results of some file operations. However, the number of clusters occupied by the RDT file can be computed from the information about the number of files on the hard disk, the number of directory entries per cluster, and the number of invalidated entries (holes) in the RDT. The number of files on the hard disk is already represented through the state-space component *fdb*. The number of directory entries per cluster is constant. In SOS*, this constant is denoted by $\mathrm{RDTEPC} \in \mathbb{N}_{32}^{+}$. Finally, the number of holes in the RDT is maintained in the state-space component *rdt-holes*:

$$\mathit{rdt\text{-}holes} : \mathbb{N}.$$

The FAT32 file-system specification does not consider file owners or file permissions. In SOS, however, we want this kind of access control. In the implementation, we maintain this information in the resource data base (RDB). Just like the RDT, the RDB is a, for user applications invisible, file on the hard disk. For each user-visible file it contains an entry. Each of these entries has a fixed size. If a file is created, a new entry is inserted into the RDB. If a file is deleted, the corresponding entry is invalidated. As for the RDT, we are not compressing the RDB. An RDB hole is overwritten, if a new file is created. Thus, the size of the RDB file sometimes increases but never shrinks. Creating a file fails, if there are no more holes in the RDB and there is not enough free space on the hard disk to increase the size of that file. In SOS*, we use the state-space component *rdb-holes* to keep track of the number of holes in the RDB.

$$\mathit{rdb\text{-}holes} : \mathbb{N}$$

As for the RDT, we need to know the number of RDB entries that can be stored in a single cluster. In SOS*, this number is denoted by $\mathrm{RDBEPC} \in \mathbb{N}_{32}^{+}$. Thus, knowing the number of visible files, the number of holes in the RDB, and RDBEPC, we can always compute the size of the (otherwise invisible) RDB file.

4.2.3 Serial Interface and Virtual Terminals

In order to allow users to interact with the system, the SOS supports a screen and a keyboard. For user applications, these two devices are combined and multiplexed to a number of so-called virtual terminals. Essentially, a user application may display characters on the screen of a virtual terminal or retrieve keyboard input from it.

In SOS⋆, the keyboard, the screen, and the corresponding device drivers are abstracted to the state-space component *tdb*, i. e. the terminal data base. This abstraction provides $\text{NT} \in \mathbb{N}_{32}^{+}$ virtual terminals, where each of them is identified by a terminal id $tid\_t = \{0, \ldots, \text{NT} - 1\}$.

Now, such a virtual terminal is the combination of an input and an output buffer.[13] Each terminal is connected to (at most) one user application. At any time, only one terminal has the focus. The terminal that currently has the focus, is stored in the state-space component *focus*:

$$focus : tid\_t.$$

If a terminal has the focus, then its output buffer is visible on the screen and the keyboard input is appended to its input buffer. The focus can be switched by pressing the special key $\text{STK} \in byte\_t$. The STK key is not appended to the input buffer. The contents of the input- and output buffer is retained while switching to other terminals but emptied if the terminal connection is passed to another application. Applications can not read from any terminal's output buffer, nor write to any terminal's input buffer. All terminals have the same set $\text{SCRC-OUT} \subset byte\_t \setminus \{\text{STK}\}$ of printable characters and the same set $\text{SCRC-IN} \subset byte\_t \setminus \{\text{STK}\}$ of allowed input characters. The size of the input queue is limited to $\text{TINMAX} \in \mathbb{N}_{32}^{+}$ characters. Finally, the dimensions of the user accessible screen are fixed to $\text{SCRX} \in \mathbb{N}_{32}^{+}$ and $\text{SCRY} \in \mathbb{N}_{32}^{+}$ and its area is denoted by $\text{SCRXY} \in \mathbb{N}_{32}^{+}$.

The type *term_t* describes a single virtual terminal. It contains the queue *in*, for keyboard input; the array *out*, representing the contents of the screen; and *pos*, for the current cursor position:

$$term\_t = \left\{ \begin{array}{l} in : byte\_t^{*}, \\ out : byte\_t^{*}, \\ pos : \mathbb{N}_{32} \end{array} \right\}.$$

Now, the terminal data base is a function that maps terminal ids to individual terminals:

$$tdb : tid\_t \rightarrow term\_t.$$

[13] In the following, if it is clear from the context, we will simply write "terminal" as a shorthand for "virtual terminal".

Note that in the implementation, the topmost row of the screen is reserved for a status line. This line contains: the information about a pending input request, the user id of the owner of the connected application, and the file id of the executable of the connected application. However, only the SOS can update this line. Thus, in SOS*, it is not included in the user-accessible screen area. Furthermore, when inspecting other state-space components, the contents of the status line can always be computed (*terminal-status* defined in § 4.3.3.1). Thus, *term_t* does not contain an explicit field for the status line.

4.2.4 Network Card and Sockets

In order to send and receive emails, a network card is included in the SOS.

In SOS*, the network card, its low level device driver, the TCP/IP implementation, and the socket interface are abstracted to the state-space component *sdb*, i. e. the socket data base. This abstraction provides a number of sockets, i. e. endpoints of connections, that may be created, accessed, and closed by means of socket calls and external input.

The type *socket_t* describes a single socket. It contains *state*, for the abstract socket state; *lpn*, for the local port number; *rna*, for the remote network address; and *rpn*, for the remote port number. Furthermore, it contains the queue *lq*, storing connection requests from remote sites; *in*, a queue holding input that was received from the outside world; and *out*, a queue holding output that was sent to the outside world. Additionally there is the counter *read*, indicating the number of bytes in the input queue that have been locally delivered and the counter *ack*, indicating the number of bytes in the output queue that have been acknowledged by the remote site:

$$socket\_t = \left\{ \begin{array}{ll} state & : sstate\_t, \\ lpn & : pn\_t, \\ lq & : (na\_t \times pn\_t)^*, \\ rna & : na\_t_\varepsilon, \\ rpn & : pn\_t_\varepsilon, \\ in & : byte\_t^*, \\ read & : \mathbb{N}, \\ out & : byte\_t^*, \\ ack & : \mathbb{N} \end{array} \right\}.$$

Here, a socket's abstract state (*sstate_t*) is BOUND, LISTEN, ACCEPTING, CONNECTING, ESTABLISHED, or REMOTE-CLOSED. The type $na\_t \subset \mathbb{N}_{32}$ is used for an abstract network address (IP addresses) and the type $pn\_t \subset \mathbb{N}_{32}$ is used for an abstract port number. A socket's remote network address and remote port are set to ε, if the socket is not part of an established connection.

Now, the socket data base is a function that maps socket ids $sid\_t \subset \mathbb{N}_{32}$ to individual sockets. Currently unassigned socket ids are mapped to ε:

$$sdb : sid\_t \rightarrow socket\_t_{\varepsilon}.$$

The implementation supports a single local network address. In SOS⋆ this address is represented by the state-space component *lna*:

$$lna : na\_t.$$

Note that the contents of *in* and *out* do not contain any protocol overhead; it represents pure payload. In addition, note that the number of unacknowledged characters in the output queue is limited to SOCK-WIN-SIZE $\in \mathbb{N}_{32}^{+}$, i. e. writing to a socket fails if this number is exceeded.

4.2.5 User Applications

The SOS supports communicating user applications. In Section 4.1, we already explained how user applications can be seen as self-contained 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: (i) a local state, i. e. the application's internal configuration, (ii) a number of process-related data structures, i. e. bookkeeping data structures about the user process maintained by the kernel, and (iii) a number of application-related data structures, i. e. bookkeeping data structures maintained by the SOS.

4.2.5.1 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 MAXPROCESSES $\in \mathbb{N}_{32}^{+}$. The kernel uses process identifiers (PIDs) to refer to specific processes. In SOS⋆, the set of all PIDs is represented by $pid\_t = \{1, \ldots, \text{MAXPROCESSES}\}$. 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 process ids are mapped to ε:

$$pdb\_t = pid\_t \setminus \{\text{OSPID}\} \rightarrow \mathcal{S}_{\text{p}} \cup \{\varepsilon\}$$

and:

$$pdb : pdb\_t.$$

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*.

4.2.5.2 Kernel Data Structures about User Processes

In SOS⋆, we inherit a number of data structures from VAMOS⋆+C0 that are necessary to characterize user applications. In the following paragraphs we discuss each of these data structures and then combine them into (SOS⋆) state space component *kds*.

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 ($hn\_t \subset \mathbb{N}_{32}$) and PIDs. This mapping is called the handle data base. In SOS⋆, the handle data base is represented by the *kds* component *hdb*:

$$hdb : pid\_t \times hn\_t \to pid\_t_{\varepsilon}.$$

There are a number of special handles provided. These are: (i) HN-NONE $\in$ hn_t, the pseudo handle identifying no process; (ii) HN-PARENT $\in$ hn_t, the handle identifying the parent process; and (iii) HN-SELF $\in$ hn_t, the handle used as self-reference.[14]

Stolen handle data base. If a process terminates, then all handles that point to this process become invalid. In order to be able to synchronously inform the affected processes about this asynchronous change in the handle data base, the so-called stolen handles are stored in the stolen handle data base.[15] In this data base, the kernel maintains for each PID the set of stolen handles. In SOS⋆, the stolen handle data base is represented by the *kds* component *sthdb*:

$$sthdb : pid\_t \to \mathcal{P}(hn\_t).$$

Rights data base. Along the road of handles are IPC rights. Even if a process has the handle for another process, it is desirable to allow a fine grained control over their communication. For that, handles are accompanied with rights. In SOS⋆, these rights are encoded by the set $rights\_t = \{\text{SND}, \text{REQ}, \text{MULT}, \text{FIN}\}$. Here:

- SND denotes the right to send a message to a certain process without waiting for an answer;
- REQ denotes the right to make a request, i. e. the right to send a message to a certain process while enforcing that the sender waits for an answer;

[14] For all user applications HN-PARENT maps to OSPID. This is because the SOS spawns all user applications.

[15] Here, "synchronously" means related / due to the computations of a particular process. The opposite is described by "asynchronously", which means independent from the computations of a particular process, but related to those of another process.

- MULT denotes the right to make a request and/or send operation multiple times; and
- FIN denotes the right to make a request and/or send operation specifying a finite timeout.

For each pair of processes, the kernel maintains the associated IPC rights. In SOS⋆, the rights data base is used to represent IPC rights. Here, the rights data base is represented by the *kds* component *rdb*:

$$rdb : pid\_t \times pid\_t_{\varepsilon} \to \mathcal{P}(rights\_t).$$

Wait data base. Finally, if a process makes an IPC request, it might take some time before an answer is returned. Meanwhile, the requesting process is not scheduled by the kernel. In SOS⋆, the wait data base is used to track the processes waiting for an answer to their requests. Here, the wait data base is represented by the *kds* component *wdb*:

$$wdb : pid\_t \to \mathbb{B}_{\varepsilon}.$$

Now, the handle data base, the stolen handle data base, the rights data base, and the wait data base are all data structures that are maintained by the kernel. In SOS⋆, they are combined in the kernel data structure type *kds_t*:

$$kds\_t = \left\{ \begin{array}{ll} hdb & : pid\_t \times hn\_t \to pid\_t_{\varepsilon}, \\ sthdb & : pid\_t \to \mathcal{P}(hn\_t), \\ rdb & : pid\_t \times pid\_t \to \mathcal{P}(rights\_t), \\ wdb & : pid\_t \to \mathbb{B}_{\varepsilon} \end{array} \right\},$$

and represented in the state space component *kds*

$$kds : kds\_t$$

Note that if a certain PID, or a handle, is unassigned, then the functions *hdb* and *wdb* map to ε, and the functions *sthdb* and *rdb* map to the empty set.

4.2.5.3 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⋆, this information is represented in a number of data structures of type *app_t*. This type contains the fields *parent*, for the handle of the parent

application; *owner*, for the owner of the application; and *term*, for the terminal id of the terminal this application is attached to. Furthermore, the field *sockets* is used to keep track of the sockets this application is connected to and the field *exec* stores the file id of the file being executed. Finally, the flag *wait* tells whether the parent application is waiting for this application to terminate and the flag *read* tells whether this application is waiting for keyboard input. If an application is not attached to a terminal, then *term* is set to ε. If an application has no parent application, as is true, for example, for the login shells, then *parent* has the value ε:

$$app\_t = \left\{ \begin{array}{ll} parent & : hn\_t_\varepsilon, \\ owner & : uid\_t, \\ term & : tid\_t_\varepsilon, \\ sockets & : \mathcal{P}(sid\_t), \\ exec & : fid\_t, \\ wait & : \mathbb{B}, \\ read & : \mathbb{B} \end{array} \right\}.$$

Now, the application data base is represented by the state-space component *adb*. It is a function that maps application handles to the associated information. Currently unassigned application handles are mapped to ε:

$$adb : hn\_t \rightarrow app\_t_\varepsilon.$$

4.2.6 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.[16]

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. $service\_t = iid\_t \times prcid\_t$.

Now, the portmapper data base is represented by the state-space component *pmdb*.

$$pmdb : pmdb\_t.$$

[16] Currently, 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 suffice to serve our needs.

Here, the type *pmdb_t* contains *serv*, the mapping between interface ids and the handles of the providing servers; *reg*, the set of registered services; and *known*, the set of known services:

$$pmdb\_t = \left\{ \begin{array}{ll} serv & : iid\_t \rightarrow hn\_t_{\varepsilon}, \\ reg & : \mathcal{P}(service\_t), \\ known & : \mathcal{P}(service\_t) \end{array} \right\}.$$

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.

4.2.7 Summing Up

Now, collecting all pieces, the SOS* state space has the following structure:

$$\mathcal{S} = \left\{ \begin{array}{ll} udb & : \mathcal{P}(uid\_t), \\ fdb & : fid\_t \rightarrow file\_t_{\varepsilon}, \\ rdt\text{-}holes & : \mathbb{N}, \\ free\text{-}clusters & : \mathbb{N}, \\ rdb\text{-}holes & : \mathbb{N}, \\ tdb & : tid\_t \rightarrow term\_t, \\ focus & : tid\_t, \\ sdb & : sid\_t \rightarrow socket\_t_{\varepsilon}, \\ lna & : na\_t, \\ pdb & : pdb\_t \\ kds & : kds\_t, \\ adb & : hn\_t \rightarrow app\_t_{\varepsilon}, \\ pmdb & : pmdb\_t \end{array} \right\}.$$

4.3 Transition Relation

There are three main classes of SOS* transitions: system calls, local computations, and transitions related to external inputs. The class of system calls is further differentiated into SOS calls and kernel calls. While kernel calls and local computations (Assembler / C0 semantics) are already described in [Tsy08, Lei08], the document at hand concentrates on SOS calls and user-visible external inputs.

SOS calls are passed to and answered by the SOS by means of IPC calls. In a sense, each SOS call has three phases: receiving the call, handling it, and returning the result. The handling of external inputs, can be broken apart in a similar way. Just like SOS calls, the SOS receives interrupt notifications through IPC calls. The transitions related to external inputs consist of a receive phase and a handle phase.[17] Now, the semantics of IPC calls are quite complex. We want to model SOS calls as atomic steps but instead of tackling all three / two phases at once, we will first of all (§ 4.3.1 – § 4.3.7) only describe the handler part. That is, we ignore the side effects of receiving SOS calls and interrupt notifications and do not describe how SOS-call results are applied to the state space of user applications, i. e. we only describe the local transitions of the user process that implements the SOS. This description of SOS-local transitions will be based on the standard alphabets of external inputs (Σ) and external outputs (Ω) and the intermediate alphabets of SOS-call inputs (Ω_{sc}) and SOS-call results (Σ_{sc}) (Figure 4.1 on the following page).

In the end (§ 4.3.8) we will integrate the SOS-local transitions into the global transition system. For that, we will add kernel calls, both as individual calls and as missing pieces under the hood of SOS calls, and extend the alphabets Ω_{sc} and Σ_{sc} to Ω_{p} and Σ_{p}. Finally, we will combine this global view of transitions related to kernel calls, SOS calls, and external inputs with local transitions of user applications. Thus, in the end, we present a unique transition relation for the entire system stack (Figure 4.9 on page 134).

Now, before we start specifying each of the SOS-call handlers, note the following. For each call, we will proceed in the same way. First, we present the C0 signature of the corresponding wrapper provided by the library Libsos, and shortly describe the behavior of the SOS call from a programmers point of view. This introduction should then serve as a motivation for the following mathematical specification of the particular SOS-call handler. Within this specification, we first of all define the necessary SOS inputs and SOS outputs, i. e. add the appropriate elements to the alphabets Ω_{sc} and Σ_{sc}, and then present the specification of the actual handler.

[17] Other than for SOS calls, for external inputs, there is usually no phase for returning results. In some cases, however, the external input resolves a pending application request (e. g. a user application waits for keyboard input) and thus results in sending a message to some user application.

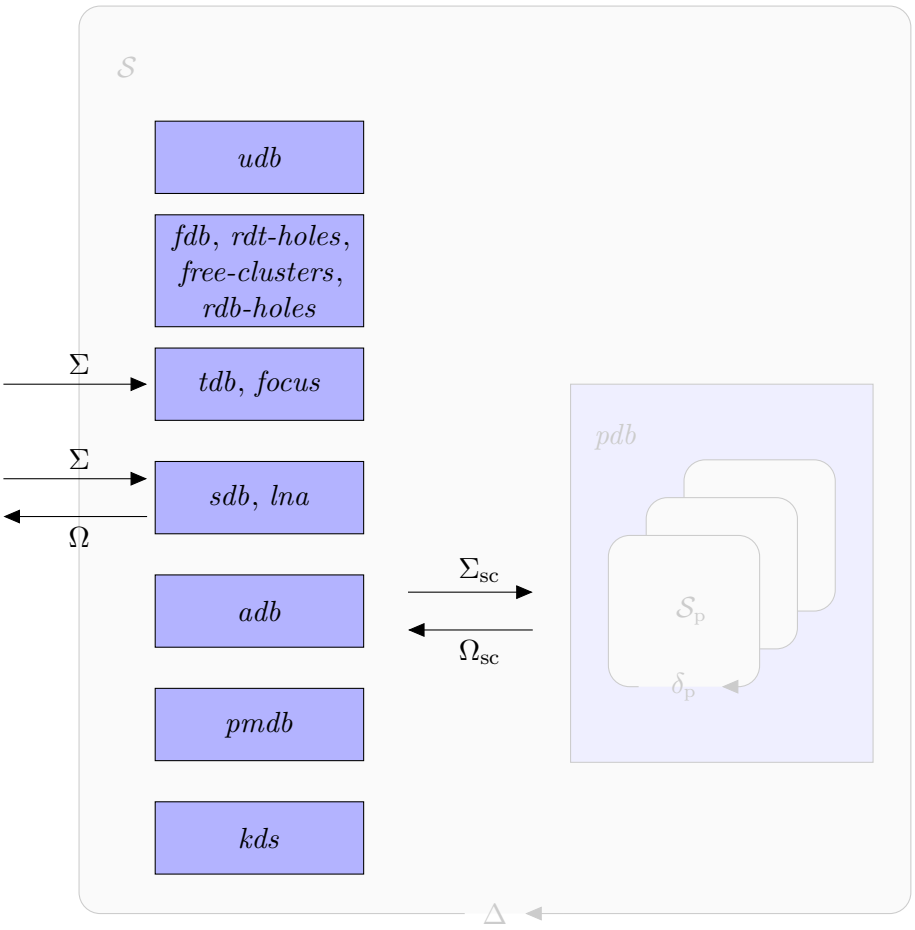

Figure 4.1: SOS⋆— The Small Picture. For the beginning, we specify the SOS-call handlers independently from the rest of the system.

4.3.1 Users

4.3.1.1 Adding a User

The library Libsos implements the following call that allows the super user to add a new user to the system:

```
int sc_user_add(unsigned int* uid).
```

In the SOS implementation, `sc_user_add` is handled by `sos_user_add`. If there is no error, then `sc_user_add` adds a new user and returns success as well the assigned user id. Note that the new user id returned via to call-by-reference parameter `uid`.

In the specification, we add UADD, as an abstract representation of the SOS call, to the input alphabet Ω_{sc}:

$$\Omega_{\text{sc}} \ni \text{UADD}$$

and SUCC-UADD *uid_t* as well as PERM and LIMIT, as abstract representations of the possible results, to the output alphabet Σ_{sc}:

$$\Sigma_{\text{sc}} \supset \{\text{SUCC-UADD } \mathit{uid}, \text{PERM}, \text{LIMIT} \mid \mathit{uid} \in \mathit{uid\_t}\}.$$

The behavior of `sos_user_add` is described by the function *uadd*. As we will see in § 4.3.8.1, it is the SOS-call dispatcher that recognizes UADD and calls *uadd*, passing along the current state and the handle of the calling application. Based on these two arguments, *uadd* computes the next state, a list of outputs to the environment, and the list of result messages:

$$\mathit{uadd} \in \mathcal{S} \times \mathit{hn\_t} \to \mathcal{S} \times \Omega^* \times (\mathit{hn\_t} \times \Sigma_{\text{sc}})^*.$$

For $\mathit{uadd}(s, hn)$, the following cases are considered:

- If the calling application a, with $a = s.adb(hn)$, is not owned by the super user, i. e. $a.owner \neq \text{SU}$, then the message (hn, PERM) is returned.
- If there is no more user ids available, i. e. if the set *free* of unassigned user ids is empty, then the message (hn, LIMIT) is returned.
- If the previous cases do not apply, then the user id *uid* is added to the user data base and a success message, including the new user id, is returned. Here, *uid* is computed to be the smallest unassigned user id, i. e. $\mathit{uid} = \mathit{min}(\mathit{free})$.

This adds up to the following definition of *uadd*:

$$uadd(s, hn) =$$

$$\begin{array}{lll} \textbf{let} & a & = s.adb(hn); \\ & free & = \{x \mid x \in uid\_t \wedge x \notin s.udb\}; \\ & uid & = min(free); \\ & s_1 & = s[\![udb := s.udb \cup \{uid\}]\!] \end{array}$$

$$\textbf{in} \quad \begin{cases} (s\ , [\,], [(hn, \text{PERM})]) & \textbf{if } a.owner \neq \text{SU}, \\ (s\ , [\,], [(hn, \text{LIMIT})]) & \textbf{else if } free = \{\,\}, \\ (s_1, [\,], [(hn, \text{SUCC-UADD } uid)]) & \textbf{else}. \end{cases}$$

Note that in the case of *uadd*, there are no outputs to the environment and no more than one result message. However, for a simpler formalization of the SOS-call dispatcher (§ 4.3.8.1), we specify all SOS handlers with the same result type. Furthermore, all handlers have at least the current state and the handle of the calling application as input arguments. Below, we will no longer explicitly mention these (standard) input arguments, nor explain the result type.

4.3.1.2 Removing a User

The library Libsos implements the following call that allows the super user to remove a user from the system:

```
int sc_user_del(unsigned int uid).
```

In the SOS implementation, `sc_user_del` is handled by `sos_user_del`. If there is no error, then, after the call is handled, `uid` is no longer registered.

In the specification, we add UDEL uid_t_ε, as an abstract representation of `sc_user_del`, to the input alphabet Ω_{sc}:

$$\Omega_{\text{sc}} \supset \{\text{UDEL } uid \mid uid \in uid\_t_\varepsilon\}$$

and SUCC and ARG, as possible results, to the output alphabet Σ_{sc}:

$$\Sigma_{\text{sc}} \supset \{\text{SUCC} \mid \text{ARG}\}.$$

The behavior of `sos_user_del` is described by the function *udel*. This function takes, as a call-specific argument, the user id of the user that should be removed:

$$udel \in \mathcal{S} \times hn\_t \times uid\_t_\varepsilon \to \mathcal{S} \times \Omega^* \times (hn\_t \times \Sigma_{\text{sc}})^*.$$

For $udel(s, hn, uid)$, the following cases are considered:

- If *uid* is not the user id of an existing user, i. e. $uid \notin s.udb$, then the message (hn, ARG) is returned.

- If the calling application is not owned by the super user, *uid* still owns some file or application, or if *uid* still exists in some file's permission set, then the message (hn, PERM) is returned.
- If none of the previous cases applies, then the *uid* is removed from the system and a success message returned. Here, removing the user from the system is as simple as removing the user id *uid* from the user data base.

This adds up to the following definition of *udel*:

$$udel(s, hn, uid) =$$

$$\textbf{let} \quad s_1 = s[\![udb := s.udb \setminus \{uid\}]\!]$$

$$\textbf{in} \quad \begin{cases} (s\ , [\,], [(hn, \text{ARG})]) & \textbf{if } uid \notin s.udb, \\ (s\ , [\,], [(hn, \text{PERM})]) & \textbf{else if } s.adb(hn).owner \neq \text{SU} \\ & \qquad \vee \exists x.\ s.fdb(x).owner = uid \\ & \qquad \vee \exists x.\ s.adb(x).owner = uid \\ & \qquad \vee \exists x, y.\ uid \in s.fdb(x).perm(y), \\ (s_1, [\,], [(hn, \text{SUCC})]) & \textbf{else}. \end{cases}$$

Note that in `sc_user_del`, the type used for the argument `uid` is `unsigned int`. In the SOS implementation, however, the largest valid user id is `127`. Thus, it would be of advantage to directly restrict the possible values in SOS's (binary) interface. Unfortunately this is not possible due to restrictions of the underlying IPC mechanism (see § 4.3.7). Instead, if an application calls `sc_user_del`, specifying a user id `>127`, then the handler `sos_user_del` returns an error. In the specification, the same happens. However, for a more intuitive signature of *udel*, we represent all values > 127 by ε. That is, all values of *uid* such that $uid \notin uid\_t$ are represented by ε. Thus, we use uid_t_ε rather than $\mathbb{N}_{32}$, for the third argument of *udel*. Below, we will use the same approach in many places, but no longer explicitly mention it.

4.3.2 File I/O

In the following subsection, we will specify SOS calls related to file I/O. We will describe calls that allow user applications to: create, lock and unlock, read and write, truncate, and remove files. We will also describe calls that allow user applications: to change the owner of a file, to change the permissions associated with files, and to retrieve the information about files.

Note that the necessary SOS* transitions are comparably easy. The reason for that is that a hard disk only responds to requests from within the system, i. e. there are no inputs from or outputs to the outside world. Furthermore,

most file operations are implemented to return a result within a single SOS cycle. That means that, except for locking a file, there is no need to maintain requests.

4.3.2.1 Create a File

The library Libsos implements the following call that allows a user application to create a new file:

```
int sc_file_creat(unsigned int fid).
```

In the SOS implementation, `sc_file_creat` is handled by `sos_file_creat`. If there is no error, then, after the call is handled, there is a new file with the id equal to `fid`.

In the specification, we add FCREAT fid_t_ε, as an abstract representation of `sc_file_creat`, to the input alphabet Ω_{sc}:

$$\Omega_{\text{sc}} \supset \{\text{FCREAT } fid \mid fid \in fid\_t_\varepsilon\}.$$

The behavior of `sos_file_creat` is described by the function *fcreat*. This function takes, as a call-specific argument, the desired file id:

$$fcreat \in \mathcal{S} \times hn\_t \times fid\_t_\varepsilon \to \mathcal{S} \times \Omega^* \times (hn\_t \times \Sigma_{\text{sc}})^*.$$

For $fcreat(s, hn, fid)$, the following cases are considered:

- If *fid* is not a valid file id, i. e. $fid = \varepsilon$, then the message (hn, ARG) is returned.
- If a file with id *fid* already exists, i. e. $s.fdb(fid) \neq \varepsilon$, then the message (hn, PERM) is returned.[18]
- Creating a new file requires an update of the RDT and the RDB. If the RDT does not contain holes, i. e. $s.rdt\text{-}holes = 0$, then it is necessary to extend the RDT data structure using an additional cluster and adding RDTEPC-1 new holes. The same is true for the RDB. If the RDB does not contain holes, i. e. $s.rdb\text{-}holes = 0$, then it is necessary to extend the RDB data structure using an additional cluster and adding RDBEPC $- 1$ new holes. That means, depending on $s.rdt\text{-}holes$ and $s.rdb\text{-}holes$, we need cl, with $cl = 2 - min(\{1, s.rdb\text{-}holes\}) - min(\{1, s.rdt\text{-}holes\})$, clusters. Now, if there are not enough free clusters, i. e. $free\text{-}clusters < cl$, then (only) the message (hn, LIMIT) is returned.

[18] In the following, we will use the phrase "file *fid*" as shorthand for "file with id *fid*". If clear from the context, we will use an analogous shorthand when referring to a user (*uid*), a terminal (*tid*), a socket (*sid*), or an application (*hn*).

- If none of the previous cases applies, then the values for *s.rdt-holes*, *s.rdb-holes*, and *free-clusters* may be updated, the new file f is added, and a success message returned. The new file is thereby owned by the user ao, it has an offset equal to 0, and the initial permissions are set to $perm'$. At this time, the file has no contents and it is not yet locked. Here, ao is the user id of the owner of the calling application, i. e. $ao = s.adb(hn).owner$, and $perm$ only allows ao to lock it, i. e. $perm' = \lambda x \in fop\_t.$ (**if** $x \neq$ LCK **then** $\{\}$ **else** $\{ao\}$).

This adds up to the following definition of *fcreat*:

$$fcreat(s, hn, fid) =$$

$$\begin{aligned}
\textbf{let}\quad & ao = s.adb(hn).owner;\\
& perm' = \lambda x \in fop_t.\ \begin{cases} \{\} & \textbf{if } x \neq \text{LCK},\\ \{ao\} & \textbf{else}; \end{cases}\\
& f = [\![owner = ao, pos = 0, con = [\,], perm = perm', lock = [\,]]\!];\\
& rdbh' = \begin{cases} s.rdb\text{-}holes - 1 & \textbf{if } s.rdb\text{-}holes > 0,\\ \text{RDBEPC} - 1 & \textbf{else}; \end{cases}\\
& rdth' = \begin{cases} s.rdt\text{-}holes - 1 & \textbf{if } s.rdt\text{-}holes > 0,\\ \text{RDTEPC} - 1 & \textbf{else}; \end{cases}\\
& cl = 2 - min(\{1, s.rdb\text{-}holes\}) - min(\{1, s.rdt\text{-}holes\});\\
& s_1 = s \left[\!\!\left[\begin{array}{ll} fdb(fid) & := f,\\ rdb\text{-}holes & := rdbh',\\ rdt\text{-}holes & := rdth',\\ free\text{-}clusters & := s.free\text{-}clusters - cl \end{array} \right]\!\!\right]
\end{aligned}$$

$$\textbf{in}\quad \begin{cases} (s\ , [\,], [(hn, \text{ARG})]) & \textbf{if } fid = \varepsilon,\\ (s\ , [\,], [(hn, \text{PERM})]) & \textbf{else if } s.fdb(fid) \neq \varepsilon,\\ (s\ , [\,], [(hn, \text{LIMIT})]) & \textbf{else if } s.free\text{-}clusters < cl,\\ (s_1, [\,], [(hn, \text{SUCC})]) & \textbf{else}. \end{cases}$$

4.3.2.2 Lock a File

The library Libsos implements the following call that allows a user application to try to lock a file, i. e. gain exclusive access to a file:

```
int sc_file_lock(unsigned int fid).
```

In the SOS implementation, `sc_file_lock` is handled by `sos_file_lock`. If there is no error, then, after the call is handled, only the caller can access the file.

In the specification, we add FLOCK $\mathit{fid\_t}_\varepsilon$, as an abstract representation of `sc_file_lock`, to the input alphabet Ω_{sc}:

$$\Omega_{\text{sc}} \supset \{\text{FLOCK } \mathit{fid} \mid \mathit{fid} \in \mathit{fid\_t}_\varepsilon\}.$$

The behavior of `sos_file_lock` is described by the function *flock*. This function takes, as a call-specific argument, the file id of the file to be locked:

$$\mathit{flock} \in \mathcal{S} \times \mathit{hn\_t} \times \mathit{fid\_t}_\varepsilon \to \mathcal{S} \times \Omega^* \times (\mathit{hn\_t} \times \Sigma_{\text{sc}})^*.$$

For $\mathit{flock}(s, \mathit{hn}, \mathit{fid})$, the following cases are considered:

- If the file f, with $f = s.\mathit{fdb}(\mathit{fid})$, does not exist, i. e. $f = \varepsilon$, then the message $(\mathit{hn}, \text{ARG})$ is returned.
- If the owner of the calling application a, with $a = s.\mathit{adb}(\mathit{hn})$, does not have the permission to lock the file, i. e. $a.\mathit{owner} \notin f.\mathit{perm}(\text{LCK})$, then the message $(\mathit{hn}, \text{PERM})$ is returned.
- If the number of files locked by a exceeds the constant MAXFLOCKS $\in \mathbb{N}_{32}^+$, i. e. $|\mathit{locks}| \geq \text{MAXFLOCKS}$, then the message $(\mathit{hn}, \text{LIMIT})$ is returned. Here, *locks* is the set of file ids x that satisfy $\mathit{hn} = s.\mathit{fdb}(x).\mathit{lock}[0]$.
- It is not considered as an error, if a already has the lock for f. Thus, in this case the message $(\mathit{hn}, \text{SUCC})$ is returned.
- If none of the previous cases applies, then *hn* is appended to $f.\mathit{lock}$. If this makes *hn* the head of the list, i. e. the new lock owner, then the message $(\mathit{hn}, \text{SUCC})$ is returned. If it is not the head of the list, then there is no immediate result. Instead, a result may be returned, if some other application releases the lock for f (*funlock* defined in § 4.3.2.3).

This adds up to the following definition of *flock*:

$$\mathit{flock}(s, \mathit{hn}, \mathit{fid}) =$$

$$\begin{aligned}
\textbf{let}\quad & a = s.\mathit{adb}(\mathit{hn});\\
& f = s.\mathit{fdb}(\mathit{fid});\\
& \mathit{locks} = \{x \mid s.\mathit{fdb}(x) \neq \varepsilon \wedge \mathit{hn} = s.\mathit{fdb}(x).\mathit{lock}[0]\};\\
& s_1 = s[\![\mathit{fdb}(\mathit{fid}).\mathit{lock} := f.\mathit{lock} \circ [\mathit{hn}]]\!]
\end{aligned}$$

in
$$\begin{cases} (s\ , [\,], [(hn, \text{ARG})]) & \textbf{if } \mathit{fid} = \varepsilon \vee f = \varepsilon, \\ (s\ , [\,], [(hn, \text{PERM})]) & \textbf{else if } a.\mathit{owner} \notin f.\mathit{perm}(\text{LCK}), \\ (s\ , [\,], [(hn, \text{LIMIT})]) & \textbf{else if } |\mathit{locks}| \geq \text{MAXFLOCKS}, \\ (s\ , [\,], [(hn, \text{SUCC})]) & \textbf{else if } hn = f.\mathit{lock}[0], \\ (s_1, [\,], [\,]) & \textbf{else if } f.\mathit{lock} \neq [\,], \\ (s_1, [\,], [(hn, \text{SUCC})]) & \textbf{else}. \end{cases}$$

Note that an application is blocked until it receives a result. Because of this, it is impossible that the calling application is already in the list $f.\mathit{lock}$, but not the head of that list. Thus, we can safely append hn without violating the predicate *inv-unique-lock-requests*. Here, the predicate *inv-unique-lock-requests* is an invariant for SOS*, which is satisfied, if the entries in the *lock* lists are unique:

$$\begin{gathered} \mathit{inv\text{-}unique\text{-}lock\text{-}requests} \in \mathcal{S} \to \mathbb{B} \\ \mathit{inv\text{-}unique\text{-}lock\text{-}requests}(s) \equiv \\ \forall \mathit{fid}.\ s.\mathit{fdb}(\mathit{fid}) \neq \varepsilon \\ \Longrightarrow \\ \forall i, j \in \mathbb{N}.\ \ 0 \leq i < j < \mathit{length}(s.\mathit{fdb}(\mathit{fid}).\mathit{lock}) \\ \wedge\ s.\mathit{fdb}(\mathit{fid}).\mathit{lock}[i] \neq s.\mathit{fdb}(\mathit{fid}).\mathit{lock}[j]. \end{gathered}$$

4.3.2.3 Unlock a File

As counterpart to `sc_file_lock`, the library Libsos provides the following call that allows a user application to release the lock on a file:

```
int sc_file_unlock(unsigned int fid).
```

In the SOS implementation, `sc_file_unlock` is handled by `sos_file_unlock`.

In the specification, we add FUNLOCK $\mathit{fid\_t}_\varepsilon$, as an abstract representation of `sc_file_unlock`, to the input alphabet Ω_{sc}:

$$\Omega_{\text{sc}} \supset \{\text{FUNLOCK } \mathit{fid} \mid \mathit{fid} \in \mathit{fid\_t}_\varepsilon\}$$

and LOCK, as an abstract representation of a possible result, to the output alphabet Σ_{sc}:

$$\Sigma_{\text{sc}} \ni \text{LOCK}.$$

The behavior of `sos_file_unlock` is described by the function *funlock*. This function takes, as a call-specific argument, the file id of the file that should be unlocked:

$$\mathit{funlock} \in \mathcal{S} \times \mathit{hn\_t} \times \mathit{fid\_t}_\varepsilon \to \mathcal{S} \times \Omega^* \times (\mathit{hn\_t} \times \Sigma_{\text{sc}})^*.$$

For $\mathit{funlock}(s, hn, \mathit{fid})$, the following cases are considered:

- If the file f, with $f = s.fdb(fid)$, does not exist, then the message (hn, ARG) is returned.
- If the calling application does not have the lock on the file, i. e. $hn \neq f.lock[0]$, then the message (hn, LOCK) is returned.
- If f is locked, but there is no other application waiting for the lock, i. e. $length(f.lock) = 1$, then the lock is released, the position within the file reset to 0, and the message (hn, SUCC) returned.
- If f is locked and there is at least one application waiting for the lock, then the lock is passed to the next application and the position in the file reset to 0. Furthermore, both applications, the one that released the lock and the new lock owner, are informed about the success.

This adds up to the following definition of *funlock*:

$$funlock(s, hn, fid) =$$

$$\textbf{let} \quad \begin{aligned} f &= s.fdb(fid); \\ hn_2 &= f.lock[1]; \\ s_1 &= s \left[\!\!\left[\begin{array}{l} fdb(fid).pos := 0, \\ fdb(fid).lock := tail(f.lock) \end{array} \right]\!\!\right] \end{aligned}$$

$$\textbf{in} \quad \begin{cases} (s\ , [\,], [(hn, \text{ARG})]) & \textbf{if } fid = \varepsilon \vee f = \varepsilon, \\ (s\ , [\,], [(hn, \text{LOCK})]) & \textbf{else if } hn \neq f.lock[0], \\ (s_1, [\,], [(hn, \text{SUCC})]) & \textbf{else if } length(f.lock) = 1, \\ (s_1, [\,], [(hn, \text{SUCC}), (hn_2, \text{SUCC})]) & \textbf{else}. \end{cases}$$

Note that because of the invariant *inv-unique-lock-requests*, we know that $hn \neq hn_2$. Therefore, we can be sure that we will not accidently send both success messages to the same application.

Further note, *funlock* either removes entries form the queue $s.fdb(fid).lock$ or does not modify it. Hence, we can be sure that *inv-unique-lock-requests* is not violated. Now, since *flock* and *funlock* are the only functions that (potentially) modify file locks, we can be sure that *inv-unique-lock-requests* is preserved throughout the whole SOS* model.

4.3.2.4 Truncate a File

Before specifying how files are truncated, two auxiliary functions should be introduced.

The predicate *faccess-legal* is used to check whether some file access is legal. *faccess-legal*(s, hn, fid, fop) is satisfied, if fid is the file id of an existing file, this

file is currently locked by the application with handle hn, and the corresponding application owner has the permission to perform the file operation fop:

$$\begin{gathered}\mathit{faccess\text{-}legal} \in \mathcal{S} \times \mathit{hn\_t} \times \mathit{fid\_t}_{\varepsilon} \times \mathit{fop\_t} \to \mathbb{B} \\ \mathit{faccess\text{-}legal}(s, \mathit{hn}, \mathit{fid}, \mathit{fop}) \equiv \\ \mathit{fid} \neq \varepsilon \wedge s.\mathit{fdb}(\mathit{fid}) \neq \varepsilon \wedge \mathit{hn} = s.\mathit{fdb}(\mathit{fid}).\mathit{lock}[0] \\ \wedge\, s.\mathit{adb}(\mathit{hn}).\mathit{owner} \in s.\mathit{fdb}(\mathit{fid}).\mathit{perm}(\mathit{fop}).\end{gathered}$$

If *faccess-legal* is not satisfied, then the function *faccess-error* returns the appropriate message to the application hn:

$$\begin{gathered}\mathit{faccess\text{-}error} \in \mathcal{S} \times \mathit{hn\_t} \times \mathit{fid\_t}_{\varepsilon} \to \mathcal{S} \times \Omega^* \times (\mathit{hn\_t} \times \Sigma_{\mathrm{sc}})^* \\ \mathit{faccess\text{-}error}(s, \mathit{hn}, \mathit{fid}) = \\ \begin{cases} (s, [\,], [(\mathit{hn}, \mathrm{ARG})]) & \textbf{if } \mathit{fid} = \varepsilon \vee s.\mathit{fdb}(\mathit{fid}) = \varepsilon, \\ (s, [\,], [(\mathit{hn}, \mathrm{LOCK})]) & \textbf{else if } \mathit{hn} \neq s.\mathit{fdb}(\mathit{fid}).\mathit{lock}[0], \\ (s, [\,], [(\mathit{hn}, \mathrm{PERM})]) & \textbf{else.} \end{cases}\end{gathered}$$

Now, for truncating a file, i. e. shortening its contents, the library Libsos implements the following call:

```
int sc_file_truncate(unsigned int fid, unsigned int len).
```

In the SOS implementation, `sc_file_truncate` is handled by `sos_file_truncate`. If there is no error, then, after the call is handled, the size of the file `fid` is at most `len` words. Only the first `len` words are kept. Any additional data is lost. If the file is smaller, then nothing changes.

