Formal Specification and Verification of a Microkernel

Dissertation

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

Jan Dörrenbächer

jandb@wjpserver.cs.uni-saarland.de

Saarbrücken, November 2010

Tag des Kolloquiums:	23.11.2010
Dekan:	Prof. Dr. Holger Hermanns
Vorsitzender des Prüfungsausschusses:	Prof. Dr.-Ing. Philipp Slusallek
1. Berichterstatter:	Prof. Dr. Wolfgang J. Paul
2. Berichterstatter:	Prof. Dr. Bernhard Beckert
3. Berichterstatter:	PD Dr. Werner Stephan
akademischer Mitarbeiter:	Dr. Eyad Alkassar

Eidesstattliche Versicherung

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, November 2010

Danksagung

Ich möchte mich an dieser Stelle bei allen bedanken, die zum Gelingen dieser Arbeit beigetragen haben.

An erster Stelle gilt dieser Dank meiner Familie, deren Unterstützung ich mir immer sicher sein konnte und die mir stets mit Rat und Tat zur Seite stand.

Herrn Prof. Wolfgang Paul danke ich für die Möglichkeit zur Promotion und für die wissenschaftliche Betreuung der Arbeit.

Ein großer Dank geht an meine Kollegen für die interessanten und fruchtbaren, manchmal aber auch anstrengenden Diskussionen, die zum Gelingen dieser Arbeit von unschätzbarem Wert waren. Im Besonderen Eyad Alkassar, Sebastian Bogan, Matthias Daum, Mark Hillebrand, Norbert Schirmer und Burkhardt Wolff. Insgesamt möchte ich mich bei allen Mitarbeitern am Lehrstuhl Paul für die gute Arbeits- und Feierabendatmosphäre bedanken.

Nicht zuletzt möchte ich mich bei meinen Freunden bedanken, die mich nicht nur tatkräftig unterstützt haben, sondern mich stets aufbauten und für die erforderliche Abwechslung sorgten.

Diese Arbeit wurde teilweise im Rahmen des Verisoft Projekts vom Bundesministerium für Bildung und Forschung (BMBF) unter dem Förderkennzeichen 01 IS C38 gefördert.

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

Abstract

This thesis basically splits up into two parts. The first part introduces the abstract model of the VAMOS kernel. The VAMOS kernel provides the infrastructure for process and memory management, priority-based round-robin scheduling, communication with external devices, as well as inter-process communication. In the second part, we formulate a simulation theorem between the abstract VAMOS model and the concrete VAMOS implementation. The crucial points of the theorem are, on the one hand, the abstraction relation connecting the datastructures of the implementation with those of the model and, on the other hand, the implementation invariant formulating validity statements on the datastructures. Besides the exact formal definitions of the abstraction relation and the implementation invariant, we prove substantial parts of the simulation theorem.

This work is part of the Verisoft project which aims at the pervasive formal verification of computer systems. For the modelling and the verification of the VAMOS kernel this entails the integration of various computational models, for instance, *Communicating Virtual Machines* (CVM) encapsulating the hardware-specific low-level functionality, and devices.

The models and proofs presented in this thesis are formalized in the uniform logical framework of the interactive theorem prover Isabelle/HOL, and hence, it is rigorously checked that all verification results fit together.

Zusammenfassung

Die vorliegende Arbeit teilt sich im Wesentlichen in zwei Teile auf. Im ersten Teil wird das abstrakte Modell des VAMOS-Kernels vorgestellt. Der VAMOS-Kernel liefert die Infrastruktur für Prozess- und Speicherverwaltung, prioritäts-basiertes Round-Robin-Scheduling, Kommunikation mit externen Geräten, sowie Interprozesskommunikation. Im zweiten Teil der Arbeit formulieren wir ein Simulationstheorem zwischen dem abstrakten VAMOS-Modell und der konkreten VAMOS-Implementierung. Kernpunkte dieses Theorems sind zum einen die Abstraktionsrelation, die die Datenstrukturen der Implementierung mit denen des Modells in Beziehung setzt und zum anderen die Implementierungsinvariante, die Gültigkeitsaussagen über die Datenstrukturen trifft. Neben den exakten Definitionen der Abstraktionsrelation und der Implementierungsinvariante, werden wesentliche Teile dieses Simulationstheorems bewiesen.

Die Arbeit wurde im Rahmen des Verisoft Projekts angefertigt, das die durchgängige formale Verifikation von Computersystemen zum Ziel hat. Für die Modellierung und Verifikation des VAMOS-Kernels hat dies zur Folge, dass diverse Berechnungsmodelle integriert werden müssen, unter anderem das Gerätemodell und Communicating Virtual Machines (CVM), das die

hardwarespezifische und systemnahe Funktionalität kapselt.

Alle Modelle und Beweise, die in dieser Arbeit vorgestellt werden, sind in dem interaktiven Theorembeweiser Isabelle/HOL formalisiert worden, womit sichergestellt ist, dass alle Ergebnisse der Verifikation zusammenpassen.

Ausführliche Zusammenfassung

Die zunehmende Komplexität und Vernetzung heutiger Computersysteme stellt immer höhere Anforderungen an die Verlässlichkeit und Fehlerfreiheit der Systeme und ihrer Komponenten. Der sicherste Ansatz, um schon beim Systementwurf konzeptionelle und menschliche Unzulänglichkeiten auszumerzen, stellt die *durchgängige formale Verifikation* dar.

Das Verisoft Projekt, in dessen Rahmen diese Arbeit entstanden ist, verfolgt genau diesen Ansatz. Zu diesem Zweck wird jede einzelne Schicht des betrachteten Systems durch ein mathematisches Modell beschrieben. Durch sogenannte Simulationsbeweise wird bewiesen, dass die jeweiligen Schichten tatsächlich diesen Modellen genügen. Durchgängigkeit bedeutet, dass zum Schluss eine Korrektheitsaussage über das Gesamtsystem steht, die alle Schichten einbezieht, wobei Annahmen in höheren Schichten durch die darunterliegenden entlastet werden. Dadurch, dass alle Beweise durch einen Computer überprüft werden, sprechen wir von formaler Verifikation.

Betriebssysteme gehören unzweifelhaft zu den essentiellen Komponenten von Computersystemen und tragen somit wesentlich zur Verlässlichkeit der Gesamtsysteme bei.

In dieser Arbeit beschreiben wir die Modellierung und formale Verifikation des Betriebssystemkernels VAMOS in einem durchgängigen Kontext. Die Funktionalität des VAMOS-Kernels umfasst Prozess- und Speicherverwaltung, prioritäts-basiertes Round-Robin-Scheduling, Kommunikation mit externen Geräten, sowie Interprozesskommunikation. Ein mathematisches Modell liefert die Spezifikation des Kernels. Dass die VAMOS-Implementierung tatsächlich diesem Modell genügt wird duch einen Simulationsbeweis gezeigt. Um die Datenstrukturen der Implementierung mit denen des Modells in Beziehung zu setzen, definieren wir eine Abstraktionsrelation. Desweiteren definieren eine Implementierungsinvariante, die Gültigkeitsaussagen über die Datenstrukturen trifft. Basierend darauf werden wesentliche Teile des Simulationstheorems bewiesen.

Die Tatsache, dass das VAMOS-Modell eine gültige Spezifikation des VAMOS-Kernels darstellt, liefert die Grundlage dafür, dass das Modell verlässlich in höheren Schichten benutzt werden kann. So werden zum Beispiel Definitionen des VAMOS-Modells direkt von Bogan [15] benutzt, um ein einfaches Betriebssystem zu spezifizieren. Desweiteren wird in [25] der Beweis erbracht, dass der VAMOS Scheduler fair ist.

Zusammengefasst umfasst die Arbeit drei Teile. Im ersten Teil beschreiben wir allgemein den Ansatz zur durchgängigen formalen Verifikation im Verisoft Projekt. Desweiteren führen wir die Berechnungsmodelle ein, die in das VAMOS-Modell integriert werden. Unter anderem ein generisches Modell für externe Geräte und *Communicating Virtual Machines* (CVM), das die hardwarespezifische und systemnahe Funktionalität kapselt.

Im zweiten Teil definieren wir das VAMOS-Modell als Übergangssystem, das die Basis für den Simulationsbeweis darstellt, mit dem wir uns im dritten Teil dieser Arbeit beschäftigen.

Alle Modelle und Beweise, die in dieser Arbeit vorgestellt werden, sind in dem interaktiven Theorembeweiser Isabelle/HOL formalisiert worden, womit sichergestellt ist, dass alle Ergebnisse der Verifikation zusammenpassen.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Contents

Without a doubt, there is a steadily growing impact of computers on our daily life. A couple decades ago, only a few people – mostly for professional reasons – were confronted with the phenomenon "computer". That has certainly changed. Meanwhile, while driving a car various programs – from the entertainment system to the airbag control – assist us and usually ensure that we reach our destination safely. We use computers for banking transactions and the state-of-the-art-technology already deals with computers assisting physicians during difficult surgeries. Even the leisure sector observes an increasing entry of technology. It seems that nowadays nearly nobody is able to ride a bike without tracking the route by GPS or run without listening to music. Not to mention that the thereby achieved performances are immediately shared with the rest of the world via social networks like Facebook.

A reason for the triumphal procession of these new technologies is that – most of the time – they ease our lives and work properly. And when sometimes things go wrong, this is annoying, but usually no severe or even live-threatening consequences are entailed.