In the specification, we add FTRUNCATE $\mathit{fid\_t}_{\varepsilon}$ $\mathbb{N}_{32}$, as an abstract representation of `sc_file_truncate`, to the input alphabet Ω_{sc}:

$$\Omega_{\mathrm{sc}} \supset \{\mathrm{FTRUNCATE}\ \mathit{fid}\ \mathit{len} \mid \mathit{fid} \in \mathit{fid\_t}_{\varepsilon} \wedge \mathit{len} \in \mathbb{N}_{32}\}.$$

The behavior of `sos_file_truncate` is described by the function *ftruncate*. This function takes, as call-specific arguments, the id of the file and the desired new length:

$$\mathit{ftruncate} \in \mathcal{S} \times \mathit{hn\_t} \times \mathit{fid\_t}_{\varepsilon} \times \mathbb{N}_{32} \to \mathcal{S} \times \Omega^* \times (\mathit{hn\_t} \times \Sigma_{\mathrm{sc}})^*.$$

For $\mathit{ftruncate}(s, \mathit{hn}, \mathit{fid}, \mathit{len})$, the following cases are considered:

- If $\mathit{faccess\text{-}legal}(s, \mathit{hn}, \mathit{fid}, \mathrm{WRITE})$ is not satisfied, then the result is computed and returned by $\mathit{faccess\text{-}error}(s, \mathit{hn}, \mathit{fid})$.
- If the previous case does not apply, then the file f, with $f = s.\mathit{fdb}(\mathit{fid})$, is truncated and the message $(\mathit{hn}, \mathrm{SUCC})$ returned. The value size', i. e. the new size of f, is thereby computed as the minimum of the desired file

length and the size of the original file. If the size of the file is reduced, clusters may need to be freed. The number cl of clusters that are freed, is computed as the difference of the number of clusters occupied by the original file and the number of clusters occupied by the truncated file. The new current position within the file is computed as the minimum of the old position and the new file size.

This adds up to the following definition of *ftruncate*:

$$\mathit{ftruncate}(s, hn, \mathit{fid}, \mathit{len}) =$$

$$\begin{array}{lll}
\textbf{let} & f & = s.\mathit{fdb}(\mathit{fid}); \\
& \mathit{size} & = \mathit{length}(f.\mathit{con}); \\
& \mathit{size}' & = \mathit{min}(\{\mathit{len}, \mathit{size}\}); \\
& cl & = \mathit{ocl}(\mathit{size}) - \mathit{ocl}(\mathit{size}'); \\
& s_1 & = s \left[\!\!\left[\begin{array}{l} \mathit{fdb}(\mathit{fid}).\mathit{con} := \mathit{take}(\mathit{size}', f.\mathit{con}), \\ \mathit{fdb}(\mathit{fid}).\mathit{pos} := \mathit{min}(\{f.\mathit{pos}, \mathit{size}'\}), \\ \mathit{free\text{-}clusters} := s.\mathit{free\text{-}clusters} + cl \end{array} \right]\!\!\right]
\end{array}$$

$$\textbf{in} \quad \begin{cases} \mathit{faccess\text{-}error}(s, hn, \mathit{fid}) & \textbf{if } \neg \mathit{faccess\text{-}legal}(s, hn, \mathit{fid}, \text{WRITE}), \\ (s_1, [\,], [(hn, \text{SUCC})]) & \textbf{else}. \end{cases}$$

4.3.2.5 Delete a File

The library Libsos implements the following call that allows a user application to delete a file:

```
int sc_file_unlink(unsigned int fid).
```

In the SOS implementation, `sc_file_unlink` is handled by `sos_file_unlink`.

In the specification, we add FUNLINK $\mathit{fid\_t}_\varepsilon$, as an abstract representation of `sc_file_unlink`, to the input alphabet Ω_{sc}:

$$\Omega_{\text{sc}} \supset \{\text{FUNLINK } \mathit{fid} \mid \mathit{fid} \in \mathit{fid\_t}_\varepsilon\}.$$

The behavior of `sos_file_unlink` is described by the function *funlink*. This function takes, as a call-specific argument, the file id of the file to delete:

$$\mathit{funlink} \in \mathcal{S} \times \mathit{hn\_t} \times \mathit{fid\_t}_\varepsilon \to \mathcal{S} \times \Omega^* \times (\mathit{hn\_t} \times \Sigma_{\text{sc}})^*.$$

For $\mathit{funlink}(s, hn, \mathit{fid})$, the following cases are considered:

- If $\mathit{faccess\text{-}legal}(s, hn, \mathit{fid}, \text{WRITE})$ is not satisfied, then the result is computed and returned by $\mathit{faccess\text{-}error}(s, hn, \mathit{fid})$.

- If the previous case does not apply, then the file f, with $f = s.fdb(fid)$, is deleted, the lock released, the previously occupied clusters freed, and a success message returned. As the file is removed from the RDT and the RDB, the number of holes in these data structures increase. Because of this, $s.rdt\text{-}holes$ and $s.rdb\text{-}holes$ are each incremented by 1. Finally, all pending requests to lock the file f are canceled, i. e. the corresponding applications receive the result ARG.

This adds up to the following definition of *funlink*:

$$funlink(s, hn, fid) =$$

$$\begin{array}{ll}
\textbf{let} & f = s.fdb(fid); \\
& s_1 = s \left[\!\!\left[\begin{array}{ll} fdb(fid) & := \varepsilon, \\ free\text{-}clusters & := s.free\text{-}clusters + ocl(length(f.con)), \\ rdt\text{-}holes & := s.rdt\text{-}holes + 1, \\ rdb\text{-}holes & := s.rdb\text{-}holes + 1 \end{array} \right]\!\!\right]; \\
& m = map((\lambda x.\ (x, \text{ARG})), tail(f.lock)) \\
\textbf{in} & \begin{cases} access\text{-}error(s, hn, fid) & \textbf{if } \neg access\text{-}legal(s, hn, fid, \text{WRITE}), \\ (s_1, [\,], (hn, \text{SUCC})\#m) & \textbf{else.} \end{cases}
\end{array}$$

As before, because of the invariant *inv-unique-lock-requests*, we know that all entries in the list $tail(f.lock)$ are unique and different from hn. Therefore, we can be sure that we will not accidently send several messages to the same application.

4.3.2.6 Retrieve Information about a File

The library Libsos implements the following call that allows a user application to retrieve information about individual files:

```
int sc_file_info( unsigned int fid, unsigned int* fidr,
                  unsigned int* owner, unsigned int* size,
                  unsigned int* lock, unsigned int* perm,
                  unsigned int* fidn).
```

In the SOS implementation, `sc_file_info` is handled by `sos_file_info`. If there was no error, then `sc_file_info` returns the id of the file that was inspected, its owner, its size, the information whether the file is currently locked by the calling application, the permissions for the owner of the calling application, and the file id of the file with the next bigger file id. If `sc_file_info` is called with `fid` equal to `SOS_NIL`, then information about the file with the smallest file

id is returned.[19] If there is no file with a bigger file id, then `SOS_NIL` is returned for `fidn`.

In the specification, we add FINFO $\mathit{fid\_t}_\varepsilon \cup \text{NIL}$, as an abstract representation of the SOS call, to the input alphabet Ω_{sc}:

$$\Omega_{\text{sc}} \supset \{\text{FINFO } \mathit{fid} \mid \mathit{fid} \in \mathit{fid\_t}_\varepsilon \cup \text{NIL}\}$$

and SUCC-FINFO $\mathit{fid\_t}\ \mathit{uid\_t}\ \mathbb{N}_{32}\ \mathbb{B}\ \mathcal{P}(\mathit{fop\_t})\ \mathit{fid\_t} \cup \text{NIL}$, as an abstract representation of a possible result, to the output alphabet Σ_{sc}:

$$\Sigma_{\text{sc}} \supset \left\{ \begin{array}{l} \text{SUCC-FINFO } \mathit{fidr}\ \mathit{uid}\ \mathit{size}\ \mathit{lock}\ \mathit{perm}\ \mathit{fidn} \mid \\ \mathit{fidr\_t} \in \mathit{fid\_t} \wedge \mathit{uid} \in \mathit{uid\_t} \wedge \mathit{size} \in \mathbb{N}_{32} \wedge \mathit{lock} \in \mathbb{B} \\ \wedge\ \mathit{perm} \in \mathcal{P}(\mathit{fop\_t}) \wedge \mathit{fidn} \in \mathit{fid\_t} \cup \text{NIL} \end{array} \right\}.$$

The behavior of `sos_file_info` is described by the function *finfo*. This function takes, as a call-specific argument, the file id of the file for which information should be retrieve:

$$\mathit{finfo} \in \mathcal{S} \times \mathit{hn\_t} \times \mathit{fid\_t}_\varepsilon \cup \text{NIL} \to \mathcal{S} \times \Omega^* \times (\mathit{hn\_t} \times \Sigma_{\text{sc}})^*.$$

For $\mathit{finfo}(s, \mathit{hn}, \mathit{fid})$, the following cases are considered:

- The set $\mathit{fids} = \{x \mid s.\mathit{fdb}(x) \neq \varepsilon\}$ is the set of all assigned file ids. If the supplied *fid* is equal to NIL, then *fidr* is set to be the smallest assigned file id, or ε, if there are no files. If the supplied *fid* is not equal to NIL, then *fidr* is set to the value of *fid*. Now, if *fidr* is not a valid file id, i. e. $\mathit{fidr} = \varepsilon$, or the file f, with $f = s.\mathit{fdb}(\mathit{fidr})$, does not exist, then the message $(\mathit{hn}, \text{ARG})$ is returned.

- If the previous case does not apply, then the information about the file f is compiled and the message SUCC-FINFO *fidr uid size lock perm fidn* returned. Here, *uid* is the owner of the file and *size* the length of its contents. The boolean flag *lock* is TRUE, if the calling application currently has the lock for the file f. The set *perm*, with $\mathit{perm} = \{x \mid \mathit{ao} \in f.\mathit{perm}(x)\}$, contains the file permissions for the owner of the calling application. The value of *fidn* is the file id of the next file, i. e. the minimum of all assigned file ids that are greater than *fidr*. If there are no assigned file ids that are greater than *fidr*, then *fidn* is set to NIL.

This adds up to the following definition of *finfo*:

[19] Combining this with the information about the next bigger file id, it is possible to gather information about all files in the file system. This is, for example, necessary to implement the `ls` or `dir` command.

$$\mathit{finfo}(s, hn, \mathit{fid}) =$$

$$
\begin{array}{lll}
\textbf{let} & \mathit{fids} & = \{x \mid s.\mathit{fdb}(x) \neq \varepsilon\}; \\
& \mathit{fidr} & = \begin{cases} \mathit{fid} & \textbf{if } \mathit{fid} \neq \text{NIL}, \\ \mathit{min}(\mathit{fids}) & \textbf{else if } \mathit{fids} \neq \{\}, \\ \varepsilon & \textbf{else}; \end{cases} \\
& f & = s.\mathit{fdb}(\mathit{fidr}); \\
& \mathit{uid} & = f.\mathit{owner}; \\
& \mathit{size} & = \mathit{length}(f.\mathit{con}); \\
& \mathit{lock} & = \begin{cases} \text{TRUE} & hn = f.\mathit{lock}[0], \\ \text{FALSE} & \textbf{else}; \end{cases} \\
& \mathit{ao} & = s.\mathit{adb}(hn).\mathit{owner}; \\
& \mathit{perm} & = \{x \mid \mathit{ao} \in f.\mathit{perm}(x)\}; \\
& \mathit{fidn} & = \begin{cases} \mathit{min}(\{x \mid x \in \mathit{fids} \wedge x > \mathit{fidr}\}) & \textbf{if } \exists x \in \mathit{fids}.\ x > \mathit{fidr}, \\ \text{NIL} & \textbf{else}; \end{cases} \\
& m & = [(hn, \text{SUCC-FINFO } \mathit{fidr}\ \mathit{uid}\ \mathit{size}\ \mathit{lock}\ \mathit{perm}\ \mathit{fidn})] \\
\textbf{in} & & \begin{cases} (s, [\,], [(hn, \text{ARG})]) & \textbf{if } \mathit{fidr} = \varepsilon \vee f = \varepsilon, \\ (s, [\,], m) & \textbf{else}. \end{cases}
\end{array}
$$

4.3.2.7 Write to a File

For writing to a file, the library Libsos implements the following call:

```
int sc_file_write( unsigned int fid, unsigned int len,
                   sos_buffer_t buf, unsigned int* pos,
                   unsigned int* size).
```

In the SOS implementation, `sc_file_write` is handled by `sos_file_write`. If there is no error, then, after the call is handled, the first `len` words from the buffer `buf` are written to the file with the id `fid`. Thereby writing starts at the current position. Any previous content is overwritten and the file is, if necessary, extended. The current position within the file is updated and the new position as well as the new file size are returned to the calling application.

In the specification, we add FWRITE $\mathit{fid\_t}_\varepsilon$ $\mathit{word\_t}^*$, as an abstract representation of `sc_file_write`, to the input alphabet Ω_{sc}. Note that we use $\mathit{word\_t}^*$ as an abstraction that combines `buf` and `len`:

$$\Omega_{\text{sc}} \supset \{\text{FWRITE } \mathit{fid}\ \mathit{words} \mid \mathit{fid} \in \mathit{fid\_t}_\varepsilon \wedge \mathit{words} \in \mathit{word\_t}^*\}$$

Furthermore, we add SUCC-FWRITE $\mathbb{N}_{32}$ $\mathbb{N}_{32}$, as an abstract representation of a possible result, to the output alphabet Σ_{sc}:

$$\Sigma_{\text{sc}} \supset \{\text{SUCC-FWRITE } \mathit{pos}\ \mathit{size} \mid \mathit{pos}, \mathit{size} \in \mathbb{N}_{32}\}.$$

The behavior of `sos_file_write` is described by the function *fwrite*. This function takes, as call-specific arguments, the id of the file and the string that should be written to the file:

$$\mathit{fwrite} \in \mathcal{S} \times \mathit{hn\_t} \times \mathit{fid\_t}_{\varepsilon} \times \mathit{word\_t}^* \to \mathcal{S} \times \Omega^* \times (\mathit{hn\_t} \times \Sigma_{\text{sc}})^*.$$

For $\mathit{fwrite}(s, \mathit{hn}, \mathit{fid}, \mathit{words})$, the following cases are considered:

- If $\mathit{faccess\text{-}legal}(s, \mathit{hn}, \mathit{fid}, \text{WRITE})$ is not satisfied, then the result is computed and returned by $\mathit{faccess\text{-}error}(s, \mathit{hn}, \mathit{fid})$.
- If a user application wants to write to a file *fid*, then this writing always starts at the current position within the file. Starting at this point, the previous contents is overwritten by the words *words*. If necessary, the file is extended. If a file needs to be extended, then, depending on the length of *words*, the position in the file, and the file size, it may be necessary to allocate *cl* additional clusters. Now, if there are not enough free clusters or the new file size would exceed the maximum file size MAXFSIZE $\in \mathbb{N}_{32}^{+}$, then the error LIMIT is reported to the calling application.
- If none of the previous cases applies, then the words are written to the file and the number of free clusters adapted as necessary. Additionally, the success message m, including the new position and the new file size, is returned.

This adds up to the following definition of *fwrite*:

$$\mathit{fwrite}(s, \mathit{hn}, \mathit{fid}, \mathit{words}) =$$

let

$$\begin{aligned}
\mathit{fc} &= s.\mathit{fdb}(\mathit{fid}).\mathit{con};\\
\mathit{pos} &= s.\mathit{fdb}(\mathit{fid}).\mathit{pos};\\
\mathit{pos}' &= \mathit{pos} + \mathit{length}(\mathit{words});\\
\mathit{fc}' &= \mathit{take}(\mathit{pos}, \mathit{fc}) \circ \mathit{words} \circ \mathit{drop}(\mathit{pos}', \mathit{fc});\\
\mathit{cl} &= \mathit{ocl}(\mathit{length}(\mathit{fc}')) - \mathit{ocl}(\mathit{length}(\mathit{fc}));\\
s_1 &= s \left[\!\!\left[\begin{array}{l} \mathit{fdb}(\mathit{fid}).\mathit{con} := \mathit{fc}',\\ \mathit{fdb}(\mathit{fid}).\mathit{pos} := \mathit{pos}',\\ \mathit{free\text{-}clusters} := s.\mathit{free\text{-}clusters} - \mathit{cl} \end{array} \right]\!\!\right];\\
m &= [(\mathit{hn}, \text{SUCC-FWRITE } \mathit{pos}'\ \mathit{length}(\mathit{fc}))]
\end{aligned}$$

$$\textbf{in} \begin{cases} \textit{faccess-error}(s, hn, \textit{fid}) & \textbf{if } \neg \textit{faccess-legal}(s, hn, \textit{fid}, \text{WRITE}), \\ (s\ , [\,], [(hn, \text{LIMIT})]) & \textbf{else if } s.\textit{free-clusters} < cl \vee \text{MAXFSIZE} < fc', \\ (s_1, [\,], m) & \textbf{else.} \end{cases}$$

4.3.2.8 Read from a File

For reading from a file, the library Libsos implements the following call:

```
int sc_file_read( unsigned int fid, unsigned int* len,
                  unsigned int* pos, sos_buffer_t* buf).
```

In the SOS implementation, `sc_file_read` is handled by `sos_file_read`. If there is no error, then, at most, `len` words are read from the file with the id `fid` and returned to the calling application. Furthermore, the new position within the file and the number of words that have been read, are returned.

In the specification, we add FREAD $\textit{fid\_t}_\varepsilon\ \mathbb{N}_{32}$, as an abstract representation of `sc_file_read`, to the input alphabet Ω_{sc}:

$$\Omega_{\text{sc}} \supset \{\text{FREAD } \textit{fid}\ \textit{len} \mid \textit{fid} \in \textit{fid\_t}_\varepsilon \wedge \textit{len} \in \mathbb{N}_{32}\}$$

and SUCC-FREAD $\textit{word\_t}^*\ \mathbb{N}_{32}$, as an abstract representation of a possible result, to the output alphabet Σ_{sc}:

$$\Sigma_{\text{sc}} \supset \{\text{SUCC-FREAD } \textit{words}\ \textit{pos} \mid \textit{words} \in \textit{word\_t}^* \wedge \textit{pos} \in \mathbb{N}_{32}\}.$$

The behavior of `sos_file_read` is described by the function *fread*. This function takes, as call-specific arguments, the id of the file and the number of words that should be read:

$$\textit{fread} \in \mathcal{S} \times \textit{hn\_t} \times \textit{fid\_t}_\varepsilon \times \mathbb{N}_{32} \to \mathcal{S} \times \Omega^* \times (\textit{hn\_t} \times \Sigma_{\text{sc}})^*.$$

For $\textit{fread}(s, hn, \textit{fid}, \textit{len})$, the following cases are considered:

- If $\textit{faccess-legal}(s, hn, \textit{fid}, \text{READ})$ is not satisfied, then the result is computed and returned by $\textit{faccess-error}(s, hn, \textit{fid})$.
- If the previous case does not apply, then a success message, containing the words that have been read and the new position $\textit{pos}'$, is returned. Here, the length $\textit{len}'$ is calculated as the minimum of how much can be read from the file, the length *len*, and the buffer size CMPC $\in \mathbb{N}_{32}^+$.[20]

[20] In the implementation, we have to statically fix the size of IPC messages used for receiving SOS calls and returning results. Thus, only a limited number of words may be read with a single `sc_file_read` call. In the specification, this number is represented by the constant CMPC.

This adds up to the following definition of *fread*:

$$\mathit{fread}(s, hn, \mathit{fid}, \mathit{len}) =$$

$$\begin{array}{lll}
\textbf{let} & f & = s.\mathit{fdb}(\mathit{fid}); \\
& \mathit{len}' & = \mathit{min}(\{\mathit{length}(f.\mathit{con}) - f.\mathit{pos}, \mathit{len}, \text{CMPC}\}); \\
& \mathit{words} & = \mathit{take}(\mathit{len}', \mathit{drop}(f.\mathit{pos}, f.\mathit{con})); \\
& \mathit{pos}' & = f.\mathit{pos} + \mathit{len}'; \\
& s_1 & = s[\![\mathit{fdb}(\mathit{fid}).\mathit{pos} := \mathit{pos}']\!]; \\
& m & = [(hn, \text{SUCC-FREAD } \mathit{words}\ \mathit{pos}')]
\end{array}$$

$$\textbf{in} \quad \begin{cases} \mathit{faccess\text{-}error}(s, hn, \mathit{fid}) & \textbf{if } \neg \mathit{faccess\text{-}legal}(s, hn, \mathit{fid}, \text{READ}), \\ (s_1, [\,], m) & \textbf{else}. \end{cases}$$

4.3.2.9 Change the Position within a File

Before specifying how the position within a file can be changed, we need an auxiliary function.

The function $\mathit{offset}(\mathit{start}, \mathit{current}, \mathit{end}, \mathit{flag}, \mathit{off})$ computes a value in the range $[\mathit{start} \ldots \mathit{current} \ldots \mathit{end}]$. Depending on the flag *flag*, the offset *off* is either added to *start*, *current*, or *end*:

$$\mathit{offset} \in \mathbb{N}_{32} \times \mathbb{N}_{32} \times \mathbb{N}_{32} \times \mathbb{Z}_{32} \times \mathbb{Z}_{32} \to \mathbb{N}_{32}$$
$$\mathit{offset}(\mathit{start}, \mathit{current}, \mathit{end}, \mathit{flag}, \mathit{off}) =$$

$$\begin{cases} \mathit{max}(\{\mathit{min}(\{\mathit{start} + \mathit{off}, \mathit{end}\}), \mathit{start}\}) & \textbf{if } \mathit{flag} < 0, \\ \mathit{max}(\{\mathit{min}(\{\mathit{current} + \mathit{off}, \mathit{end}\}), \mathit{start}\}) & \textbf{else if } \mathit{flag} = 0, \\ \mathit{max}(\{\mathit{min}(\{\mathit{end} + \mathit{off}, \mathit{end}\}), \mathit{start}\}) & \textbf{else}. \end{cases}$$

Note that $\mathit{off} < 0$ is possible. For example, $\mathit{offset}(0, 5, 10, 1, -1) = 9$.

Now, for changing the position within a file, the library Libsos implements the following call:

```
int sc_file_seek( unsigned int fid, int flag, int off,
                  *unsigned int pos).
```

In the SOS implementation, `sc_file_seek` is handled by `sos_file_seek`. If there is no error, then the position within the file `fid` is changed and the new position returned to the calling application. The calculation for the new position `pos` depends on `flag`. If `flag` is smaller than `0`, then `pos=off`. If `flag` is equal to `0`, then `off` is added to the current position within the file. Finally, if `flag` is greater than `0`, then `off` is added to the position at the end of the file. In any case, the new position is 'cropped' to point to a position within the file.

In the specification, we add FSEEK $\mathit{fid\_t}_\varepsilon$ $\mathbb{Z}_{32}$ $\mathbb{Z}_{32}$, as an abstract representation of `sc_file_seek`, to the input alphabet Ω_{sc}:

$$\Omega_{\text{sc}} \supset \{\text{FSEEK } \mathit{fid}\ \mathit{flag}\ \mathit{off} \mid \mathit{fid} \in \mathit{fid\_t}_\varepsilon \wedge \mathit{flag}, \mathit{off} \in \mathbb{Z}_{32}\}$$

and SUCC-FSEEK $\mathbb{N}_{32}$, as an abstract representation of a possible result, to the output alphabet Σ_{sc}:

$$\Sigma_{\text{sc}} \supset \{\text{SUCC-FSEEK } \mathit{pos} \mid \mathit{pos} \in \mathbb{N}_{32}\}.$$

The behavior of `sos_file_seek` is described by the function *fseek*. This function takes, as call-specific arguments, the id of the file, the flag indicating the mode of operation, and the offset:

$$\mathit{fseek} \in \mathcal{S} \times \mathit{hn\_t} \times \mathit{fid\_t}_\varepsilon \times \mathbb{Z}_{32} \times \mathbb{Z}_{32} \to \mathcal{S} \times \Omega^* \times (\mathit{hn\_t} \times \Sigma_{\text{sc}})^*.$$

For $\mathit{fseek}(s, \mathit{hn}, \mathit{fid}, \mathit{flag}, \mathit{off})$, the following cases are considered:

- If $\mathit{faccess\text{-}legal}(s, \mathit{hn}, \mathit{fid}, \text{READ})$ is not satisfied, then the result is computed and returned by $\mathit{faccess\text{-}error}(s, \mathit{hn}, \mathit{fid})$.
- If the previous case does not apply, then the new position pos' is calculated using the function *offset*, the file position is updated. Furthermore, a success message, containing the new position within the file, is returned to the calling application.

This adds up to the following definition of *fseek*:

$$\mathit{fseek}(s, \mathit{hn}, \mathit{fid}, \mathit{flag}, \mathit{off}) =$$

let

$$\begin{aligned} f &= s.\mathit{fdb}(\mathit{fid}); \\ \mathit{pos}' &= \mathit{offset}(0, f.\mathit{pos}, \mathit{length}(f.\mathit{con}), \mathit{flag}, \mathit{off}); \\ s_1 &= s[\![\mathit{fdb}(\mathit{fid}).\mathit{pos} := \mathit{pos}']\!] \end{aligned}$$

in

$$\begin{cases} \mathit{faccess\text{-}error}(s, \mathit{hn}, \mathit{fid}) & \textbf{if } \neg \mathit{faccess\text{-}legal}(s, \mathit{hn}, \mathit{fid}, \text{READ}), \\ (s_1, [\,], [(\mathit{hn}, \text{SUCC-FSEEK } \mathit{pos}')]) & \textbf{else}. \end{cases}$$

4.3.2.10 Change the Permissions associated with a File

For changing the permissions associated with a file, the library Libsos implements the following call:

```
int sc_file_chmod( unsigned int fid, unsigned int fop,
                   unsigned int uid, int flag).
```

In the SOS implementation, `sc_file_chmod` is handled by `sos_file_chmod`. If there is no error, then the permissions associated with the file `fid` may be changed. If `flag` is greater than or equal to `0`, then the user `uid` receives the permission to perform the file operation `fop`. If `flag` is less than `0`, then the permission is revoked. Note that the SOS call also returns successfully if the permissions do not need to be updated, i. e. if the user already has the permission and the calling application tries to add it, or if the user does not have the permission and the calling application tries to remove it.

In the specification, we add FCHMOD fid_t_ε fop_t_ε uid_t_ε $\mathbb{Z}_{32}$, as an abstract representation of `sc_file_chmod`, to the input alphabet Ω_{sc}:

$$\Omega_{\text{sc}} \supset \left\{ \begin{array}{l} \text{FCHMOD } fid\ fop\ uid\ flag \mid \\ fid \in fid\_t_\varepsilon \wedge fop \in fop\_t_\varepsilon \wedge uid \in uid\_t_\varepsilon \wedge flag \in \mathbb{Z}_{32} \end{array} \right\}.$$

The behavior of `sos_file_chmod` is described by the function *fchmod*. This function takes, as call-specific arguments, the id of the file, the file operation, the user id, and the flag indicating the mode of operation:

$$\begin{aligned} fchmod \in \mathcal{S} \times hn\_t \times fid\_t_\varepsilon \times fop\_t_\varepsilon \times uid\_t_\varepsilon \times \mathbb{Z}_{32} \\ \to \mathcal{S} \times \Omega^* \times (hn\_t \times \Sigma_{\text{sc}})^*. \end{aligned}$$

For $fchmod(s, hn, fid, fop, uid, flag)$, the following cases are considered:

- If *fid* is not a valid file id or such a file does not exist or if *fop* is not a valid file operation or *uid* does not exist, then the message (hn, ARG) is returned.
- If the calling application *hn* does not have the lock for the file, then the message (hn, LOCK) is returned.
- If the owner of the calling application is neither the owner of the file nor entitled to change its permissions, then the message (hn, PERM) is returned.
- If none of the previous cases applies, then *uid* is either added to or removed from the set of users entitled to perform the file operation *fop* on the file f. Whether *uid* is added or removed depends on the flag *flag*. If $flag \geq 0$, then *uid* is added, otherwise *uid* is removed. Furthermore, a success message is returned.
- If the previous case applies and it is some file-lock permission that has been revoked, i. e. $fop = \text{LCK} \wedge flag < 0$, then additionally the list of lock requests is inspected to cancel all requests that are no longer valid, i. e. the corresponding applications receive the result PERM. For that, we *filter* the list of $f.lock$ to collect all applications that are owned by *uid* and then create a list of error messages using the *map* operator.

Furthermore, all those applications that receive an error message are removed from the list of lock requests.

This adds up to the following definition of *fchmod*:

$$\mathit{fchmod}(s, hn, \mathit{fid}, \mathit{fop}, \mathit{uid}, \mathit{flag}) =$$

$$\begin{array}{lll}
\textbf{let} & ao & = s.adb(hn).owner; \\
& f & = s.fdb(\mathit{fid}); \\
& \mathit{perm}' & = \begin{cases} f.perm(\mathit{fop}) \cup \{\mathit{uid}\} & \textbf{if } \mathit{flag} \geq 0, \\ f.perm(\mathit{fop}) \setminus \{\mathit{uid}\} & \textbf{else}; \end{cases} \\
& s_1 & = s \ [\![\mathit{fdb}(\mathit{fid}).perm(\mathit{fop}) := \mathit{perm}']\!]; \\
& \mathit{hns} & = \mathit{filter}((\lambda x.\ s.adb(x).owner = \mathit{uid}), \mathit{tail}(f.lock)); \\
& m & = \mathit{map}((\lambda x.\ (x, \text{PERM})), \mathit{hns}); \\
& \mathit{lock}' & = hn \# \mathit{filter}((\lambda x.\ s.adb(x).owner \neq \mathit{uid}), \mathit{tail}(f.lock)); \\
& s_2 & = s_1 \, [\![\mathit{fdb}(\mathit{fid}).lock := \mathit{lock}']\!]
\end{array}$$

$$\textbf{in} \quad \begin{cases} (s\ , [\,], [(hn, \text{ARG})]) & \textbf{if } \mathit{fid} = \varepsilon \vee \mathit{fop} = \varepsilon \vee f = \varepsilon \vee \mathit{uid} \notin s.udb, \\ (s\ , [\,], [(hn, \text{LOCK})]) & \textbf{else if } hn \neq f.lock[0], \\ (s\ , [\,], [(hn, \text{PERM})]) & \textbf{else if } ao \neq f.owner \wedge ao \notin f.perm(\text{CHMOD}), \\ (s_1, [\,], [(hn, \text{SUCC})]) & \textbf{else if } \mathit{fop} \neq \text{LCK} \vee \mathit{flag} \geq 0, \\ (s_2, [\,], (hn, \text{SUCC}) \# m) & \textbf{else.} \end{cases}$$

Again, because of the invariant *inv-unique-lock-requests*, we know that all entries in the list $\mathit{tail}(f.lock)$ are unique and different from hn. Therefore, we can be sure that we will not accidently send several messages to the same application.

4.3.2.11 Change the Owner of a File

The library Libsos implements the following call that allows the super user to change the owner of a file:

```
int sc_file_chown( unsigned int fid, unsigned int uid).
```

In the SOS implementation, `sc_file_chown` is handled by `sos_file_chown`. If there is no error, then, after the call is handled, the file `fid` is owned by `uid`.

In the specification, we add FCHOWN $\mathit{fid\_t}_\varepsilon$ $\mathit{uid\_t}_\varepsilon$, as an abstract representation of `sc_file_chown`, to the input alphabet Ω_{sc}:

$$\Omega_{\text{sc}} \supset \{\text{FCHOWN } \mathit{fid}\ \mathit{uid} \mid \mathit{fid} \in \mathit{fid\_t}_\varepsilon \wedge \mathit{uid} \in \mathit{uid\_t}_\varepsilon\}.$$

The behavior of `sos_file_chown` is described by the function *fchown*. This function takes, as call-specific arguments, the id of the file and the user id of the new owner:

$$\mathit{fchown} \in \mathcal{S} \times \mathit{hn\_t} \times \mathit{fid\_t}_\varepsilon \times \mathit{uid\_t}_\varepsilon \to \mathcal{S} \times \Omega^* \times (\mathit{hn\_t} \times \Sigma_{\mathrm{sc}})^*.$$

For $\mathit{fchown}(s, \mathit{hn}, \mathit{fid}, \mathit{uid})$, the following cases are considered:

- If the file *fid* does not exist, or the user *uid* does not exist, then the message $(\mathit{hn}, \text{ARG})$ is returned.
- If the calling application *hn* does not have the lock for the file, then the message $(\mathit{hn}, \text{LOCK})$ is returned.
- If the calling application is not owned by the super user, then the message $(\mathit{hn}, \text{PERM})$ is returned.
- If none of the previous cases applies, then the file data base is updated such that *uid* is the new owner of the file *fid*, and a success is message returned.

This adds up to the following definition of *fchown*:

$$\mathit{fchown}(s, \mathit{hn}, \mathit{fid}, \mathit{uid}) =$$

$$\textbf{let} \qquad s_1 = s[\![\mathit{fdb}(\mathit{fid}).\mathit{owner} := \mathit{uid}]\!]$$

$$\textbf{in} \quad \begin{cases} (s\ , [\,], [(\mathit{hn}, \text{ARG})]) & \textbf{if } \mathit{fid} = \varepsilon \vee s.\mathit{fdb}(\mathit{fid}) = \varepsilon \vee \mathit{uid} \notin s.\mathit{udb}, \\ (s\ , [\,], [(\mathit{hn}, \text{LOCK})]) & \textbf{else if } \mathit{hn} \neq f.\mathit{lock}[0], \\ (s\ , [\,], [(\mathit{hn}, \text{PERM})]) & \textbf{else if } s.\mathit{adb}(\mathit{hn}).\mathit{owner} \neq \text{SU}, \\ (s_1, [\,], [(\mathit{hn}, \text{SUCC})]) & \textbf{else}. \end{cases}$$

4.3.2.12 How Does It Compare

In § 1.2.1, we mentioned a number of projects that focused on file systems. Since Yang et al. [YTEM06] essentially did rule-based testing and Joshi and Holzmann [JH07] have not yet published any results, only the works of Bevier et al. [BCT95, BC96] and Arkoudas et al. [AZKR04] can be compared with our file system specification.

The specifications provided by Bevier et al. are to a certain degree similar to our specification. Disregarding the fact that they used the specification languages Z and ACL2, they represented files in a similar fashion and formalized similar set of file operations. On the one hand, the file system they specified has more features than ours (e. g. multi level directories and the concept of

file descriptors), but, on the other hand, their specification is intended as a programmer's manual and thus lacks the completeness of our specification. For example, they did not consider any resource limits, excluded error reporting, and ignored concurrent file access.

In [AZKR04], Arkoudas et al. established a simulation relation between two differently abstract models of a file system. Although they managed to prove simulation, their specification is far from complete. In fact, they only considered a simple read operation and a simple write operation. Even for these two operations, they did not consider anything like processes, users, or permissions.

4.3.3 Virtual Terminals

In the following subsection, we will specify SOS calls that allow user applications to get keyboard input, write to the screen, change the position of the cursor, and retrieve information about the terminal.

Other than the hard disk, a keyboard depends on the outside world. If a user presses a key on the keyboard, then this is considered to be an external input to the SOS*. Hence, in the following we will also describe how such input is treated, i. e. specify the keyboard-interrupt handler.

4.3.3.1 Get Keyboard Input

Before we describe what happens if a user application wants to read keyboard input, we need some auxiliary functions.

The function *terminal-owner*(s, tid) returns the handle of the application that is connected to the virtual terminal *tid*. If the terminal is not connected to any application, then ε is returned:[21]

$$\textit{terminal-owner} \in \mathcal{S} \times \textit{tid\_t} \to \textit{hn\_t}_\varepsilon$$

$$\textit{terminal-owner}(s, tid) =$$

$$\begin{cases} \epsilon\{x \mid s.adb(x).term = tid\} & \textbf{if } \exists x.s.adb(x).term = tid, \\ \varepsilon & \textbf{else.} \end{cases}$$

Note that in *terminal-owner* we use the Hilbert Choice operator ϵ to choose one handle *hn* from the set of handles of applications that are connected to the terminal *tid*. This is only deterministic if there is (at most) one such *hn*. That means, any virtual terminal should only be connected to (at most) one

[21] A virtual terminal may not be connected to an application, if the initial user application that was started for this terminal, i. e. the login shell, terminated.

application. Thus, we need the following invariant:

$$\begin{gathered}
\mathit{inv\text{-}unambiguous\text{-}terminal\text{-}owner} \in \mathcal{S} \to \mathbb{B} \\
\mathit{inv\text{-}unambiguous\text{-}terminal\text{-}owner}(s) \equiv \\
\forall hn_1, hn_2.\ s.adb(hn_1).term = s.adb(hn_2).term \\
\wedge\ s.adb(hn_1).term \neq \varepsilon \\
\Longrightarrow \\
hn_1 = hn_2.
\end{gathered}$$

In § 4.3.6 we will see that the owner of a terminal only changes if a new application is created (*aexec* and *afork* defined in § 4.3.6.1 and § 4.3.6.2, respectively) or if an existing application is terminated (*aexit* defined in § 4.3.6.4). In all three cases, a terminal connection may be passed from one application to another but never newly assigned. Thus, we can be sure that *inv-unambiguous-terminal-owner* is preserved throughout the whole SOS* model.

In the implementation, each virtual terminal displays a status line (see § 4.2.3). The contents of this line can always be computed using the function *terminal-status*. If the terminal is connected to some application a, then this line displays whether a has a pending input request, the terminal id, the user that owns the connected application, and the file id of the executable of the application. If the terminal is not connected to an application, then only the terminal id is displayed:

$$\begin{gathered}
\mathit{terminal\text{-}status} \in \mathcal{S} \times tid_t \to (\mathbb{B}_\varepsilon, tid_t, uid_t_\varepsilon, fid_t_\varepsilon) \\
\mathit{terminal\text{-}status}(s, tid) = \\
\textbf{let} \quad hn = \mathit{terminal\text{-}owner}(s, tid); \\
\qquad\ a \ = s.adb(hn) \\
\textbf{in} \quad \begin{cases} (a.read, tid, a.owner, a.exec) & \textbf{if } hn \neq \varepsilon, \\ (\varepsilon, tid, \varepsilon, \varepsilon) & \textbf{else.} \end{cases}
\end{gathered}$$

Now, the library Libsos implements the following call that allows a user application to read keyboard input from a connected terminal:

```
int sc_term_read(char* c).
```

In the SOS implementation, `sc_term_read` is handled by `sos_term_read`. If there is no error, then the oldest character from the input queue of the connected terminal is returned to the calling application.

In the specification, we add TREAD, as an abstract representation of the SOS call `sc_term_read`, to the input alphabet Ω_{sc}:

$$\Omega_{\text{sc}} \ni \text{TREAD}$$

and SUCC-TREAD $byte\_t$, as an abstract representation of a possible result, to the output alphabet Σ_{sc}:

$$\Sigma_{sc} \supset \{\text{SUCC-TREAD } b \mid b \in byte\_t\}.$$

The behavior of `sos_term_read` is described by the function *tread*:

$$tread \in \mathcal{S} \times hn\_t \to \mathcal{S} \times \Omega^* \times (hn\_t \times \Sigma_{sc})^*.$$

For $tread(s, hn)$, the following cases are considered:

- If the calling application is not connected to a terminal, i. e. $tid = s.adb(hn).term$ and $tid = \varepsilon$, then the message (hn, PERM) is returned.
- If the input queue $t.in$, with $t = s.tdb(tid)$, is empty, then the *read* flag in the caller's application data structure is set to TRUE to signal the pending input request. In this case, there is no immediate result. Instead, a result may be returned by the event handler that treats keyboard input (*event-tkbd* defined in § 4.3.3.2).
- If none of the previous cases applies, then the oldest character, i. e. $t.in[0]$, is removed from $t.in$ and a success message, containing this character, returned.

This adds up to the following definition of *tread*:

$$tread(s, hn) =$$

let

$$\begin{aligned} tid &= s.adb(hn).term; \\ t &= s.tdb(tid); \\ s_1 &= s[\![adb(hn).read := \text{TRUE}]\!]; \\ s_2 &= s[\![tdb(tid).in := tail(t.in)]\!] \end{aligned}$$

in

$$\begin{cases} (s\ , [\,], [hn, \text{PERM}]) & \textbf{if } tid = \varepsilon, \\ (s_1, [\,], [\,]) & \textbf{else if } t.in = [\,], \\ (s_2, [\,], [hn, \text{SUCC-TREAD } t.in[0]]) & \textbf{else}. \end{cases}$$

4.3.3.2 Keyboard Input — KBD

If the user presses a key on the connected keyboard, then an interrupt is raised and delivered to the SOS. In the SOS implementation, such an interrupt is handled by the keyboard-interrupt handler `sos_term_int`. This interrupt handler first of all reads the input from the device and then processes it.

In SOS⋆, we add KBD *na_t byte_t*, as an abstract representation of a keyboard interrupt, to the input alphabet Σ:

$$\Sigma \supset \{\text{KBD } dna \; byte \mid dna \in na\_t \wedge byte \in byte\_t\}.$$

Here, KBD identifies the input as keyboard input, the network address identifies the system the input is intended for, and the byte is the actual input.[22]

Now, we want to present an abstraction of `sos_term_int`. First of all, note that receiving keyboard input is problematic as there are two scenarios where keyboard input might get lost:

- If the latency of the interrupt handler is high, the UART's internal receive queue may overflow. Even if there was hardware flow control, a user that is fast enough and that does not follow some protocol, could fill up any hardware buffer. In this case, input from key strokes is lost independently from the currently focused virtual terminal.
- If an applications does not consume the input, then the software buffer might overflow. In this case, input from key strokes is only lost, if the input buffer of the currently focused virtual terminal is full.

We will model these two bottlenecks at different places in SOS⋆. The hardware buffer overflow will be modeled at the dispatcher level (see § 4.3.8.2). There, the transition relation Δ nondeterministically drops keyboard input. The software buffer overflow will be modeled along with the specification of the keyboard-interrupt handler.

Thus, considering the software buffer overflow, the interrupt handler `sos_term_int` is described by the function *event-tkbd*. This function takes, as event-specific arguments, the (destination) network address and the actual input:

$$\textit{event-tkbd} \in \mathcal{S} \times na\_t \times byte\_t \to \mathcal{S} \times \Omega^* \times (hn\_t \times \Sigma_{\text{sc}})^*.$$

For $\textit{event-tkbd}(s, dna, byte)$, the following cases are considered:

- If the keyboard input is not intended for the local system, i. e. $dna \neq s.lna$; if the key is neither the STK key nor does it translate to a valid input character; if the input queue of the currently focused terminal $t = s.tdb(s.focus)$ is full, i. e. $\textit{length}(t.in) = \text{TINMAX}$; or if t is not connected to any application; then the key is simply ignored.
- If the key is the STK key, then the focus is moved to the next terminal.
- If the key translates to a valid input character, and there is no pending input request from the connected application, then the character is appended to the input queue of the currently focused terminal

[22] The network address is necessary as soon as there are multiple instances of SOS⋆. In this case, we need a way to match keyboard input and target system (see Chapter 5).

- If the key translates to a valid input character, and there is a pending input request, then the request is removed and a success message, including the new character, returned to the waiting application. Note, there can only be a pending input request, if the input queue is empty (*tread* defined in § 4.3.3.1).

This adds up to the following definition of *event-tkbd*:

$$\textit{event-tkbd}(s, \textit{dna}, \textit{byte}) =$$

$$\begin{array}{ll}
\textbf{let} & \textit{tid} = s.\textit{focus}; \\
& t \;\; = s.\textit{tdb}(\textit{tid}); \\
& \textit{hn} = \textit{terminal-owner}(s, \textit{tid}); \\
& s_1 \;\; = s[\![\textit{focus} := (s.\textit{focus} + 1)\%\text{NT}]\!]; \\
& s_2 \;\; = s[\![\textit{tdb}(\textit{tid}).\textit{in} := t.\textit{in} \circ [\textit{byte}]]\!]; \\
& s_3 \;\; = s[\![\textit{adb}(\textit{hn}).\textit{read} := \text{FALSE}]\!]
\end{array}$$

in

$$\begin{cases}
(s\ , [\,], [\,]) & \textbf{if}\ \ \textit{dna} \neq s.\textit{lna} \\
& \quad \vee (\textit{byte} \neq \text{STK} \wedge \textit{byte} \notin \text{SCRC-IN}) \\
& \quad \vee \textit{length}(t.\textit{in}) = \text{TINMAX} \\
& \quad \vee \textit{hn} = \varepsilon, \\
(s_1, [\,], [\,]) & \textbf{else if}\ \ \textit{byte} = \text{STK}, \\
(s_2, [\,], [\,]) & \textbf{else if}\ \ \neg \textit{adb}(\textit{hn}).\textit{read}, \\
(s_3, [\,], [(\textit{hn}, \text{SUCC-TREAD}\ \textit{byte})]) & \textbf{else}.
\end{cases}$$

4.3.3.3 Write to the Screen

The library Libsos implements the following call that allows a user application to write to the screen of a connected terminal:

```
int sc_term_write(char c, int flag).
```

In the SOS implementation, `sc_term_write` is handled by `sos_term_write`. If there is no error, then the character `c` is printed on the screen and the cursor position incremented. The exact placement of `c` and the contents of the remaining screen depend on the flag `flag`. If `flag` is less or equal to `0`, then `c` is printed at the current cursor position. If `flag` is equal to `1`, then the row of the current cursor position is cleared and `c` printed at the beginning of that row. Finally, if `flag` is greater or equal to `2`, then the whole screen is cleared and `c` printed in the upper left corner.

In the specification, we add TWRITE *byte_t* $\mathbb{Z}_{32}$, as an abstract representation of `sc_term_write`, to the input alphabet Ω_{sc}:

$$\Omega_{\text{sc}} \supset \{\text{TWRITE } \mathit{byte}\ \mathit{flag} \mid \mathit{byte} \in \mathit{byte\_t} \wedge \mathit{flag} \in \mathbb{Z}_{32}\}.$$

The behavior of `sos_term_write` is described by the function *twrite*. This function takes, as call-specific arguments, the character that should be printed and the flag indicating the mode of operation:

$$\mathit{twrite} \in \mathcal{S} \times \mathit{hn\_t} \times \mathit{byte\_t} \times \mathbb{Z}_{32} \to \mathcal{S} \times \Omega^* \times (\mathit{hn\_t} \times \Sigma_{\text{sc}})^*.$$

For $\mathit{twrite}(s, \mathit{hn}, \mathit{byte}, \mathit{flag})$, the following cases are considered:

- If the calling application is not connected to a virtual terminal, then the message $(\mathit{hn}, \text{PERM})$ is returned.
- If *byte* is not a printable character, i. e. $\mathit{byte} \notin$ SCRC-OUT, then the message $(\mathit{hn}, \text{ARG})$ is returned.
- If none of the previous cases applies, then the screen contents is updated and a success message returned. For that, we first of all compute the position pos' where the character has to be placed. Then, the new contents of the screen is computed by concatenating *beg*, the prefix of the old contents; the character *byte*; possibly some white spaces; and *end*, the postfix the old contents (see Figure 4.2 on the facing page). Finally, the cursor is moved behind the character that has been printed. If *byte* is printed at the lower right corner, then the cursor is moved to the upper left corner.

This adds up to the following definition of *twrite*:

$$\mathit{twrite}(s, \mathit{hn}, \mathit{byte}, \mathit{flag}) =$$

$$\begin{aligned}
\textbf{let}\quad & \mathit{tid} = s.\mathit{adb}(\mathit{hn}).\mathit{term};\\
& t = s.\mathit{tdb}(\mathit{tid});\\
& \mathit{tc} = s.\mathit{tdb}(\mathit{tid}).\mathit{out};\\
& \mathit{pos}' = \begin{cases} t.\mathit{pos} & \textbf{if } \mathit{flag} <= 0,\\ t.\mathit{pos} - (t.\mathit{pos}\%\text{SCRX}) & \textbf{else if } \mathit{flag} = 1,\\ 0 & \textbf{else}; \end{cases}\\
& \mathit{beg} = \mathit{take}(\mathit{pos}', \mathit{tc});\\
& \mathit{end} = \begin{cases} \mathit{drop}(\mathit{pos}' + 1, \mathit{tc}) & \textbf{if } \mathit{flag} <= 0,\\ \text{SPACE}^{\text{SCRX}-1} \circ \mathit{drop}(\mathit{pos}' + \text{SCRX}, \mathit{tc}) & \textbf{else if } \mathit{flag} = 1,\\ \text{SPACE}^{\text{SCRXY}-1} & \textbf{else}; \end{cases}\\
& s_1 = s \left[\!\!\left[\begin{array}{l} \mathit{tdb}(\mathit{tid}).\mathit{out} := \mathit{beg} \circ [\mathit{byte}] \circ \mathit{end},\\ \mathit{tdb}(\mathit{tid}).\mathit{pos} := (\mathit{pos}' + 1)\%\text{SCRXY} \end{array} \right]\!\!\right]
\end{aligned}$$

in
$$\begin{cases} (s\ , [\,], [(hn, \text{PERM})]) & \textbf{if } tid = \varepsilon, \\ (s\ , [\,], [(hn, \text{ARG})]) & \textbf{else if } byte \notin \text{SCRC-OUT}, \\ (s_1, [\,], [(hn, \text{SUCC})]) & \textbf{else.} \end{cases}$$

Here, we use SPACE $\in$ SCRC-OUT to denote the white-space character. Further, we use SPACE^x to denote a list $l \in byte\_t^x$ such that $l = [\text{SPACE}, \ldots, \text{SPACE}]$.

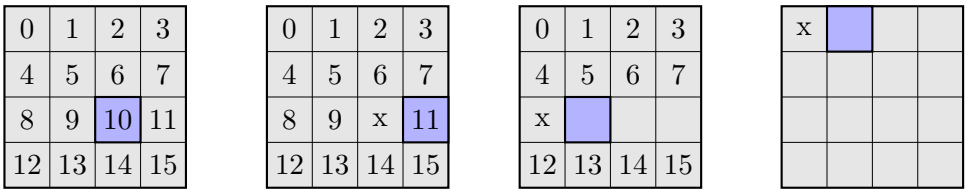

Figure 4.2: Writing to a Terminal (*twrite*). If (a) depicts some terminal's screen contents $s.tdb(t).out$, then (b), (c), and (d) depict $s_1.tdb(t).out$, if the application hn was connected to t and successfully called $twrite(s, hn, x, 0)$, $twrite(s, hn, x, 1)$, or $twrite(s, hn, x, 2)$, respectively.

4.3.3.4 Change the Position on the Screen

For moving the cursor, the library Libsos implements the following call:

```
int sc_term_seek(int flag, int off, unsigned int* pos).
```

In the SOS implementation, `sc_term_seek` is handled by `sos_term_seek`. If there is no error, then the position of the cursor on a connected terminal is updated and the new position returned to the calling application. The calculation for the new position `pos` depends on `flag`. If `flag` is smaller than `0`, then `pos=off`. If `flag` is equal to `0`, then `off` is added to the current cursor position. If `flag` is greater than `0`, then `off` is added to the position at the end of the screen. Note, as for `sc_term_seek`, `off` may be negative, in which case the cursor is moved backward. In any case, the new position is 'cropped' to point to a position on the screen.

In the specification, we add TSEEK $\mathbb{Z}_{32}$ $\mathbb{Z}_{32}$, as an abstract representation of `sc_term_seek`, to the input alphabet Ω_{sc}:

$$\Omega_{\text{sc}} \supset \{\text{TSEEK } flag\ off \mid flag, off \in \mathbb{Z}_{32}\}$$

and SUCC-TSEEK $\mathbb{N}_{32}$, as an abstract representation of a possible result, to the output alphabet Σ_{sc}:

$$\Sigma_{\text{sc}} \supset \{\text{SUCC-TSEEK } pos \mid pos \in \mathbb{N}_{32}\}.$$

The behavior of `sos_term_seek` is described by the function *tseek*. This function takes, as call-specific arguments, the flag indicating the mode of operation and the offset:

$$\mathit{tseek} \in \mathcal{S} \times \mathit{hn\_t} \times \mathbb{Z}_{32} \times \mathbb{Z}_{32} \to \mathcal{S} \times \Omega^* \times (\mathit{hn\_t} \times \Sigma_{\mathrm{sc}})^*.$$

For $\mathit{tseek}(s, hn, \mathit{flag}, \mathit{off})$, the following cases are considered:

- If the calling application is not connected to a virtual terminal, then the message (hn, PERM) is returned.
- If the previous case does not apply, then the new position pos' is calculated using the function *offset*, the cursor position updated accordingly, and a success message, containing the new position, returned.

This adds up to the following definition of *tseek*:

$$\mathit{tseek}(s, hn, \mathit{flag}, \mathit{off}) =$$

$$\begin{array}{ll}
\textbf{let} & tid \;\; = s.adb(hn).term; \\
& pos \;\; = s.tdb(tid).pos; \\
& pos' = \mathit{offset}(0, pos, \text{SCRXY} - 1, \mathit{flag}, \mathit{off}); \\
& s_1 \;\; = s[\![tdb(tid).pos := pos']\!] \\
\textbf{in} & \begin{cases} (s\ , [\,], [(hn, \text{PERM})]) & \textbf{if } tid = \varepsilon, \\ (s_1, [\,], [(hn, \text{SUCC-TSEEK } pos')]) & \textbf{else.} \end{cases}
\end{array}$$

4.3.3.5 Retrieve Information about a Terminal

The library Libsos implements the following call that allows a user application to retrieve information about a connected terminal:

```
int sc_term_info( unsigned int* width, unsigned int* height,
                  unsigned int* pos).
```

In the SOS implementation, `sc_term_info` is handled by `sos_term_info`. If there is no error, then `sc_term_info` returns the width and height of the (user accessible) screen area and the current cursor position.

In the specification, we add TINFO, as an abstract representation of the SOS call, to the input alphabet Ω_{sc}:

$$\Omega_{\mathrm{sc}} \ni \text{TINFO}$$

and SUCC-TINFO $\mathbb{N}_{32}$ $\mathbb{N}_{32}$ $\mathbb{N}_{32}$, as an abstract representation of a possible result, to the output alphabet Σ_{sc}:

$$\Sigma_{\mathrm{sc}} \supset \{\text{SUCC-TINFO } \mathit{width}\ \mathit{height}\ \mathit{pos} \mid \mathit{width}, \mathit{height}, \mathit{pos} \in \mathbb{N}_{32}\}.$$

The behavior of `sos_term_info` is described by the function *tinfo*:

$$\mathit{tinfo} \in \mathcal{S} \times \mathit{hn\_t} \to \mathcal{S} \times \Omega^* \times (\mathit{hn\_t} \times \Sigma_{\mathrm{sc}})^*.$$

For $\mathit{tinfo}(s, hn)$, the following cases are considered:

- If the calling application is not connected to a virtual terminal, then the message (hn, PERM) is returned.
- If the previous case does not apply, then a success message, including SCRX, SCRY, and the position of the cursor on the terminal, is returned.

This adds up to the following definition of *tinfo*:

$$\mathit{tinfo}(s, hn) =$$

let
$$\begin{aligned} \mathit{tid} &= s.\mathit{adb}(hn).\mathit{term}; \\ \mathit{pos} &= s.\mathit{tdb}(\mathit{tid}).\mathit{pos} \end{aligned}$$

in
$$\begin{cases} (s, [\,], [(hn, \text{PERM})]) & \textbf{if } \mathit{tid} = \varepsilon, \\ (s, [\,], [(hn, \text{SUCC-TINFO SCRX SCRY } \mathit{pos})]) & \textbf{else.} \end{cases}$$

4.3.4 Sockets

We want to allow user applications to communicate with the outside world. For that, the SOS implementation provides a socket interface. In the following subsection, we will first of all give some background information about the TCP layer, its assumptions, and its guarantees. Based on that, we will introduce abstract network packets, state fundamental invariants about sockets, and finally specify SOS handlers and event handlers related to sockets.

Note that the necessary SOS* transitions are fairly complex. Modeling a network card and its device drivers requires us to deal with external input as well as external output and it is also necessary to consider network properties. This makes the abstraction less intuitive and the simulation relation more complex. For a better understanding, we will give more explanation than for transitions related to file I/O or virtual terminals. In several places we will point out how the abstraction relates to the implementation and why certain abstractions may be valid.

4.3.4.1 TCP Assumptions and Guarantees

In general, TCP provides reliable, in-order delivery of a stream of bytes [Ste93]. Our implementation, (additionally) guarantees Safety and Liveness for the

opening phase, the transmission phase, and the closing phase [Cai06].[23] In order to guarantee these properties a number of assumptions are made. Among them are:

- Network Liveness: every IP packet that is sent infinitely often will eventually be received.
- Network Safety: the source of an IP packet can not be faked.
- Time to Live: IP packets expire in a way that guarantees unique sequence numbers.
- Internal Liveness: the TCP layer is called infinitely often.
- Absence of Timeouts: there are no external timeouts, i. e. the socket layer does not set up timeouts for data submitted to the TCP layer (e. g. there are no timeouts for for requesting and accepting a connection or a connection does not terminate after a certain time without traffic).

These assumptions impose requirements on the network as well as on the implementation of the network protocol stack of all the communication partners. At this point, we are actually arguing about several systems (see Chapter 5 and Chapter 6). Although the above assumptions very much restrict our implementation, we were still able to implement a relevant subset of the socket calls specified in the POSIX standard [IEE04]. This implementation is, on the one hand, powerful enough to communicate with any of the standard implementations (e. g. the implementations within Linux and Windows XP), and, on the other hand, restrictive enough to meet the above assumptions. That means that our operating system can be tested / used in the 'real world', but, at the same time, we are able to take advantage of the TCP guarantees within SOS* and DSOS*. The latter allows us to represent communication via sockets on a very abstract level.

4.3.4.2 Abstract Network Packets

As in TCP, the communication via sockets can be divided into three phases, i. e. the opening phase, the communication phase, and the closing phase:

- In the opening phase, a socket is created and bound to a local address and local port number. Furthermore, in this phase, a connection is established. How a connection is established depends on the character of the participating applications. A server signals its willingness to accept connections on a certain local port by listening on the corresponding socket. Then, a connection is established, if there is a client that requests

[23] A short overview of the "Reactive Properties of the TCP Subsystem of the Simple Operating System" is presented in the Verisoft-internal Technical Report #69.

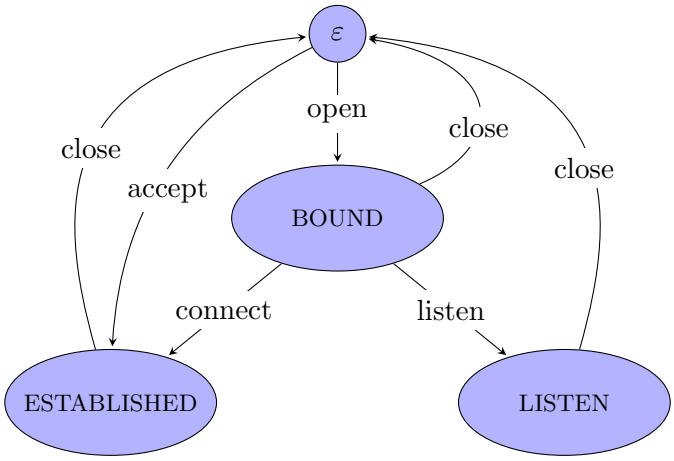

Figure 4.3: Live Cycle of a Socket. Sockets end up waiting for connection requests or may become part of a connection. (Figure 4.8 on page 100, at the end of this subsection, presents a more precise picture.)

a connection and the server accepts that request. On the client side, the original socket is used as an endpoint of the newly established connection. On the server side, the original socket remains in listen mode and a new socket is used as an endpoint of the newly established connection. (see Figure 4.3)

- In the communication phase, sockets are used to send and receive data.
- In the closing phase, a connection is terminated, if at least one of the endpoints explicitly closes it. In this case, the closing side releases it's socket and the opposite side is informed. Then, the latter changes it's socket state to reflect this (partially) closed connection.

In all three phases, the state of a socket depends on the socket calls executed on the local machine as well as on external events. In the implementation, the socket layer is at the top of the TCP / IP stack. At the bottom of that stack is a network card that receives data from the outside world and that sends data to the outside world. Receiving data via the network card is very similar to receiving data via the serial interface; a remote system sending data is similar to a user pressing a key on a connected keyboard. Hence, in SOS*, we model data coming in via the network card as external inputs and data going out as external outputs. Depending on the level of abstraction these inputs and outputs may be TCP packets, IP packets or one of the units of even lower layers. In SOS*, we hide as much as possible of the underlying protocols. Here, we no longer talk about TCP packets but introduce abstract network

packets. A single abstract network packet is either a packet used to establish a connection (REQ and READY), a packet used to exchange or acknowledge data (DATA and ACK), or a packet used to close a connection (CLOSE). In SOS⋆, the type *np_t* is used to represent abstract network packets. It is an abstract data type that contains (at least) the packet identifier (constructor) REQ, READY, DATA, ACK, or CLOSE; the sender's port and network address; and the receiver's port and network address. A data packet additionally contains a list of bytes, the actual payload, and a data acknowledge packet additionally contains the number of bytes that should be acknowledged:

$$\begin{aligned} np\_t = \quad & \text{REQ } na\_t\ pn\_t\ na\_t\ pn\_t \\ \mid\ & \text{READY } na\_t\ pn\_t\ na\_t\ pn\_t \\ \mid\ & \text{DATA } na\_t\ pn\_t\ na\_t\ pn\_t\ byte\_t^* \\ \mid\ & \text{ACK } na\_t\ pn\_t\ na\_t\ pn\_t\ \mathbb{N}_{32} \\ \mid\ & \text{CLOSE } na\_t\ pn\_t\ na\_t\ pn\_t. \end{aligned}$$

Note that the correspondence between abstract network packets and TCP packets is non-trivial. For example, in the implementation, a client may send several `SYN` packets before one of them is received by the server. In SOS⋆, however, we represent all of these `SYN` packets by a single REQ packet, i. e. we hide the packets that are lost. That means that in SOS⋆, a REQ packet simply declares the willingness to connect to a remote site. Thus, instead of (re-) modeling the opening phase of the underlying TCP protocol, we take advantage of the TCP guarantees and use the abstract network packets REQ and READY only as means of synchronization. Hence, there is no need to send a REQ packet several times. This is also the reason why the final `ACK` packet, the third way of the so-called three-way handshake [Ste93], is invisible in SOS⋆. In Appendix A.2, we formally proved that, for SOS⋆ and DSOS⋆, this two-way handshake is indeed a valid abstraction of the three-way handshake.[24] Similar abstractions are used for the communication- and closing phase.

Having abstract network packets in place, we are now able to add NET *np*, as an abstract representation of network input, to the input alphabet Σ:

$$\Sigma \supset \{\text{NET } np \mid np \in np\_t\}$$

and as an abstract representation of network output to the output alphabet Ω:

$$\Omega = \{\text{NET } np \mid np \in np\_t\}.$$

4.3.4.3 Socket Invariants

Before we can finally get to the actual calls and handlers, we still need to formulate a number of necessary socket invariants. These invariants primarily

[24] The proof in Appendix A.2 is taken from the Verisoft Technical Report #5 [Bog08d].

rely on the fact that, in our implementation, the only sockets that are shared are endpoints of established connections. That is, in SOS*, while forking an application, only those sockets that are in the state ESTABLISHED or in the state REMOTE-CLOSED are made accessible to the child application (*afork* defined in § 4.3.6.2). Furthermore, shared sockets can not be reused to establish new connections, i. e. the state of such a socket can not be changed (back) to BOUND, LISTEN, or ACCEPTING. Finally, opening a socket on a port that is already in use is prohibited:

- If a socket is in the state BOUND or CONNECTING, then there is no other socket associated with the same local port:

$$\begin{gathered}
\mathit{inv\text{-}unique\text{-}socket\text{-}bound\text{-}connecting} \in \mathcal{S} \to \mathbb{B} \\
\mathit{inv\text{-}unique\text{-}socket\text{-}bound\text{-}connecting}(s) \equiv \\
\forall \mathit{sid}_1, \mathit{sid}_2.\ s.\mathit{sdb}(\mathit{sid}_1).\mathit{state} \in \{\text{BOUND}, \text{CONNECTING}\} \\
\wedge\ s.\mathit{sdb}(\mathit{sid}_2) \neq \varepsilon \wedge \mathit{sid}_1 \neq \mathit{sid}_2 \\
\Longrightarrow \\
s.\mathit{sdb}(\mathit{sid}_1).\mathit{lpn} \neq s.\mathit{sdb}(\mathit{sid}_2).\mathit{lpn}.
\end{gathered}$$

- If a socket is in state LISTEN, then there is no other socket listening on the same local port:

$$\begin{gathered}
\mathit{inv\text{-}unique\text{-}state\text{-}listening} \in \mathcal{S} \to \mathbb{B} \\
\mathit{inv\text{-}unique\text{-}state\text{-}listening}(s) \equiv \\
\forall \mathit{sid}_1, \mathit{sid}_2.\ s.\mathit{sdb}(\mathit{sid}_1).\mathit{state} = \text{LISTEN} \\
\wedge\ s.\mathit{sdb}(\mathit{sid}_2) \neq \varepsilon \wedge \mathit{sid}_1 \neq \mathit{sid}_2 \\
\Longrightarrow \\
s.\mathit{sdb}(\mathit{sid}_2).\mathit{lpn} \neq s.\mathit{sdb}(\mathit{sid}_1).\mathit{lpn} \\
\vee\ s.\mathit{sdb}(\mathit{sid}_2).\mathit{state} \neq \text{LISTEN}.
\end{gathered}$$

- If a socket is in state ACCEPTING, then there is no other socket accepting on the same local port:

$$\begin{gathered}
\mathit{inv\text{-}unique\text{-}state\text{-}accepting} \in \mathcal{S} \to \mathbb{B} \\
\mathit{inv\text{-}unique\text{-}state\text{-}accepting}(s) \equiv \\
\forall \mathit{sid}_1, \mathit{sid}_2.\ s.\mathit{sdb}(\mathit{sid}_1).\mathit{state} = \text{ACCEPTING} \\
\wedge\ s.\mathit{sdb}(\mathit{sid}_2) \neq \varepsilon \wedge \mathit{sid}_1 \neq \mathit{sid}_2 \\
\Longrightarrow \\
s.\mathit{sdb}(\mathit{sid}_2).\mathit{lpn} \neq s.\mathit{sdb}(\mathit{sid}_1).\mathit{lpn} \\
\vee\ s.\mathit{sdb}(\mathit{sid}_2).\mathit{state} \neq \text{ACCEPTING}.
\end{gathered}$$

- Connections are unique, i. e. there are no two sockets that are in the state ESTABLISHED and that have the same local port, the same remote port, and the same remote network address:

$$\begin{aligned}
&\textit{inv-unique-connection} \in \mathcal{S} \to \mathbb{B} \\
&\textit{inv-unique-connection}(s) \equiv \\
\forall \textit{sid}_1, \textit{sid}_2.\ & s.sdb(\textit{sid}_1).\textit{state} \in \{\text{ESTABLISHED}\} \\
&\wedge\ s.sdb(\textit{sid}_2).\textit{state} \in \{\text{ESTABLISHED}\} \\
&\wedge\ \textit{sid}_1 \neq \textit{sid}_2 \\
&\Longrightarrow \\
& s.sdb(\textit{sid}_1).\textit{lpn} \neq s.sdb(\textit{sid}_2).\textit{lpn} \\
&\vee\ s.sdb(\textit{sid}_1).\textit{rpn} \neq s.sdb(\textit{sid}_2).\textit{rpn} \\
&\vee\ s.sdb(\textit{sid}_1).\textit{rna} \neq s.sdb(\textit{sid}_2).\textit{rna}.
\end{aligned}$$

- If there are two application that have access to the same socket, then this socket is in the state ESTABLISHED, or in the state REMOTE-CLOSED:

$$\begin{aligned}
&\textit{inv-only-established-shared} \in \mathcal{S} \to \mathbb{B} \\
&\textit{inv-only-established-shared}(s) \equiv \\
\forall \textit{sid}.\ \exists \textit{hn}_1, \textit{hn}_2.\ & \textit{sid} \in s.adb(\textit{hn}_1).\textit{sockets} \\
&\wedge\ \textit{sid} \in s.adb(\textit{hn}_2).\textit{sockets} \\
&\wedge\ \textit{hn}_1 \neq \textit{hn}_2 \\
&\Longrightarrow \\
& s.sdb(\textit{sid}).\textit{state} \in \{\text{ESTABLISHED}, \text{REMOTE-CLOSED}\}.
\end{aligned}$$

- Since TCP provides in-order delivery of packets, we can be sure that a CLOSE packet is only (successfully) received, after all DATA packets have been received and acknowledged. Hence, if a socket's state is REMOTE-CLOSED, then there can not be any unacknowledged data:

$$\begin{aligned}
&\textit{inv-close-in-order} \in \mathcal{S} \to \mathbb{B} \\
&\textit{inv-close-in-order}(s) \equiv \\
\forall \textit{sid}.\ & s.sdb(\textit{sid}).\textit{state} = \text{REMOTE-CLOSED} \\
&\Longrightarrow \\
& (s.sdb(\textit{sid}).\textit{in} - s.sdb(\textit{sid}).\textit{read}) \leq \text{SOCK-WIN-SIZE}.
\end{aligned}$$

After we have discussed assumptions and guarantees of the TCP layer, introduced abstract network packets, and stated some socket invariants, we will now specify SOS calls and event handlers related to Socket I/O. While specifying the individual calls and handlers we will use the (above-defined) socket invariants and show how they are maintained.

4.3.4.4 Open a Socket

As described earlier, the first steps towards an established connection are creating and binding a socket. Usually this is achieved by two separate system calls. In SOS, however, this is achieved by a single call. For that, the library Libsos implements the following call that creates a socket and immediately binds it to a port:

```
int sc_socket_open(unsigned int pn, unsigned int* sid).
```

In the SOS implementation, `sc_socket_open` is handled by `sos_socket_open`. If there is no error, then a socket is initialized, it is bound to the local port `pn`, and the socket id is returned to the calling application.

In the specification, we add SOPEN pn_t_ε, as an abstract representation of `sc_socket_open`, to the input alphabet Ω_{sc}:

$$\Omega_{\text{sc}} \supset \{\text{SOPEN } pn \mid pn \in pn\_t_\varepsilon\}$$

and SUCC-SOPEN sid_t as well as SOCK, as abstract representations of possible results, to the output alphabet Σ_{sc}:

$$\Sigma_{\text{sc}} \supset \{\text{SUCC-SOPEN } sid, \text{SOCK} \mid sid \in sid\_t\}.$$

The behavior of `sos_socket_open` is described by the function *sopen*. This function takes, as a call-specific argument, the desired port number:

$$sopen \in \mathcal{S} \times hn\_t \times pn\_t_\varepsilon \to \mathcal{S} \times \Omega^* \times (hn\_t \times \Sigma_{\text{sc}})^*.$$

For $sopen(s, hn, pn)$, the following cases are considered:

- If there are no more socket ids available, i. e. if the set of unassigned socket ids *free* is empty, or the calling application a has reached the maximum number of sockets per application $\text{MSPA} \in \mathbb{N}_{32}^+$, i. e. $|a.sockets| = \text{MSPA}$, then the message (hn, LIMIT) is returned.
- If pn is not a valid port number, i. e. $pn = \varepsilon$, then the message (hn, ARG) is returned.
- If pn is already in use, i. e. $\exists x.\ s.sdb(x).lpn = pn$, then the message (hn, SOCK) is returned.
- If none of the previous cases apply, then a new socket, with the socket id sid_{n}, is initialized and added to the socket data base. Furthermore, sid_{n} is added to the application's socket references and the success message m, including the new socket's id, is returned.

This adds up to the following definition of *sopen*:

$$sopen(s, hn, pn) =$$

$$\begin{array}{lll}
\textbf{let} & a & = s.adb(hn); \\
& free & = \{x \mid x \in sid_t \wedge s.sdb(x) = \varepsilon\}; \\
& sid_{\mathrm{n}} & = min(free); \\
& s_1 & = s \left[\!\!\left[\begin{array}{lll} sdb(sid_{\mathrm{n}}) & := & \left[\!\!\left[\begin{array}{ll} state & = \text{BOUND}, \\ lpn & = pn, \\ lq & = [\,], \\ rna & = \varepsilon, \\ rpn & = \varepsilon, \\ in & = [\,], \\ read & = 0, \\ out & = [\,], \\ ack & = 0 \end{array} \right]\!\!\right], \\ adb(hn).sockets & := & a.sockets \cup \{sid_{\mathrm{n}}\} \end{array} \right]\!\!\right]; \\
& m & = \text{SUCC-SOPEN } sid_{\mathrm{n}}
\end{array}$$

$$\textbf{in} \quad \begin{cases} (s\ , [\,], [(hn, \text{LIMIT})]) & \textbf{if } free = \{\,\} \vee |a.sockets| = \text{MSPA}, \\ (s\ , [\,], [(hn, \text{ARG})]) & \textbf{else if } pn = \varepsilon, \\ (s\ , [\,], [(hn, \text{SOCK})]) & \textbf{else if } \exists x.\ s.sdb(x).lpn = pn, \\ (s_1, [\,], [(hn, m)]) & \textbf{else.} \end{cases}$$

4.3.4.5 Change a Socket to Listen Mode

Before specifying how a socket's state may be changed to listening, two auxiliary functions should be introduced.

Similar to *faccess-legal*, the predicate *saccess-legal* is used to check whether some socket access is legal. *saccess-legal*$(s, hn, sid, states)$ is satisfied, if sid is a valid socked id, sid is in the set of socket references of application hn, and the socket's state is in the set $states$:

$$saccess\text{-}legal \in \mathcal{S} \times hn\_t \times sid\_t_{\varepsilon} \times \mathcal{P}(sstate\_t) \to \mathbb{B}$$

$$saccess\text{-}legal(s, hn, sid, states) \equiv$$

$$sid \neq \varepsilon \wedge sid \in s.adb(hn).sockets$$
$$\wedge\ (s.sdb(sid).state \in states \vee states = \{\,\}).$$

Note, if *saccess-legal* is called with $states = \{\}$, then the socket's state does not matter. We will use this, when closing a socket (*sclose* defined in § 4.3.4.14).

The function *saccess-error* is the counterpart to *saccess-legal*. It returns the correct error message in the case when the predicate *saccess-legal* is not satisfied:

$$\textit{saccess-error} \in \mathcal{S} \times \textit{hn\_t} \times \textit{sid\_t}_\varepsilon \to \mathcal{S} \times \Omega^* \times (\textit{hn\_t} \times \Sigma_{\text{sc}})^*$$

$$\textit{saccess-error}(s, hn, sid) =$$

$$\begin{cases} (s, [\,], [(hn, \text{ARG})]) & \textbf{if } sid = \varepsilon, \\ (s, [\,], [(hn, \text{LOCK})]) & \textbf{else if } sid \notin s.adb(hn).sockets, \\ (s, [\,], [(hn, \text{SOCK})]) & \textbf{else.} \end{cases}$$

The library Libsos implements the following call that allows a user application to change a socket's state to listening:

```
int sc_socket_listen(unsigned int sid).
```

In the SOS implementation, `sc_socket_listen` is handled by `sos_socket_listen`. If there is no error, then, after the call is handled, the socket `sid` is in listen mode.

In the specification, we add SLISTEN $\textit{sid\_t}_\varepsilon$, as an abstract representation of `sc_socket_listen`, to the input alphabet Ω_{sc}:

$$\Omega_{\text{sc}} \supset \{\text{SLISTEN } sid \mid sid \in \textit{sid\_t}_\varepsilon\}.$$

The behavior of `sos_socket_listen` is described by the function *slisten*. This function takes, as a call-specific argument, the socket id:

$$\textit{slisten} \in \mathcal{S} \times \textit{hn\_t} \times \textit{sid\_t}_\varepsilon \to \mathcal{S} \times \Omega^* \times (\textit{hn\_t} \times \Sigma_{\text{sc}})^*.$$

For $\textit{slisten}(s, hn, sid)$, the following cases are considered:

- If $\textit{saccess-legal}(s, hn, sid, \{\text{BOUND}\})$ is not satisfied, then the result is computed and returned by $\textit{saccess-error}(s, hn, sid)$.
- If the previous case does not apply, then the socket's state is changed and a success message returned.

This adds up to the following definition of *slisten*:

$$\textit{slisten}(s, hn, sid) =$$

$$\textbf{let} \quad s_1 = s[\![sdb(sid).state := \text{LISTEN}]\!]$$

$$\textbf{in} \quad \begin{cases} \textit{saccess-error}(s, hn, sid) & \textbf{if } \neg\textit{saccess-legal}(s, hn, sid, \{\text{BOUND}\}), \\ (s_1, [\,], [(hn, \text{SUCC})]) & \textbf{else.} \end{cases}$$

Note, because of *inv-unique-socket-bound-connecting*, we can be sure that there is no other socket bound to the same local port. Hence, we can be sure that, after the call, *inv-unique-state-listening* holds.

4.3.4.6 Establish a Connection

The library Libsos implements the following call that allows a user application to establish a connection:

```
int sc_socket_connect( unsigned int sid, unsigned int ip,
                       unsigned int pn).
```

In the SOS implementation, `sc_socket_connect` is handled by the SOS-call handler `sos_socket_connect`. If there is no error, then, after the call returns to the calling application, there exists an established connection between the local system and the system with the IP address `ip`. For that, the three way handshake is initiated by sending a `SYN` packet to the remote site. After that, the local system performs other tasks and waits for the remote site to answer with a corresponding `SYN/ACK` packet. If this packet is received, then the three way handshake is completed by sending the final `ACK` packet. Finally, if all goes well, the SOS call returns and informs the calling application about the success.

In the specification, we add SCONNECT sid_t_ε na_t_ε pn_t_ε, as an abstract representation of `sc_socket_connect`, to the input alphabet Ω_{sc}:

$$\Omega_{\text{sc}} \supset \{\text{SCONNECT } sid\ na\ pn \mid sid \in sid\_t_\varepsilon \wedge na \in na\_t_\varepsilon \wedge pn \in pn\_t_\varepsilon\}.$$

The behavior of `sos_socket_connect` is described by the function *sconnect*. This function takes, as call-specific arguments, the socket id, the remote network address, and the remote port number:

$$\begin{aligned} sconnect \in\ & \mathcal{S} \times hn\_t \times sid\_t_\varepsilon \times na\_t_\varepsilon \times pn\_t_\varepsilon \\ & \to \mathcal{S} \times \Omega^* \times (hn\_t \times \Sigma_{\text{sc}})^*. \end{aligned}$$

For $sconnect(s, hn, sid, na, pn)$, the following cases are considered:[25]

- If $saccess\text{-}legal(s, hn, sid, \{\text{BOUND}\})$ is not satisfied, then the result is computed and returned by $saccess\text{-}error(s, hn, sid)$.
- If the previous case does not apply, but the remote network address is invalid, i. e. $na = \varepsilon$, or the remote port number is invalid, i. e. $pn = \varepsilon$, then the message (hn, ARG) is returned.

[25] The parameter na from $sconnect(s, hn, sid, na, pn)$ corresponds to the `ip` parameter in the Libsos call. This is because we choose to name the corresponding type (abstract) network address (na_t) rather than IP address.

- If none of the previous cases apply, then the remote network address and the remote port number are stored in the socket *sid* and the abstract network packet *np* is sent to the remote site. Furthermore, in order to indicate the pending connection attempt, the state of the socket *sid* is changed to CONNECTING.

This adds up to the following definition of *sconnect*:

$$\mathit{sconnect}(s, hn, sid, na, pn) =$$

$$\begin{array}{lll}
\textbf{let} & sock & = s.sdb(sid); \\
& sock' & = sock[\![state := \text{CONNECTING}, rna := na, rpn := pn]\!]; \\
& np & = \text{REQ } s.lna\ sock.lpn\ na\ pn; \\
& s_1 & = s[\![sdb(sid) := sock']\!]
\end{array}$$

$$\textbf{in} \quad \begin{cases} \mathit{access\text{-}error}(s, hn, sid) & \textbf{if } \neg \mathit{access\text{-}legal}(s, hn, sid, \{\text{BOUND}\}), \\ (s\ , [\,], [(hn, \text{ARG})]) & \textbf{else if } na = \varepsilon \vee pn = \varepsilon, \\ (s_1, [\text{NET } np], [\,]) & \textbf{else}. \end{cases}$$

Note, as for *slisten*, because of *inv-unique-socket-bound-connecting*, we can be sure that there is no other socket bound to the same local port. Hence, we can be sure that, after the call returns, *inv-unique-connection* holds. Similar to *tread*, it is a side effect of the event handler *event-sready* (defined in § 4.3.4.9) to complete the process of establishing a connection and to return a result to the calling application.

4.3.4.7 Accept incoming Connection Requests

Before we go on to describe what happens if a REQ packet is received, we want to introduce another SOS call.

The library Libsos implements the following call that allows a user application, in this case a server, to accept a new connection:

```
int sc_socket_accept( unsigned int sid, unsigned int* ip
                      unsigned int* pn, unsigned int* new_sid).
```

In the SOS implementation, `sc_socket_accept` is handled by `sos_socket_accept`. There are two main scenarios for `sc_socket_accept`.

- If there are no errors and there is some remote site, in this case a client that has already requested a connection, then the connection is established by sending the `SYN/ACK` packet, and the SOS call returns immediately. Thereby, the remote IP address `ip` and the remote port number `pn` as well as the socket id of the socket that is used for the new connection (`new_sid`), are reported to the calling application.

Note that the new socket is necessary, as the original socket `sid` remains in a listening state. This is the standard behavior. In combination with a fork operation, servers, such as SMTP servers or HTTP servers, commonly use this to have a single process accepting all incoming requests and then serve these requests (in parallel) in separate processes. In fact, the SMTP server implemented on top of the SOS does that too.

- If there are no errors and (so far) no client has requested a connection, then the local system performs other tasks and waits for some remote site to request one.

In the specification, we add SACCEPT sid_t_ε, as an abstract representation of `sc_socket_accept`, to the input alphabet Ω_{sc}:

$$\Omega_{\text{sc}} \supset \{\text{SACCEPT } sid \mid sid \in sid\_t_\varepsilon\}$$

and SUCC-SACCEPT $sid\_t\ na\_t\ pn\_t$, as an abstract representation of a possible result, to the output alphabet Σ_{sc}:

$$\Sigma_{\text{sc}} \supset \{\text{SUCC-SACCEPT } sid\ na\ pn \mid sid \in sid\_t \wedge na \in na\_t \wedge pn \in pn\_t\}.$$

The behavior of `sos_socket_connect` is described by the function *saccept*. This function takes, as a call-specific argument, the id of the socket:

$$saccept \in \mathcal{S} \times hn\_t \times sid\_t_\varepsilon \to \mathcal{S} \times \Omega^* \times (hn\_t \times \Sigma_{\text{sc}})^*.$$

For $saccept(s, hn, sid)$, the following cases are considered:

- If $saccess\text{-}legal(s, hn, sid, \{\text{LISTEN}\})$ is not satisfied, then the result is computed and returned by $saccess\text{-}error(s, hn, sid)$.
- If there are no more socket ids available or the maximum number of sockets per application is reached, then the message (hn, LIMIT) is returned.
- If none of the previous cases apply but the listen queue is empty, i. e. up to now there was no REQ packet received for that socket, then the socket $sock_{\text{n}}$ with id sid_{n} is reserved for a later connection. That means that $sock_{\text{n}}$ is initialized and added to the socket data base and sid_{n} is added to the set of socket references of the calling application. The new socket's state is thereby set to ACCEPTING, and the remote network address and the remote port are set to ε.

  In this case there is no immediate result. Instead, a result is returned if an appropriated REQ packet is received and the connection can be established. That is, if the event handler that treats REQ packets (*event-sreq* defined in § 4.3.4.8) receives an appropriate packet on the local port pn.

Note that because of *inv-unique-state-listening* there can not be another socket listening on the same port. Moreover, *saccept* does not return until the connection via sid_{n} is established. Hence, we can be sure that (this case of) *saccept* does not violate the invariant *inv-unique-state-accepting*.

- Finally, if the predicate *saccess-legal* is satisfied, (na, pn) is the oldest entry in the listen queue, and no limits are exceeded, then the following happens. The new socket $sock_{\mathrm{n}}$ is initialized; the abstract network packet np is sent to the remote site; and the success message m, containing the remote port, remote network address, and the new socket's id, is returned. Furthermore, the head of the listen queue is removed, the new socket added to the socket data base, and the new socket's id added to the set of socket references of the calling application. Other than in the previous case, the new socket's state is set to ESTABLISHED, and the remote network address and remote port number are set to na and pn.

  Note that because of the TCP assumption 'Absence of Timeouts', we can be sure that the remote site is still waiting and that it will send the final ACK packet. Because of this, in SOS*, we can abstract the remaining part of the three way handshake and claim that the connection has been established (Appendix A.2).

  Further, note that because we assume that *sconnect* ensures *inv-unique-connection* at the remote site, *saccept* does so too on the local site.