However, there are definitely computer systems where malfunctions could result in dramatic consequences. One just has to think of systems flying airplanes, controling nuclear power plants or pacemakers dictating the beat of human hearts. Thanks to outstanding engineering skills, until now, we remained widely spared from any severe scenarios. Nevertheless, it is no longer a rarity that the media reports on flaws in computer systems with far-reaching consequences. At the beginning of the year 2010, for instance, nearly 30 million bank customers in Germany were incapable of withdrawing money at cash points. The reason for that could be traced back to a programming error in a security chip on the bank cards. Another software bug was recently reported in conjunction with the acceleration mechanism in vehicles of Toyota's model Prius. Such occurrences certainly scratch on the image of the companies concerned, without mentioning the financial drawbacks.

There is some indication that such reports will accumulate in the future. That is not because the system engineers today are worse than in the past or became lazy in testing their systems. The reason is simply founded in the increasing complexity of such systems which makes it more and more difficult for human beings to keep the overview. Furthermore, the new technologies reach out into new areas. A few years ago it would have been unthinkable that computers assist in medical surgeries or computer-controlled robots even accomplish them. Today we accept computer-assisted surgical planning as normal.

If, however, such a system fails, severe consequences are possible. Therefore, it is expected that the demand for truly robust, safe, and secure systems will constantly grow in the next years. In order to achieve this, a detailed and profound analysis of such computer systems is required. Previous approaches to satisfy the desire for reliable systems often focused on exhaustive testing of the systems. However, only limited reliability can be achieved this way, because testing never proves the absence of errors and often relies on assumptions on other system components (e. g., underlying system software) which may behave differently than expected.

A different approach constitutes the pervasively formalization and verification of computer systems. In [55], J. S. Moore, principal researcher of the CLI stack project [12], declares the formal verification of a "practical computing system" as a grand challenge problem.

The Verisoft projects exactly tackles this problem and aims at the pervasive modeling, implementation, and verification of a complete computer system reaching from gate-level hardware to applications running on top of an operating system. Certainly, a corner stone of this system is the microkernel constituting the interface between the hardware and the operating system.

In this thesis, we describe the modeling and verification of the microkernel in a pervasive context.

1.1 Document Outline

In the remainder of this chapter, we discuss related work and present the motivation and contribution of this thesis. It concludes with some basic notation. Chapter 2 describes background information about the context and prerequisites of this thesis. It introduces the languages C0 and assembly which are used to implement the kernel and sketches the verification approach. Furthermore, it presents the computational models for the devices and the communicating virtual machines (CVM). The chapter concludes with an overview on the design and implementation of the VAMOS kernel.

Chapter 3 gives a bird's eye view on the abstract system layer, which is concretized in Chapter 4 with the model of the VAMOS kernel.

Chapter 5 states the VAMOS correctness theorem and Chapter 6 deals with the actual proofs.

We conclude in Chapter 7 with a summary and an outlook on future work.

1.2 Related Work

Early attempts regarding the formal verification of operating system kernels already took place in the 1970s and early 1980s. The Provably Secure Operating System (PSOS) comprised hardware and software and aimed at a useful general purpose operating system with demonstrable security properties, as a retrospective report [58] from 2003 describes. Even if no code proofs were undertaken, PSOS introduced concepts for OS verification and aimed at applying formal methods throughout the whole implementation of the OS.

The verification of a kernel supporting threads, a capability-based access control, virtual memory, and device accesses, was the aim of the UCLA Secure Unix project [79]. It relied on the assumptions that the compiler and the hardware work correct. They report that about 90% of the specifications and 20% of the code proofs were completed.

The KIT kernel [11, 12, 55] is the first kernel that can be considered as formally verified. However, it can only be referred as groundbreaking to the area of pervasive operating-system verification. The main difference to more recent verification attempts of "real software" is that it relies on a LISP execution model that is nowadays considered fairly abstract. Moreover, the corner stone theorem of this work is on memory separation of processes.

While they pioneered the field, none of the above projects comes up with a realistic operating system kernel (written in an imperative language) with full implementation proofs. The main reason for this were the missing or to a large extend underdeveloped tool environments. It took until the early 2000s to remedy this deplorable state of affairs, before projects again engaged in increasing the confidence in system software by means of formal methods. Various approaches cover only specific aspects of an operating system kernel and apply formal methods to it. Though these approaches are promising, they would not produce a formally verified operating system kernel. Thus, we do not consider them in the context of this thesis but rather concentrate on the approaches examining complete kernels. Subsequently, we refer to the most prominent projects. For the interested reader, we also recommend the excellent and comprehensive overview by Klein [43].

Most of the projects deal with the verification of an operating system kernel by following the single-layer approach. They apply formal methods to the kernel layer with the aim to show, that this layer fulfills a certain specification and (possibly) some properties. In contrast to the pervasive approach (we are aiming at), the lower layers are considered as correct and respective properties are assumed to hold. Additionally, while defining the particular specifications, it is usually not ensured (at least not apparently) how these specifications could be integrated or used in higher layers, for instance, in order to specify an operating system.