This adds up to the following definition of *saccept*:

$$saccept(s, hn, sid) =$$

$$\begin{aligned}
\textbf{let}\quad & sock = s.sdb(sid);\\
& lq = sock.lq;\\
& free = \{x \mid x \in sid_t \wedge s.sdb(x) = \varepsilon\};\\
& sid_{\mathrm{n}} = min(free);\\
& sockets = s.adb(hn).sockets;\\
& (na, pn) = \begin{cases} lq[0] & \textbf{if } lq \neq [\,], \\ (\varepsilon, \varepsilon) & \textbf{else}; \end{cases}\\
& state' = \begin{cases} \text{ESTABLISHED} & \textbf{if } lq \neq [\,], \\ \text{ACCEPTING} & \textbf{else}; \end{cases}\\
& sock_{\mathrm{n}} = sock \left[\!\!\left[\begin{array}{ll} state & := state', \\ lq & := [\,], \\ rna & := na, \\ rpn & := pn \end{array} \right]\!\!\right];
\end{aligned}$$

$$
\begin{aligned}
s_1 &= s\left[\!\!\left[\begin{array}{ll} sdb(sid_{\mathrm{n}}) & := sock_{\mathrm{n}}, \\ adb(hn).sockets & := sockets \cup \{sid_{\mathrm{n}}\} \end{array}\right]\!\!\right]; \\
s_2 &= s_1[\![sdb(sid).lq := tail(lq)]\!]; \\
m &= [(hn, \text{SUCC-SACCEPT } sid_{\mathrm{n}}\ na\ pn)]; \\
np &= \text{READY } s.lna\ sock.lpn\ na\ pn
\end{aligned}
$$

in

$$
\begin{cases}
saccess\text{-}error(s, hn, sid) & \textbf{if } \neg saccess\text{-}legal(s, hn, sid, \{\text{LISTEN}\}), \\
(s\ , [\,], [(hn, \text{LIMIT})]) & \textbf{else if } free = \{\,\} \vee |sockets| = \text{MSPA}, \\
(s_1, [\,], [\,]) & \textbf{else if } lq = [\,], \\
(s_2, [\text{NET } np], m) & \textbf{else.}
\end{cases}
$$

4.3.4.8 Network Input — REQ

Before we go on to describe the first socket-related event handler, we need to introduce an auxiliary function.

The function $match\text{-}socket(s, state, lpn, rna, rpn)$ returns the socket id of the socket that matches the arguments. If several sockets match, then the smallest socket id is returned. If no match can be found, then ε is returned:

$$
\begin{gathered}
match\text{-}socket \in \mathcal{S} \times \mathcal{P}(sstate\_t) \times pn\_t_\varepsilon \times na\_t_\varepsilon \times pn\_t_\varepsilon \to sid\_t_\varepsilon \\
match\text{-}socket(s, state, lpn, rna, rpn) =
\end{gathered}
$$

let

$$
matches = \left\{
\begin{array}{l}
x \mid x \in sid\_t \wedge s.sdb(x) \neq \varepsilon \\
\qquad \wedge (s.sdb(x).state \in state \vee state = \{\,\}) \\
\qquad \wedge (s.sdb(x).lpn = lpn \vee lpn = \varepsilon) \\
\qquad \wedge (s.sdb(x).rna = rna \vee rna = \varepsilon) \\
\qquad \wedge (s.sdb(x).rpn = rpn \vee rpn = \varepsilon)
\end{array}
\right\}
$$

in

$$
\begin{cases}
min(matches) & \textbf{if } matches \neq \{\,\}, \\
\varepsilon & \textbf{else.}
\end{cases}
$$

If the network card receives some packet, an interrupt is raised and delivered to the SOS. At the highest level of our TCP / IP implementation, such an interrupt is treated by the interrupt handler `sos_socket_int`. Here, the interrupt is first of all passed down through the different layers of the TCP / IP stack. At the lowest level, the actual data is read from the device. Then, as results are passed back up, each level of the TCP / IP stack might process the new data. The implementation of the different interrupt handlers is quite complex, but, in the end, it is possible to group all transitions into five distinct classes.

As described earlier, each of these classes is represented by a different type of abstract network packet, i. e. REQ, READY, DATA, ACK, and CLOSE.

Now, if a remote site wants to connect to the local system, i. e. a SYN packet has been received, then, in SOS*, this is modeled as external input that is an abstract network packet of type REQ. The behavior of the corresponding interrupt handler(s) is described by the function *event-sreq*. This function takes, as event-specific arguments, the sender's network address and port number as well as the destination network address and port number:

$$\textit{event-sreq} \in \mathcal{S} \times \textit{na\_t} \times \textit{pn\_t} \times \textit{na\_t} \times \textit{pn\_t} \to \mathcal{S} \times \Omega^* \times (\textit{hn\_t} \times \Sigma_{\text{sc}})^*.$$

For $\textit{event-sreq}(s, \textit{sna}, \textit{spn}, \textit{dna}, \textit{dpn})$, the following cases are considered:

- If the packet that was received is not intended for the local system, i. e. $\textit{dna} \neq s.\textit{lna}$, or if there is no application listening on the destination port *dpn*, i. e. $\textit{match-socket}(s, \{\text{LISTEN}\}, \textit{dpn}, \varepsilon, \varepsilon) = \varepsilon$, then this packet is dropped.
- If there is no pending accept request, then the connection attempt is stored in the listen queue of the listening socket.
- If there is a pending accept request, then the connection is established and the abstract network packet *np* sent to the remote site. Furthermore, the success message *m*, containing the remote port, remote network address and the new socket's id, is returned to the application waiting for the accept to complete.

This adds up to the following definition of *event-sreq*:

$$\textit{event-sreq}(s, \textit{sna}, \textit{spn}, \textit{dna}, \textit{dpn}) =$$

$$\begin{aligned}
\textbf{let}\quad & \textit{sid}_1 = \textit{match-socket}(s, \{\text{LISTEN}\}, \textit{dpn}, \varepsilon, \varepsilon);\\
& \textit{sid}_2 = \textit{match-socket}(s, \{\text{ACCEPTING}\}, \textit{dpn}, \varepsilon, \varepsilon);\\
& s_1 \;= s[\![\textit{sdb}(\textit{sid}_1).\textit{lq} := s.\textit{sdb}(\textit{sid}_1).\textit{lq} \circ [(\textit{sna}, \textit{spn})]]\!];\\
& s_2 \;= s \left[\!\!\left[\begin{array}{ll} \textit{sdb}(\textit{sid}_2).\textit{state} & := \text{ESTABLISHED},\\ \textit{sdb}(\textit{sid}_2).\textit{rna} & := \textit{sna},\\ \textit{sdb}(\textit{sid}_2).\textit{rpn} & := \textit{spn} \end{array} \right]\!\!\right];\\
& \textit{hn} \;= \epsilon\{x \mid \textit{sid}_2 \in s.\textit{adb}(x).\textit{sockets}\};\\
& m \;= [(\textit{hn}, \text{SUCC-SACCEPT } \textit{sid}_2\ \textit{sna}\ \textit{spn})];\\
& \textit{np} \;= \text{READY } \textit{dna}\ \textit{dpn}\ \textit{sna}\ \textit{spn}
\end{aligned}$$

$$\textbf{in}\quad \begin{cases} (s\ , [\,], [\,]) & \textbf{if } \textit{dna} \neq s.\textit{lna} \vee (\textit{sid}_1 = \varepsilon \wedge \textit{sid}_2 = \varepsilon),\\ (s_1, [\,], [\,]) & \textbf{else if } \textit{sid}_2 = \varepsilon,\\ (s_2, [\text{NET } \textit{np}], m) & \textbf{else.} \end{cases}$$

Note that in the implementation, listen queues are bounded, but, because of the assumption 'Absence of Timeouts', we can be sure that the connection attempt will eventually be added to the listen queue — even a bounded one. Hence, unbounded listen queues are a valid abstraction. Also note that if there exists a socket waiting for a REQ packet, i. e. $sid_2 \neq \varepsilon$, then, because of the invariants *inv-unique-state-accepting* and *inv-only-established-shared*, there can only be one application x such that $sid_2 \in s.adb(x).sockets$. Hence, it is deterministic to use ϵ to discover the handle of this application.

4.3.4.9 Network Input — READY

If some remote site responds to a connect request, i. e. a SYN/ACK packet is received, then, in SOS*, this is modeled as external input, which is an abstract network packet of type READY. The behavior of the corresponding interrupt handler(s) is described by the function *event-sready*. This function takes, as event-specific arguments, the sender's network address and port number as well as the destination network address and port number:

$$\textit{event-sready} \in \mathcal{S} \times \textit{na\_t} \times \textit{pn\_t} \times \textit{na\_t} \times \textit{pn\_t} \to \mathcal{S} \times \Omega^* \times (\textit{hn\_t} \times \Sigma_{\text{sc}})^*.$$

For $\textit{event-sready}(s, sna, spn, dna, dpn)$, the following cases are considered:

- If the packet that was received is not intended for the local system, or none of the applications on the local system tries to connect to the remote system, i. e. $\textit{match-socket}(s, \{\text{CONNECTING}\}, dpn, sna, spn) = \varepsilon$, then this packet is dropped.
- If the previous case does not apply, i. e. there exists an application hn that wants to connect to the source of the READY packet via the socket sid, then the socket's state is set to ESTABLISHED, and a success message returned.

This adds up to the following definition of *event-sready*:

$$\textit{event-sready}(s, sna, spn, dna, dpn) =$$

$$\begin{aligned}
\textbf{let}\quad & sid = \textit{match-socket}(s, \{\text{CONNECTING}\}, dpn, sna, spn);\\
& hn = \epsilon\{x \mid sid \in s.adb(x).sockets\};\\
& s_1 = s[\![sdb(sid).state := \text{ESTABLISHED}]\!]
\end{aligned}$$

$$\textbf{in}\quad \begin{cases} (s\ , [\,], [\,]) & \textbf{if } dna \neq s.lna \vee sid = \varepsilon,\\ (s_1, [\,], [(hn, \text{SUCC})]) & \textbf{else.} \end{cases}$$

Note that, as mentioned earlier, the final ACK packet is hidden in our abstraction (Appendix A.2). Furthermore, if there exists a socket waiting for a READY

packet, i. e. $sid \neq \varepsilon$, then, because of *inv-unique-socket-bound-connecting* and *inv-only-established-shared*, there can only be one application x such that $sid \in s.adb(x).sockets$. Hence, it is deterministic to use ϵ to discover the handle of this application.

4.3.4.10 Write to a Socket

The library Libsos implements the following call that allows a user application to write to a socket, i. e. send data to a remote site via an established connection:

```
int sc_socket_write( unsigned int sid, unsigned int len
                     sos_buffer_t buf).
```

In the SOS implementation, `sc_socket_write` is handled by `sos_socket_write`. If there is no error, then, after the call is handled, (at most) the first `len` bytes from the buffer `buf` are appended to the output buffer of the socket `sid`.

In the specification, we add SWRITE sid_t_ε $byte\_t^*$, as an abstract representation of `sc_socket_write`, to the input alphabet Ω_{sc}:

$$\Omega_{\text{sc}} \supset \{\text{SWRITE } sid \ bytes \mid sid \in sid\_t_\varepsilon \wedge bytes \in byte\_t^*\}.$$

The behavior of `sos_socket_write` is described by the function *swrite*. This function takes, as call-specific arguments, the socket id and the bytes that should be written:

$$swrite \in \mathcal{S} \times hn\_t \times sid\_t_\varepsilon \times byte\_t^* \to \mathcal{S} \times \Omega^* \times (hn\_t \times \Sigma_{\text{sc}})^*.$$

For $swrite(s, hn, sid, bytes)$, the following cases are considered:

- If $access\text{-}legal(s, hn, sid, \text{ESTABLISHED})$ is not satisfied, then the result is computed and returned by $access\text{-}error(s, hn, sid)$.
- If the bytes *bytes* do not fit into the socket's output buffer, i. e. there are too many bytes in the output buffer that have not yet been acknowledged by the remote site, then the message (hn, LIMIT) is returned (see Figure 4.4 on the following page).
- If none of the previous cases apply, then the bytes are appended to the socket's output buffer, the abstract network packet *np* sent to the remote site, and a success message returned.

This adds up to the following definition of *swrite*:

$$swrite(s, hn, sid, bytes) =$$

let
$$\begin{aligned} sock &= s.sdb(sid); \\ s_1 &= s[\![sdb(sid).out := sock.out \circ bytes]\!]; \\ np &= \text{DATA } s.lna \ sock.lpn \ sock.rna \ sock.rpn \ bytes \end{aligned}$$

in

$$\begin{cases} \mathit{saccess\text{-}error}(s, hn, sid) & \textbf{if } \neg \mathit{saccess\text{-}legal}(s, hn, sid, \text{ESTABLISHED}), \\ (s\ , [\,], [(hn, \text{LIMIT})]) & \textbf{else if } \mathit{length}(sock.out) - sock.ack \\ & \qquad + \mathit{length}(bytes) > \text{SOCK-WIN-SIZE}, \\ (s_1, [\text{NET } np], [(hn, \text{SUCC})]) & \textbf{else}. \end{cases}$$

Note that any partitioning of data is abstracted away. Furthermore, because of the liveness property for the TCP transmission phase, we can pretend that the data only need to be sent once.

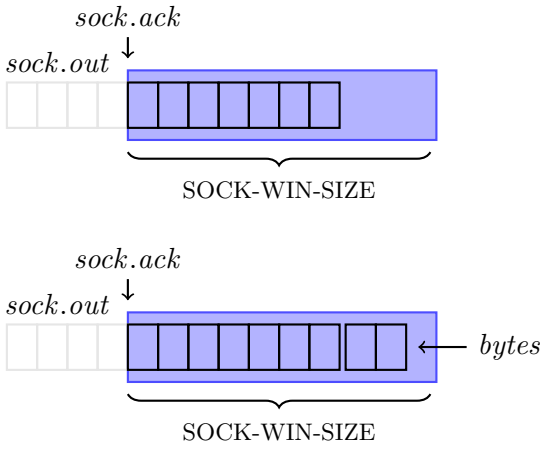

Figure 4.4: Writing to a Socket (*swrite*). Two bytes are appended to the output buffer *sock.out*. Appending these bytes increases the number of bytes that have not yet been acknowledged. Here, *swrite* would be successful because SOCK-WIN-SIZE is not exceeded.

4.3.4.11 Network Input — DATA

If some remote site sends some data, then, in SOS*, this is modeled as external input, which is an abstract network packet of type DATA. The behavior of the corresponding interrupt handler(s) is described by the function *event-sdata*. This function takes, as event-specific arguments, the sender's network address and port number, the destination network address and port number, and a list of bytes:

$$\begin{aligned} \mathit{event\text{-}sdata} \in\ & \mathcal{S} \times na\_t \times pn\_t \times na\_t \times pn\_t \times byte\_t^* \\ & \to \mathcal{S} \times \Omega^* \times (hn\_t \times \Sigma_{\text{sc}})^*. \end{aligned}$$

For $\mathit{event\text{-}sdata}(s, sna, spn, dna, dpn, bytes)$, the following cases are considered:

- If the packet that was received is not intended for the local system, or if the packet does not belong to an established connection between local- and remote system, i. e. $\mathit{match\text{-}socket}(s, \{\text{ESTABLISHED}\}, \mathit{dpn}, \mathit{sna}, \mathit{spn}) = \varepsilon$, then this packet is dropped.
- If the previous case does not apply, then the list of bytes *bytes* is appended to the input queue of the appropriate socket *sock* (see Figure 4.5 on the next page).
- Depending on the difference of the number of bytes that were received in the past and the number of bytes that have been locally delivered, *ack* bytes are acknowledged to the remote site. Here, *ack* is the number of new bytes that would still fit into the bounded input buffer in the implementation, i. e. the number of new bytes visible in the sliding window.

This adds up to the following definition of *event-sdata*:

$$\mathit{event\text{-}sdata}(s, \mathit{sna}, \mathit{spn}, \mathit{dna}, \mathit{dpn}, \mathit{bytes}) =$$

let

$$\begin{aligned}
\mathit{sid} &= \mathit{match\text{-}socket}(s, \{\text{ESTABLISHED}\}, \mathit{dpn}, \mathit{sna}, \mathit{spn});\\
\mathit{sock} &= s.\mathit{sdb}(\mathit{sid});\\
s_1 &= s[\![\mathit{sdb}(\mathit{sid}).\mathit{in} := \mathit{sock}.\mathit{in} \circ \mathit{bytes}]\!];\\
\mathit{ack} &= \mathit{min}((\mathit{sock}.\mathit{read} + \text{SOCK-WIN-SIZE} - \mathit{length}(\mathit{sock}.\mathit{in})), \mathit{length}(\mathit{bytes}));\\
\mathit{np} &= \text{ACK } \mathit{dna}\ \mathit{dpn}\ \mathit{sna}\ \mathit{spn}\ \mathit{ack}
\end{aligned}$$

in

$$\begin{cases}
(s\ , [\,], [\,]) & \textbf{if } \mathit{dna} \neq s.\mathit{lna} \vee \mathit{sid} = \varepsilon,\\
(s_1, [\,], [\,]) & \textbf{else if } \mathit{ack} \leq 0,\\
(s_1, [\text{NET } \mathit{np}], [\,]) & \textbf{else.}
\end{cases}$$

Note that because of the liveness and safety properties for the TCP transmission phase, we can be sure that data that was sent will be appended to the input buffer at one point. Hence, an unbounded input queue with a bounded acknowledgment is a valid abstraction. Further, note that because of *inv-unique-connection*, we can be sure that the data is appended to the right socket.

4.3.4.12 Read from a Socket

The library Libsos implements the following call that allows a user application to read from a socket:

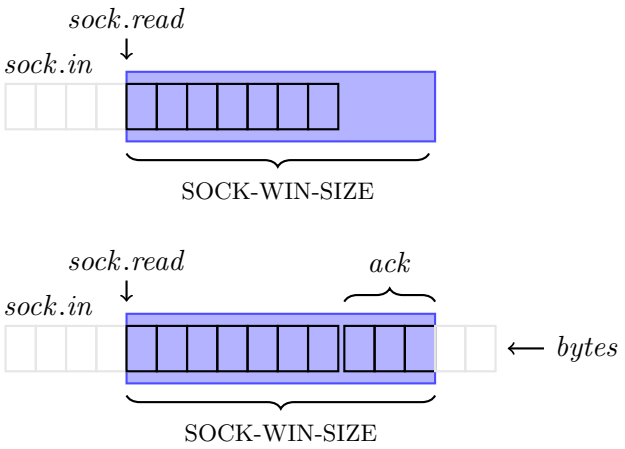

Figure 4.5: Receiving Data for a Socket (*event-sdata*). Five bytes are appended to the input queue *sock.in*. The first three of them, the ones already visible in SOCK-WIN-SIZE, are acknowledged. The remaining two will be acknowledged if some local application reads from the input queue (*sread*) and thereby increments the counter *sock.read*, i. e. if the window slides to the right.

```
int sc_socket_read( unsigned int sid, unsigned int* len,
                    sos_buffer_t* buf).
```

In the SOS implementation, `sc_socket_read` is handled by `sos_socket_read`. If there is no error, then at most `len` bytes are read from the socket with the id `sid` and returned to the calling application. The SOS only supports non-blocking reads on sockets. That means that `sos_socket_read` returns immediately. If there are less then `len` bytes in the input buffer, only these ones, together with the number of bytes that could be read, are returned.

In the specification, we add SREAD sid_t_ε $\mathbb{N}_{32}$, as an abstract representation of `sc_socket_read`, to the input alphabet Ω_{sc}:

$$\Omega_{\text{sc}} \supset \{\text{SREAD } sid\ len \mid sid \in sid\_t_\varepsilon \wedge len \in \mathbb{N}_{32}\}$$

and SUCC-SREAD $byte\_t^*$, as an abstract representation of a possible result, to the output alphabet Σ_{sc}:

$$\Sigma_{\text{sc}} \supset \{\text{SUCC-SREAD } bytes \mid bytes \in byte\_t^*\}.$$

The behavior of `sos_socket_read` is described by the function *sread*. This function takes, as call-specific arguments, the id of the socket and the number of bytes that should be read:

$$sread \in \mathcal{S} \times hn\_t \times sid\_t_\varepsilon \times \mathbb{N}_{32} \to \mathcal{S} \times \Omega^* \times (hn\_t \times \Sigma_{\text{sc}})^*.$$

For $sread(s, hn, sid, len)$, the following cases are considered:

- If *access-legal*$(s, hn, sid, leagl\text{-}states)$, with *leagl-states* containing the states ESTABLISHED and REMOTE-CLOSED, is not satisfied, then the result is computed and returned by *access-error*(s, hn, sid).
- If the previous case does not apply, then a success message m, including the list of bytes that could be read, is returned. The length len' is thereby calculated as the minimum of: *unread-data*, the number of up-to-now unread bytes; the length *len*; the buffer size $4 * \text{CMPC}$; and the window size SOCK-WIN-SIZE. As len' bytes are read, the counter *sock.read* needs to be increased by len' (see Figure 4.6 on the following page).
- At any time, there are at most SOCK-WIN-SIZE unread bytes that are acknowledged. Thus, there are *unack-data* = *unread-data* − SOCK-WIN-SIZE bytes in the input queue that have not yet been acknowledged. That means, after reading len' bytes, $ack = min(\{unack\text{-}data, len'\})$ bytes can be acknowledged. Hence, if $ack > 0$, then the abstract network packet *np* is sent to the remote site.

This adds up to the following definition of *sread*:

$$sread(s, hn, sid, len) =$$

$$\begin{array}{lll}
\textbf{let} & legal\text{-}states & = \{\text{ESTABLISHED}, \text{REMOTE-CLOSED}\}; \\
& sock & = s.sdb(sid); \\
& unread\text{-}data & = length(sock.in) - sock.read; \\
& len' & = min(\{unread\text{-}data, len, 4 * \text{CMPC}, \text{SOCK-WIN-SIZE}\}); \\
& bytes & = take(len', drop(sock.read, sock.in)); \\
& s_1 & = s[\![sdb(sid).read := sock.read + len']\!]; \\
& m & = [(hn, \text{SUCC-SREAD } bytes)]; \\
& unack\text{-}data & = unread\text{-}data - \text{SOCK-WIN-SIZE}; \\
& ack & = min(\{unack\text{-}data, len'\}); \\
& np & = \text{ACK } s.lna\ sock.lpn\ sock.rna\ sock.rpn\ ack
\end{array}$$

$$\textbf{in} \begin{cases} saccess\text{-}error(s, hn, sid) & \textbf{if } \neg saccess\text{-}legal(s, hn, sid, legal\text{-}states), \\ (s_1, [\,], m) & \textbf{else if } ack \leq 0, \\ (s_1, [\text{NET } np], m) & \textbf{else.} \end{cases}$$

Note that a socket's state is REMOTE-CLOSED, if the remote partner closed its side of the connection (*event-sclose* defined in § 4.3.4.15). In this case, we may no longer send data to the remote site. However, reading data that was already received is still possible. Because of *inv-close-in-order*, we know that if a socket's state is REMOTE-CLOSED, then *unack-data* ≤ 0. Therefore, we can be sure that we will not send an (abstract) ACK packet to a closed socket.

Even if a socket's state is ESTABLISHED, only those bytes that have already been acknowledged may be read. Thus, we require $len' \leq$ SOCK-WIN-SIZE. Finally, we may return $4 * \text{CMPC}$ bytes, rather than only CMPC bytes, since $|word\_t| = 4 * |byte\_t|$.

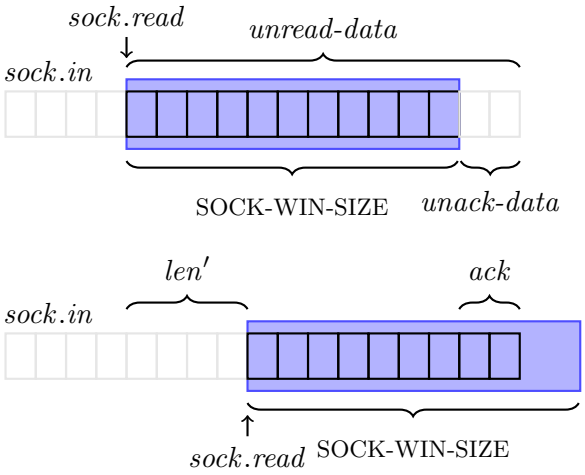

Figure 4.6: Reading From a Socket (*sread*). len' bytes are read from the input queue *sock.in* and the counter *sock.read* is incremented accordingly. Additionally, *ack* bytes, which were received but not yet acknowledged, can be acknowledged.

4.3.4.13 Network Input — ACK

If some remote site acknowledges data, then, in SOS⋆, this is modeled as external input that is an abstract network packet of type ACK. The behavior of the corresponding interrupt handler(s) is described by the function *event-sack*. This function takes, as event-specific arguments, the sender's network address and port number, the destination network address and port number, and the number of acknowledged bytes:

$$\textit{event-sack} \in \mathcal{S} \times na\_t \times pn\_t \times na\_t \times pn\_t \times \mathbb{N} \rightarrow \mathcal{S} \times \Omega^* \times (hn\_t \times \Sigma_{\text{sc}})^*.$$

For $\textit{event-sack}(s, sna, spn, dna, dpn, ack)$, the following cases are considered:

- If the packet that was received is not intended for the local system, or if the packet does not belong to an established connection between the local and the remote system, then this packet is dropped.
- If the previous case does not apply, then the counter *ack* of the appropriate socket *sock* is increased (also see Figure 4.7 on the next page).

This adds up to the following definition of *event-sack*:

$$\textit{event-sack}(s, sna, spn, dna, dpn, ack) =$$

$$\begin{array}{ll}
\textbf{let} & sid \;\; = \textit{match-socket}(s, \{\text{ESTABLISHED}\}, dpn, sna, spn); \\
& sock = s.sdb(sid); \\
& s_1 \;\; = s[\![sdb(sid).ack := sock.ack + ack]\!]
\end{array}$$

$$\textbf{in} \quad \begin{cases} (s\ , [\,], [\,]) & \textbf{if } dna \neq s.lna \vee sid = \varepsilon, \\ (s_1, [\,], [\,]) & \textbf{else}. \end{cases}$$

Note that because of *inv-unique-connection*, we can be sure that the correct socket is updated.

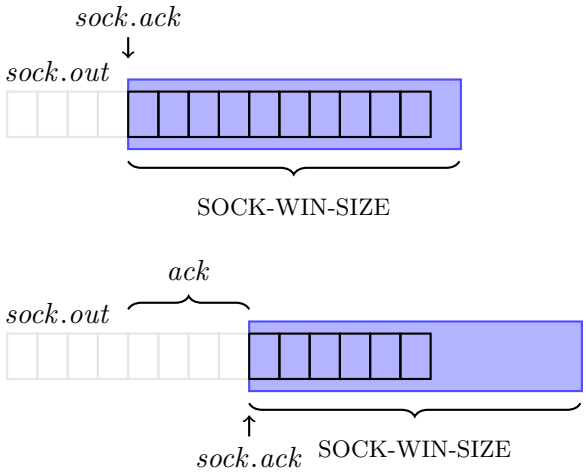

Figure 4.7: Receiving an Acknowledgment (*event-sack*). Here, *ack* bytes are acknowledged. That means, the counter *sock.ack* can be incremented by *ack*.

4.3.4.14 Close a Socket

The library Libsos implements the following call that allows a user application to close a socket:

```
int sc_socket_close(unsigned int sid).
```

In the SOS implementation, `sc_socket_close` is handled by `sos_socket_close`. There are two main scenarios for `sc_socket_close`.

- If there are no errors and the socket is not shared, then the socket is removed and the remote site informed.

- If there are no errors but the socket is shared, then only the calling application looses its reference to the socket. In this case the socket itself is not touched.

Note that, in the POSIX standard, the corresponding call `shutdown` allows to partially close a socket. There, it is possible to only disable one part of the full-duplex connection, i. e. a user may prevent subsequent send or receive operations but otherwise keep the socket. Here, we only support the 'full close', i. e. no more send or receive operations on this socket. This is also the reason why *swrite* fails for a socket that is in state REMOTE-CLOSED. In the specification, we add SCLOSE sid_t_ε, as an abstract representation of `sc_socket_close`, to the input alphabet Ω_{sc}:

$$\Omega_{\text{sc}} \supset \{\text{SCLOSE } sid \mid sid \in sid\_t_\varepsilon\}.$$

The behavior of `sos_socket_close` is described by the function *sclose*. This function takes, as a call-specific argument, the id of the socket:

$$sclose \in \mathcal{S} \times hn\_t \times sid\_t_\varepsilon \to \mathcal{S} \times \Omega^* \times (hn\_t \times \Sigma_{\text{sc}})^*.$$

For $sclose(s, hn, sid)$, the following cases are considered:

- If $saccess\text{-}legal(s, hn, sid, \{\})$ is not satisfied, then the result is computed and returned by $saccess\text{-}error(s, hn, sid)$.
- If $saccess\text{-}legal(s, hn, sid, \{\})$ is satisfied and there exists some application $x \neq hn$ that has a reference to the socket, then sid is only removed from the set of sockets accessible by hn.
- If $saccess\text{-}legal(s, hn, sid, \{\})$ is satisfied and hn is the only one that has access to the socket sid, then the socket as well as its reference are removed.
- Finally, unless the remote site has already closed its part of the connection, i. e. $s.sdb(sid).state = \text{REMOTE-CLOSED}$, the abstract network packet $np = (\text{CLOSE}, s.lna, sock.lpn, na, pn)$ is sent.

This adds up to the following definition of *sclose*:

$$sclose(s, hn, sid) =$$

$$\begin{aligned}
\textbf{let} \quad & s_1 = s[\![adb(hn).sockets := s.adb(hn).sockets \setminus \{sid\}]\!]; \\
& s_2 = s_1[\![sdb(sid) := \varepsilon]\!]; \\
& np = \text{CLOSE } s.lna \ s.sdb(sid).lpn \ na \ pn
\end{aligned}$$

in

$$\begin{cases} \textit{saccess-error}(s, hn, sid) & \textbf{if } \neg \textit{saccess-legal}(s, hn, sid, \{\}), \\ (s_1, [\,], [(hn, \text{SUCC})]) & \textbf{else if } \exists x \in hn\_t.\ x \neq hn \wedge s.adb(x) \neq \varepsilon \\ & \qquad \wedge\ sid \in s.adb(x).sockets, \\ (s_2, [\,], [(hn, \text{SUCC})]) & \textbf{else if } s.sdb(sid).state = \text{REMOTE-CLOSED}, \\ (s_2, [\text{NET } np], [(hn, \text{SUCC})]) & \textbf{else.} \end{cases}$$

Note that there are no restrictions on the state of a socket that should be closed. However, because *sconnect* (§ 4.3.4.6) and *saccept* (§ 4.3.4.7) are blocking calls and only those sockets that are part of an established connection are shared (*inv-only-established-shared*), we know that the socket's state is neither CONNECTING nor ACCEPTING.

4.3.4.15 Network Input — CLOSE

If some remote site closes its part of an established connection, then, in SOS*, this is modeled as external input that is an abstract network packet of type CLOSE. The behavior of the corresponding interrupt handler(s) is described by the function *event-sclose*. This function takes, as event-specific arguments, the sender's network address and port number and the destination network address and port number:

$$\textit{event-sclose} \in \mathcal{S} \times na\_t \times pn\_t \times na\_t \times pn\_t \to \mathcal{S} \times \Omega^* \times (hn\_t \times \Sigma_{\text{sc}})^*.$$

For $\textit{event-sclose}(s, sna, spn, dna, dpn)$, the following cases are considered:

- If the packet that was received is not intended for the local system, or if the packet does not belong to an established connection between the local and the remote system, then this packet is dropped.
- If the previous case does not apply, then the state of the corresponding socket *sock* is changed to REMOTE-CLOSED.

This adds up to the following definition of *event-sclose*:

$$\textit{event-sclose}(s, sna, spn, dna, dpn) =$$

$$\begin{array}{ll} \textbf{let} & sid = \textit{match-socket}(s, \{\text{ESTABLISHED}\}, dpn, sna, spn); \\ & s_1 \ = s \left[\!\!\left[sdb(sid).state := \text{REMOTE-CLOSED}, \right]\!\!\right] \end{array}$$

$$\textbf{in} \quad \begin{cases} (s\ , [\,], [\,]) & \textbf{if } dna \neq s.lna \vee sid = \varepsilon, \\ (s_1, [\,], [\,]) & \textbf{else.} \end{cases}$$

Note, because of *inv-unique-connection*, we can be sure to update the correct socket.

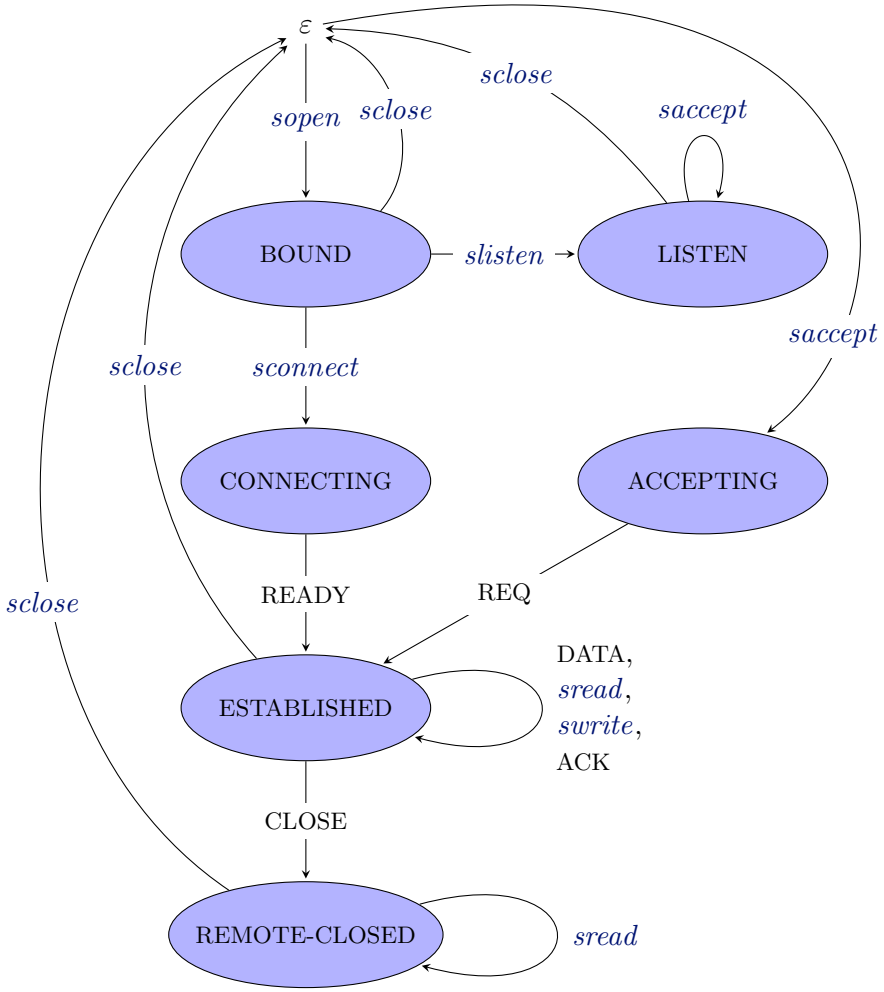

Figure 4.8: Socket States and Transitions. Socket states and transitions resulting from socket calls and external input.

4.3.4.16 How Does It Compare

In [Smi96b, Smi96a], Smith provides an abstract specification for TCP / IP transport level protocols and then proves that different (more concrete) models of TCP satisfy this specification. This work is very much different from what we did. He specifies TCP on the protocol level (rather than on the socket level) and he does not consider TCP in the context of an entire operating system. Thus, while he has to reveal more of the TCP internals (e. g. sequence numbers or various scenarios in the closing phase), he does not have to deal with TCP as the back end of sockets. He is (only) locking at a single connection and sockets being associated with different ports and applications do not matter for him. Although he was able to show simulation between differently abstract TCP models, he did not consider an actual implementation.

Later, in 2002, Smith et al. [SR02] specified the SACK extension for the TCP standard. However, this work can not be compared with ours as it only deals with an extension to TCP. In fact, we have not included any of the congestion avoidance optimizations in our system.

The NETSEM project [Net08] is not aiming at proving correctness of a particular implementation. However, their service-level specification [BFN+07] covers way more aspects of the standard socket implementations than ours does. For example, they specify blocking and non-blocking semantics for socket calls, they consider timeouts, and, above all, they specify the full range of socket calls for TCP- and UDP sockets. Still, their specification (summarized in [RNS08]) formalizes the communication via sockets at the same level of abstraction as ours does. If one ignores the specialities that are due to the more comprehensive approach, then their specification looks similar to ours. They also use higher order logic and thus, some of the transition rules they specify seem familiar. As we do, they divide transitions into those that are due to socket calls and those that are due to external inputs. However, they consider more cases and each of these cases is more complex. Thus, other than we do, they (further) split the specification of the socket calls and input handlers into individual rules. Then each of these rules only describes a specific success- or failure case. For example, while our specification of the socket-accept call (*saccept* defined in § 4.3.4.7) contains four different cases, they specify seven separate rules. However, the cases we consider (in *saccept*) are also present in the specification of Bishop et al. [BFN+07]. In the end, they provide a specification that is, on the one hand, more comprehensive but, on the other hand, not targeted at pervasive verification.

4.3.5 Portmapper

Remember, in order to implement RPCs we need a runtime mapping from service names to service providers. In the following subsection, we will specify SOS calls allow user applications to register, look up, and unregister a service.

4.3.5.1 Register a Service

The library Libsos implements the following call that allows a user application to register as a server providing some service:

```
int sc_pm_reg(unsigned int iid, unsigned int prcid).
```

In the SOS implementation, `sc_pm_reg` is handled by `sos_pm_reg`. If there is no error, then, after the call is handled, the calling application is registered as a server for the interface `iid`, or registered to provide the service (`iid`, `prcid`). Whether the service or only the interface is registered depends on the value of `prcid`.

In the specification, we add PMREG iid_t_ε $prcid\_t_\varepsilon \cup \{\text{PMINT}\}$, as an abstract representation of `sc_pm_reg`, to the input alphabet Ω_{sc}:

$$\Omega_{\text{sc}} \supset \{\text{PMREG } iid \; prcid \mid iid \in iid\_t_\varepsilon \wedge prcid \in prcid\_t_\varepsilon \cup \{\text{PMINT}\}\},$$

where the special procedure id PMINT indicates that only the interface should be registered.

The set of possible SOS call results Σ_{sc} is extended by PMDUP and PMOTHER:

$$\Sigma_{\text{sc}} \supset \{\text{PMDUP}, \text{PMOTHER}\}.$$

The behavior of `sos_pm_reg` is described by the function *pmreg*. This function takes, as call-specific arguments, the interface id and the procedure id:

$$\begin{aligned} pmreg \in \mathcal{S} \times hn\_t \times iid\_t_\varepsilon \times prcid\_t_\varepsilon \cup \{\text{PMINT}\} \\ \to \mathcal{S} \times \Omega^* \times (hn\_t \times \Sigma_{\text{sc}})^*. \end{aligned}$$

For $pmreg(s, hn, iid, prcid)$, the following cases are considered:

- If the application only wants to register the interface, i. e. $prcid = \text{PMINT}$, but the interface id is unknown, or if the service $(iid, prcid)$ is unknown, then the message (hn, ARG) is returned.
- If the application is already registered as a server for a different interface, then the message (hn, PMDUP) is returned.
- If the interface is already registered by a different application, then the message $(hn, \text{PMOTHER})$ is returned.
- If none of the previous cases apply and the application only wants to register for the interface, then it is ensured that $pmdb.serv(iid)$ points to the handle of the calling application. Furthermore, a success message is returned.
- If none of the previous cases apply and the application wants to register a particular service, it is ensured that $pmdb.serv(iid)$ points to the handle of the calling application and that the service $(iid, prcid)$ is in the set of registered services. As in the previous case, a success is message returned.

This adds up to the following definition of *pmreg*:

$$pmreg(s, hn, iid, prcid) =$$

$$\begin{array}{ll}
\textbf{let} & serv = s.pmdb.serv, \\
& known = s.pmdb.known, \\
& s_1 = \begin{cases} s[\![pmdb.serv(iid) := hn]\!] & \textbf{if } serv(iid) \neq hn, \\ s & \textbf{else;} \end{cases} \\
& s_2 = s_1[\![pmdb.reg := s.pmdb.reg \cup \{(iid, prcid)\}]\!]
\end{array}$$

in

$$\begin{cases}
(s\ , [\,], [(hn, \text{ARG})]) & \textbf{if } (prcid = \text{PMINT} \wedge \nexists x.\ (iid, x) \in known) \\
& \quad \vee (prcid \neq \text{PMINT} \wedge (iid, prcid) \notin known), \\
(s\ , [\,], [(hn, \text{PMDUP})]) & \textbf{else if } \exists x \neq iid.\ serv(x) = hn, \\
(s\ , [\,], [(hn, \text{PMOTHER})]) & \textbf{else if } serv(iid) \neq hn \wedge serv(iid) \neq \varepsilon, \\
(s_1, [\,], [(hn, \text{SUCC})]) & \textbf{else if } prcid = \text{PMINT}, \\
(s_2, [\,], [(hn, \text{SUCC})]) & \textbf{else.}
\end{cases}$$

4.3.5.2 Lookup a Service

The library Libsos implements the following call that allows a user application to look up the handle of a service-providing server:

```
sc_pm_lookup( unsigned int iid, unsigned int prcid,
              unsigned int* handle).
```

In the SOS implementation, `sc_pm_lookup` is handled by `sos_pm_lookup`. If there is no error and there exists a server providing the service (`iid`, `prcid`), then a handle for that server is returned.