An exponent of the single-layer approach is the MASK project [54] standing for *mathematically analyzed separation kernel.* It guarantees a separation property that is provided by construction in the highest-level model, and by multiple formal refinement proofs down to a low-level design, which is close to an implementation. Code proofs, however, were not attempted.

The VFIASCO project [39] aims at the verification of the microkernel Fiasco implemented in a subset of C++. Besides model checking of basic safety properties in Spin [31] for a strongly limited and simplified version of Fiasco IPC, three properties concerning ready and waiting lists have been verified in PVS [67]. However, this work is less than full code verification since C++ code was transformed (only for this purpose) into PVS without defining a formal semantics of the used C++ subset. Even some functions were only described by their semantical effects, and hence the obtained implementation model is rather a specification.

With EROS/Coyotos [70], Shapiro and Weber define a capability based kernel system and prove certain security policies to hold. For the successor project COYOTOS, a complete formal specification of the microkernel and the used programming language BitC has been finished. No formal proofs have been reported yet.

Other projects, for instance, EMBEDDED DEVICE [37, 36], were successful in the verification but did not reach down to the implementation level. In the FLINT project [60], an assembly code verification framework is developed and code for context switching on a x86 machine was formally proven to be correct. The presented code is in functionality and size very similar to the process switch implemented in the CVM, and also reported verification efforts are comparable to those in Verisoft. Although a verification logic for assembler code is presented, no integration of results into high level programming languages has been undertaken yet.

Besides the verification of the VAMOS kernel, the Verisoft project also dealt with the correctness of the real-time operating system OLOS. The functional correctness of this tiny real-time operating system also relies on the Cvm layer and has completely been shown [29]. Compared to the Vamos kernel, however, the complexity of Olos with only three rather low-level system calls is much lower.

Further verification projects are L4.verified and Verisoft XT both aiming at the complete code-level verification of operating system kernels.

L4.verified. L4.verified focuses on the seL4 kernel [35], which is based on the ARM11 platform. It is a third generation microkernel of L4 provenance and comprises 8,700 lines of C and 600 lines of assembly code. Relying on [42, 44], the L4.verified project is providing a machine-checked proof of the functional correctness of the seL4 microkernel with respect to a high level, formal description of its expected behaviour.

The lowest level of the verification and of the refinement is a high-performance C and ARM assembly implementation of the seL4 microkernel. The next level up, the low-level design, is a detailed, executable specification of the intended behaviour of the C implementation. This executable specification is derived automatically from a prototype of the kernel which was written in the high-level, functional programming language Haskell. While trying to avoid messy specifics of how data structures and code are optimized in C, the executable specification still represents a pretty concrete view on the kernel implementation. For instance, it still contains doubly-linked lists, i. e., pointer structures. The highest refinement layer is the high-level design, an abstract, operational specification of the kernel. Usually, this is accompanied with a high level of abstraction. It is reported, however, that this layer precisely describes argument formats, encodings and error reporting. Thus, for instance, some of the C-level size restrictions become visible on this level. On the other hand, the specification uses non-determinism in order to avoid an explicit description of the system. For example, neither a scheduling policy is defined at the abstract level nor the correctness of the interrupt controller management is modeled in detail. The refinement proofs are machine-checked in the interactive theorem prover Isabelle/HOL.

In [18, 30], they also defined an abstract access control model of seL4 that captures how capabilities (the kernel's access control mechanism) are distributed in the system and showed the isolation of security domains based on this model. The access control model, however, is not formally connected to the high-level design of seL4 which is used in the refinement proof.

Without doubt, the results of the L4.verified project are very promissing. However, the following issues have to be observed: Firstly, the approach assumes the correctness of the C compiler, the assembly code, and the hardware. Furthermore, in contrast to the Verisoft project, there is no obvious technology to integrate and combine all these entities into one coherent theory. Secondly, the abstraction level of the high-level design seems to be comparatively low (which reduces the effort regarding the refinement proof) and unsuitable to show, for instance, properties of the access control mechanism. Instead, these proofs rely on a (probably) more abstract model which is not formally connected to the high-level design in the refinement proof. Thirdly, the abstract models are not applied, in the sense that they are used to specify an operating system, for instance.

Based on these issues, we define the scope of our work. The model of the VAMOS kernel presented in this work is completely integrated into a model stack reaching from applications down to the hardware. Relying literally on definitions of the lower layers, on the one hand, the VAMOS model exports definitions to higher layers, on the other hand. Bogan [15], for instance, based his operating system model on VAMOS and uses definitions from it literally.

The seamless integration into the model stack is not the only remarkable difference between the high-level design of seL4 and the VAMOS model. Similar as above, the VAMOS model serves as basis for the functional correctness of the VAMOS implementation. Thereby, the model establishes an abstraction level that is high enough, such that it can be used to show a pretty complex property of the VAMOS kernel, namely the fairness of the VAMOS scheduler [25, 27].

Verisoft XT. The VERISOFT XT project aims at the complete code-level verification of the following operating system kernels:

PIKEOS kernel [41]. This L4-derivate comprises 6,000 lines of code (loc) and is part of a commercial product available for Intel's x86, PowerPC, and ARM. The proofs are carried out by the VCC verification environment [21] (a descendant of the Spec# program-verification environment of Microsoft Research), which uses a trusted tool chain comprising the automatic verifier for concurrent C code *VCC*, the verification condition generator *Boogie* [16, 17], and the automated theorem prover *Z3* [56]. The supported C fragment is a large fragment of ANSI C.

Recent publications [8, 7] mainly introduce the tool chain and methodology. Apart from the verification of a simple system call which changes the priority of a thread, there are no reports yet on the portion of verified code.

HYPER-V [66]. The kernel comprises 100,000 lines of C and 5,000 lines of assembly code and is part of a commercial virtualization environment. The hypervisor turns a single real multi-processor x64 machine (with AMD SVM or Intel VMX virtualization extensions) into a number of virtual multi-processor x64 machines. The verification is realized by the VCC verification environment (see above); the supported C fragment is a large fragment of ANSI C. Recent publications [20, 22, 23] in this context mainly focus on the methodology of VCC and its application.

BABY HYPER-V. Alkassar *et al.* [5], report on the successful verification of the so-called baby hypervisor. In comparison to the HYPER-V, the baby hypervisor and the architecture it virtualizes are very simple because it only includes the initialization of the guest partitions and a simple shadow page table algorithm for memory virtualization. However, it played an important role of driving the development of the VCC technology and applying it to system verification.

The VCC technology [22] demands a high level of trust: The current VCC version uses an axiomatization of the execution model consisting of various axioms, which introduce some rather abstract concepts like concurrency and ownership of references. Though critical subproblems of the foundation are tackled by informal as well as formal proof methods, the integration into a uniform foundational theory is significantly less prioritized. In this respect, the tool chains, methodologies etc. are driven by the need to deal with the existing code that can only be changed, if errors have been revealed.

1.3 Motivation and Contribution

Summing up the last section, we can state that real system-code verification represents a grand challenge. Current approaches – including ours – compromise in one way or the other: the logical foundations are quite problematic, the computer architectures are simpler than industrial-strength processors, the code size is fairly small, or the underlying execution model of C and assembly makes severe simplifications. In addition, we have to deal with an ambiguous notion of functional correctness which mainly concerns the abstract model, the functional correctness relies on. Thus, we have to be aware of the following issues:

- is the abstract model confirmed by underlying models or is it based on the informal understanding of the lower layers and respective properties
- which level of abstraction provides the model and which parts of the concrete system are actually modeled
- is the model tailor-made to show a particular system property, or is it able to serve as basis for higher layers, e. g., an operating system model

Taking up these issues, the abstract model for the VAMOS kernel, which we present in this thesis, is completely integrated into a model stack reaching

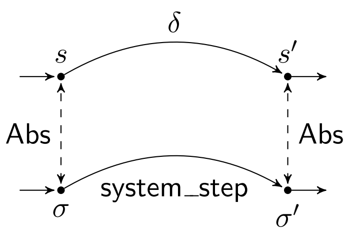
$$\begin{tikzcd} \arrow[r] & s \arrow[rr, bend left, "\delta"] & & s' \arrow[r] & {} \\ \arrow[r] & \sigma \arrow[u, dashed, "\mathsf{Abs}"] \arrow[rr, bend left, "\mathsf{system\_step}"'] & & \sigma' \arrow[u, dashed, "\mathsf{Abs}"'] \arrow[r] & {} \end{tikzcd}$$

Figure 1.1: Kernel step refinement (simplified)

from applications down to the hardware (cf. Section 2.1). While defining the model, we literally use definitions from the lower layers, for instance, in order to describe visible parts of the processor. The abstraction level of the VAMOS model is so high, that it is possible to show a pretty complex property of the VAMOS kernel, namely the fairness of the scheduler [25, 27]. In addition, Bogan [15] based his operating system model on VAMOS and uses definitions from it literally. The integration into the overall model stack certainly justifies the relevance of the VAMOS model. This relevance, however, will be further backed by a simulation proof establishing the relation between this abstract model and the concrete, fairly realistic implementation of the VAMOS kernel.