In the specification, we add PMLOOKUP iid_t_ε $prcid\_t_\varepsilon$, as an abstract representation of `sc_pm_lookup`, to the input alphabet Ω_{sc}:

$$\Omega_{\text{sc}} \supset \{\text{PMLOOKUP } iid\ prcid \mid iid \in iid\_t_\varepsilon \wedge prcid \in prcid\_t_\varepsilon\},$$

and SUCC-PMLOOKUP hn_t as well as PMNOTREG, as abstract representations of possible results, to the output alphabet Σ_{sc}:

$$\Sigma_{\text{sc}} \supset \{\text{SUCC-PMLOOKUP } hn, \text{PMNOTREG} \mid hn \in hn\_t\}.$$

The behavior of `sos_pm_lookup` is described by the function *pmlookup*. This function takes, as call-specific arguments, the interface id and the procedure id of the service in question:

$$pmlookup \in \mathcal{S} \times hn\_t \times iid\_t_\varepsilon \times prcid\_t_\varepsilon \to \mathcal{S} \times \Omega^* \times (hn\_t \times \Sigma_{\text{sc}})^*.$$

For *pmlookup*$(s, hn, iid, prcid)$, the following cases are considered:

- If the service $(iid, prcid)$ is unknown, then the message (hn, ARG) is returned.
- If there exists no application that has registered for this service, then the message $(hn, \text{PMNOTREG})$ is returned.
- If there exists an application that has registered for this service, then a success message, including the handle hn_2 of the service provider, is returned.

This adds up to the following definition of *pmlookup*:

$$pmlookup(s, hn, iid, prcid) =$$

$$\textbf{let} \quad hn_2 = s.pmdb.serv(iid)$$

$$\textbf{in} \begin{cases} (s, [\,], [(hn, \text{ARG})]) & \textbf{if } (iid, prcid) \notin s.pmdb.known, \\ (s, [\,], [(hn, \text{PMNOTREG})]) & \textbf{else if } (iid, prcid) \notin pmdb.reg, \\ (s, [\,], [(hn, \text{SUCC-PMLOOKUP } hn_2])) & \textbf{else.} \end{cases}$$

4.3.5.3 Unregister a Service

The library Libsos implements the following call that allows a user application to unregister a single service or a whole interface:

```
int sc_pm_unreg(unsigned int iid, unsigned int prcid).
```

In the SOS implementation, `sc_pm_unreg` is handled by `sos_pm_unreg`. If there is no error, then, after the call is handled, the caller is no longer registered to provide the service (`iid`, `prcid`) or no longer registered to serve the interface at all. As for `sc_pm_reg`, the exact behavior depends on the value of `prcid`.

In the specification, we add PMUNREG iid_t_ε $prcid\_t_\varepsilon \cup \{\text{PMINT}\}$, as an abstract representation of `sc_pm_reg`, to the input alphabet Ω_{sc}:

$$\Omega_{\text{sc}} \supset \{\text{PMUNREG } iid \; prcid \mid iid \in iid\_t_\varepsilon \wedge prcid \in prcid\_t_\varepsilon \cup \{\text{PMINT}\}\}.$$

The behavior of `sos_pm_unreg` is described by the function *pmunreg*. This function takes, as call-specific arguments, the interface id and the procedure id:

$$pmunreg \in \mathcal{S} \times hn\_t \times iid\_t_\varepsilon \times prcid\_t_\varepsilon \cup \{\text{PMINT}\} \\ \to \mathcal{S} \times \Omega^* \times (hn\_t \times \Sigma_{\text{sc}})^*.$$

For *pmunreg*$(s, hn, iid, prcid)$, the following cases are considered:

- If the calling application tries to unregister an interface or service it is not registered for, then the message (hn, ARG) is returned.
- If the previous case does not apply and the calling application only wants to unregister the service $(iid, prcid)$, then this service is removed from the set of registered services, and a success message returned.
- If none of the precious cases apply, then the interface iid is entirely unregistered and a success message returned. Here, "entirely unregistered" means, $pmdb.serv(iid)$ is set to ε and all services (x, y), where $x = iid$, are removed from the set of registered services.

This adds up to the following definition of *pmunreg*:

$$pmunreg(s, hn, iid, prcid) =$$

$$\textbf{let} \quad s_1 = s[\![pmdb.reg := s.pmdb.reg \setminus \{(iid, prcid)\}]\!];$$
$$s_2 = s \left[\!\!\left[\begin{array}{ll} pmdb.serv(iid) & := \varepsilon, \\ pmdb.reg & := \{(x,y) \mid (x,y) \in s.pmdb.reg \wedge x \neq iid\} \end{array} \right]\!\!\right]$$

in

$$\begin{cases} (s\ , [\,], [(hn, \text{ARG})]) & \textbf{if } s.pmdb.serv(iid) \neq hn \\ & \quad \vee (prcid \neq \text{PMINT} \wedge (iid, prcid) \notin s.pmdb.reg), \\ (s_1, [\,], [(hn, \text{SUCC})]) & \textbf{else if } prcid \neq \text{PMINT}, \\ (s_2, [\,], [(hn, \text{SUCC})]) & \textbf{else}. \end{cases}$$

4.3.6 Applications

In the following subsection, we will specify SOS calls that allow us to start new applications, clone existing ones, wait for child applications, and terminate applications.

4.3.6.1 Start an Application

Before specifying how a new application may be started, we need to look at the file format of (the underlying) executables.

An SOS executable contains the text segment and the data segment of the application. By the time of loading an executable, most of the data segment, i. e. the stack and the heap, is empty. In our case, the data segment is located at the end of an executable file and it is filled with 0s. In order to handle such files more efficiently, the compiler / assembler reduces the file size by clipping trailing 0s and then appending the number of 0s that were clipped, i. e. the clipp length. Now, for executing an application, the SOS first of all

loads the contents of the file. Based on the value of the clipp length and the file size, it allocates the necessary number of virtual memory pages, and copies the file contents into these pages. Since newly allocated memory pages are copy-on-write, initially mapping to a 0-filled page, the desired (complete) process image, including stack and heap, is realized. Obviously, while copying, the last word, which represents the clipp length, is omitted.

In SOS⋆, this (re-) construction of the process image is represented by the function *unpack-img*:

$$\begin{aligned} &\mathit{unpack\text{-}img} \in \mathit{word\_t}^* \to \mathit{word\_t}^* \\ &\mathit{unpack\text{-}img}(\mathit{con}) = \\ \textbf{let} \quad &\mathit{len} \quad = \mathit{length}(\mathit{con}); \\ &\mathit{clipp\text{-}len} = \mathit{con}[\mathit{len}-1] \\ \textbf{in} \quad &\mathit{take}(\mathit{len}-1, \mathit{con}) \circ 0^{\mathit{clipp\text{-}len}}. \end{aligned}$$

Here, we use $0^{\mathit{clipp\text{-}len}}$ to denote a list $l \in \mathit{word\_t}^{\mathit{clipp\text{-}len}}$, such that $l = [0, \ldots, 0]$. We will use the function *unpack-img* in the following specification of the SOS call `sc_app_exec` to (re-) construct a process image *img* from the contents of an executable file.[26]

After this preparatory work, we are now able to describe `sc_app_exec`. This call is implemented by the library Libsos. It allows a user application to start a new application:

```
int sc_app_exec( unsigned int fid, unsigned int uid,
                 unsigned int flag, unsigned int* handle).
```

In the SOS implementation, `sc_app_exec` is handled by `sos_app_exec`. If there is no error, then, after the call is handled, there is a new application, i. e. the child application executing the file `fid`. The super user has the right to assign a different application owner when starting a new application. Thus, if the calling application is owned by the super user, then the child application may be owned by `uid`, otherwise it is owned by the owner of the calling application. If the calling application has access to a terminal and `flag!=0`, then the terminal is passed to the child application. In case of success, the calling application receives a handle for the child and the full set of communication rights.

In the specification, we add AEXEC $\mathit{fid\_t}_\varepsilon$ $\mathit{uid\_t}_\varepsilon$ $\mathbb{N}_{32}$, as an abstract representation of `sc_app_exec`, to the input alphabet Ω_{sc}:

$$\Omega_{\mathrm{sc}} \supset \{\mathrm{AEXEC}\ \mathit{fid}\ \mathit{uid}\ \mathit{flag} \mid \mathit{fid} \in \mathit{fid\_t}_\varepsilon \wedge \mathit{uid} \in \mathit{uid\_t}_\varepsilon \wedge \mathit{flag} \in \mathbb{N}_{32}\}$$

[26] Note, it is the function *interpret* (Section 4.1) that maps such a (complete) process image to an instance of the application state space ($\mathcal{S}_{\mathrm{p}}$).

and SUCC-AEXEC hn_t as well as KERNEL, as abstract representations of possible results, to the output alphabet Σ_{sc}:

$$\Sigma_{\text{sc}} \supset \{\text{SUCC-AEXEC } hn, \text{KERNEL} \mid hn \in hn\_t\}.$$

The implementation of most SOS handlers relies on kernel calls. For the SOS handlers that have been specified so far, the underlying kernel calls were entirely hidden through our abstraction. For `sos_app_exec`, however, this is not possible. Here, we split the specification of `sos_app_exec` into a kernel part and an SOS part, and, in the end, combine both parts in *aexec*.

Kernel

The handler `sos_app_exec` relies on the kernel call `vc_process_create`. Inside the kernel, `vc_process_create` is handled by `process_create`. Essentially `process_create` tries to allocate a number of virtual memory pages, copies the content of some memory region (from the calling process) into these pages, and then initializes the kernel data structures. Within VAMOS⋆+C0, the behavior of `process_create` is specified by *process_create*. Unfortunately, some of the VAMOS⋆ data structures are no longer visible in SOS⋆; they are hidden by our abstraction. Thus, it is not possible to directly reuse *process_create*. However, instead of redefining *process_create* in terms of SOS⋆ data structures, which would be the 'same' as *process_create* only omitting updates of the hidden data structures, we take *aexec-kernel*, as SOS⋆ counter part of *process_create*, for granted.[27] We assume that *aexec-kernel* describes the changes of the (in SOS⋆ still visible) VAMOS⋆ data structures that result from creating a new process. Thus, if $(kds, pdb, img, kds', pdb', hn_{\text{n}}) \in$ *aexec-kernel*, then kds, pdb, and img represent the kernel data structures, the process data base, and process image; and kds', pdb', and hn_{n} represent the updated kernel data structures, the updated process data base, and the handle of the new process. If for some reason the kernel can not create the new process, then $kds' = kds$, $pdb' = pdb$, and $hn_{\text{n}} = \varepsilon$:

$$\textit{aexec-kernel} \subset kds\_t \times pdb\_t \times word\_t^* \times kds\_t \times pdb\_t \times hn\_t_\varepsilon.$$

Note that in VAMOS⋆+C0 the scheduling data base is abstracted and the scheduler replaced by some fairness property. Among other things, this means, in SOS⋆, we can no longer predict the exact process id and handle of the new process. Hence, *aexec-kernel* is a relation and not a function.

[27] In the Isabelle specification of the SOS [Bog08c] we actually construct *aexec-kernel*. There, we use the knowledge about the SOS implementation to relate SOS⋆- and VAMOS⋆+C0 states. Thus, in Isabelle we specify kernel calls, by deriving a VAMOS⋆+C0 state, applying the corresponding VAMOS⋆+C0 transition, and then (re-) constructing the SOS⋆ state (also see § 6.1.2).

SOS

The function *aexec-sos* specifies the SOS part of `sos_app_exec`:

$$\begin{aligned} \textit{aexec-sos} \in\ & \mathcal{S} \times \textit{hn\_t} \times \textit{fid\_t}_{\varepsilon} \times \textit{uid\_t}_{\varepsilon} \times \mathbb{N}_{32} \times \textit{kds\_t} \times \textit{pdb\_t} \times \textit{hn\_t}_{\varepsilon} \\ & \to \mathcal{S} \times \Omega^* \times (\textit{hn\_t} \times \Sigma_{\mathrm{sc}})^*. \end{aligned}$$

For $\textit{aexec-sos}(s, \textit{hn}, \textit{fid}, \textit{uid}, \textit{flag}, \textit{kds}', \textit{pdb}', \textit{hn}_{\mathrm{n}})$, the following cases are considered:

- If $\textit{faccess-legal}(s, \textit{hn}, \textit{fid}, \mathrm{EXEC})$ is not satisfied, then the result is computed and returned by $\textit{faccess-error}(s, \textit{hn}, \textit{fid})$.
- In SOS, we limit the number of simultaneously running applications owned by a single user. In SOS⋆, this number is denoted by $\mathrm{MAPU} \in \mathbb{N}_{32}^{+}$.

  Since applications need to be loaded into the virtual memory of the SOS, and we want to avoid out-of-memory situations while using dynamically allocated memory, the maximum size of executables is limited. In SOS⋆, this value is denoted by $\mathrm{MSOE} \in \mathbb{N}_{32}^{+}$.

  If the calling application a is owned by the super user and *uid* exists, then the new owner $\textit{uid}'_{\mathrm{n}}$ is equal to *uid*. If one of these preconditions is not satisfied, then $\textit{uid}'$ is equal to the owner of the calling application.

  Now, if the number of applications owned by $\textit{uid}'$ reaches the maximum number of applications per user, or if the size of the file *fid* exceeds the maximum size for executables, then the message $(\textit{hn}, \mathrm{LIMIT})$ is returned.
- If the kernel call fails, i. e. $\textit{hn}_{\mathrm{n}} = \varepsilon$, then the message $(\textit{hn}, \mathrm{KERNEL})$ is returned.
- If none of the previous cases apply, then the kernel data structures, the process data base, and the application data base are updated, and a success message, including the handle $\textit{hn}_{\mathrm{n}}$ of the new application, returned. Here, the updated kernel data structures and the updated process data base are already an argument to *aexec-sos*. The application data base is updated such that a' represents the updated calling application and a'_{n} represents the new application. For a'_{n}, *hn* is the handle of the parent application, its owner is $\textit{uid}'$, it has no socket references, and its parent does not wait for it. Furthermore, if the flag *flag* is different from 0, then a (potential) terminal connection is passed from a to a'_{n}.

This adds up to the following definition of *aexec-sos*:

$$\mathit{aexec\text{-}sos}(s, hn, \mathit{fid}, \mathit{uid}, \mathit{flag}, \mathit{kds}', \mathit{pdb}', hn_{\mathrm{n}}) =$$

$$
\begin{array}{lll}
\textbf{let} & a & = s.\mathit{adb}(hn); \\
& ao & = a.\mathit{owner}; \\
& \mathit{uid}' & = \begin{cases} ao & \textbf{if } ao \neq \text{SU} \vee s.\mathit{udb}(\mathit{uid}) = \varepsilon, \\ \mathit{uid} & \textbf{else}; \end{cases} \\
& \mathit{up} & = \{x \mid s.\mathit{adb}(x).\mathit{owner} = \mathit{uid}'\}; \\
& a' & = \begin{cases} a[\![\mathit{term} := \varepsilon]\!] & \textbf{if } \mathit{flag} \neq 0, \\ a & \textbf{else}; \end{cases} \\
& a'_{\mathrm{n}} & = \left[\!\!\left[\begin{array}{ll} \mathit{parent} & = hn, \\ \mathit{owner} & = \mathit{uid}', \\ \mathit{term} & = \begin{cases} a.\mathit{term} & \textbf{if } \mathit{flag} \neq 0, \\ \varepsilon & \textbf{else}, \end{cases} \\ \mathit{sockets} & = \{\,\}, \\ \mathit{exec} & = \mathit{fid}, \\ \mathit{wait} & = \text{FALSE}, \\ \mathit{read} & = \text{FALSE} \end{array} \right]\!\!\right]; \\
& s_1 & = s[\![\mathit{kds} := \mathit{kds}', \mathit{pdb} := \mathit{pdb}', \mathit{adb}(hn) := a', \mathit{adb}(hn_n) := a'_{\mathrm{n}}]\!]; \\
& m & = [(hn, \text{SUCC-AEXEC } hn_{\mathrm{n}})]
\end{array}
$$

$$
\textbf{in} \begin{cases} \mathit{faccess\text{-}error}(s, hn, \mathit{fid}) & \textbf{if } \neg \mathit{faccess\text{-}legal}(s, hn, \mathit{fid}, \text{EXEC}), \\ (s\ , [\,], [(hn, \text{LIMIT})]) & \textbf{else if } |\mathit{up}| \geq \text{MAPU} \\ & \qquad \vee\ \mathit{length}(s.\mathit{fdb}(\mathit{fid}).\mathit{con}) > \text{MSOE}, \\ (s\ , [\,], [(hn, \text{KERNEL})]) & \textbf{else if } hn_{\mathrm{n}} = \varepsilon, \\ (s_1, [\,], m) & \textbf{else}. \end{cases}
$$

Note that a file must be locked in order to execute it (see *faccess-legal* definined in § 4.3.2.4). However, after the application was started the caller may unlock the file. In this case the contents of the file may be modified, which means that the file id, displayed in the status line, is not very helpful. It is the responsibility of the user application to decide whether to keep the file locked, or to arrange file permissions in an appropriate way.

Now, we combine *aexec-kernel* and *aexec-sos* to specify the behavior of `sos_app_exec`. The relation *aexec* relates an SOS* state, the handle of a calling application, the id of the file that should be executed, the desired application owner, the flag indicating whether to pass terminal access the child; to a new SOS* state, external output, and messages that need to be sent to user

applications:

$$\mathit{aexec} \subset \mathcal{S} \times \mathit{hn\_t} \times \mathit{fid\_t}_{\varepsilon} \times \mathit{uid\_t}_{\varepsilon} \times \mathbb{Z}_{32} \times \mathcal{S} \times \Omega^* \times (\mathit{hn\_t} \times \Sigma_{\mathrm{sc}})^*.$$

Essentially, *aexec* calls the kernel to create a new process and considers the results of the kernel call while updating the SOS data structures. In other words, we (pre-) compute the changes that would result from executing the kernel call (*aexec-kernel*) and then use the updated kernel data structures within the definition of *aexec-sos*:

$$\mathit{aexec} = \left\{ \begin{array}{l} (s, \mathit{hn}, \mathit{fid}, \mathit{uid}, \mathit{flag}, s', o', m') \mid \\ (s.\mathit{kds}, s.\mathit{pdb}, \mathit{unpack\text{-}img}(s.\mathit{fdb}(\mathit{fid}).\mathit{con}), \mathit{kds}', \mathit{pdb}', \mathit{hn}_{\mathrm{n}}) \in \mathit{aexec\text{-}kernel} \\ \wedge\, (s', o', m') = \mathit{aexec\text{-}sos}(s, \mathit{hn}, \mathit{fid}, \mathit{uid}, \mathit{flag}, \mathit{kds}', \mathit{pdb}', \mathit{hn}_{\mathrm{n}}) \end{array} \right\}.$$

4.3.6.2 Fork an Application

The library Libsos implements the following call that allows a user application, to fork itself:

```
int sc_app_fork(unsigned int flag, unsigned int* handle).
```

In the SOS implementation, `sc_app_fork` is handled by `sos_app_fork`. If there is no error, then, after the call is handled, there exists a copy of the calling application. As a result of the SOS call, the calling application receives a handle for the child and the full set of communication rights ($\{\mathrm{SND}, \mathrm{REQ}, \mathrm{MULT}, \mathrm{FIN}\}$). Also, the new application receives a success message, but in this case the handle is equal to `VAMOS_HANDLE_NONE`. Except for the register holding the returned handle, the two virtual machines are identical. For the corresponding applications this is not true. For example, the associated entries in the rights data base are different and at most one of them is connected to a terminal.

In the specification, we add AFORK $\mathbb{N}_{32}$, as an abstract representation of `sc_app_fork`, to the input alphabet Ω_{sc}:

$$\Omega_{\mathrm{sc}} \supset \{\mathrm{AFORK}\ \mathit{flag} \mid \mathit{flag} \in \mathbb{N}_{32}\}$$

and SUCC-AFORK *hn_t* as abstract representations of a possible result, to the output alphabet Σ_{sc}:

$$\Sigma_{\mathrm{sc}} \supset \{\mathrm{SUCC\text{-}AFORK}\ \mathit{hn} \mid \mathit{hn} \in \mathit{hn\_t}\}.$$

As for `sos_app_exec`, it is necessary to expose the underlying kernel call. Thus, just like we did before, we split the specification of `sos_app_fork` into a kernel part and an SOS part, and, in the end, combine both parts in *afork*.

Kernel

The handler `sos_app_fork` relies on the kernel call `vc_process_clone`. Inside the kernel, `vc_process_clone` is handled by `process_clone`. Basically `process_clone` allocates virtual memory pages, copies the contents of the virtual memory of an existing process, and updates the kernel data structures. Within VAMOS*+C0, the behavior of `process_clone` is specified by *process_clone*. Just like we did for `process_create`, and for the same reasons, we assume *afork-kernel* to be given. We assume that *afork-kernel* describes the changes of the (in SOS* still visible) VAMOS* data structures that result from cloning a process. Thus, if $(kds, pdb, hn, kds', pdb', hn_\mathrm{n}) \in$ *afork-kernel*, then kds, pdb, and hn represent the kernel data structures, the process data base, and the handle of the process that should be cloned; and kds', pdb', and hn_n represent the updated kernel data structures, the updated process data base, and the handle of the new process. If, for some reason, the kernel can not create the new process, then $kds' = kds$, $pdb' = pdb$, and $hn_\mathrm{n} = \varepsilon$:

$$\textit{afork-kernel} \subset kds\_t \times pdb\_t \times hn\_t \times kds\_t \times pdb\_t \times hn\_t_\varepsilon.$$

Again, because of the abstracted scheduler, *afork-kernel* needs to be a relation.

SOS

The function *afork-sos* specifies the SOS part of `sos_app_fork`:

$$\textit{afork-sos} \in \mathcal{S} \times hn\_t \times \mathbb{N}_{32} \times kds\_t \times pdb\_t \times hn\_t_\varepsilon \to \mathcal{S} \times \Omega^* \times (hn\_t \times \Sigma_\mathrm{sc})^*.$$

For $\textit{afork-sos}(s, hn, \mathit{flag}, kds', pdb', hn_\mathrm{n})$, the following cases are considered:

- If the number of applications, owned by the owner of the calling application, reaches the maximum number of applications per user, then the message $(hn, \textsc{limit})$ is returned.
- If the kernel call fails, i. e. $hn_\mathrm{n} = \varepsilon$, then the message $(hn, \textsc{kernel})$ is returned.
- If none of the previous cases apply, then the kernel data structures, the process data base, and the application data base are updated, and two success messages returned. The first message is sent to the parent application, and it contains the handle of the child. The second message is sent to the child, and it contains the special handle HN-NONE. The updated kernel data structures and the updated process data base are already an argument to *afork-sos*. The application data base is updated such that a' represents the updated calling application and a'_n represents the new application. Here, a'_n is essentially a copy of a. However, the child's parent is set to hn, the sockets es are shared, and the parent does not wait for its child. Here, es is the subset of the parent's sockets

(references) which only contains the sockets that are part of an established or previously established connection (which ensures *inv-only-established-shared*). Furthermore, if the flag *flag* is different from 0, then a (potential) terminal connection is passed from a to a'_{n}.

This adds up to the following definition of *afork-sos*:

$$\mathit{afork\text{-}sos}(s, hn, \mathit{flag}, \mathit{kds}', \mathit{pdb}', hn_{\mathrm{n}}) =$$

let

$$\begin{aligned}
a &= s.\mathit{adb}(hn);\\
\mathit{up} &= \{x \mid s.\mathit{adb}(x).\mathit{owner} = \mathit{uid}\};\\
\mathit{es} &= \{x \mid x \in a.\mathit{sockets} \wedge s.\mathit{sdb}(x).\mathit{state} \in \{\text{ESTABLISHED}, \text{REMOTE-CLOSED}\}\};\\
a' &= \begin{cases} a[\![\mathit{term} := \varepsilon]\!] & \textbf{if } \mathit{flag} \neq 0,\\ a & \textbf{else};\end{cases}\\
a'_{\mathrm{n}} &= a\left[\!\!\left[\begin{array}{ll} \mathit{parent} & := hn,\\ \mathit{term} & := \begin{cases} a.\mathit{term} & \textbf{if } \mathit{flag} \neq 0,\\ \varepsilon & \textbf{else},\end{cases}\\ \mathit{sockets} & := \mathit{es},\\ \mathit{wait} & := \text{FALSE}\end{array}\right]\!\!\right];\\
s_1 &= s[\![\mathit{kds} := \mathit{kds}', \mathit{pdb} := \mathit{pdb}', \mathit{adb}(hn) := a', \mathit{adb}(hn_n) := a'_{\mathrm{n}}]\!];\\
m &= [(hn, \text{SUCC-AFORK } hn_{\mathrm{n}}), (hn_{\mathrm{n}}, \text{SUCC-AFORK HN-NONE})]
\end{aligned}$$

in

$$\begin{cases} (s\ , [\,], [(hn, \text{LIMIT})]) & \textbf{if } |\mathit{up}| \geq \text{MAPU},\\ (s\ , [\,], [(hn, \text{KERNEL})]) & \textbf{else if } hn_{\mathrm{n}} = \varepsilon,\\ (s_1, [\,], m) & \textbf{else}.\end{cases}$$

Now we combine *afork-kernel* and *afork-sos* to specify the behavior of the handler `sos_app_fork`. The relation *afork* relates an SOS* state, the handle of a calling application, and the flag indicating whether to pass terminal access to the child; to a new SOS* state, external output and messages that need to be sent to user applications:

$$\mathit{afork} \subset \mathcal{S} \times \mathit{hn\_t} \times \mathbb{N}_{32} \times \mathcal{S} \times \Omega^* \times (\mathit{hn\_t} \times \Sigma_{\mathrm{sc}})^*.$$

Basically, *afork* calls the kernel to clone the process and considers the results of the kernel call while updating the SOS data structures:

$$\mathit{afork} =$$

$$\left\{ \begin{array}{l} (s, hn, \mathit{flag}, s', o', m') \mid\\ (s.\mathit{kds}, s.\mathit{pdb}, hn, \mathit{kds}', \mathit{pdb}', hn_{\mathrm{n}}) \in \mathit{afork\text{-}kernel}\\ \wedge\, (s', o', m') = \mathit{afork\text{-}sos}(s, hn, \mathit{flag}, \mathit{kds}', \mathit{pdb}', hn_{\mathrm{n}}) \end{array} \right\}.$$

4.3.6.3 Wait for an Application to terminate

The library Libsos implements the following call that allows a user application, to wait for one of its children:

```
int sc_app_wait(unsigned int handle, int* ec).
```

In the SOS implementation, `sc_app_wait` is handled by `sos_app_wait`. If there is no error, then the call returns as soon as the application `hn` terminates. In this case `ec` contains the exit code.

In the specification, we add AWAIT hn_t, as an abstract representation of `sc_app_wait`, to the input alphabet Ω_{sc}:[28]

$$\Omega_{\text{sc}} \supset \{\text{AWAIT } hn \mid hn \in hn\_t\}$$

and SUCC-AWAIT $\mathbb{Z}_{32}$ as abstract representations of a possible result, to the output alphabet Σ_{sc}:

$$\Sigma_{\text{sc}} \supset \{\text{SUCC-AWAIT } ec \mid ec \in \mathbb{Z}_{32}\}.$$

The behavior of `sos_app_wait` is described by the function *await*. This function takes, as a call-specific argument, the handle of the application that should be waited for:

$$await \in \mathcal{S} \times hn\_t \times hn\_t \to \mathcal{S} \times \Omega^* \times (hn\_t \times \Sigma_{\text{sc}})^*.$$

For $await(s, hn, hn_{\text{w}})$, the following cases are considered:

- If hn_{w} is not the handle of a child application of the calling application, then the message (hn, PERM) is returned.
- If the previous case does not apply, then the child's application data base entry is updated to indicate the waiting parent. In this case, there is no immediate result. Instead, a result may be returned, if the child terminates (*aexit* defined in § 4.3.6.4).

This adds up to the following definition of *await*:

$$await(s, hn, hn_{\text{w}}) =$$

$$\begin{aligned}
&\textbf{let} && a \;\; = s.adb(hn_{\text{w}});\\
& && s_1 = s[\![adb(hn_{\text{w}}).wait := \text{TRUE}]\!]\\
&\textbf{in} && \begin{cases} (s\;\;, [\,], [(hn, \text{PERM})]) & \textbf{if } a.parent \neq hn,\\ (s_1, [\,], [\,]) & \textbf{else.} \end{cases}
\end{aligned}$$

[28] In the implementation of `sc_app_wait`, the `handle` argument is passed to the SOS using the additional-handle argument of the IPC-request call. Handles unknown to the calling application are already captured by the kernel. That means, if the SOS receives this call, we can be sure to receive a valid handle. Hence, the type of the supplied handle is hn_t and not hn_t_{ε}.

4.3.6.4 Exit an Application

Before we specify how a user application may terminate, we need to introduce two auxiliary functions.

The function *unlock-all-files* releases all file locks that are owned by a certain application, and collects the success messages for the new lock owners. For *unlock-all-files*(s, hn), the following happens. Within each recursion step, we first of all compute the set *fids* of file ids of files that are (still) locked by the application hn. Afterwards, we apply *funlock* to the smallest of these ids. Finally, we concatenate the results of the current recursion step with results from deeper recursion steps. *unlock-all-files* terminates, if there are no more files that are locked by hn:

$$\begin{array}{c} \textit{unlock-all-files} \in \mathcal{S} \times hn\_t \to \mathcal{S} \times (hn\_t \times \Sigma_{\text{sc}})^* \\ \textit{unlock-all-files}(s, hn) = \end{array}$$

$$\begin{array}{ll} \textbf{let} & \begin{array}{ll} \textit{fids} & = \{x \mid hn = s.fdb(x).lock[0]\}; \\ (s_1, x, m_1) & = \textit{funlock}(s, hn, \textit{min}(\textit{fids})); \\ (s_2, m_2) & = \textit{unlock-all-files}(s_1, hn) \end{array} \\ \textbf{in} & \begin{cases} (s\ , [\,]) & \textbf{if } \textit{fids} = \{\,\}, \\ (s_2, \textit{tail}(m_1) \circ m_2) & \textbf{else.} \end{cases} \end{array}$$

Similar to *unlock-all-files*, the function *close-all-sockets* applies *sclose* to all socket references owned by a certain application. Other than *unlock-all-files*, *close-all-sockets* collects abstract network packets that need to be sent to remote endpoints of established connections. *close-all-sockets* terminates, if all socket references that were previously owned by hn are processed:

$$\begin{array}{c} \textit{close-all-sockets} \in \mathcal{S} \times hn\_t \to \mathcal{S} \times \Omega^* \\ \textit{close-all-sockets}(s, hn) = \end{array}$$

$$\begin{array}{ll} \textbf{let} & \begin{array}{ll} (s_1, n_1, x) & = \textit{sclose}(s, hn, \textit{min}(s.adb(hn).sockets)); \\ (s_2, n_2) & = \textit{close-all-sockets}(s_1, hn) \end{array} \\ \textbf{in} & \begin{cases} (s\ , [\,]) & \textbf{if } s.adb(hn).sockets = \{\,\}, \\ (s_2, n_1 \# n_2) & \textbf{else.} \end{cases} \end{array}$$

Now, the library Libsos implements the following call that allows a user application to terminate:

```
int sc_app_exit(int ec).
```

In the SOS implementation, `sc_app_exit` is handled by `sos_app_exit`. After the call is handled, the calling application no longer exists, and a potentially waiting parent receives the exit code `ec`. Thereby, any traces of the calling application are removed, i. e. all socket references removed and exclusive connections closed, all file locks released, and registered services unregistered.

In the specification, we add AEXIT $\mathbb{Z}_{32}$, as an abstract representation of the call `sc_app_exit`, to the input alphabet Ω_{sc}:

$$\Omega_{\text{sc}} \supset \{\text{AEXIT } ec \mid ec \in \mathbb{Z}_{32}\}.$$

As before, it is necessary to expose the underlying kernel call. Thus, again, we split the specification of `sos_app_exit` into a kernel part and an SOS part, and, in the end, combine both parts in *aexit*.

Kernel

The handler `sos_app_exit` relies on the kernel call `vc_process_kill`. Inside the kernel, `vc_process_kill` is handled by `process_kill`. Basically, `process_kill`'s task is to remove a process and all its traces from the kernel data structures. Within VAMOS⋆+C0, the behavior of `process_kill` is specified by *process_kill*. At first this might seem an easy task, but killing a process has a number of drawbacks. Among other things, it is, for example, necessary to resolve pending IPC operations. Almost all kernel data structures are affected while killing a process. Thus, just like we did before, we are not going to redefine *process_kill* in terms of SOS⋆ data structures. Instead, we assume *aexit-kernel*, as SOS⋆ pendant of *process_kill*, to be given. We assume, *aexit-kernel* describes the changes of the (in SOS⋆ still visible) VAMOS⋆ data structures that result from killing a process. Thus, if $(kds, pdb, hn, kds', pdb') \in$ *aexit-kernel*, then *kds*, *pdb*, and *hn* represent the kernel data structures, the process data base, and the handle of the process that should be killed; and kds' and pdb' represent the updated kernel data structures and the updated process data base. Other than for *aexec-kernel* and *afork-kernel*, kernel errors are impossible:

$$\textit{aexit-kernel} \subset kds\_t \times pdb\_t \times hn\_t \times kds\_t \times pdb\_t.$$

SOS

The function *aexit-sos* specifies the SOS part of `sos_app_exit`:

$$\textit{aexit-sos} \in \mathcal{S} \times hn\_t \times \mathbb{N}_{32} \times kds\_t \times pdb\_t \to \mathcal{S} \times \Omega^* \times (hn\_t \times \Sigma_{\text{sc}})^*.$$

Just like the kernel part, the SOS part of `sos_app_exit` always succeeds. For $\textit{aexit-sos}(s, hn, ec, kds', pdb')$, the following cases are considered.

- First of all, all files that are locked by the calling application a are unlocked (s_1). Then, all sockets a has a reference for are closed (s_2). Then, if a is registered as service provider, then all services are unregistered

(s_3). After that, all children of a are inherited by a's parent, a potential terminal connection is passed to a's parent, and the application data base entry for a (itself) reset to ε. Finally, the kernel data structures and the process data base are updated according to the arguments kds' and pdb'.

- If a's parent waits for a to exit, then (additionally) the parent receives a success message. This message includes the error code ec.

This adds up to the following definition of *aexit-sos*:

$$\textit{aexit-sos}(s, hn, ec, kds', pdb') =$$

let

$$\begin{array}{lll}
(s_1, m) & = \textit{unlock-all-files}(s, hn); & \\
(s_2, n) & = \textit{close-all-sockets}(s_1, hn); & \\
iids & = \{x \mid s_2.pmdb.serv(x) = hn\}; & \\
(s_3, y, z) & = \begin{cases} \textit{pmunreg}(s_2, hn, \epsilon(iids), \text{PMINT}) & \textbf{if } |iids| = 1, \\ (s_2, [\,], [\,]) & \textbf{else}; \end{cases} & \\
a & = s_3.adb(hn); & \\
hn_\text{p} & = a.parent; & \\
adb' & = \lambda x \in hn_t. \begin{cases} s_3.adb(x)[\![parent := hn_\text{p}]\!] & \textbf{if } s_3.adb(x).parent = hn, \\ s_3.adb(x)[\![term := a.term]\!] & \textbf{else if } x = hn_\text{p} \wedge hn_\text{p} \neq \varepsilon \\ & \qquad \wedge\, a.term \neq \varepsilon, \\ \varepsilon & \textbf{else if } x = hn, \\ s_3.adb(x) & \textbf{else}; \end{cases} & \\
s_4 & = s_3[\![kds := kds', pdb := pdb', adb := adb']\!] &
\end{array}$$

in

$$\begin{cases} (s_4, n, m) & \textbf{if } \neg a.wait, \\ (s_4, n, m \circ [(hn_\text{p}, \text{SUCC-AWAIT } ec)]) & \textbf{else}. \end{cases}$$

Note, the calling application can not have a pending request (e. g. a request to lock a file or a request to receive keyboard input), because in this case it would not be able to call `sc_app_exit`. That means that none of the sockets associated with the calling application can be in the state ACCEPTING or CONNECTING. Hence, we can be sure that exiting an application does not violate the TCP requirement 'Absence of Timeouts'. For the same reason, we can also be sure that the calling application is not waiting for a file lock, keyboard input, or another application to terminate. That means that no additional cleanup is necessary in this respect. Additionally, note that it is possible that $hn_\text{p} = \varepsilon$. In this case, all the children of hn would become orphans. Furthermore, if hn

is connected to a virtual terminal and there is no parent to inherit it, then this terminal will never again be connected to any application.

Now we combine *aexit-kernel* and *aexit-sos* to specify the behavior of `sos_app_exit`. The relation *aexit* relates an SOS* state, the handle of a calling application, and the exit code; to a new SOS* state, external output and messages that need to be sent to user applications:

$$aexit \subset \mathcal{S} \times hn\_t \times \mathbb{Z}_{32} \times \mathcal{S} \times \Omega^* \times (hn\_t \times \Sigma_{\text{sc}})^*.$$

Basically, *aexit* calls the kernel to kill the process and then considers the results of the kernel call while updating the SOS data structures:

$$aexit = \left\{ \begin{array}{l} (s, hn, ec, s', o', m') \mid \\ (s.kds, s.pdb, hn, kds', pdb') \in aexit\text{-}kernel \\ \wedge\, (s', o', m') = aexit\text{-}sos(s, hn, ec, kds', pdb') \end{array} \right\}.$$

4.3.7 Undefined SOS Calls

For receiving SOS calls, the SOS provides an initialized IPC buffer of a fixed size. If the size of this buffer is too small to fit a particular IPC message, then the calling application receives a kernel-call error.[29] If, however, the buffer is large enough, then, after receiving the call, the content of this buffer will be 'casted' into a value of type `complex_t`:[30]

```
#define SOS_COMPLEX_BUF_SIZE 256u
typedef unsigned int sos_buffer_t[SOS_COMPLEX_BUF_SIZE];
struct complex_t_ {
  int i[3u];
  unsigned int u[10u];
  sos_buffer_t a;
};
typedef struct complex_t_ complex_t;
```

Then, in the implementation of the SOS dispatcher, the SOS interprets the value of `u[0u]` as the SOS-call number. According to this number, the arguments of the particular SOS call are extracted from the remaining fields of `i`, `u`, and `a`. Thus, the available SOS calls and SOS-call results as well as associated arguments, their types, and their positions within the arrays `i`, `u`, and `a` constitute the SOS's application (binary) interface. The specification of this interface is encoded in the alphabets Ω_{sc} and Σ_{sc}. So far, however, we have not

[29] In SOS*, these unsuccessful IPC operations are represented by (unsuccessful) kernel-call transitions.

[30] Note, the implementation constant `SOS_COMPLEX_BUF_SIZE` corresponds to CMPC in SOS*

yet described how to translate the elements of Ω_{sc} and Σ_{sc} to values of type `complex_t`, i. e. how to get from the specification to the implementation. This will be done below.

All elements of Ω_{sc} and Σ_{sc} are built from a constructor and zero or more parameters. These parameters either have the type $\mathbb{Z}_{32}$, $\mathbb{N}_{32}$, *word_t**, or *byte_t**, or they have a type that was derived from $\mathbb{Z}_{32}$ or $\mathbb{N}_{32}$. Now, in the SOS implementation, we use `int`, whenever the specification uses $\mathbb{Z}_{32}$ or one of its subtypes. Likewise, we use `unsigned int`, whenever the specification uses $\mathbb{N}_{32}$ or one of its subtypes. We use the type `sos_buffer_t` as an implementation for the SOS⋆ types *word_t** and *byte_t**. Finally, we use `unsigned int` values to implement the constructors used in Ω_{sc}, and `int` values to implement the constructors used in Σ_{sc}.

Now, the order and the type of the arguments, as they are specified by Ω_{sc}, dictate their position in the structure `complex_t`. Remember, for example,:

$$\Omega_{\text{sc}} \supset \{\text{FSEEK } \mathit{fid}\ \mathit{flag}\ \mathit{off} \mid \mathit{fid} \in \mathit{fid\_t}_{\varepsilon} \wedge \mathit{flag}, \mathit{off} \in \mathbb{Z}_{32}\}.$$

Here, FSEEK represents the SOS-call number. That means that if the SOS receives an IPC message that contains the numerical representation of FSEEK in `u[0u]`, then the SOS implementation should behave as specified by *fseek*. For that, the SOS handler `sos_file_seek` expects the values corresponding to *fid*, *flag*, and *off* in `u[1u]`, `i[0u]`, and `i[1u]`, respectively. This is because $\mathit{fid} \in \mathit{fid\_t} \cup \varepsilon$, $\mathit{fid\_t} \subset \mathbb{N}_{32}$ and $\mathit{flag}, \mathit{off} \in \mathbb{Z}_{32}$, i. e. in the implementation, *fid* is represented by a value of type `unsigned int`, and *flag* and *off* are represented by values of type `int`. Since `u[0u]` is already occupied by the representation of FSEEK, *fid* must be in `u[1u]`. Furthermore, since *flag* is mentioned before *off* and there are no other `int` values, *flag* must be in `i[0u]`, and *off* must be in `i[1u]`.

Similar to SOS calls, the elements of Σ_{sc}, specify the message layout for SOS-call results. We use the same rules, based on order and type, to compute the position of a particular value within the structure `complex_t`. Thus, in case of:

$$\Sigma_{\text{sc}} \supset \{\text{SUCC-FSEEK } \mathit{pos} \mid \mathit{pos} \in \mathbb{N}_{32}\},$$

SUCC-FSEEK represents the overall result of the SOS call. A user application can expect to find the overall result of an SOS call in `i[0u]`. As $\mathit{pos} \in \mathbb{N}_{32}$, the user application can expect to find the value, corresponding to *pos*, in `u[0u]`.

Now, the only interesting case that is left to explain is a value of type *word_t** or *byte_t**. In SOS⋆, we use these types in a way such that they simultaneously encode the contents and the length of a particular value. In the implementation, however, the type `sos_buffer_t` has a fixed length. Thus, in the implementation, we use a single `unsigned int` value to specify the number of 'useful' values in the `a`-component of the `complex_t` structure. This number is appended after all other `unsigned int` values. Thus, for example, for:

$$\Omega_{\text{sc}} \supset \{\text{SWRITE } \mathit{sid}\ \mathit{bytes} \mid \mathit{sid} \in \mathit{sid\_t}_{\varepsilon} \wedge \mathit{bytes} \in \mathit{byte\_t}^*\},$$

the length of *bytes* must be in `u[2u]`. Similarly, in the case of:

$$\Sigma_{\text{sc}} \supset \{\text{SUCC-FREAD } words\ pos \mid words \in word\_t^* \wedge pos \in \mathbb{N}_{32}\}.$$

an application can expect to find the number of words that could be read in `u[1u]`. Thus, in this case, `a[0u],\dots,a[u[1u]]` represents the contents of *words*.