The availability of a hardware-processor model represents the foundational layer of our work. This model describes, at gate level, the transitions of the RISC processor VAMP. A small piece of software executed on the VAMP, called CVM, provides an abstract layer running concurrently a system process as well as a number of user processes. The implemented microkernel VAMOS is executed as this system process. Its implementation uses the C fragment *C0*, which can be translated into assembly code by a verified compiler [49, 48]. VAMOS provides a process scheduler, an infrastructure for communication with hardware devices, and message passing between processes. All software layers are formally specified; simulation relations correlate the adjacent layers such that eventually, all specification layers can be mapped down to our hardware-processor model. The whole model stack is formalized in the theorem prover Isabelle/HOL [61].

In more detail, CVM and the VAMOS implementation together realize an abstract transition δ by an implementation function system_step as shown in Figure 1.1. The transition function δ represents the execution of a non-empty portion of a user process including a possible process switch. The simulation proof establishes that the implementation with its underlying state σ indeed realizes the high-level state transition δ over the corresponding abstract state s, where abstract and concrete states are linked via a formally defined abstraction relation Abs.

To our knowledge, such a proof based on a model stack of this concrete level of detail and with such a clean, seamless logical foundation has been undertaken for the first time.

1.4 Preliminaries

The formalizations presented in this thesis are mechanized and checked within the generic interactive theorem prover *Isabelle*[62] [1]. Isabelle is called generic as it provides a framework to formalize various *object logics* that are declared via natural deduction style inference rules within Isabelle's meta-logic *Pure*. The object logic that we employ for our formalization is the higher order logic of *Isabelle/HOL*[61].

This thesis is written using Isabelle's document generation facilities, which guarantees that the presented theorems correspond to formally proven ones. We distinguish formal entities typographically from other text. We use a sans serif font for types and constants (including functions and predicates), e. g., replicate, a slanted serif font for free variables, e. g., x, and a slanted sans serif font for bound variables, e. g., x. Small capitals are used for data type constructors, e. g., Foo. HOL keywords are typeset in type-writer font, e. g., **let**.

As Isabelle's inference kernel manipulates rules and theorems at the Pure level the meta-logic becomes visible to the user and also in this article when we present theorems and lemmas. The Pure logic itself is intuitionistic higher order logic, where universal quantification is $\bigwedge$, implication is $\Longrightarrow$ and equality is $\equiv$. Nested implications like $P_1 \Longrightarrow P_2 \Longrightarrow P_3 \Longrightarrow C$ are abbreviated with $[\![P_1; P_2; P_3]\!] \Longrightarrow C$, where we refer to P_1, P_2, and P_3 as the premises and to C as the conclusion.

In the object logic HOL universal quantification is $\forall$, implication is $\longrightarrow$ and equality is $=$. Isabelle/HOL provides a library of standard types like Booleans, natural numbers, integers, total functions, pairs, lists, and sets as well as packages to define new data types and records. We only present them shortly in the following.

Functions. In HOL all functions are total. An unamed function can be specified using the λ-operator, e. g., $\lambda x.\ x$ is the identiy function. In general, we prefer curried functions over functions taking an n-tupel as argument, e. g., $f\ a\ b$ instead of $f(a, b)$. Note that function application binds tighter than any other operator, i. e., $f\ i + g\ j$ means $(f\ i) + (g\ j)$. Function update is $f(y := v) \equiv \lambda x.$ **if** $x = y$ **then** v **else** $f\ x$ and function composition is $f \circ g \equiv \lambda x.\ f\ (g\ x)$. Partial functions are usually formalized in HOL with an optional type. This type is a data type with two constructors, one to inject values of the base type, e. g., $\lfloor x \rfloor$, and the additional element $\bot$. With $\lceil y \rceil$, the original base value x can be obtained such that $y = \lfloor x \rfloor$. There

[1]For taming Isabelle's powerful document-generation mechanism in general, and in particular for the major part of this section, we are indebted to Schirmer *et al.* [69, 4]

is no base value x iff $y = \bot$. As HOL is a total logic the term $\lceil\bot\rceil$ is still a well-defined yet unspecified value. Partial functions can be represented by the type $'a \Rightarrow{} 'b$ option, abbreviated as $'a \rightharpoonup 'b$. For an update of a partial function, we write $f(y \mapsto x)$. For data types, we write the structural case distinction over some value v as **case** v **of** $\bot \Rightarrow g \mid \lfloor x\rfloor \Rightarrow f\ x$. Some data types might have many constructors, which are all treated in the same fashion in a certain situation. In this situation, we may write **case** v **of** Foo $\Rightarrow x \mid \_ \Rightarrow y$, which means value x, if v has the value Foo, and otherwise value y.

Sets and Intervalls. Sets come along with the standard operations for union, i. e., $A \cup B$, intersection, i. e., $A \cap B$ and membership, i. e., $x \in A$. The set image f ` A yields a new set by applying function f to every element in set A.

We denote the intervall of the numbers a and b excluding the endpoints by $\{a{<}..{<}b\}$, and including the endpoints by $\{a..b\}$. Furthermore, we write $a \leq c < b$ to denote that a number c is in the interval $\{a..{<}b\}$.

Lists. The syntax and the operations for lists are similar to functional programming languages like ML or Haskell. The empty list is [], with $x \cdot xs$ the element x is 'consed' to the list xs, the head of list xs is hd xs and the remainder, its tail, is tl xs. With xs @ ys list ys is appended to list xs. With map f xs the function f is applied to all elements in xs. The length of a list is length xs, the n-th element of a list can be selected with xs ! n and updated via $xs[n := v]$. With set xs we obtain the set of elements in list xs. Filtering those elements from a list for which predicate P holds is achieved by $[x{\in}xs\ .\ P\ x]$. Removing an element e from a list xs is also realized by filtering, i. e., $[x{\in}xs\ .\ x \neq e]$. The function takeWhile P xs yields elements from a list xs while a property P is true and then skips the remainder of the sequence. The function dropWhile P xs removes elements from a list xs while a property P is true and returns the remainder of the sequence. With replicate n e we denote a list that consists of n elements e. The function list_sum xs sums up the elements of list xs.

Records and Tupels. A record is constructed by assigning all of its fields, e. g., $(\!| \mathsf{fld}_1 = v_1, \mathsf{fld}_2 = v_2 |\!)$. Field fld_1 of record r is selected by $r.\mathsf{fld}_1$ and updated with a value x via $r(\!| \mathsf{fld}_1 := x |\!)$.

The first and second component of a pair can be accessed with the functions fst and snd. Tuples with more than two components are pairs nested to the right.

Chapter 2

Background

Contents

The present work was done in the context of the German Verisoft project, a large scale effort bringing together industrial and academic partners to push the state of the art in formal verification for realistic computer systems, comprising hard- and software. Our microkernel belongs to Verisoft's so-called *Academic System* (as opposed to systems in subprojects with partners from industry), which is a distributed computer system for the exchange and management of signed and encrypted e-mails. This computer system is designed as part of an open network with a number of trusted computers. While the system is open to receive e-mails from arbitrary computers in the network, each trusted computer uses identical hardware and the same operating-system software. As application software, we implemented and verified an e-mail client [10, 9], an e-mail server [45, 46], and a cryptography server that run on top of this operating system. These applications might be arbitrarily distributed over the trusted computers in the network.

This section continues with a short introduction of the implementation layers in our computer system. Furthermore, we summarize the general verification approach that has been developed for sequential programs within Verisoft in Section 2.2. A key feature of this approach is, that it can cope with mixed-language implementations as they frequently occur in system-level software. In our case, these languages are C0 and assembly. Assembly is used in the lowest software layer, a low-level microkernel. The device

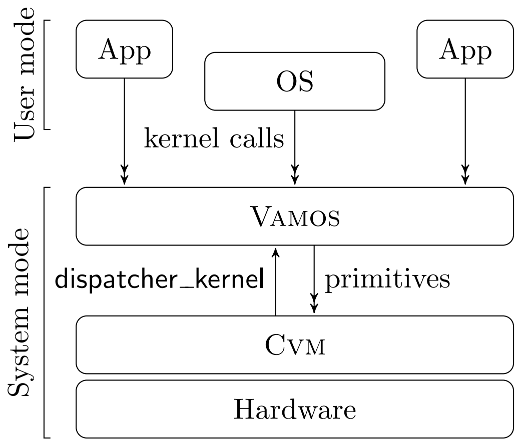

Figure 2.1: System stack

model in Section 2.3 together with the computational model *Communicating Virtual Machines* (CVM) introduced in Section 2.4, abstract from this lowest software layer. On top of it, Section 2.5 introduces the VAMOS microkernel which is completely implemented in C0. It relies on the CVM framework and represents the top-level microkernel with utilities like interprocess communication, memory management and process scheduling.

2.1 The Academic System: Software Stack

Figure 2.1 depicts the hard- and software layers of a single computer in the system. The lowest software layer is called *communicating virtual machines* (CVM). This layer encapsulates all the hardware-specific low-level kernel functionality, which uses inlined assembly. Technically, this software constitutes the interrupt-service routine of the processor. CVM's major task is memory virtualization and process separation. Hence, CVM includes a page-fault handler with a simple memory swapping facility [6]. The remaining functionality of our microkernel is implemented by the hardware-independent kernel part. CVM exports an interface of so-called *primitives* for the access to and manipulation of user processes to this kernel part. We have thereby established a solid framework for microkernel construction.

Using this framework, we have implemented our microkernel VAMOS in the C variant *C0* without inlined assembly. When an interrupt occurs, CVM preserves the old processor context, establishes a suitable C0 environment and calls the function dispatcher_kernel of VAMOS. For the manipulation of the user memory or registers, VAMOS may call the primitives of CVM. The return value of dispatcher_kernel determines which process resumes when the kernel execution finishes.