Now, although it does not really matter for the specification of the SOS, it is worthwhile noting that the calls implemented in the library Libsos, also follow the above scheme. Thus, coming back to the example of `fseek`, the signature of the corresponding Libsos call is:

```
int sc_file_seek( unsigned int fid, int flag, int off,
                  *unsigned int pos).
```

Here, `flag` comes before `off` since `sos_file_seek` expects the values for `fid`, `flag`, and `off` in `i[0u]`, and `i[1u]`, respectively.

Now, knowing the message layout of IPC messages that are used to transfer SOS calls, we need to look at the case where a user application sends something but the content of `u[0u]` is not a valid SOS-call number. In SOS⋆ we represented such calls by the Ω_{sc} element UNDEFINED-SC:

$$\Omega_{\text{sc}} \ni \text{UNDEFINED-SC}.$$

The SOS receives such calls but does not handle them, i. e. no error message is returned. Thus, no handler is needed. The calling application will simply wait forever to receive a result (see § 4.3.8.1). That is, it will be 'punished' for not using the calls provided by the library Libsos but keeping the SOS busy with undefined calls.

4.3.8 Putting Together the Transition Relation

Now we want to collect all pieces and present the global transition relation Δ. For that, we first of all integrate the specification of the SOS-local transitions, i. e. we combine the phase of handling SOS calls with the receiving phase and the phase of returning results (compare with the introduction to Section 4.3). While doing that, we extend the intermediate alphabets of SOS inputs (Ω_{sc}) and SOS outputs (Σ_{sc}) to the alphabets of application outputs (Ω_{p}) and application inputs (Σ_{p}). Based on these, we specify the SOS-call dispatcher and the dispatcher for external inputs. In the end, we assemble SOS calls, the treatment of external inputs, kernel calls, and local computations of user applications.

4.3.8.1 SOS Calls

As said earlier, in the implementation, SOS calls are passed to and answered by the SOS via IPC kernel calls. In SOS⋆, we would like to hide this mechanism

and provide an abstraction, where SOS calls are not different from kernel calls. In most cases this works, but some artefacts of the underlying IPC mechanism remain.

Earlier we mentioned that IPC calls are also used to communicate handles and IPC rights. Furthermore, the kernel appends notifications to IPC messages. These additional IPC features need to be considered in the phase of receiving SOS calls and in the phase of returning results. Below, we first of all discuss the receiving phase and the phase of returning results, and then describe the SOS dispatcher.

Receiving SOS Calls

In general, the calls implemented in the library Libsos do not send (additional) handles, nor do they update IPC rights.[31] However, user applications may use IPC calls directly and, thus, construct (valid) SOS calls that affect the handle- and rights data base. Fortunately, they are unprivileged and, therefore, not allowed to modify the IPC rights of third party applications. Furthermore, the SOS knows all applications and, thus, it can not be surprised by an unknown application handle (sent via the additional handle). Hence, the only thing that could happen is that a user application modifies the IPC rights for the SOS, i. e. the rights of the SOS for calling this user application.[32] That means that all those elements of Ω_{p} that represent SOS calls have to have a field for IPC rights.

Besides handles and IPC rights, the kernel uses IPC messages to transfer kernel notifications. Every time a user process receives an IPC message, the kernel appends a notification about stolen handles, i. e. the information whether user processes, previously known to the recipient, have terminated meanwhile. This notification is simply a boolean flag that is set to `TRUE`, if there are stolen handles, and set to `FALSE`, if there are none. Stolen handles can be acknowledged using the kernel call `read_kernel_info`. After acknowledging a handle as a stolen handle, the kernel no longer flags it as stolen. If a user processes, with (unacknowledged) stolen handles, calls IPC receive, then the kernel immediately responds. Even if there is no rendezvous situation, i. e. currently there is no matching IPC send, the kernel directly delivers the stolen-handle notification. In this case, the contents of the IPC message is empty, and the message sender is the kernel. However, a handle is not considered to be stolen, if one user process (actively) kills another user processes, and thereby synchronously invalidates one of its handles. Thus, in this case, there will not be stolen-handle notification.

Now, the SOS is the only user process entitled to kill other user processes. That means that in general, the SOS does not receive stolen-handle notifica-

[31] *await*, defined in § 4.3.6.3, is the only exception. There, the additional handle is used to transmit the handle of the application to wait for.

[32] Using an IPC operation, rights may be granted but not revoked.

tions. However, if the kernel kills a user application because it miss-behaves (e. g. it triggers an internal interrupt like missalignment, illegal instruction, or segmentation fault), then the SOS will be notified. Fortunately, a user application can only trigger such an interrupt, if it's active, which means, the SOS must be waiting for input. Thus, in this case, the SOS is notified immediately, i. e. the SOS is woken up and the kernel notification delivered directly. When receiving such a notification, the SOS investigates the handle of the application that has failed and then updates its bookkeeping data structures accordingly. These updates are the same as if the killed application called `sc_app_exit`. The only difference is that in the case of `sc_app_exit` the kernel kills an application because the SOS is asking for it, whereas in the case of a stolen-handle notification the kernel has already killed an application and only informs the SOS about it. In the end, both cases result in the same updates of kernel- and SOS data structures. Thus, in SOS*, we treat such miss-behavior as if the application called `sc_app_exit(-1000)` (§ 4.3.6.4). That means that although stolen-handle notifications are appended to every IPC message, in SOS* they are not part of standard SOS calls. Instead, they only appear in the form of *aexit* calls.

Thus, in order to integrate the SOS-local transitions into the global transition system, we (only) need to consider the case where a user application modifies the IPC rights for SOS. Hence:[33]

$$\Omega_{\mathrm{p}} \supset \Omega_{\mathrm{sc}} \times \mathcal{P}(\mathit{rights\_t}).$$

Thus, the set of application outputs that are SOS calls is defined as follows:

$$\begin{aligned}
\Omega_{\mathrm{p}} \supset \{\ & (\textsc{uadd}, r),\ (\textsc{udel}\ \mathit{uid}, r),\\
& (\textsc{fcreat}\ \mathit{fid}, r),\ (\textsc{flock}\ \mathit{fid}, r),\ (\textsc{funlock}\ \mathit{fid}, r),\\
& (\textsc{ftruncate}\ \mathit{fid}\ \mathit{len}, r),\ (\textsc{funlink}\ \mathit{fid}, r),\ (\textsc{finfo}\ \mathit{fid\text{-}nil}, r),\\
& (\textsc{fwrite}\ \mathit{fid}\ \mathit{words}, r),\ (\textsc{fread}\ \mathit{fid}\ \mathit{len}, r),\\
& (\textsc{fseek}\ \mathit{fid}\ \mathit{zflag}\ \mathit{off}, r),\ (\textsc{fchmod}\ \mathit{fid}\ \mathit{fop}\ \mathit{uid}\ \mathit{zflag}, r),\\
& (\textsc{fchown}\ \mathit{fid}\ \mathit{uid}, r),\\
& (\textsc{tread}, r),\ (\textsc{twrite}\ \mathit{byte}\ \mathit{zflag}, r),\ (\textsc{tseek}\ \mathit{zflag}\ \mathit{off}, r),\\
& (\textsc{tinfo}, r),\\
& (\textsc{sopen}\ \mathit{pn}, r),\ (\textsc{slisten}\ \mathit{sid}, r),\ (\textsc{sconnect}\ \mathit{sid}\ \mathit{na}\ \mathit{pn}, r),\\
& (\textsc{saccept}\ \mathit{sid}, r),\ (\textsc{swrite}\ \mathit{sid}\ \mathit{bytes}, r),\\
& (\textsc{sread}\ \mathit{sid}\ \mathit{len}, r),\ (\textsc{sclose}\ \mathit{sid}, r),
\end{aligned}$$

[33] Note that $\Omega_{\mathrm{p}} \neq \Omega_{\mathrm{sc}} \times \mathcal{P}(\mathit{rights\_t})$ because Ω_{p} also contains kernel calls. In § 4.3.8.3, we will add kernel calls (e. g. IPC calls) to Ω_{p} and discuss the corresponding SOS* transitions.

$$
\begin{aligned}
&(\text{PMREG } \mathit{iid}\ \mathit{prcid\text{-}int}, r),\ (\text{PMLOOKUP } \mathit{iid}\ \mathit{prcid}, r),\\
&(\text{PMUNREG } \mathit{iid}\ \mathit{prcid\text{-}int}, r),\\
&(\text{AEXEC } \mathit{fid}\ \mathit{uid}\ \mathit{nflag}, r),\ (\text{AFORK } \mathit{nflag}, r),\ (\text{AWAIT } \mathit{hn}, r),\\
&(\text{AEXIT } \mathit{ec}, r),\\
&(\text{UNDEFINED-SC},\ r)\\
\mid\ & r \in \mathcal{P}(\mathit{rights\_t}) \wedge \mathit{byte} \in \mathit{byte\_t} \wedge \mathit{bytes} \in \mathit{byte\_t}^* \wedge \mathit{words} \in \mathit{word\_t}^*\\
&\wedge \mathit{fop} \in \mathit{fop\_t}_\varepsilon \wedge \mathit{uid} \in \mathit{uid\_t}_\varepsilon \wedge \mathit{fid} \in \mathit{fid\_t}_\varepsilon \wedge \mathit{fid\text{-}nil} \in \mathit{fid\_t}_\varepsilon \cup \{\text{NIL}\}\\
&\wedge \mathit{zflag}, \mathit{off}, \mathit{ec} \in \mathbb{Z}_{32} \wedge \mathit{len}, \mathit{nflag} \in \mathbb{N}_{32} \wedge \mathit{pn} \in \mathit{pn\_t}_\varepsilon \wedge \mathit{na} \in \mathit{na\_t}_\varepsilon\\
&\wedge \mathit{sid} \in \mathit{sid\_t}_\varepsilon \wedge \mathit{hn} \in \mathit{hn\_t} \wedge \mathit{iid} \in \mathit{iid\_t}_\varepsilon\\
&\wedge \mathit{prcid\text{-}int} \in \mathit{prcid\_t}_\varepsilon \cup \{\text{PMINT}\} \wedge \mathit{prcid} \in \mathit{prcid\_t}_\varepsilon
\end{aligned}
$$

}.

Returning Results of SOS Calls

So far, we have only looked at receiving SOS calls. The returning of results, however, also requires some special considerations.

When returning results to user application, the good thing is that we know exactly what the SOS does. Thus, although `sc_pm_lockup`, `sc_app_exec`, and `sc_app_fork` return handles and rights (§ 4.3.5.2,§ 4.3.6.1, and § 4.3.6.2), we can easily compute the necessary changes and update the handle data base and the rights data base as it would be done by the IPC call (*result* defined in § 4.3.8.1). That means that the form of the SOS-call results, i. e. the alphabet Σ_{sc}, does not change in this respect.

Note that if a user application did not use a Libsos call but used IPC calls to manually constructed an SOS call (see § 4.3.7), then it could be that the size of the specified receive buffer is too small to fit the result. In this case, the SOS's IPC-send operation would fail. Nevertheless, since the SOS only uses immediate send operations, the SOS remains working. In fact, currently, the SOS does not even care about the success or failure of this IPC operation. The user application, however, remains blocked, waiting for its IPC request to be answered. Since user applications are forced to use IPC requests, rather than independed send- and receive operations, while talking to the SOS, and these requests must not have a finite timeout, we can be sure that returning a result either succeeds immediately or fails ultimately.

Now, in terms of stolen handles, the situation is more difficult. User processes receive the notification about stolen handles with every IPC receive operation. That means, they also receive this notification when receiving results of SOS calls. Other than in Ω_{p}, in Σ_{p} we can not hide this notification. Thus, in order to integrate the SOS-local transitions into the global transition system, SOS-call results must be extended by a flag indicating stolen handles:

Hence:[34]

$$\Sigma_{\mathrm{p}} \supset \Sigma_{\mathrm{sc}} \times \mathbb{B}.$$

Thus, the set of application inputs that are SOS-call results is defined as follows:

$$\begin{aligned}
\Sigma_{\mathrm{p}} \supset \{\ & (\text{SUCC-UADD } \mathit{uid_t}, \mathit{sth}), \\
& (\text{SUCC-FINFO } \mathit{fid}\ \mathit{uid}\ \mathit{size}\ \mathit{lock}\ \mathit{perm}\ \mathit{fid\text{-}nil}, \mathit{sth}), \\
& (\text{SUCC-FSEEK } \mathit{pos}, \mathit{sth}), (\text{SUCC-FWRITE } \mathit{pos}\ \mathit{size}, \mathit{sth}), \\
& (\text{SUCC-FREAD } \mathit{words}\ \mathit{pos}, \mathit{sth}), \\
& (\text{SUCC-TREAD } \mathit{byte}, \mathit{sth}), (\text{SUCC-TSEEK } \mathit{pos}, \mathit{sth}), \\
& (\text{SUCC-TINFO } \mathit{width}\ \mathit{height}\ \mathit{pos}, \mathit{sth}), \\
& (\text{SUCC-SOPEN } \mathit{sid}, \mathit{sth}), (\text{SUCC-SACCEPT } \mathit{sid}\ \mathit{na}\ \mathit{pn}, \mathit{sth}), \\
& (\text{SUCC-SREAD } \mathit{bytes}, \mathit{sth}), \\
& (\text{SUCC-PMLOOKUP } \mathit{hn}, \mathit{sth}), \\
& (\text{SUCC-AEXEC } \mathit{hn}, \mathit{sth}), (\text{SUCC-AFORK } \mathit{hn}, \mathit{sth}), \\
& (\text{SUCC-AWAIT } \mathit{ec}, \mathit{sth}), \\
& (\text{SUCC}, \mathit{sth}) \\
& (\text{ARG}, \mathit{sth}), (\text{PERM}, \mathit{sth}), (\text{LIMIT}, \mathit{sth}), (\text{LOCK}, \mathit{sth}), (\text{SOCK}, \mathit{sth}), \\
& (\text{KERNEL}, \mathit{sth}), (\text{PMDUP}, \mathit{sth}), (\text{PMOTHER}, \mathit{sth}), \\
& (\text{PMNOTREG}, \mathit{sth}) \\
\mid\ & \mathit{sth}, \mathit{lock} \in \mathbb{B} \wedge \mathit{byte} \in \mathit{byte_t} \wedge \mathit{bytes} \in \mathit{byte_t}^* \wedge \mathit{perm} \in \mathcal{P}(\mathit{fop_t}) \\
& \wedge \mathit{words} \in \mathit{word_t}^* \wedge \mathit{uid} \in \mathit{uid_t} \wedge \mathit{pn} \in \mathit{pn_t} \wedge \mathit{na} \in \mathit{na_t} \\
& \wedge \mathit{ec} \in \mathbb{Z}_{32} \wedge \mathit{sid} \in \mathit{sid_t} \wedge \mathit{fid} \in \mathit{fid_t} \wedge \mathit{fid\text{-}nil} \in \mathit{fid_t} \cup \{\text{NIL}\} \\
& \wedge \mathit{pos}, \mathit{with}, \mathit{height}, \mathit{size}, \mathit{len}, \mathit{nflag} \in \mathbb{N}_{32} \wedge \mathit{hn} \in \mathit{hn_t} \\
\}.&
\end{aligned}$$

SOS-Call Dispatcher

Up to now, we have defined the SOS-related alphabets of application outputs and application inputs. Other than Ω_{sc} and Σ_{sc}, the alphabets Σ_{p} and Ω_{p} also concern the underlying IPC mechanism. Before we can look at actual transitions, we still need a number of auxiliary predicates.

Remember that $\omega_{\mathrm{p}} \in \mathcal{S}_{\mathrm{p}} \to \Omega_{\mathrm{p}} \cup \{\varepsilon\}$ computes the output of a user application based on its state. If, for some (existing) user application, this output is different from ε, then the user application either wants to call the system, or the system call of this application is currently being handled. Furthermore, this system call is an SOS call, rather than a kernel call, if the

[34] As for SOS calls, $\Sigma_{\mathrm{p}} \neq \Sigma_{\mathrm{sc}} \times \mathbb{B}$ because Σ_{p} also contains kernel call results § 4.3.8.3.

output is an element of $\Omega_{\text{sc}} \times \mathcal{P}(\mathit{rights\_t})$. Now, if it is an SOS call, it could be that:

- the call was not yet received by the SOS,
- the call was received but could not be answered immediately, or
- the call was received and handled but returning the result failed.

For now, we are only interested in the first case. Since we know that user applications must use IPC-request operations with infinite timeouts, we can be sure that if an SOS call was (already) received, then there must be a corresponding entry in the wait data base. Vice versa, for a new SOS call, the application output must be an element of $\Omega_{\text{sc}} \times \mathcal{P}(\mathit{rights\_t})$ and there cannot be a corresponding entry in the wait data base. In SOS⋆, the predicate $\mathit{new\text{-}soscall}(s, hn)$ is satisfied, if the output of the application hn indicates a new SOS call:

$$\mathit{new\text{-}soscall} \in \mathcal{S} \times \mathit{hn\_t} \to \mathbb{B}$$

$$\mathit{new\text{-}soscall}(s, hn) \equiv$$

let $$\mathit{pid} = s.\mathit{kds}.\mathit{hdb}(\text{OSPID}, hn)$$

in $$\omega_{\text{p}}(s.\mathit{pdb}(\mathit{pid})) \in \Omega_{\text{sc}} \times \mathcal{P}(\mathit{rights\_t}) \wedge \neg s.\mathit{kds}.\mathit{wdb}(\mathit{pid}).$$

If there is an application that wants the SOS to perform an SOS call, then the SOS needs to receive and handle this call. From what was said earlier, it is clear that, in terms of the SOS⋆ state space, receiving a call is nothing more than updating the rights data base and adding an appropriate entry to the wait data base. Thus, the function $\mathit{receive}(s, hn)$ receives an SOS call from the application hn, updates the rights- and wait data base, and returns the updated state space and the call that was received:

$$\mathit{receive} \in \mathcal{S} \times \mathit{hn\_t} \to \mathcal{S} \times \Omega_{\text{sc}}$$

$$\mathit{receive}(s, hn) =$$

let
$$\begin{aligned} \mathit{pid} &= s.\mathit{kds}.\mathit{hdb}(\text{OSPID}, hn); \\ (c, r) &= \omega_{\text{p}}(s.\mathit{pdb}(\mathit{pid})); \\ s_1 &= s \left[\!\!\left[\begin{array}{ll} \mathit{kds}.\mathit{rdb}(\text{OSPID}, \mathit{pid}) & := s.\mathit{kds}.\mathit{rdb}(\text{OSPID}, \mathit{pid}) \cup r, \\ \mathit{kds}.\mathit{wdb}(\mathit{pid}) & := \text{TRUE} \end{array} \right]\!\!\right] \end{aligned}$$

in $$(s_1, c).$$

Note that *receive* assumes that *new-soscall* is satisfied.

Most of the SOS handlers return a list of messages that need to be send to user applications. In order to specify complete transitions we still need

to describe how the state space changes while returning these results. The function *result* returns a single message to one of the user applications:

$$\mathit{result} \in \mathcal{S} \times \mathit{hn\_t} \times \Omega_{\mathrm{sc}} \to \mathcal{S}.$$

For $\mathit{result}(s, hn, o)$, the following cases are considered:

- Earlier, we described the situation where returning a result fails because the application's receive buffer is too small. Now, assume there is a predicate $\mathit{result\text{-}fit}(s, hn, o)$ that is only satisfied, if the application hn has provided a receive buffer large enough to fit the result o. Then, returning a result fails, if *result-fit* is not satisfied. In this case, the state s remains unchanged.
- If the receive buffer is large enough, then returning a result will be successful. In this case, the results, including the flag about stolen handles, are applied to the application's state and the wait data base is updated.
- If the result of the SOS call contains a handle, as is true in case of SUCC-PMLOOKUP, SUCC-AEXEC, and SUCC-AFORK, and this handle is not the special handle HN-NONE, then we also need to update the rights data base and the handle data base.[35] In such case, it is necessary to translate the additional handle hn_{n} from a handle valid for the SOS into a handle valid for the user application. Currently, the handle data base is, except for special handles, the identity function hn_{n}. Thus, this translation is simple.

This adds up to the following definition of *result*:

$$\mathit{result}(s, hn, o) =$$

$$\begin{array}{lll}
\textbf{let} & \mathit{pid} & = s.\mathit{kds}.\mathit{hdb}(\mathrm{OSPID}, hn); \\
 & (x\ hn_{\mathrm{n}}) = & \begin{cases} o & \textbf{if} \quad (\ \ o = \text{SUCC-AEXEC}\ hn_{\mathrm{n}} \\ & \qquad\quad \vee\ o = \text{SUCC-AFORK}\ hn_{\mathrm{n}} \\ & \qquad\quad \vee\ o = \text{SUCC-PMLOOKUP}\ hn_{\mathrm{n}}) \\ & \qquad \wedge\ hn_{\mathrm{n}} \neq \text{HN-NONE}, \\ (\varepsilon, \varepsilon) & \textbf{else}; \end{cases} \\
 & \mathit{pid}_{\mathrm{n}} & = s.\mathit{kds}.\mathit{hdb}(\mathrm{OSPID}, hn_{\mathrm{n}}); \\
 & \mathit{sth} & = (s.\mathit{kds}.\mathit{sthdb}(\mathit{pid}) \neq \{\,\});
\end{array}$$

[35] In SOS, only `sos_app_fork` might return a special handle. If `sos_app_fork` returns a special handle, then it is HN-NONE. However, for HN-NONE we do not need to update the handle data base or rights data base, nor do we need to translate it.

$$s_1 = s \left[\!\!\left[\begin{array}{ll} kds.wdb(pid) & := \text{FALSE}, \\ pdb(pid) & := \delta_{\text{p}}(s.pdb(pid), (o, sth)) \end{array} \right]\!\!\right];$$

$$s_2 = s_1 \left[\!\!\left[\begin{array}{l} kds.hdb(pid, hn_{\text{n}}) := pid_{\text{n}}, \\ kds.rdb(pid, pid_{\text{n}}) := \{\text{SND}, \text{REQ}, \text{MULT}, \text{FIN}\} \end{array} \right]\!\!\right]$$

in

$$\begin{cases} s & \textbf{if } \neg \mathit{result\text{-}fit}(s, hn, o), \\ s_1 & \textbf{else if } x = \varepsilon, \\ s_2 & \textbf{else.} \end{cases}$$

SOS handlers return lists of messages. The function *results* recursively calls *result* to send all the messages of a list, and updates the state space appropriately:

$$\mathit{results} \in \mathcal{S} \times (hn\_t \times \Omega_{\text{sc}})^* \to \mathcal{S}$$

$$\mathit{results}(s, l) =$$

$$\begin{cases} s & \textbf{if } l = [\,], \\ \mathit{results}(\mathit{result}(s, l[0]), \mathit{tail}(l)) & \textbf{else.} \end{cases}$$

Now, we have everything in place to describe the transition relation for SOS calls. If there exists a new SOS call sc from the user application hn, then this call is received, dispatched to the appropriate handler, and the result messages m are returned to user applications. Doing that, the next state s' and the outputs o' are computed:

$$\begin{aligned} \Delta \supset \{\, (s, \varepsilon, s', o') \mid \exists hn \in hn\_t. \\ & \mathit{new\text{-}soscall}(s, hn) \\ & \wedge (s_1, sc) = \mathit{receive}(s, hn), \\ & \wedge (\quad (sc = \text{UADD} \wedge (s_2, o', m) = \mathit{uadd}(s_1, hn)) \\ & \quad \vee (sc = \text{UDEL } \mathit{uid} \wedge (s_2, o', m) = \mathit{udel}(s_1, hn, \mathit{uid})) \\ & \quad \vee (sc = \text{FCREAT } \mathit{fid} \wedge (s_2, o', m) = \mathit{fcreat}(s_1, hn, \mathit{fid})) \\ & \quad \vee (sc = \text{FLOCK } \mathit{fid} \wedge (s_2, o', m) = \mathit{flock}(s_1, hn, \mathit{fid})) \\ & \quad \vee (sc = \text{FUNLOCK } \mathit{fid} \wedge (s_2, o', m) = \mathit{funlock}(s_1, hn, \mathit{fid})) \\ & \quad \vee (sc = \text{FTRUNCATE } \mathit{fid}\ \mathit{len} \wedge (s_2, o', m) = \mathit{ftruncate}(s_1, hn, \mathit{fid}, \mathit{len})) \\ & \quad \vee (sc = \text{FUNLINK } \mathit{fid} \wedge (s_2, o', m) = \mathit{funlink}(s_1, hn, \mathit{fid})) \\ & \quad \vee (sc = \text{FINFO } \mathit{fid} \wedge (s_2, o', m) = \mathit{finfo}(s_1, hn, \mathit{fid})) \\ & \quad \vee (sc = \text{FWRITE } \mathit{fid}\ \mathit{words} \wedge (s_2, o', m) = \mathit{fwrite}(s_1, hn, \mathit{fid}, \mathit{words})) \\ & \quad \vee (sc = \text{FREAD } \mathit{fid}\ \mathit{len} \wedge (s_2, o', m) = \mathit{fread}(s_1, hn, \mathit{fid}, \mathit{len})) \\ & \quad \vee (sc = \text{FSEEK } \mathit{fid}\ \mathit{flag}\ \mathit{off} \wedge (s_2, o', m) = \mathit{fseek}(s_1, hn, \mathit{fid}, \mathit{flag}, \mathit{off})) \\ & \quad \vee (sc = \text{FCHMOD } \mathit{fid}\ \mathit{fop}\ \mathit{uid}\ \mathit{flag} \wedge (s_2, o', m) = \mathit{fchmod}(s_1, hn, \mathit{fid}, \mathit{fop}, \mathit{uid}, \mathit{flag})) \\ & \quad \vee (sc = \text{FCHOWN } \mathit{fid}\ \mathit{uid} \wedge (s_2, o', m) = \mathit{fchown}(s_1, hn, \mathit{fid}, \mathit{uid})) \end{aligned}$$

$$
\begin{aligned}
&\vee\ (sc = \text{TREAD} \wedge (s_2, o', m) = \mathit{tread}(s_1, hn))\\
&\vee\ (sc = \text{TWRITE}\ \mathit{byte}\ \mathit{flag} \wedge (s_2, o', m) = \mathit{twrite}(s_1, hn, \mathit{byte}, \mathit{flag}))\\
&\vee\ (sc = \text{TSEEK}\ \mathit{flag}\ \mathit{off} \wedge (s_2, o', m) = \mathit{tseek}(s_1, hn, \mathit{flag}, \mathit{off}))\\
&\vee\ (sc = \text{TINFO} \wedge (s_2, o', m) = \mathit{tinfo}(s_1, hn))\\
&\vee\ (sc = \text{SOPEN}\ pn \wedge (s_2, o', m) = \mathit{sopen}(s_1, hn, pn))\\
&\vee\ (sc = \text{SLISTEN}\ \mathit{sid} \wedge (s_2, o', m) = \mathit{slisten}(s_1, hn, \mathit{sid}))\\
&\vee\ (sc = \text{SCONNECT}\ \mathit{sid}\ na\ pn \wedge (s_2, o', m) = \mathit{sconnect}(s_1, hn, \mathit{sid}, na, pn))\\
&\vee\ (sc = \text{SACCEPT}\ \mathit{sid} \wedge (s_2, o', m) = \mathit{saccept}(s_1, hn, \mathit{sid}))\\
&\vee\ (sc = \text{SWRITE}\ \mathit{sid}\ \mathit{bytes} \wedge (s_2, o', m) = \mathit{swrite}(s_1, hn, \mathit{sid}, \mathit{bytes}))\\
&\vee\ (sc = \text{SREAD}\ \mathit{sid}\ \mathit{len} \wedge (s_2, o', m) = \mathit{sread}(s_1, hn, \mathit{sid}, \mathit{len}))\\
&\vee\ (sc = \text{SCLOSE}\ \mathit{sid} \wedge (s_2, o', m) = \mathit{sclose}(s_1, hn, \mathit{sid}))\\
&\vee\ (sc = \text{PMREG}\ \mathit{iid}\ \mathit{prcid} \wedge (s_2, o', m) = \mathit{pmreg}(s_1, hn, \mathit{iid}, \mathit{prcid}))\\
&\vee\ (sc = \text{PMLOOKUP}\ \mathit{iid}\ \mathit{prcid} \wedge (s_2, o', m) = \mathit{pmlookup}(s_1, hn, \mathit{iid}, \mathit{prcid}))\\
&\vee\ (sc = \text{PMUNREG}\ \mathit{iid}\ \mathit{prcid} \wedge (s_2, o', m) = \mathit{pmunreg}(s_1, hn, \mathit{iid}, \mathit{prcid}))\\
&\vee\ (sc = \text{AEXEC}\ \mathit{fid}\ \mathit{uid}\ \mathit{flag} \wedge (s_1, hn, \mathit{fid}, \mathit{uid}, \mathit{flag}, s_2, o', m) \in \mathit{aexec})\\
&\vee\ (sc = \text{AFORK}\ \mathit{flag} \wedge (s_1, hn, \mathit{flag}, s_2, o', m) \in \mathit{afork})\\
&\vee\ (sc = \text{AWAIT}\ hn_{\text{w}} \wedge (s_2, o', m) = \mathit{await}(s_1, hn, hn_{\text{w}}))\\
&\vee\ (sc = \text{AEXIT}\ ec \wedge (s_1, hn, ec, s_2, o', m) \in \mathit{aexit})\\
&\vee\ (sc = \text{UNDEFINED-SC} \wedge (s_2, o', m) = s_1, [\,], [\,])\\
&)\\
&\wedge\ s' = \mathit{results}(s_2, m)\}.
\end{aligned}
$$

Note that the user application that is served is chosen nondeterministically. However, fairness between user applications is guaranteed by the scheduler which is, in SOS*, expressed through SOS* runs (*app-fairness* defined in § 4.5.1).

4.3.8.2 External Inputs

Now, we describe SOS* transitions due to external inputs.

Just like SOS calls, external inputs are received by the SOS by means of IPC. However, for external inputs, nothing of this underlying mechanism is exposed. That means, the alphabets Σ and Ω do not need to be adapted. They remain as they were introduced while defining the individual handlers (§ 4.3.1 – § 4.3.7):

$$\Sigma = \text{KBD}\ \mathit{na\_t}\ \mathit{byte\_t} \mid \text{NET}\ \mathit{np\_t}$$

and

$$\Omega = \text{NET}\ \mathit{np\_t}.$$

Now, if there exists some input i, then this input is dispatched to the appropriate handler, and possible result messages m are returned to user applications. Doing that, the next state s' and the outputs o' are computed:

$$
\begin{aligned}
\Delta \supset \{\, (s, i, s', o') \mid \\
\quad (\quad & (i = \text{KBD } \mathit{dna}\ \mathit{byte} \\
& \quad \wedge ((s_1, o', m) = (s, [\,], [\,]) \vee (s_1, o', m) = \mathit{event\text{-}tkbd}(s, \mathit{dna}, \mathit{byte}))) \\
& \vee (i = \text{NET REQ } \mathit{sna}\ \mathit{spn}\ \mathit{dna}\ \mathit{dpn} \\
& \quad \wedge (s_1, o', m) = \mathit{event\text{-}sreq}(s, \mathit{sna}, \mathit{spn}, \mathit{dna}, \mathit{dpn})) \\
& \vee (i = \text{NET READY } \mathit{sna}\ \mathit{spn}\ \mathit{dna}\ \mathit{dpn} \\
& \quad \wedge (s_1, o', m) = \mathit{event\text{-}sready}(s, \mathit{sna}, \mathit{spn}, \mathit{dna}, \mathit{dpn})) \\
& \vee (i = \text{NET DATA } \mathit{sna}\ \mathit{spn}\ \mathit{dna}\ \mathit{dpn}\ \mathit{bytes} \\
& \quad \wedge (s_1, o', m) = \mathit{event\text{-}sdata}(s, \mathit{sna}, \mathit{spn}, \mathit{dna}, \mathit{dpn}, \mathit{bytes})) \\
& \vee (i = \text{NET ACK } \mathit{sna}\ \mathit{spn}\ \mathit{dna}\ \mathit{dpn}\ \mathit{ack} \\
& \quad \wedge (s_1, o', m) = \mathit{event\text{-}sack}(s, \mathit{sna}, \mathit{spn}, \mathit{dna}, \mathit{dpn}, \mathit{ack})) \\
& \vee (i = \text{NET CLOSE } \mathit{sna}\ \mathit{spn}\ \mathit{dna}\ \mathit{dpn} \\
& \quad \wedge (s_1, o', m) = \mathit{event\text{-}sclose}(s, \mathit{sna}, \mathit{spn}, \mathit{dna}, \mathit{dpn})) \\
&) \\
\wedge\ & s' = \mathit{results}(s_1, m)\}.
\end{aligned}
$$

Note, in § 4.3.3.2, we described two types of buffer overflows related to receiving keyboard input. While the software buffer overflow was already modeled in *event-tkbd*, the hardware buffer overflow is represented above by nondeterministically loosing keyboard input ($(s_1, o', m) = (s, [\,], [\,])$ in the case that $i = \text{KBD } \mathit{dna}\ \mathit{byte}$).

4.3.8.3 Kernel Calls

As said earlier, kernel calls are already specified in VAMOS⋆ and VAMOS⋆+C0. The transitions described there nicely fit into SOS⋆. Instead of redefining them, we will only extend the alphabets Ω_p and Σ_p, and simply assume there is a function *handle-kernelcall* that handles kernel calls.

Thus, in order to represent kernel calls that are still available to user applications in the presence of the SOS, we extend Ω_p in the following way:[36]

$$
\begin{aligned}
\Omega_\text{p} \supset \{ \quad & \text{SND } \mathit{hn}_\text{r}\ \mathit{rights}_\text{s}\ \mathit{msg}\ \mathit{hn}_\text{a}\ \mathit{rights}_\text{a}\ \mathit{to}_\text{s}, \\
& \text{RCV } \mathit{hn}_\text{s}\ \mathit{buf}\ \mathit{to}_\text{r},
\end{aligned}
$$

[36] The types used for the arguments *msg* and *buf* are those of memory objects and memory buffers. While the earlier is used to refer to actual data, the latter is used to describe a buffer of a certain size to store data. Both types are chosen to abstract as much as possible from the machine-internal representation and focus on the actual semantics. A detailed discussion is given in the VAMOS⋆+C0 specification.

$$
\begin{aligned}
&\text{SNDRCV } hn_{\mathrm{r}}\ rights_{\mathrm{s}}\ msg\ hn_{\mathrm{a}}\ rights_{\mathrm{a}}\ to_{\mathrm{s}}\ hn_{\mathrm{s}}\ buf\ to_{\mathrm{r}},\\
&\text{CHRIGHTS } hn_{\mathrm{sub}}\ hn_{\mathrm{obj}}\ grant\ rights,\\
&\text{KINFO},\\
&\text{PRIVILEGED, UNDEFINED-KC}\\
&\mid rights_{\mathrm{s}}, rights_{\mathrm{a}} \in \mathcal{P}(rights\_t) \cup \{\varepsilon\} \wedge to_{\mathrm{s}}, to_{\mathrm{r}} \in \mathbb{Z}_{32} \cup \text{INF}\\
&\quad \wedge grant \in \mathbb{B} \wedge hn_{\mathrm{r}}, hn_{\mathrm{a}}, hn_{\mathrm{s}}, hn_{\mathrm{sub}}, hn_{\mathrm{obj}} \in hn\_t\\
&\quad \wedge msg \in byte\_t^{*} \cup \{\text{UNDEFINED, UNAVAILABLE}\}\\
&\quad \wedge buf \in \mathbb{N}_{32} \cup \{\text{UNDEFINED, UNAVAILABLE}\}\\
\}.
\end{aligned}
$$

Briefly, these calls allow user applications to do inter-process communication (SND,RCV, and SNDRCV), change communication rights (CHRIGHTS), and retrieve information from the kernel (KINFO).[37] The kernel provides more calls, but as user applications are never privileged, only a subset of the kernel calls is available to them. Calls that require the caller to be privileged are represented in SOS* by the Ω_{p} element PRIVILEGED. Finally, calls that are unknown to the kernel are represented by the Ω_{p} element UNDEFINED-KC.

As we added kernel calls to the alphabet Ω_{p}, we also need to extend Σ_{p}. The following additional elements of Σ_{p} represent the possible results of the aforementioned kernel calls:

$$
\begin{aligned}
\Sigma_{\mathrm{p}} \supset \{\ &\text{SUCCESS},\\
&\text{SUCC-RCV } hn_{\mathrm{s}}\ reuse_{\mathrm{s}}\ rights_{\mathrm{s}}\ msg\ hn_{\mathrm{a}}\ reuse_{\mathrm{a}}\ rights_{\mathrm{a}}\ sth,\\
&\text{ERR-UNPRIVILEGED},\\
&\text{ERR-INVALID-ARGS},\\
&\text{ERR-INVALID-HANDLE},\\
&\text{ERR-INVALID-SUBJ-HANDL, ERR-INVALID-OBJ-HANDLE},\\
&\text{ERR-SND-INVALID-HANDLE, ERR-RCV-INVALID-HANDLE},\\
&\text{ERR-SND-TIMEOUT, ERR-RCV-TIMEOUT},\\
&\text{ERR-SND-BUFFER-OVL, ERR-RCV-BUFFER-OVL},\\
&\text{ERR-SND-SEGV, ERR-RCV-SEGV},\\
&\mid msg \in byte\_t^{*}\\
&\quad \wedge hn_{\mathrm{new}}, hn_{\mathrm{s}}, hn_{\mathrm{a}} \in hn\_t
\end{aligned}
$$

[37] Note, the presence / availability of these kernel calls (SND,RCV, and SNDRCV) make the definition of receiving SOS calls (*receive*) and returning SOS-call results (*result*) so difficult. This is because, we are facing two levels of granularity. On the one hand, we would like to hide the implementation of the Libsos calls but, on the other hand, applications may use these kernel calls to manually construct SOS calls.

$$\begin{aligned}
&\wedge\ reuse_{\mathrm{s}}, reused_{\mathrm{a}}, sth \in \mathbb{B} \\
&\wedge\ rights_{\mathrm{s}}, rights_{\mathrm{a}} \in \mathcal{P}(rights_t)
\end{aligned}$$
$$\}.$$

Before we talk about actual kernel call transitions, we introduce the auxiliary predicate *new-kernelcall*. Similarly to *new-soscall*, *new-kernelcall*(s, pid) is satisfied, if the output of the user application *pid* indicates a new kernel call:

$$new\text{-}kernelcall \in \mathcal{S} \times pid\_t \to \mathbb{B}$$

$$new\text{-}kernelcall(s, pid) \equiv$$

let $$kc = \omega_{\mathrm{p}}(s.pdb(pid))$$

in $$kc \in \Omega_{\mathrm{p}} \wedge kc \notin \Omega_{\mathrm{sc}} \times \mathcal{P}(rights\_t) \wedge \neg s.kds.wdb(pid).$$

Now, we assume *handle-kernelcall* describes the changes of the (in SOS⋆ still visible) VAMOS⋆ data structures that would result from handling the kernel call of a particular process (see Section 6.1 for a more details about *handle-kernelcall*). Thus, if $(kds, pdb, pid, kds', pdb') \in$ *handle-kernelcall*, then kds, pdb, and pid represent the kernel data structures, the process data base, and the PID of the process; and kds' and pdb' represent the updated kernel data structures and process data base. Now, we can specify kernel-call-related SOS⋆ transitions as follows. If there exists an application pid, whose output is a new kernel call, then the updates to the kernel data structures and the process data base are computed by *handle-kernelcall*, and then applied to the SOS⋆ state:

$$\begin{aligned}
\Delta \supset \{ (s, \varepsilon, s', \varepsilon) \mid \exists pid \in pid_t. \\
&new\text{-}kernelcall(s, pid) \\
&\wedge (s.kds, s.pdb, pid, kds', pdb') \in handle\text{-}kernelcall \\
&\wedge s' = s[\![kds := kds', pdb := pdb']\!]\}.
\end{aligned}$$

Note that kernel calls that are triggered by user applications do not produce any user-visible external output. Hence, the output is by default ε. Further, note that, as for SOS calls, fairness between user applications is guaranteed through SOS⋆ runs.

4.3.8.4 Local Computation

Last but not least, there are SOS⋆ transitions that represent local computations of user applications.

If there exists an application *pid*, such that its output is equal to ε, then this application may do a local step:

$$\begin{aligned}\Delta \supset \{\, (s, \varepsilon, s', \varepsilon) \mid \exists pid \in pid\_t.\\ \varepsilon = \omega_{\mathrm{p}}(s.pdb(pid))\\ \wedge\, s' = s[\![pdb(pid) := \delta_{\mathrm{p}}(s.pdb(pid), \varepsilon)]\!]\}.\end{aligned}$$

Again, no user-visible external output is produced, and fairness between user applications is guaranteed through SOS* runs.

4.4 Initial States

The set $\mathcal{S}^0$ of initial states is almost as large as $\mathcal{S}$. Any configuration s that does not contain any socket in an ACCEPTING, CONNECTING, ESTABLISHED, or REMOTE-CLOSED state, qualifies as an initial state:

$$\begin{aligned}\mathcal{S}^0 = \{s \mid s \in \mathcal{S}\\ \wedge\, (\forall x \in sid\_t.\ s.sdb(x) = \varepsilon \vee s.sdb(x).state \in \{\text{BOUND}, \text{LISTEN}\})\}.\end{aligned}$$

4.5 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 $r \in (\mathcal{S} \times \Sigma)^*$, with $r[0][0] \in \mathcal{S}^0$ and $\forall n \in \mathbb{N}.\ (s^n, i^n) = r[n] \wedge (s^{n+1}, i^{n+1}) = r[n+1]$ such that $\exists o \in \Omega_\varepsilon$ and $(s^n, i^n, s^{n+1}, o) \in \Delta$. We define a (valid) SOS* run to be a run r such that $\mathcal{R}(r)$, i. e. a sequence of states and inputs that is 'covered' by the SOS* transition relation, and that satisfies the predicate $\mathcal{R}$.

4.5.1 Fairness Between User Applications

Fairness between user applications is an important property. However, while introducing C0 machines as user processes, it was necessary to abstract the scheduler (Section 3.5). 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. There are the following exceptions:

- an application might wait infinitely long for another application to match an IPC operation,

- an application does not progress if it terminates, i. e. if it successfully calls AEXIT (again, also matching the situation where the kernel kills the application (see § 4.3.6.4 and § 4.3.8.1)),
- an SOS call may not be answered, because a requested resource, such as a file lock, is never available, or a
- the results of an SOS call can not be returned to an application, because the provided receive buffer is too small.

Processes that might not make any progress because of one of the aforementioned reasons can be discovered using the predicate *possibly-no-progress*. The predicate *possibly-no-progress*(s, pid) is satisfied, if the output o, with $o = \omega_\mathrm{p}(s.pdb(pid))$, reveals an IPC call with infinite timeout (INF) or the SOS call (AEXIT, ...) or if the output matches some SOS call and there is an entry in the wait data base. Note that the last case matches the 3rd and 4th exception.

$$\textit{possibly-no-progress} \in \mathcal{S} \times \textit{pid\_t} \to \mathbb{B}$$

$$\textit{possibly-no-progress}(s, pid) \equiv$$

$$\begin{aligned}
&\exists o \in \Omega_\mathrm{p}.\ o = \omega_\mathrm{p}(s.pdb(pid)) \\
&\wedge (\quad (o = \text{SNDRCV}\ \ldots to_\mathrm{s}\ \ldots to_\mathrm{r}\ \wedge (to_\mathrm{s} = \text{INF} \vee to_\mathrm{r} = \text{INF})) \\
&\quad \vee (o = \text{SND}\ \ldots to \wedge to = \text{INF}) \\
&\quad \vee (o = \text{RCV}\ \ldots to \wedge to = \text{INF}) \\
&\quad \vee o = \text{AEXIT}\ \ldots \\
&\quad \vee (o \in \Omega_\mathrm{sc} \times \mathcal{P}(\textit{rights_t}) \wedge s.kds.wdb(pid))).
\end{aligned}$$

Besides identifying processes that might not make progress, we also need to characterize progress. In SOS*, we can simply assume that a process has progressed, between s and s', if s' could be the result of applying δ_p to s and some input i:[38]

$$\textit{progress} \in \mathcal{S}_\mathrm{p} \times \mathcal{S}_\mathrm{p} \to \mathbb{B}$$
$$\textit{progress}(s, s') \equiv \exists i \in \Sigma_\mathrm{p} \cup \{\varepsilon\}.\ s' = \delta_\mathrm{p}(s, i).$$

Note, depending on the particular process abstraction, it may be that $s = \delta_\mathrm{p}(s, i)$. Therefore, in *progress*, we do not require $s' \neq s$. This is sound, as in the context of scheduler fairness, it does not matter whether a process was scheduled but did not change its state or whether it was not scheduled at all. Without the state changing while progressing, the lack of fairness can not be

[38] On lower layers there exists a special case. There, one process might change the size of virtual memory available to another processes. In this case, the state of the latter would be changed without the process actually progressing. In our implementation, however, the SOS, which is the only one entitled to do that, does not make use of the feature. Hence, we do not need to further inspect the particular input.

measured anyways. Thus, even if $s = \delta_{\mathrm{p}}(s, i)$, we might as well assume the process has progressed.

Now, being able to identify processes that may not progress and processes that have progressed, we can state fairness between user applications as a predicate that must be satisfied by all valid SOS* runs. The predicate *app-fairness*(r) is satisfied, if each process progresses infinitely often or gets stuck due to one of the exceptions covered by the predicate *possibly-no-progress*:[39]

$$\begin{gathered}
\textit{app-fairness} \in (\mathcal{S} \times \Sigma)^* \to \mathbb{B} \\
\textit{app-fairness}(r) \equiv \\
\forall i \in \mathbb{N}, \textit{pid} \in \textit{pid_t}.\ r[i][0].\textit{pdb}(\textit{pid}) \neq \varepsilon \wedge \neg \textit{possibly-no-progress}(r[i][0], \textit{pid}) \\
\Longrightarrow \\
\exists j \in \mathbb{N}, j \geq i.\ \textit{progress}(r[j][0].\textit{pdb}(\textit{pid}), r[j+1][0].\textit{pdb}(\textit{pid})).
\end{gathered}$$

The proof of *app-fairness* is discussed in Section 6.2.

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

$$\begin{gathered}
\mathcal{R} \in (\mathcal{S} \times \Sigma)^* \to \mathbb{B} \\
\mathcal{R}(r) \equiv \textit{app-fairness}.
\end{gathered}$$

4.6 Summary

In the beginning of this chapter we introduced SOS* as the following parameterized transition system:

$$\begin{gathered}
\mathrm{SOS}^\star(\mathcal{S}_{\mathrm{p}}, \delta_{\mathrm{p}}, \omega_{\mathrm{p}}, \textit{vm-size}, \textit{interpret}) = \\
(\mathcal{S}, \mathcal{S}^0, \Sigma, \Omega, \Delta, \mathcal{R}, \Sigma_{\mathrm{p}}, \Omega_{\mathrm{p}}).
\end{gathered}$$

At this point, we have defined all its elements. Thus, Figure 4.9 on the following page should serve as a visual summary of what has been defined.

[39] *app-fairness* is still quite coarse grained. However, depending on the needs that arise when proving properties about user applications, one can easily diversify this predicate.

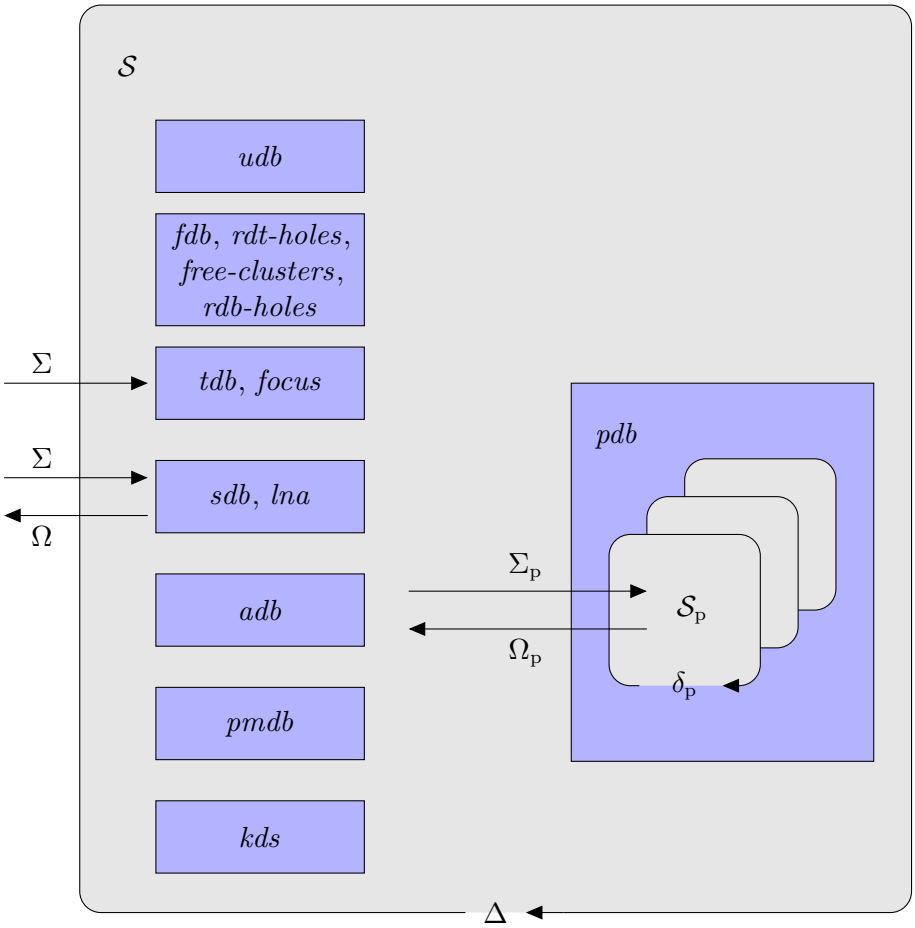

Figure 4.9: SOS*— The Big Picture. The transition relation Δ modifies the state space $\mathcal{S}$. External inputs (Σ) affect the state-space components related to virtual terminals and sockets. Communication via sockets also produces external outputs (Ω). The process data base *pdb* stores the states of the individual user processes. These user processes are modeled by them selves as self-contained I/O automata, communicating via Σ_p and Ω_p with the rest of the SOS* model.

Chapter

5

DSOS*

Contents

In Chapter 4 we spent a great deal of effort specifying a single system. But, besides specifying the properties of single systems, it is also desirable to look at properties of (such) systems in a distributed environment. Below, we will define DSOS*; a model of a distributed system containing a number of SOS* instances.

5.1 Overview

Similar to SOS*, DSOS* is defined as a transition system:

$$\text{DSOS}^\star = (\mathcal{S}_{\text{ds}}, \mathcal{S}^0_{\text{ds}}, \Sigma_{\text{ds}}, \Delta_{\text{ds}}, \mathcal{R}_{\text{ds}}).$$

5.2 Components of DSOS*

5.2.1 State Space

The state space $\mathcal{S}_{\text{ds}}$ has two components. These are: *nodes*, a set of SOS* instances and *bus*, a kind of central bus connecting the outputs and inputs of the individual nodes:

$$\mathcal{S}_{\text{ds}} = \{nodes : \mathcal{P}(\mathcal{S}), bus : \Sigma^*\}.$$

Note that we will use *bus* to store the outputs of individual nodes and later distribute these outputs as inputs to other nodes. For that, we rely on the fact

that $\Omega \subset \Sigma$. Further, note that, in order to clearly identify individual nodes, we require unique network addresses. Thus, *inv-unique-network-address* must be an invariant of DSOS*:

$$\begin{gathered}\textit{inv-unique-network-address} \in \mathcal{S}_{\text{ds}} \to \mathbb{B} \\ \textit{inv-unique-network-address}(ds) \equiv \\ \forall s_1, s_2 \in ds.nodes.\ s_1.lna = s_2.lna \implies s_1 = s_2.\end{gathered}$$

5.2.2 External Inputs

In DSOS*, the only external input that needs to be considered is keyboard input. This is because network traffic is only allowed between nodes of the system. Thus, Σ_{ds}, the set of external inputs for DSOS*, is defined as follows:

$$\Sigma_{\text{ds}} = \{\text{KBD}\ dna\ byte \mid dna \in na\_t \wedge byte \in byte\_t\}.$$

Which means, $\Sigma_{\text{ds}} \subset \Sigma$.

5.2.3 Transition Relation

Before we go on to specify the transition relation Δ_{ds}, we need to introduce an auxiliary function.

The function $\textit{strip-input}(l, na)$ returns the first element from the list l which is an input for the node with the network address na. If there exists such an element, then it will be removed from the list and both, the stripped list and the match, are returned. If the list does not contain a matching element, then the original list l and ε are returned:

$$\begin{gathered}\textit{strip-input} \in \Sigma^* \times na\_t \to \Sigma_\varepsilon \times \Sigma^* \\ \textit{strip-input}(l, na) \equiv \\ \textbf{let} \qquad (x, y) = \textit{strip-input}(\textit{tail}(l), na) \\ \textbf{in} \quad \begin{cases} (\varepsilon, [\,]) & \textbf{if } l = [\,], \\ (l[0], \textit{tail}(l)) & \textbf{else if } \exists\ byte \in byte\_t, bytes \in byte\_t^*, ack \in \mathbb{N}. \\ & \qquad l[0] = \text{KBD}\ na\ byte \\ & \qquad \vee\ l[0] = \text{NET}\ \ldots na \\ & \qquad \vee\ l[0] = \text{NET}\ \ldots na\ bytes \\ & \qquad \vee\ l[0] = \text{NET}\ \ldots na\ ack, \\ (x, l[0] \# y) & \textbf{else.} \end{cases}\end{gathered}$$

Now, the transition relation Δ_{ds} essentially allows one of the nodes of the distributed system to do a single step. For that, we append any external input

σ to the bus $ds.bus$, strip the oldest input for the system $s.lna$, apply the transition relation Δ, and append the resulting output (o) to the bus:

$$\Delta_{\text{ds}} \subset \mathcal{S}_{\text{ds}} \times \Sigma^*_{\text{ds}} \times \mathcal{S}_{\text{ds}} \times na\_t$$

$$\begin{aligned}\Delta_{\text{ds}} = \{\, (ds, \sigma, ds', na) \mid \exists s, s', i, bus', o.\ & s \in ds.nodes \\ \wedge\ & (i, bus') = \textit{strip-input}(ds.bus \circ \sigma, s.lna) \\ \wedge\ & (s, i, s', o) \in \Delta \\ \wedge\ & ds' = [\![nodes := ds.nodes \setminus \{s\} \cup \{s'\}, bus := bus' \circ o]\!] \\ \wedge\ & na = s.lna\}.\end{aligned}$$

Note that the node which is allowed to do a step is chosen nondeterministically. This node is identified by na. Similarly to SOS⋆, we formalize fairness through runs. In this case, DSOS⋆ runs. Within the formalization of fairness between nodes (§ 5.2.5) we will need na to identify the system that has progressed in a certain transition of a DSOS⋆ run.

5.2.4 Initial States

For the set $\mathcal{S}^0_{\text{ds}}$, of initial states, we require that all of the individual systems are in an initial state and there are no untreated inputs on the bus:

$$\mathcal{S}^0_{\text{ds}} = \{ds \mid ds \in \mathcal{S}_{\text{ds}} \wedge ds.bus = [\,] \wedge \forall s \in ds.nodes.\ s \in \mathcal{S}^0\}.$$

5.2.5 Runs

$\mathcal{R}_{\text{ds}}$ is a predicate that characterizes valid DSOS⋆ runs. Similarly to SOS⋆, we define a valid DSOS⋆ run to be an infinite sequence of states and inputs that is 'covered' by the DSOS⋆ transition relation and that satisfies the predicate $\mathcal{R}_{\text{ds}}$.

In DSOS⋆, we use runs to formalize fairness between nodes of the distributed system. The predicate $\mathcal{R}_{\text{ds}}$ is satisfied, if in the course of an infinite DSOS⋆ run, every node na that has something to do is chosen infinitely often. That is, if, in state i, there exists a system s, with $s.lna = na$, that could make progress, i. e. there exists a process pid such that $\neg\textit{possibly-no-progress}(s, pid)$, then there (also) exists a state $j \geq i$ where s progresses, i. e. s changes its state in the transition between j and $j + 1$:

$$\begin{gathered}\mathcal{R}_{\text{ds}} \in (\mathcal{S}_{\text{ds}} \times \Sigma^*_{\text{ds}})^* \to \mathbb{B} \\ \mathcal{R}_{\text{ds}}(r) \equiv \\ \forall i \in \mathbb{N}, na \in na\_t.\ \exists s \in r[i][0].nodes, pid \in pid\_t. \\ s.lna = na \wedge s.pdb(pid) \neq \varepsilon \wedge \neg\textit{possibly-no-progress}(s, pid) \\ \Longrightarrow \\ \exists j \in \mathbb{N}.\ j \geq i \wedge (r[j][0],\ r[j][1],\ r[j+1][0],\ na) \in \Delta_{\text{ds}}.\end{gathered}$$

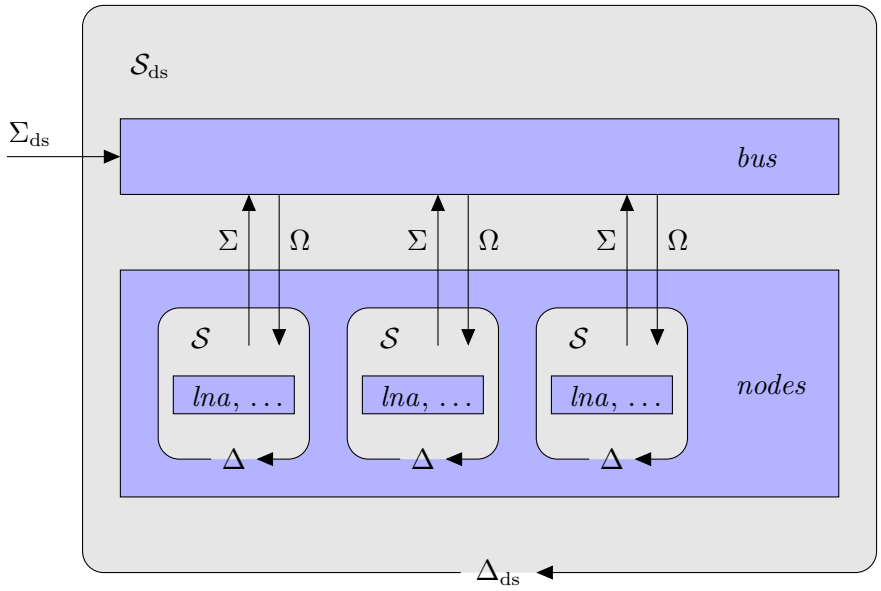

Figure 5.1: DSOS⋆—The Big Picture. The transition relation Δ_{ds} modifies the state space $\mathcal{S}_{\text{ds}}$. The only external inputs (Σ_{ds}) to consider are those from keyboards. Keyboard inputs and abstract network packages are collected and distributed by means of the *bus* component. The *nodes* component stores the states of the individual SOS⋆ instances.

5.3 Summary

In Chapter 4 we introduced SOS⋆. In the present chapter we combined several SOS⋆ instances into a distributed system, i. e. DSOS⋆. Based on Figure 4.9 on page 134, Figure 5.1 presents a visual summary of DSOS⋆.

Chapter

6

Verification

Contents

In this chapter we want to outline the most important verification obligations and sketch their proofs. Here, we will only consider the SOS* model.

6.1 Simulation

Eventually, we want to prove that the implementation is correct with respect to its specification. For that, we rely on VAMOS*+C0. We need to prove that VAMOS*+C0, instantiated by SOS, is an implementation of SOS*. That is, SOS* simulates VAMOS*+C0 plus SOS implementation.

First of all, we need to define an abstraction function that maps implementation states to specification states. Let $\mathcal{S}_{\text{vc}}$ denote the VAMOS*+C0 state space, then the abstraction function *abs* has the following signature:

$$abs \in \mathcal{S}_{\text{vc}} \to \mathcal{S}.$$

Using *abs*, we have to show that every run in the implementation has an equivalent SOS* run. Let $\mathcal{R}_{\text{vc}}$ denote the set of valid VAMOS*+C0 runs. Then, we have to prove that for every run $r_{\text{vc}} \in \mathcal{R}_{\text{vc}}$ there exists a valid SOS* run $r \in \mathcal{R}$ and a monotonous function ζ such that for all SOS* steps i there exists a corresponding number $\zeta(i)$ of VAMOS*+C0 steps such that $r[i] = abs(r_{\text{vc}}[\zeta(i)])$.

Thus, the following is the main SOS* theorem to prove:

$$\forall r_{\text{vc}} \in \mathcal{R}_{\text{vc}}$$
$$\Longrightarrow$$
$$\exists r \in \mathcal{R}.\ \forall i \in \mathbb{N}.\ r[i] = \mathit{abs}(r_{\text{vc}}[\zeta(i)])$$

In order to show this theorem, we need to prove that this statement holds for each of the four transition types, i. e. SOS calls, external inputs, kernel calls, and application-local computations. Thus, below, we inspect each of them.

6.1.1 SOS Calls and External Inputs

For SOS calls and external inputs, the proof obligations are twofold. (i) We need to show the functional correctness of the individual handlers and (ii) we need to show that handling an SOS call or external input can be modeled as an atomic step.

6.1.1.1 Functional Correctness

Functional correctness of SOS handlers boils down to proving C0 code correctness.

The difficult cases, while proving handler correctness, are those handlers that are related to external devices. This is because of the different device representations in VAMOS*+C0 and SOS*. While VAMOS*+C0 recognizes devices as a number of I/O registers and a source of interrupts, SOS* actually incorporates a set of well known devices. That means, we first of all need a hardware model for each of the incorporated devices. Then, we need to specify and prove intermediate abstractions that 'bridge' the large 'gap' between hardware model and SOS* state space:

- For the file system, we need (i) a hard-disk model, (ii) an abstraction hiding the register-based device I/O, and (iii) an abstraction hiding the cluster-based partitioning of files. Only then, we can add users and permissions and thus derive the file system as it is specified in SOS*.

  So far, [HRP05], Hillebrand et al. have presented paper-and-pencil formalizations of a system with devices for the gate level and the assembler level. Using a hard disk as a specific device, they were able to prove (on paper) the correctness of a simple disk driver.

  The correspondence between VAMOS*+C0 plus hard-disk model instantiated by a user-mode hard-disk driver, on the one hand, and the abstraction hiding the register-based device I/O, on the other hand, has been proven by Elena Petrova et al. . Unfortunately, they have not yet published their results.

The formalization of a file system abstraction hiding the cluster-based partitioning of files, i. e. a model of the FAT32 file system, is included in the Isabelle / HOL specification of the SOS [Bog08b].

- For virtual terminals, we need (i) a model of a serial interface controller and (ii) the abstraction of a serial device driver. Only then, we can multiplex the keyboard and the screen and provide virtual terminals to different user applications.

  So far, Alkassar et al. have developed a formal model of the serial interface controller UART 16550A. Furthermore, they combined this device model with the ISA model and achieved a programming model for a serial device drivers. Using this programming model, they proved the correctness of a simple serial device driver [AHK+07, Alk08].

- For sockets, we need (i) a model of a network card, (ii) an abstraction of the network layer, (iii) an abstraction of the IP layer, and (iv) an abstraction of the TCP layer. Then, on top of the TCP layer, we will be able to establish abstract network packages and application-based access to endpoints of connections.

  So far, little has been done about the intermediate layers of the network stack. [Cai06] formalized and proved some properties of the TCP layer but, as of now, there is no complete Isabelle / HOL model.[40]

Note, in general, the intermediate specification layers correspond to the intermediate implementation layers.

6.1.1.2 Atomic Steps

Now, there are a number of arguments for the atomic character of handling SOS calls and external inputs. Below, we will point out the main ones:

- Because the SOS has the highest priority, it can not be scheduled away in favor of a user application.

- If there is an interrupt, then the context of the SOS is saved, the interrupt handler marks the appropriate interrupt type as pending, and then the SOS context is restored. Due to the synchronous nature of the interrupt delivery (see Section 3.5), the SOS does not have to deal with the interrupt immediately. Although the SOS may be interrupted in the middle of a C0 statement, semantics is preserved, as will be shown by the VAMOS*+C0 model.

[40] An overview of the desired properties of the TCP layer is also presented in the Verisoft-internal Technical Report #69.

- SOS-call handlers return their results to user applications by means of an IPC-send operation. This call will immediately return because we know that user applications must be waiting for the results of SOS calls. Even if this call fails, namely, if a user application's receive buffer is too small, it will not affect the atomic character of the SOS-call handler. Only the user application will wait infinitely long for its SOS call to return.

- If the SOS implementation uses a kernel call other than IPC, then this call will be treated immediately. After the call, control is passed back to the SOS. In the meantime, no user application can be scheduled.

6.1.2 Kernel Calls

Essentially, in SOS*, the semantics of kernel calls remains the same as in VAMOS*+C0. However, a few cases can be excluded, and thus, their representation can be simplified in the following way.

- The user applications that are found in SOS* are non-privileged processes in VAMOS*+C0. Hence, all calls that require the calling process to be privileged can be represented in SOS* by the Ω_p element PRIVILEGED. The result for such a call is ERR-UNPRIVILEGED.

- In our implementation, only the SOS process can be registered as device driver. Thus, only the SOS receives interrupt notifications. User applications have no means to observe interrupts. Hence, in SOS* we abandoned the interrupt data structure and instead reveal selected types of interrupts as external inputs.[41] Thus, as interrupts are no longer visible, we can use a simplified representation of kernel call return values.

We want to take advantage of the theorems proved in VAMOS* and VAMOS*+C0, i. e. the functional correctness of the kernel calls, the correctness of the scheduler abstraction, and the correctness of the kernel-call wrappers. Thus, instead of redefining each of the kernel calls in terms of the SOS* state space, we use the knowledge about the SOS implementation to map a certain SOS* state to a corresponding VAMOS*+C0 state, perform the appropriate VAMOS*+C0 transition, and then (re-) construct an SOS* state. That is:

- We take the kernel data structures and the process data base from the current SOS* state as well as specifically crafted values for the remaining VAMOS*+C0 state-space components (e.g. the interrupt data structure, the privilege data base, and the SOS process) and construct a valid VAMOS*+C0 state ($\mathit{lower} \in \mathit{kds\_t} \times \mathit{pdb\_t} \times \mathit{pid\_t} \to \mathcal{S}_\mathrm{vc}$).

[41] Only selected types of interrupts are revealed because, depending on the abstraction chosen for a certain device, its interrupts may no longer be visible. This is, for example, true for the hard disk; user applications do not see the difference between a file system implemented in memory and a file system that relies on a hard disk.

- Then, on the resulting VAMOS*+C0 state, we perform a single Δ_{vc} transition, executing the kernel call of a particular process.
- After the transition, we strip the new VAMOS*+C0 state ($\mathit{lift} \in \mathcal{S}_{\text{vc}} \to \mathit{kds\_t} \times \mathit{pdb\_t}$) and incorporate the updated kernel data structures and process data base into the old SOS* state, and thereby gain the next SOS* state.

Note, this is the informal definition of the previously declared relation *handle-kernelcall*. In the Isabelle / HOL specification of the SOS, we actually define *handle-kernelcall*, and thus, formally integrate VAMOS*+C0 into SOS*.

Now, in order to reuse the theorems proved for VAMOS*+C0, we need to show that the functions *lower* and *lift* preserve the kernel call semantics in terms of kernel data structures and process data base. That means, if, in a given VAMOS*+C0 state vc and the corresponding SOS* state s, there exists a new kernel call from the application pid, then the changes to $s.kds$ and $s.pdb$ that result from applying Δ_{vc} to the 'lowered' SOS* state (vc_{g}), on the one hand, and the actual implementation state (vc), on the other hand, must be the same (Figure 6.1 on the next page):

$$
\begin{aligned}
&\forall vc \in \mathcal{S}_{\text{vc}}.\\
&\exists s, s' \in \mathcal{S}, pid \in \mathit{pid\_t}, vc_{\text{g}}, vc', vc'_{\text{g}} \in \mathcal{S}_{\text{vc}}, \mathit{kds}' \in \mathit{kds\_t}, \mathit{pdb}' \in \mathit{pdb\_t}.\\
&\qquad\qquad s = \mathit{abs}(vc)\\
&\qquad\qquad \wedge\ \mathit{new\text{-}kernelcall}(s, pid)\\
&\qquad\qquad \wedge\ vc_{\text{g}} = \mathit{lower}(s.\mathit{kds}, s.\mathit{pdb}, pid)\\
&\qquad\qquad \wedge\ (vc_{\text{g}}, pid, vc'_{\text{g}}) \in \Delta_{\text{vc}}\\
&\qquad\qquad \wedge\ (\mathit{kds}', \mathit{pdb}') = \mathit{lift}(vc'_{\text{g}})\\
&\qquad\qquad \wedge\ s' = s[\![\mathit{kds} := \mathit{kds}', \mathit{pdb} := \mathit{pdb}']\!]\\
&\qquad\qquad \wedge\ (vc, \ldots, vc') \in \Delta_{\text{vc}}\\
&\qquad\qquad\qquad\qquad \Longrightarrow\\
&\qquad\qquad\qquad\qquad s' = \mathit{abs}(vc')
\end{aligned}
$$

6.1.3 Local Computations

In SOS* we inherit the process abstractions from VAMOS*+C0. We model local computations of user application in the same way as VAMOS*+C0 models user processes. That means, in the case of a local computation, VAMOS*+C0 plus SOS, on the one hand, and SOS*, on the other hand, update the process data base in the very same way, i. e. the implementation and the specification do the same. Hence, in terms of application-local computations, the simulation theorem obviously holds.

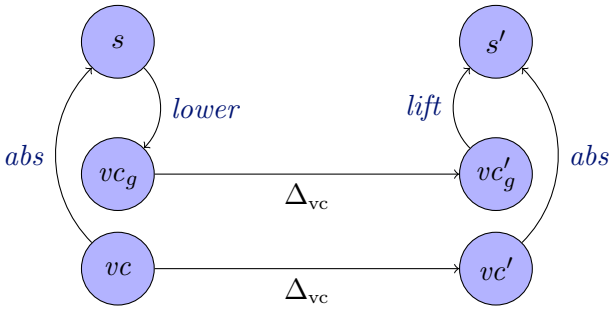

Figure 6.1: Translating Between Models. The functions *lower* and *lift* preserve the kernel call semantics in terms of kernel data structures and process data base.

6.2 Fairness Between User Applications

In § 4.5.1, we formalized fairness between user applications in terms of SOS* runs (*app-fairness*). There, we stated that an application eventually progresses unless it satisfies the predicate *possibly-no-progress*. VAMOS*+C0 ensures a similar property for user processes. In VAMOS*+C0, however, an additional requirement must be met in order to guarantee fairness. There, a user process only progresses infinitely often, if (besides the things also specified in *possibly-no-progress*) it has the maximum priority infinitely often.

Now, in order to derive the fairness between user applications from the fairness between user processes, we need to show that this additional requirement is discharged by the SOS implementation. For that, we have to show that all user applications run with the same priority and we have to show that, in the course of an infinite SOS run, the SOS idles infinitely often and thereby passes control to lower priority processes, i. e. the user applications.

The former should be easy to prove considering the implementation of the SOS calls for starting and forking user applications (§ 4.3.6.2 and § 4.3.6.1). The latter, however, is not as trivial. However, considering that the SOS idles, if there is no pending work, i. e. there is neither an SOS call nor an interrupt notification, this proof should not be too hard either. Because:

- SOS calls can only be issued, if user applications are scheduled. Even if the SOS is currently processing an SOS call, then, from the functional correctness for the particular handler, we know that this handler eventually terminates. After that, the SOS must be waiting for new work.

- Interrupt notifications are only received because of external input (e. g. keyboard input or input from the network card) or as a consequence of an earlier SOS call (e. g. a file-system call). But, in the SOS implementation, external input is buffered in bounded buffers. Thus, unless there are SOS calls, these buffers eventually fill up. If a buffer has reached its maximum,

the SOS disables the corresponding interrupt. Such a disabled interrupt is only re-enabled, if some SOS call drains the corresponding buffer. If the buffers for each of the devices have filled up, then the SOS can no longer be busy because of interrupt notifications. Hence, the SOS can not be 'flooded' by interrupt notifications. Which means, at some point, a user application must be scheduled.

Adding up, this means, there is no way that the SOS starves user applications. Thus, the additional requirement (that was necessary in VAMOS⋆+C0) is already satisfied by the SOS implementation. Hence, we can lift fairness between user processes (VAMOS⋆+C0) to fairness between user applications (SOS⋆).

6.3 SOS⋆+C0/Libsos Correctness

Proving the correctness of the library Libsos should be straightforward. We need to prove: (i) that the C0 wrapper code constructs IPC messages that respect the syntax (§ 4.3.7) required by the SOS call dispatcher, and (ii) that results are correctly incorporated into the C0 state of the user application. These proofs should be very similar to the proofs that show functional correctness of the Libvamos kernel-call wrappers. The concept of shifting the scheduling decisions was already established in VAMOS⋆+C0. Thus, the correctness of the SOS-call wrappers is (solely) a code correctness proof.

6.4 Other Properties

Depending on the way correctness of individual applications shall be proven, it may be desirable to extract some of the SOS⋆ properties. That is, properties that are inherent to SOS⋆ may be needed in a more explicit form. Proofs for those properties only rely on the SOS⋆ model itself. As an example for such a property, we have formally proven that the cursor can not be moved outside the visible screen (Appendix A.3).

Chapter

7

Conclusion

Contents

7.1 Summary

Within the Verisoft project, we aim at the pervasive modeling, implementation, and verification of a complete computer system, from gate-level hardware to applications running on top of an operating system. As an adequate representative for such a system we choose a system for writing, signing, and sending emails.

The starting point of our work was a processor together with its assembly language, a compiler for a type safe C variant and a micro kernel. The goal of our work was to develop a (user-mode) operating system that bridges the gap between micro kernel and user applications. That is, formally specify and implement a system that, on the one hand, is built right on top of our micro kernel and, on the other hand, provides everything necessary for user applications such as an SMTP server, a signing server, and an email client. Furthermore, the design of this system should support its verification in a pervasive context.

Within this document, we presented the formal specification of such an operating system, i. e. the model SOS*. Furthermore, we discussed the foundations of our work, briefly described the implementation of the SOS, and outlined the verification obligations. The specification at hand is complemented by a corresponding Isabelle / HOL specification [Bog08c] and the C0 implementation of the SOS [Bog08a].

There have been numerous attempts to increase confidence in system software by means of formal methods. In Section 1.2 we argued that these projects usually restrict themselves to a particular part of a computer system. On the one hand, there are projects that tackle selected system components (e. g. the file system or the network stack) and show that their specification satisfies certain high-level properties (e. g. [BCT95, BC96, AZKR04] and [Smi96a, BFN+07, RNS08]). However, projects in this category fail to reliably relate their specification to a particular implementation. In a sense, they are stuck at the upper part of their systems because they are missing a solid foundation. On the other hand, there are projects that start at the bottom. But no matter whether they have pervasive verification in mind or (only) focus on a particular micro-kernel implementation, none of the projects that we know of, reaches a level above micro kernels (e. g. [Bev89, HT05, SDN+04, HEK+07]). Thus, we are the first that specified and implemented an operating system that reaches so far up and, at the same time, is part of an integrated system stack. In a sense, our work connects both worlds.

While developing the SOS, the main problem was to resolve the dependencies between pervasive verification and implementation. This is because design decisions for one layer usually inflicted consequences for other layers — both, for implementation and verification. In the course of our work, it was, for example, several times necessary to entirely redesign the SOS* model and to adapt the SOS implementation in order to successfully integrate it with the remaining system. In many places, the VAMOS implementation is influenced by the verification goals of SOS* (e. g. IPC rights and privileged kernel calls). Likewise, the SOS implementation is much influenced by the limitations of C0 and the restrictive set of kernel calls (e. g. the lack of pointer arithmetic and the lack of shared memory). However, in the end, we were still able to provide a comparatively small specification (~3000 lines of Isabelle / HOL theories) and an appropriate implementation.

Except for the network- and the IP layer, our implementation is complete. Among other things, it includes a simple TCP- and socket implementation, a FAT32-compatible file system with additional access control lists, and virtual terminals. Furthermore, it provides means to manage users and applications and supports remote procedure calls. All in all, there are 31 SOS calls. These calls have been used to implement applications, such as a basic shell, an SMTP server, a signing server, and an email client. Running these applications, we were able to demonstrate two SOS instances sending and receiving signed emails via the Internet [Bog07].

Our specification, the model SOS*, establishes a computational model for communicating user applications. This model has been formalized as a transition system, where each of the transitions represents the handling of a system call, the handling of input from an external device, or the local computation of one of the user applications. We were able to specify most of the SOS calls as deterministic atomic steps. Only at the dispatcher level

nondeterminism was introduced. Along with the SOS calls, we formalized various invariants and showed how they are maintained. Within SOS*, user applications are incorporated as self-contained I/O automata. Via appropriate model parameters arbitrary process abstractions can be included. In order to connect our specification to the underlying VAMOS*+C0 model, we developed a means to translate between state spaces. Using this method we were able to actually incorporate VAMOS*+C0 into the SOS* model and recycle many of its definitions. Conversely, our specification has been used to formally verify parts of the email client [BHW06] as well as the SMTP- and signing server [LNRS07]. That means, our specification is really part of an integrated stack.

Last but not least, in the document at hand, we used only a small set of mathematical tools to present the specification of a whole operating system. We developed a simple formalism that allowed us to specify SOS calls in a dense and precise manner. After studying the preliminaries, even non-mathematicians should be able to use this document as a reference manual.

7.2 Future Work

So far, we have only outlined the verification obligations and sketched the proofs for the top level theorems. In order to reach the goal of a pervasively verified computer system, one needs to formally prove these theorems. That is, we need to define the abstraction relation and show that the simulation relation holds.

Appendix

A

Appendix

A.1 Lost in Translation

In the following we provide the most important translations between Math, Isabelle / HOL, and C0.

Math	Isabelle / HOL	C0
adb	`s_ainfodb`	
aexec	`s_aexec`	`sos_app_exec`
aexec-kernel	`s_create`	
aexit	`s_aexit`	`sos_app_exit`
aexit-kernel	`s_kill`	
afork	`s_afork`	`sos_app_fork`
afork-kernel	`s_clone`	
await	`s_await`	`sos_app_wait`
Δ	`s_`Δ	
Δ_{ds}	`ds_`Δ	
δ_{p}	δ`proc_sos`	
event-sack	`s_event_sack`	`sos_socket_int`
event-sclose	`s_event_sclose`	`sos_socket_int`
event-sdata	`s_event_sdata`	`sos_socket_int`
event-sready	`s_event_sready`	`sos_socket_int`

Math	Isabelle / HOL	C0
event-sreq	*s_event_sreq*	sos_socket_int
event-tkbd	*s_event_tkbd*	sos_term_int
faccess-error	*s_faccess_error*	sos_file_access
faccess-legal	*s_is_legal_faccess*	sos_file_access
fchmod	*s_fchmod*	sos_file_chmod
fchown	*s_fchown*	sos_file_chown
fcreat	*s_fcreate*	sos_file_creat
fdb	*s_filedb*	
finfo	*s_finfo*	sos_file_info
flock	*s_flock*	sos_file_lock
focus	*s_focus*	
fread	*s_fread*	sos_file_read
free-clusters	*f32_fc*	
fseek	*s_fseek*	sos_file_seek
ftruncate	*s_ftruncate*	sos_file_truncate
funlink	*s_funlink*	sos_file_unlink
funlock	*s_funlock*	sos_file_unlock
fwrite	*s_fwrite*	sos_file_write
handle-kernelcall	*s_syscall_vc*	
hdb	*v_hdb rightsdb*	
interpret	*init_proc*	
lift	*s_lift*	
lna	*s_lna*	
lower	*s_lower*	
match-socket	*s_get_socket*	
Ω	*s_*Ω	
Ω_{p}	Ω*proc_sos*	
Ω_{sc}	*s_scT*	
ω_{p}	ω*proc_sos*	

Math	Isabelle / HOL	C0
ocl	*f32_ocl*	
offset	*s_calc_offset*	
pdb	*s_procdb*	
pmdb.known	*s_pmlist*	
pmdb.reg	*s_pmregistered*	
pmdb.serv	*s_pmserver*	
pmlookup	*s_pmlookup*	sos_pm_lookup
pmreg	*s_pmreg*	sos_pm_reg
pmunreg	*s_pmunreg*	sos_pm_unreg
$\mathcal{R}$	*s_RT*	
$\mathcal{R}_{\text{ds}}$	*ds_RT*	
rdb	*v_rdb rightsdb*	
rdb-holes	*s_rdb_holes*	
rdt-holes	*f32_holes*	
Σ	*s_*Σ	
Σ_{ds}	*ds_*Σ	
Σ_{p}	Σ*proc_sos*	
Σ_{sc}	*s_scrT*	
$\mathcal{S}$	*s_ST*	
$\mathcal{S}_{\text{ds}}$	*'a ds_ST*	
$\mathcal{S}_{\text{p}}$	*'a::proc_conf*	
saccept	*s_saccept*	sos_socket_accept
saccess-error	*s_saccess_error*	sos_socket_access
saccess-legal	*s_is_legal_saccess*	sos_socket_access
sclose	*s_sclose*	sos_socket_close
sconnect	*s_sconnect*	sos_socket_connect
sdb	*s_socketdb*	
slisten	*s_slisten*	sos_socket_listen
sopen	*s_sopen*	sos_socket_open
sread	*s_sread*	sos_socket_read
sthdb	*v_stolen rightsdb*	
swrite	*s_swrite*	sos_socket_write

Math	Isabelle / HOL	C0
tdb	*s_termdb*	
tinfo	*s_tinfo*	sos_term_info
tread	*s_tread*	sos_term_read
tseek	*s_tseek*	sos_term_seek
twrite	*s_twrite*	sos_term_write
uadd	*s_uadd*	sos_user_add
udb	*s_userdb*	
udel	*s_udel*	sos_user_del
vm-size	*size_proc*	
wdb	*s_sndstatdb*	

A.2 Two-Way Handshake

In the following section, we prove that, for SOS* and DSOS*, the two-way handshake is indeed a valid abstraction of the three-way handshake (Figure A.1 on page 161). For that, we need to argue on the TCP layer (rather than the socket layer). However, so far, there is no Isabelle / HOL specification of the TCP layer. Hence, we model the relevant parts of the TCP layer ourself and then use this small (problem-tailored) formalization to justify the two-way handshake.[42]

```
theory tcp
imports Main
begin
```

First we declare a number of (abstract) TCP states.

```
consts
  Accept                :: "nat⇒bool"
  CA                    :: "nat=>bool"
  CC                    :: "nat=>bool"
  InfinitelySendAck     :: "nat⇒nat⇒nat=>bool"
  InfinitelySendSyn     :: "nat⇒nat⇒nat=>bool"
  InfinitelySendSynAck  :: "nat⇒nat⇒nat=>bool"
  Initial               :: "nat⇒bool"
  RecvAck               :: "nat⇒nat⇒nat=>bool"
  RecvSyn               :: "nat⇒nat⇒nat=>bool"
  RecvSynAck            :: "nat⇒nat⇒nat=>bool"
  ScAccept              :: "nat⇒bool"
  ScConnect             :: "nat⇒nat⇒nat=>bool"
  SendSyn               :: "nat⇒nat⇒nat=>bool"
  SendSynAck            :: "nat⇒nat⇒nat=>bool"
```

We assume `Initial` denotes a pristine TCP state and `ScAccept` denotes a call to the TCP subsystem to accept incoming connections. The resulting TCP state is denoted by `Accept`. Thus, if the system `a` is in an initial state (`Initial a`) and the TCP is called to accept incomming connections (`ScAccept a`), then the new state is `Accept a`.

```
constdefs
  "ItoA ≡
   ∀ a. Initial a ∧ ScAccept a ⟶ Accept a"

   lemma LItoA:
   "⟦ ItoA; Initial a;  ScAccept a⟧ ⟹ Accept a"
   apply (simp add: ItoA_def)
   done
   declare LItoA [simp]
```

If the system `a` is in the `Accept` state and it receives a `SYN` packet from the system `b`, here denoted by `RecvSyn a b a`, then it changes its state to `SendSynAck`

[42] This proof has been taken from the Verisoft Technical Report #5 [Bog08d].

a a b and returns a SYNACK packet (as an acknowledgment to the opposite side b). Note, we assume that the SYNACK packet matches the SYN packet that was received. Further note, in *RecvSyn* a b a, the first "a" identifies the system (that received the packet), the following "b" identifies the source of the packet, and the final "a" identifies the destination of the packet. Thus this quadruple can be interpreted as "<*state*> <*system*> <*source*> <*destination*>". We will use the same scheme throughout the rest of this proof.

constdefs *"AtoSSA* ≡
  ∀ a b. a≠b ∧ *Accept* a ∧ *RecvSyn* a b a ⟶ *SendSynAck* a a b"

lemma *LAtoSSA:*
  "⟦ *AtoSSA; a≠b ; Accept* a ; *RecvSyn* a b a⟧ ⟹ *SendSynAck* a a b"
  apply *(simp add: AtoSSA_def)*
  done

declare *LAtoSSA [simp]*

If the system a is in the *SendSynAck* a a b state and it receives a matching ACK packet (*RecvAck* a b a), i.e. an ACK packet from the site it originally received the SYN packet from and it sent the SYNACK packet to, then a connection is considered to be established. Here, we denote such an established connection by *CA*:

constdefs *"SSAtoCA* ≡
  ∀ a b. a≠b ∧ *SendSynAck* a a b ∧ *RecvAck* a b a ⟶ *CA* a"

lemma *LSSAtoCA:*
  "⟦ *SSAtoCA; a≠b ; SendSynAck* a a b ; *RecvAck* a b a⟧ ⟹ *CA* a"
  apply *(simp add: SSAtoCA_def)*
  apply *(auto)*
  done

declare *LSSAtoCA [simp]*

Lets switch from the server side, the one accepting incoming connections, to the client side, the one initiating a connection. Assume the client b is in the initial state and the TCP subsystem is asked to connect to the remote side a (*ScConnect* b b a). In this case, a SYN packet will be sent to the remote side and the state is changed to *SendSyn* b b a:

constdefs *"ItoSS* ≡
  ∀ a b. a≠b ∧ *Initial* b ∧ *ScConnect* b b a ⟶ *SendSyn* b b a"

lemma *LItoSS:*
  "⟦ *ItoSS; a≠b ; Initial* b ∧ *ScConnect* b b a⟧ ⟹ *SendSyn* b b a"
  apply *(simp add: ItoSS_def)*
  done

declare *LItoSS [simp]*

If a SYN packet was sent and a corresponding SYNACK packet received (*RecvSynAck* b a b), then the connection is considered to be established (from the local point

of view). The resulting state is denoted by *CC*.

Note, depending on the role of a party of an established connection (either server or client), this final state is named differently. The reason for this will be explained later.

```
constdefs "SStoCC ≡
  ∀ a b. a≠b ∧ SendSyn b b a ∧ RecvSynAck b a b ⟶ CC b"

lemma LSStoCC:
  "⟦ SStoCC; a≠b ; SendSyn b b a ; RecvSynAck b a b⟧ ⟹ CC b"
  apply (simp add: SStoCC_def)
  apply (auto)
  done

declare LSStoCC [simp]
```

So far we have only considered an ideal world, where no packets are lost. In the document at hand, however, we will also consider the case that packets get lost or discarded because of some sort of error. Still, we will assume that not all packets get lost. That means, if we sent infinitely many packets, then infinitely many packets are received. That also means, if we sent a particular packet often enough, then, eventually, it will be received.

Thus, knowing that, we will now go back and consider the cases, where packets are lost and therefore need to be resent.

Assume the client side sent a `SYN` packet (*SendSyn b b a*) and the server side really accepts incoming connections (*Accept a*), then the latter may receive this packet (*RecvSyn a b a*) and acknowledge it or the client starts resending the packet. Our implementation does not support timeouts, thus, it potentially resends the packet infinitely often (*InfinitelySendSyn b b a*).

```
constdefs "SStoISS ≡
  ∀ a b. a≠b ∧ SendSyn b b a ∧ Accept a
  ⟶ RecvSyn a b a  ∨ InfinitelySendSyn b b a"
```

However, we know that each packet will be received, if it is sent often enough. That means:

```
constdefs "ISStoRS ≡
  ∀ a b. a≠b ∧ InfinitelySendSyn b b a ⟶ RecvSyn a b a"

lemma LSStoRS:
  "⟦ SStoISS; ISStoRS; a≠b ∧ SendSyn b b a ∧ Accept a ⟧
  ⟹ RecvSyn a b a"
  apply (simp add: SStoISS_def ISStoRS_def)
  apply (auto)
  done

declare LSStoRS [simp]
```

On the way back, we follow the same argument. If a server accepts a connection and returns an appropriate `SYNACK` packet, then the packet may be received (*RecvSynAck b a b*) and acknowledged by the client or the server starts

resending the packet. As for *SStoISS*, our implementation does not support timeouts, thus, also the server potentially resends the SYNACK packet infinitely often (*InfinitelySendSynAck a a b*).

```
constdefs "SSAtoISSA ≡
  ∀ a b. a≠b ∧ SendSynAck a a b ∧ SendSyn b b a
  ⟶ RecvSynAck b a b ∨ InfinitelySendSynAck a a b"
```

But, as before, if sent often enough, then, eventually, it will be received. Thus:

```
constdefs "ISSAtoRS ≡
  ∀ a b. a≠b ∧ InfinitelySendSynAck a a b ⟶ RecvSynAck b a b"

lemma LISSAtoRS:
  "⟦ SSAtoISSA; ISSAtoRS; a≠b ∧ SendSynAck a a b ∧ SendSyn b b a ⟧
  ⟹ RecvSynAck b a b"
  apply (simp add: SSAtoISSA_def ISSAtoRS_def)
  apply (auto)
  done

declare LISSAtoRS [simp]
```

Finally, also the ACK packet may be lost. Thus, if a client sent an ACK packet, then the packet may be received (*RecvAck a b a*) by the server or the client starts resending the packet (*InfinitelySendAck b b a*).

```
constdefs "CCtoISA ≡
  ∀ a b. a≠b ∧ CC b ∧ SendSynAck a a b
  ⟶ RecvAck a b a ∨ InfinitelySendAck b b a"
```

Again, if sent often enough, eventually, it will be received. Thus:

```
constdefs "ISAtoRA ≡
  ∀ a b. a≠b ∧ InfinitelySendAck b b a ⟶ RecvAck a b a"

lemma LCCtoRA:
  "⟦ CCtoISA; ISAtoRA; a≠b ∧ CC b ∧ SendSynAck a a b ⟧
  ⟹ RecvAck a b a"
  apply (simp add: CCtoISA_def ISAtoRA_def)
  apply (auto)
  done

declare LCCtoRA [simp]
```

Note, usually the client does not sent the ACK just by itself. Normally, this packet already contains actual payload. Here, we are not interested in the payload and simply abstract from it.

Furthermore, in order to be able to express the resending-\small{}ACK behaviour, we differentiate *CC* and *CA*. However, a connection is established (*Established*) if *CC b ∧ CA a*.

```
constdefs "Established a b ≡
  CC b ∧ CA a"
```

```
lemma LEestablished:
  "⟦ a≠b; CC b; CA a ⟧ ⟹ Established a b"
  apply (simp add: Established_def)
  done

declare LEestablished [simp]
```

Now, having formalized the different transitions and properties of the network, we will derive some higher level properties.

The theorem *EventuallyConnected* states that if two parties have initially a pristine TCP state and one of them (a) allows for incoming connections, i.e. it calls accept, while the other one (b) tries to establish a corresponding connection, i.e. it calls connect, then a connection will be established (*Established a b*):

```
theorem EventuallyConnected:
  "⟦ ItoA; ItoSS; SStoISS; ISStoRS; AtoSSA; SSAtoISSA;
     ISSAtoRS; SStoCC; CCtoISA;  ISAtoRA; SSAtoCA
   ⟧ ⟹ a≠b ∧ Initial a ∧ Initial b ∧ ScAccept a ∧ ScConnect b b a
        ⟶ Established a b"
  apply (auto)
  done
```

The proof for *EventuallyConnected* is trivial as we only have to "chain" together the previously decalared axioms (*ItoA*, *ItoSS*, etc.).

Besides the assumption, that the network actually transmits some packets, we further assume that (i) the network does not invent packets, (ii) it is impossible to fake the source of a packet, and (iii) a packet can not be tampered with on its way through the net. That means, we can trust received packets. Thus, the reception of a packet implies that the opposite side must have sent this packet:

```
constdefs
  "SSfromRS ≡ ∀ a b. a≠b ∧ RecvSyn a b a ⟶ SendSyn b b a"
  "SSAfromRSA ≡ ∀ a b. a≠b ∧ RecvSynAck b a b ⟶ SendSynAck a a b"
lemma LSSfromRS:
  "⟦ SSfromRS; a≠b; RecvSyn a b a ⟧ ⟹ SendSyn b b a"
  apply (simp add: SSfromRS_def)
  done

lemma LSSAfromRSA:
  "⟦ SSAfromRSA; a≠b; RecvSynAck b a b ⟧ ⟹ SendSynAck a a b"
  apply (simp add: SSAfromRSA_def)
  done

declare LSSfromRS [simp]
declare LSSAfromRSA [simp]
```

Knowing that, we can conclude that if a SYN packet is received by a server, then the connection will be established:

```
theorem SynToEstablished:
  "⟦ SStoISS; ISStoRS; AtoSSA; SSAtoISSA;
     ISSAtoRS; SStoCC; CCtoISA;  ISAtoRA; SSAtoCA;
```

```
    SSfromRS; SSAfromRSA; SAfromRA
  ⟧ ⟹ a≠b ∧ Accept a ∧ RecvSyn a b a ⟶ Established a b"
apply (auto)
done
```

Also the proof for *SynToEstablished* is trivial. We only have to apply *SSfromRS* to infer that *b* really sent a SYN packet and then (again) "chain" together *SStoISS* and friends.

Similarly, the connection will be established, if a SYNACK packet is received by a client.

```
theorem SynAckToEstablished:
  "⟦ SStoISS; ISStoRS; AtoSSA; SSAtoISSA;
     ISSAtoRS; SStoCC; CCtoISA;  ISAtoRA; SSAtoCA;
     SSfromRS; SSAfromRSA; SAfromRA
   ⟧ ⟹ a≠b ∧ SendSyn b b a ∧ RecvSynAck b a b ⟶ Established a b"
  apply auto
  done
```

Finally, the proof for *SynAckToEstablished* works as the previous one.

Now, if, on the one hand, a server can be sure that a connection will be established as soon as it receives a SYN packet and, on the other hand, a client can be sure that the connection will be established as soon as it receives the (matching) SYNACK packet, then we can abstract away the final ACK packet. Hence, under the given assumptions, a two-way handshake is a valid abstraction of the three-way handshake (Figure A.1 on the next page). Note, if the ACK packet contains payload, then, in SOS*, it is simply represented by a DATA packet; if not, then it is entirely hidden.

=2pt

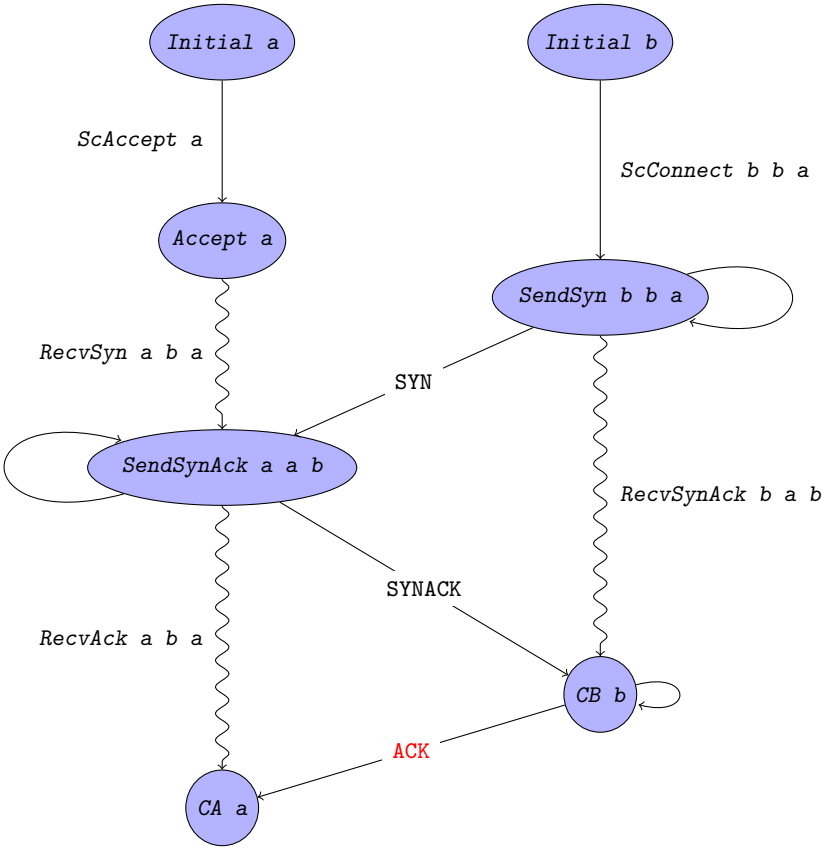

Figure A.1: Three-Way Handshake

A.3 Property Example

We prove a (high-level) property of SOS* using its Isabelle / HOL specification.[43]

```
theory sCursor
imports "../sDelta"
begin
```

We prove that, if in `s` for all terminals the cursor was within `0` and `S_SCRX*S_SCRY`, i.e. `0 ≤ (s_tpos ((s_termdb s) t)) ≤ S_SCRX*S_SCRY`, then the same holds for `s'`. That means, `s_Δ` never moves the cursor to a position outside of the screen area.

```
declare s_app_tup_def [simp]
declare s_app_the_tid_def [simp]
declare s_is_app_tno_def [simp]
declare S_SCRX_def [simp]
declare S_SCRY_def [simp]
declare S_SCRXY_def [simp]
declare s_file_up_def [simp]
declare s_socketdb_up_def [simp]

lemma min_smaller: "(a::int) ≥ min a b"
  apply (arith)
done

lemma max_bigger: "(a::int) ≤ max a b"
  apply (arith)
done

lemma max_min_bigger: "(a::int) ≤ max (min b c) a"
  apply (arith)
done
```

`s_calc_offset` calculates a value between `start` and `stop`.

```
lemma s_calc_offset_in_bounds:
        "start<stop ⟹
         start ≤ s_calc_offset start cur stop rel off
         ∧ s_calc_offset start cur stop rel off≤stop"
        apply (simp add: s_calc_offset_def)
        apply (simp add: max_min_bigger)
        apply (simp add: min_smaller)
done
```

`s_tread` does not change the cursor position.

```
lemma s_tread_tpos_equal:
  "s_tpos ((s_termdb s) x) = s_tpos ((s_termdb (fst(s_tread s hn))) x)"
  apply (simp add: s_tread_def Let_def)
  apply (case_tac "s_aterm ⌊s_ainfodb s hn⌋")
    apply (simp+)
  apply (case_tac "s_tin (s_termdb s (s_app_the_tid s hn))")
```

[43] This proof is included in the Isabelle / HOL specification of the SOS [Bog08c].

```
    apply (simp)
    apply (simp add: split_def)
  apply (simp)
  apply (simp add: split_def)
done
```

None of the system call handlers moves the cursor beyond *S_SCRX*S_SCRY* (if called by *s_dispatch_sc*).

```
lemma s_dispatch_sc_tpos_in_bounds:
        "⟦(s_tpos ((s_termdb s) t)) ≤ (S_SCRX*S_SCRY)⟧ ⟹
   (s_tpos ((s_termdb (fst (s_dispatch_sc s sc hn))) t))
   ≤ (S_SCRX*S_SCRY)"
  apply (simp (no_asm) add: s_dispatch_sc_def)
  apply (split s_scT.split)
  apply (auto)
    (* twrite *)
    apply (simp (no_asm) add: s_twrite_def)
    apply (simp add: Let_def)
    apply (auto)
    (* tseek *)
    apply (simp (no_asm) add: s_tseek_def)
    apply (simp add: Let_def)
    apply (auto)
    apply (subst zle_int [THEN sym])
    apply (simp)
    apply (rule impI)
    apply (simp add: s_calc_offset_in_bounds)
    (* tinfo *)
    apply (simp (no_asm) add: s_tinfo_def Let_def)
    (* tread *)
    apply (simp add: s_tread_tpos_equal[THEN sym])
    (* fcreat *)
    apply (simp add: s_fcreate_def f32_create_def Let_def)
    (* ftruncate *)
    apply (simp add: s_ftruncate_def f32_truncate_def Let_def)
    (* ftruncate *)
    apply (simp add: s_funlink_def f32_unlink_def Let_def)
    (* finfo *)
    apply (simp add: s_finfo_def Let_def)
    (* fwrite *)
    apply (simp add: s_fwrite_def  f32_write_def Let_def)
    (* fseek *)
    apply (simp add: s_fseek_def  Let_def)
    (* fread *)
    apply (simp add: s_fread_def  Let_def)
    (* flock *)
    apply (simp add: s_flock_def  Let_def)
    (* funlock *)
    apply (simp add: s_funlock_def  Let_def)
    (* fchmod *)
    apply (simp add: s_fchmod_def  Let_def)
    (* fchown *)
    apply (simp add: s_fchown_def  Let_def)
```

```
(* sopen *)
apply (simp add: s_sopen_def  Let_def)
(* slisten *)
apply (simp add: s_slisten_def  Let_def)
(* sconnect *)
apply (simp add: s_sconnect_def  Let_def)
(* saccept *)
apply (simp add: s_saccept_def Let_def)
apply (simp add: split_def)
apply (simp add: s_new_connection_def Let_def)
(* sread *)
apply (simp add: s_sread_def  Let_def)
(* swrite *)
apply (simp add: s_swrite_def  Let_def)
(* sclose *)
apply (simp add: s_sclose_def  Let_def)
(* uadd *)
apply (simp add: s_uadd_def  Let_def)
(* udel *)
apply (simp add: s_udel_def  Let_def)
(* pmreg *)
apply (simp add: s_pmreg_def  Let_def)
(* pmlkp *)
apply (simp add: s_pmlookup_def  Let_def)
(* pmunreg *)
apply (simp add: s_pmunreg_def  Let_def)
(* aexec *)
apply (simp add: s_aexec_def)
apply (simp add: s_create_def)
apply (simp add: Let_def)
apply (simp add: s_inherit_term_def)
apply (rule conjI)
  apply (rule impI)
  apply (rule conjI)
    apply (intro impI)
    apply (simp add: Let_def)
  apply (intro impI)
  apply (simp add: Let_def)
apply (intro impI)
apply (simp add: Let_def)
apply (intro impI)
apply (rule conjI)
  apply (intro impI)
  apply (simp add: Let_def)
apply (intro impI)
apply (simp add: Let_def)
(* afork *)
apply (simp add: s_afork_def  Let_def)
apply (simp add: s_clone_def)
apply (simp add: Let_def)
apply (intro impI)
apply (simp add: s_inherit_term_def)
apply (rule conjI)
  apply (rule impI)
```

```
    apply (simp add: Let_def)
   apply (rule impI)
   apply (simp add: Let_def)
   apply (rule impI)
   apply (simp)
   (* await *)
   apply (simp add: s_await_def  Let_def)
   (* aexit *)
   apply (simp add: s_aexit_def s_kill_def Let_def)
done


declare S_SCRX_def [simp del]
declare S_SCRY_def [simp del]
declare S_SCRXY_def [simp del]
```

Returning results to user applications (*s_send_list*) does not move the cursor beyond *S_SCRX*S_SCRY*.

```
lemma s_send_list_tpos_in_bounds [rule_format]:
  "∀ s t. (s_tpos ((s_termdb s) t)) ≤ (S_SCRX*S_SCRY) ⟶
   s_tpos (s_termdb (s_send_list s xs) t) ≤ S_SCRX * S_SCRY"
  apply (induct_tac xs)
    apply (simp)
    apply (simp)
  apply (intro allI)
  apply (rule impI)
  apply (simp add: s_send_def)
  apply (simp add: split_def)
  apply (simp add: Let_def)
done
```

Handling an SOS call (*s_syscall_sos*) does not move the cursor beyond *S_SCRX*S_SCRY*.

```
lemma s_syscall_sos_tpos_in_bounds:
        "⟦(s_tpos ((s_termdb s) t)) ≤ (S_SCRX*S_SCRY)⟧ ⟹
   (s_tpos ((s_termdb (fst(s_syscall_sos s p sc rights_snd hn_add))) t))
   ≤ (S_SCRX*S_SCRY)"
  apply (simp add: s_syscall_sos_def)
  apply (simp add: Let_def)
  apply (simp add: split_def)
  apply (rule s_send_list_tpos_in_bounds)
  apply (rule s_dispatch_sc_tpos_in_bounds)
  apply (simp add: s_receive_def)
  apply (simp add: Let_def)
done
```

Handling a kernel call (*s_syscall_vc*) does not move the cursor beyond *S_SCRX*S_SCRY*.

```
lemma s_syscall_vc_tpos_equal:
  "(s,p,s') ∈ s_syscall_vc ⟹
   (s_tpos ((s_termdb s') t))=(s_tpos ((s_termdb s) t))"
  apply (simp add: s_syscall_vc_def)
```

```
  apply (simp add: Δvc_def)
  apply (simp add: Let_def)
  apply (simp add: s_lower_def)
  apply (simp add: Let_def)
  apply (simp add: s_lift_def)
  apply (erule exE)
  apply (simp)
done
```

Neither SOS call nor kernel call move the cursor beyond `S_SCRX*S_SCRY` (`s_syscall`).

```
lemma s_syscall_tpos_in_bounds:
        "⟦(s_tpos ((s_termdb s) t)) ≤ (S_SCRX*S_SCRY);
    (s,p,s',output) ∈ s_syscall⟧ ⟹
   (s_tpos ((s_termdb s') t)) ≤ (S_SCRX*S_SCRY)"
  apply (simp add: s_syscall_def)
  apply (case_tac "ωproc_sos s p")
    apply (simp)
    apply (drule_tac f=fst in arg_cong)
    apply (simp)
    apply (simp add: s_syscall_sos_tpos_in_bounds)
  apply (simp)
  apply (auto)
  apply (simp add: s_syscall_vc_tpos_equal)
done
```

None of the event handlers moves the cursor beyond `S_SCRX*S_SCRY` (if called by `s_dispatch_event`).

```
lemma s_dispatch_event_tpos_in_bounds:
        "⟦(s_tpos ((s_termdb s) t)) ≤ (S_SCRX*S_SCRY)⟧ ⟹
   (s_tpos ((s_termdb (fst(s_dispatch_event s a))) t)) ≤ (S_SCRX*S_SCRY)"
  apply (simp (no_asm) add: s_dispatch_event_def)
  apply (split s_Σ.split)
  apply (rule conjI)
    (* event tkbd *)
    apply (simp add: Let_def)
    apply (rule impI)
    apply (rule allI)
    apply (rule impI)
    apply (simp add: s_event_tkbd_def)
    apply (rule conjI)
      apply (auto)
      apply (simp add: split_def)
      apply (rule conjI)
        apply (rule impI)
        apply (simp add: Let_def)
        apply (intro impI)
        apply (simp)
      apply (rule impI)
      apply (simp add: Let_def)
      apply (rule impI)
      apply (simp)
    apply (simp add: Let_def)
```

```
    apply (simp add: split_def)
    apply (rule impI)
    apply (simp)
  apply (split s_npT.split)
  apply (auto)
         (* event sreq *)
         apply (simp add: s_event_sreq_def)
         apply (simp add: Let_def)
        (* event sready *)
        apply (simp add: s_event_sready_def)
        apply (simp add: Let_def)
      (* event sdata *)
      apply (simp add: s_event_sdata_def)
      apply (simp add: Let_def)
    (* event sack *)
    apply (simp add: s_event_sack_def)
    apply (simp add: Let_def)
  (* event sclose *)
  apply (simp add: s_event_sclose_def)
  apply (simp add: Let_def)
done
```

s_Δ keeps the cursor within *0* and *S_SCRX*S_SCRY*.

```
theorem s_Δ_tpos_in_bounds:
  "⟦ 0≤(s_tpos ((s_termdb s) t));
          (s_tpos ((s_termdb s) t))≤(S_SCRX*S_SCRY);
     (s,inp,s',outp) ∈ s_Δ ⟧ ⟹
   0≤(s_tpos ((s_termdb s) t))
   ∧(s_tpos ((s_termdb s') t)) ≤ (S_SCRX*S_SCRY)"
  apply (simp add: s_Δ_def)
      apply (auto)
      apply (simp add: s_syscall_tpos_in_bounds)
    apply (simp add: s_input_def)
    apply (simp add: Let_def)
    apply (simp add: split_def)
    apply (rule s_send_list_tpos_in_bounds)
                apply (simp add: s_dispatch_event_tpos_in_bounds)
  apply (simp add: s_syscall_vc_tpos_equal)
done

declare s_app_tup_def [simp del]
declare s_app_the_tid_def [simp del]
declare s_is_app_tno_def [simp del]
declare s_file_up_def [simp del]
declare s_socketdb_up_def [simp del]

end
```

Bibliography

[AHK+07] Eyad Alkassar, Mark A. Hillebrand, Steffen Knapp, Rostislav Rusev, and Sergey Tverdyshev. Formal device and programming model for a serial interface. In Bernhard Beckert, editor, *4th International Verification Workshop, Bremen, Germany*, pages 4–20, 2007.

[Alk08] Eyad Alkassar. *Formal Functional Verification of Device Drivers*. PhD thesis, Saarland University, Saarbrücken, 2008. to appear.

[ANS83] ANSI. ANSI/IEEE 770x3.97-1983: The Pascal programming language, 1983.

[ANS99] ANSI. ANSI/ISO/IEC 9899-1999: Programming languages — C, 1999.

[ASS08] Eyad Alkassar, Norbert W. Schirmer, and Artem Starostin. Formal pervasive verification of a paging mechanism. In Juris Hartmanis Gerhard Goos and Jan van Leeuwen, editors, *14th intl Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *LNCS*, pages 109–123. Springer, 2008.

[AZKR04] Konstantine Arkoudas, Karen Zee, Viktor Kuncak, and Martin Rinard. Verifying a file system implementation. In *Sixth International Conference on Formal Engineering Methods*, volume 3308 of *LNCS*, pages 373–390, 2004.

[BB04] Bernhard Beckert and Gerd Beuster. Formal specification of security-relevant properties of user interfaces. In *Proceedings, 3rd International Workshop on Critical Systems Development with UML, Lisbon, Portugal*, Munich, Germany, 2004. TU Munich Technical Report TUM-I0415.

[BBuMW07] Gerd Beuster, Thorsten Bormer, and Pia Breuer und Markus Wagner. Code-level verification of an email client. `http://www.verisoft.de/.rsrc/VerisoftRepository/vemail-trunk-r15868.tar.gz`, 2007.

[BC96] William R. Bevier and Richard Cohen. An executable model of the Synergy file system. Technical Report 121, Computational Logic Inc., 1996.

[BCT95] William R. Bevier, Richard Cohen, and Jeff Turner. A specification for the Synergy file system. Technical Report 120, Computational Logic Inc., 1995.

[BDD+92] Manfred Broy, Frank Dederich, Claus Dendorfer, Max Fuchs, Thomas Gritzner, and Rainer Weber. The design of distributed systems - An introduction to FOCUS. Technical Report TUM-I9202, Technical University of Munich, jan 1992.

[Bev89] William R. Bevier. Kit and the short stack. *Journal of Automated Reasoning*, 5(4):519–530, 1989.

[Bey05] Sven Beyer. *Putting it all together - Formal Verification of the VAMP*. PhD thesis, Saarland University, Saarbrücken, 2005.

[BFN+05] Steve Bishop, Matthew Fairbairn, Michael Norrish, Peter Sewell, Michael Smith, and Keith Wansbrough. Rigorous specification and conformance testing techniques for network protocols, as applied to TCP, UDP, and sockets. In *Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 265–276, New York, NY, USA, 2005. ACM.

[BFN+06] Steve Bishop, Matthew Fairbairn, Michael Norrish, Peter Sewell, Michael Smith, and Keith Wansbrough. Engineering with logic: HOL specification and symbolic-evaluation testing for TCP implementations. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 55–66, New York, NY, USA, 2006. ACM.

[BFN+07] Steve Bishop, Matthew Fairbairn, Michael Norrish, Peter Sewell, Michael Smith, and Keith Wansbrough. TCP, UDP, and sockets: Rigorous and experimentally-validated behavioural specification: Volume 3: The service-level specification. Technical report, University of Cambridge, 2007.

[BHL+96] J. Pillip Bowen, C. A. R. Hoare, Hans Langmaack, Ernst-Rüdiger Olderog, and Anders. P. Ravn. A ProCoS II project final report: ESPRIT Basic Research project 7071. *Bulletin of the European Association for Theoretical Computer Science*, 59, June 1996.

[BHW06] Gerd Beuster, Niklas Henrich, and Markus Wagner. Real world verification — experiences from the Verisoft email client. In *Proceedings of the Workshop on Empirical Succesfully Computerized Reasoning (ESCoR 2006)*, 2006.

[BJK+03] Sven Beyer, Chris Jacobi, Daniel Kroening, Dirk C. Leinenbach, and Wolfgang J. Paul. Instantiating uninterpreted functional units and memory system: Functional verification of the VAMP. In *Correct Hardware Design and Verification Methods, 12th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2003, L'Aquila, Italy, October 21-24, 2003, Proceedings*, volume 2860 of *LNCS*, pages 51–65. Springer, 2003.

[BJK+05] Sven Beyer, Christian Jacobi, Daniel Kröning, Dirk C. Leinenbach, and Wolfgang J. Paul. Putting it all together - formal verification of the VAMP. *International Journal on Software Tools for Technology Transfer*, 2005.

[Bog07] Sebastian Bogan. Academic System — Demo, 2007. Talk given at the 3rd German Verification Day, Berlin, July 1, 2007, http://www-wjp.cs.uni-sb.de/leute/private_homepages/sebastian/Demo2.mov.

[Bog08a] Sebastian Bogan. C0 implementation of the SOS. http://www.verisoft.de/VerisoftRepository.html, 2008. to appear.

[Bog08b] Sebastian Bogan. Isabelle/HOL specification of the FAT32 device driver. Technical Report 4, Saarland University, Saarbrücken, 2008. (included in [Bog08c]).

[Bog08c] Sebastian Bogan. Isabelle/HOL specification of the SOS. http://www.verisoft.de/.rsrc/VerisoftRepository/sos-trunk-r22974.tar.gz, 2008.

[Bog08d] Sebastian Bogan. The two-way handshake within SOS*. Technical Report 5, Saarland University, Saarbrücken, 2008. (included in [Bog08c]).

[Cai06] Yiwen Cai. Verification of TCP open and close phase. Master's thesis, Saarland University, Saarbrücken, 2006.

[Cro82] David H. Crocker. RFC 822: Standard for the format of ARPA Internet text messages, 1982.

[Dal06] Iakov Dalinger. *Formal Verification of a Processor with Memory Management Units*. PhD thesis, Saarland University, 2006.

[DHP05] Iakov Dalinger, Mark A. Hillebrand, and Wolfgang J. Paul. On the verification of memory management mechanisms. In D. Borrione and Wolfgang J. Paul, editors, *Correct Hardware Design and Verification Methods, 13th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2005, Saarbrücken, Germany, October 3-6, 2005, Proceedings*, LNCS. Springer, 2005.

[Dör06] Jan Dörrenbächer. 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.

[EKD+07] Kevin Elphinstone, Gerwin Klein, Philip Derrin, Timothy Roscoe, and Gernot Heiser. Towards a practical, verified kernel. In *11th Workshop on Hot Topics in Operating Systems*, page 6, San Diego, CA, USA, may 2007.

[GHLP05] Mauro Gargano, Mark A. Hillebrand, Dirk C. Leinenbach, and Wolfgang J. Paul. On the correctness of operating system kernels. In Joe Hurd and Thomas F. Melham, editors, *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005, Oxford, UK, August 22-25, 2005, Proceedings*, LNCS. Springer, 2005.

[Grü03] Andreas Grünbacher. POSIX Access Control Lists on Linux. In *USENIX Annual Technical Conference, FREENIX Track*, pages 259–272, 2003.

[HEK+07] Gernot Heiser, Kevin Elphinstone, Ihor Kuz, Gerwin Klein, and Stefan M. Petters. Towards trustworthy computing systems: Taking microkernels to the next level. *ACM SIGOPS Operating Systems Review*, 41(3):9, 2007.

[HHF+05] Hermann Härtig, Michael Hohmuth, Norman Feske, Christian Helmuth, Adam Lackorzynski, Frank Mehnert, and Michael Peter. The Nizza secure-system architecture. In *Proceedings of the 1st International Conference on Collaborative Computing: Networking, Applications and Worksharing*, 2005.

[Hil05] Mark A. Hillebrand. *Address Spaces and Virtual Memory: Specification, Implementation, and Correctness.* PhD thesis, Saarland University, Saarbrücken, 2005.

[HLS+96] Dieter Hutter, Bruno Langenstein, Claus Sengler, Jörg H. Siekmann, Werner Stephan, and Andreas Wolpers. Deduction in the verification support environment (VSE). In Marie-Claude Gaudel and Jim Woodcock, editors, *FME '96: Industrial Benefit and Advances in Formal Methods, Third International Symposium of Formal Methods Europe, Oxford, UK, March 18-22, 1996*, volume 1051 of *LNCS*, pages 268–286. Springer, 1996.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[Hol08] HOL automated proof system for higher order logic. `http://hol.sourceforge.net/`, 2008.

[HRP05] Mark A. Hillebrand, Tom In der Rieden, and Wolfgang J. Paul. Dealing with I / O devices in the context of pervasive system verification. In *ICCD '05: Proceedings of the 2005 International Conference on Computer Design*, pages 309–316, 2005.

[HT05] Michael Hohmuth and Hendrik Tews. The VFiasco approach for a verified operating system. In *2nd ECOOP Workshop on Programm Languages and Operating Systems*, July 2005.

[HTS02] Michael Hohmuth, Hendrik Tews, and Shane G. Stephens. Applying source-code verification to a microkernel: the VFiasco project. In *EW10: Proceedings of the 10th workshop on ACM SIGOPS European workshop*, pages 165–169. ACM, 2002.

[IEE85] IEEE. ANSI/IEEE std. 754, 1985 edition. IEEE standard for binary floating-point arithmetic, 1985.

[IEE04] IEEE. 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.

[Jac88] Van Jacobson. Congestion avoidance and control. In *SIGCOMM '88: Symposium proceedings on Communications architectures and protocols*, pages 314–329, New York, NY, USA, 1988. ACM.

[JB05] Christian Jacobi and Christoph Berg. Formal verification of the VAMP floating point unit. *Journal of Formal Methods in System Design*, 26(3):227–266, 2005.

[JH07] Rajeev Joshi and Gerard J. Holzmann. A mini challenge: Build a verifiable filesystem. *Formal Aspects of Computing*, 19(2):269–272, 2007.

[KMM00] Matt Kaufmann, J. Strother Moore, and Panagiotis Manolios. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[KP07] Steffen Knapp and Wolfgang J. Paul. Pervasive verification of distributed realtime systems. In M. Broy, J. Grünbauer, and C. A. R. Hoare, editors, *Software System Reliability and Security*,

volume 9 of *NATO Security Through Science Series. Sub-Series: Information and Communication*. IOS Press, 2007.

[L407] The L4 μ-kernel family. http://os.inf.tu-dresden.de/L4/impl.html, 2007.

[Lam02] Leslie Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[Lei08] Dirk C. Leinenbach. *Compiler Verification in the Context of Pervasive System Verification*. PhD thesis, Saarland University, Saarbrücken, 2008.

[LNRS07] Bruno Langenstein, Andreas Nonnengart, Georg Rock, and Werner Stephan. Verification of distributed applications. In Francesca Saglietti and Norbert Oster, editors, *Computer Safety, Reliability, and Security, 26th International Conference, SAFECOMP 2007, Nuremberg, Germany, September 18-21, 2007*, volume 4680 of *LNCS*, pages 315–328. Springer, 2007.

[LP08] Dirk C. Leinenbach and Elena Petrova. Pervasive compiler verification — From verified programs to verified systems. In *3rd International Workshop on Systems Software Verification*, 2008.

[LPP05] Dirk C. Leinenbach, Wolfgang J. Paul, and Elena Petrova. 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*, 2005.

[Mic07] Microsoft. Microsoft extensible firmware initiative FAT32 file system specification. http://www.microsoft.com/whdc/system/platform/firmware/fatgen.mspx, 2007.

[MMFR96] Matthew Mathis, Jamshid Mahdavi, Sally Floyd, and Allyn Romanow. RFC 2018: TCP selective acknowledgment options (SACK), October 1996.

[Moo02] J. Strother Moore. A grand challenge proposal for formal methods: A verified stack. In *10th Anniversary Colloquium of UNU/IIST*, pages 161–172, 2002.

[Net08] NETSEM: Rigorous semantics for real systems. http://www.cl.cam.ac.uk/~pes20/Netsem/, 2008.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[NYS07] Zhaozhong Ni, Dachuan Yu, and Zhong Shao. Using XCAP to certify realistic systems code: Machine context management. In Klaus Schneider and Jens Brandt, editors, *Theorem Proving in Higher Order Logics, 20th International Conference, TPHOLs 2007, Kaiserslautern, Germany, September 10-13, 2007, Proceedings*, pages 189–206. LNCS, September 2007.

[ORS92] Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *Automated Deduction - CADE-11, 11th International Conference on Automated Deduction*, volume 607 of *LNCS*, pages 748–752. Springer, 1992.

[Pet07] Elena Petrova. *Verification of the C0 Compiler Implementation on the Source Code Level*. PhD thesis, Saarland University, Saarbrücken, 2007.

[Pos80] Jonathan B. Postel. RFC 768: User datagram protocol (UDP), 1980.

[Pos81a] Jonathan B. Postel. RFC 791: Internet protocol (IP), 1981.

[Pos81b] Jonathan B. Postel. RFC 793: Transmission control protocol (TCP), 1981.

[Pos82] Jonathan B. Postel. RFC 821: Simple mail transfer protocol (SMTP), 1982.

[PRS+01] Birgit Pfitzmann, James Riordan, Christian Stüble, Michael Waidner, and Arnd Weber. The PERSEUS system architecture. In Dirk Fox, Marit Köhntopp, and Andreas Pfitzmann, editors, *VIS 2001, Sicherheit in komplexen IT-Infrastrukturen*, pages 1–18. Vieweg, 2001.

[Qmu07] QEMU open source processor emulator. `http://fabrice.bellard.free.fr/qemu/about.html`, 2007.

[Rie08] Tom In der Rieden. *CVM — A Formally Verified Framework for Microkernel Programmers*. PhD thesis, Saarland University, Saarbrücken, 2008. to appear.

[RNS08] Tom Ridge, Michael Norrish, and Peter Sewell. A rigorous approach to networking: TCP, from implementation to protocol to service. In *FM 2008: Formal Methods, 15th International*

Symposium on Formal Methods, Turku, Finland, May 26-30, 2008, Proceedings, 2008.

[RT08] Tom In der Rieden and Alexandra Tsyban. CVM - A verified framework for microkernel programmers. In *3rd International Workshop on Systems Software Verification*, 2008.

[Sch05] Norbert W. Schirmer. *Verification of Sequential Imperative Programs in Isabelle/HOL*. PhD thesis, Technical University of Munich, 2005.

[SDN+04] Jonathan S. Shapiro, M. Scott Doerrie, Eric Northup, Swaroop Sridhar, and Mark S. Miller. Towards a verified, general-purpose operating system kernel. In G. Klein, editor, *Proc. NICTA FM Workshop on OS Verification. Technical Report 0401005T-1*, pages 1–19. National ICT Australia, 2004.

[Sha06] Andrey Shadrin. Design and implementation of the portmapper and RPC primitives in the context of the SOS. Master's thesis, Saarland University, Saarbrücken, 2006.

[Smi96a] Mark A. Smith. Formal verification of communication protocols. In Reinhard Gotzhein and Jan Bredereke, editors, *Formal Description Techniques IX: Theory, application and tools, IFIP TC6 WG6.1 International Conference on Formal Description Techniques IX / Protocol Specification, Testing and Verification XVI, Kaiserslautern, Germany, 8-11 October 1996*, volume 69 of *IFIP Conference Proceedings*, pages 129–144. Chapman & Hall, 1996.

[Smi96b] Mark A. Smith. *Formal Verification of TCP and T/TCP*. PhD thesis, Massachusetts Institute of Technology, 1996.

[Spi92] J. Michael Spivey. *The Z Notation: A reference manual*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1992.

[Spi98] Katharina Spies. *Eine Methode zur formalen Modellierung von Betriebssystemkonzepten*. PhD thesis, Technical University of Munich, 1998.

[SR02] Mark A. Smith and K. K. Ramakrishnan. Formal specification and verification of safety and performance of TCP selective acknowledgement. *IEEE/ACM Trans. Netw.*, 10(2):193–207, 2002.

[ST08] Artem Starostin and Alexandra Tsyban. Formal verification of operating system microkernel primitives. In *3rd International Workshop on Systems Software Verification*, 2008.

[Ste93] W. Richard Stevens. *TCP/IP Illustrated: The Protocols*, volume 1. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1993.

[SW00] Jonathan S. Shapiro and Sam Weber. Verifying the EROS confinement mechanism. In *SP '00: Proceedings of the 2000 IEEE Symposium on Security and Privacy*, pages 166–176. IEEE, 2000.

[Tan01] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, second edition, 2001.

[Tew07a] Hendrik Tews. Formal methods in the Robin project: Specification and verification of the Nova microhypervisor. http://robin.tudos.org/publications/short.ps, 2007.

[Tew07b] Hendrik Tews. Micro hypervisor verification: Possible approaches and relevant properties. http://robin.tudos.org/publications/hyperveri.pdf, 2007.

[Tom67] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11(1), 1967.

[Tsy08] Alexandra Tsyban. *Formal Verification of a Framework for Microkernel Programmers*. PhD thesis, Saarland University, Saarbrücken, 2008. to appear.

[Tun07] Tun / Tap. http://vtun.sourceforge.net/, 2007.

[Tve08] Sergey Tverdyshev. *Formal Verification of Gate-Level Computer Systems*. PhD thesis, Saarland University, Saarbrücken, 2008. to appear.

[Uar07] UART. http://www.national.com/ds/PC/PC16550D.pdf, 2007.

[Ver07] The Verisoft project. http://www.verisoft.de/, 2007.

[YTEM06] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using model checking to find serious file system errors. *ACM Transactions on Computer Systems*, 24(4):393–423, 2006.

Index

R

S