While CVM and VAMOS run in the privileged system mode of the proces-

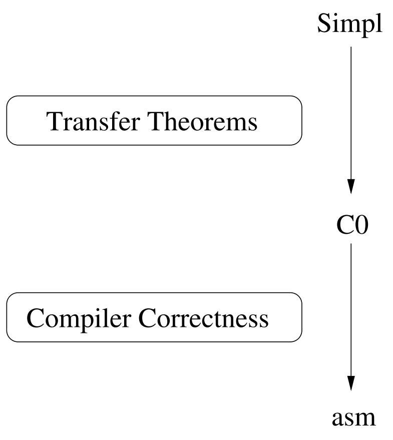

Figure 2.2: Semantic Stack

sor, processes run in the unprivileged user mode. In the figure, we labeled one process "OS" for the operating system and the others "App" as abbreviation for application (e. g., e-mail client or server). The OS process constitutes the highest layer of the operating system [15]. It features an advanced rights management with different users, implements a sophisticated access control to kernel services like process creation and provides further services like file-system and network access. All processes interact with the kernel via *kernel calls*. The instruction set architecture (ISA) provides a special instruction TRAP causing an exception, which is handled in VAMOS. VAMOS can examine and alter the state of the process using CVM primitives, thus identifying the process's specific request and storing the kernel's corresponding response.

2.2 The Academic System: Semantic Stack

As Figure 2.1 depicts, a computer system comprises a stack of hardware and software layers. Along the implementation layers, we assemble computational models which we use to specify and verify the system. Each of these layers introduce a higher abstraction level to improve the effectiveness of reasoning. In addition, these layers often employ different implementation languages. CVM, for example, provides the abstraction of separate processes with virtual memory, and is implemented in a mixture of C0 and inlined assembly. The reason for two implementation languages is apparent: C0 is a C-like high-level programming language and can be comfortably employed at a reasonable abstraction layer for most programming tasks. Specific low-level functionality, however, cannot be expressed in C0. For these tasks, we employ the assembly language.

In a large-scale verification project, the key to success is a verification approach that follows the system design. Applied to the Verisoft project, this means that verification does not only extend along the software levels but also takes into account the mixed-language architecture [3]. The verification approach is two-dimensional: The first axis, called system stack, traces the different implementation layers. Along the implementation layers, we assemble computational models which we use to specify and verify the system. Figure Figure 2.2 depicts the so-called *semantics stack* [4] combining all the different language semantics.

The assembly language marks the bottom layer, followed by C0 and Simpl at the top. Despite of the former two, Simpl is a generic programming language model for sequential imperative programming languages. This generic model provides the basis for the versatile verification environment Isabelle/Simpl. The major challenge of a pervasive verification is to develop a coherent theory that allows to transfer the correctness results down to the lowest layer. Thus, an important result of the Verisoft project is the correctness theorem of a simple non-optimizing compiler that links C0 and the assembly language [49, 63, 50]. Furthermore, transfer theorems [69] establish an embedding of C0 into the generic programming model Simpl.

In the following sections, we present the – for microkernel programming relevant – implementation languages C0 and VAMP assembly. The chapter concludes with the introduction of the verification environment Isabelle/Simpl which is used to conduct the actual proofs.

In the scope of this work, we solely focus on the correctness proofs performed in Isabelle/Simpl. Mapping down the results achieved would rely on the previously mentioned transfer theorems and the compiler correctness. Starostin [72] already successfully applied the semantic stack in the context of the formal verification of the demand paging mechanism in CVM.

2.2.1 The Language C0

ANSI C has a complex and highly underspecified semantics. However, low-level kernel programs such as drivers explicitly *use* properties of a particular compiler on a target hardware, for example, its little-endian-ness or a particular atomicity of assembly operations. They can therefore not be verified based only on the vague ANSI C semantics. In our approach, we constrained ourselves to the C-like imperative language *C0*, which has sufficient features to implement low-level software, but which is interpreted by a mathematically well-defined semantics. It is type safe and designed with verification in mind. An elaborate description of the big-step semantics is given in [69] and the small-step semantics is introduced in [48]. An overview on the simulation theorems between the semantical layers can be found in [3].

C0's most important limitations compared to ANSI C are:

- expressions must be free of side effects and do not contain function calls,

- there are no implicit type conversions, especially not from arrays to pointers,
- pointers are strongly typed and must not point to functions or stack variables (i. e., there are neither void pointers nor pointer arithmetic), and
- low-level data types (like unions and bit fields) and control-flow statements (like switch and goto) are not supported.

Syntax. C0 supports fundamental types, aggregate types and pointers. The first category comprises Booleans, 8-bit-wide characters, as well as signed and unsigned 32-bit integers. Aggregate types in C0 are arrays and structures. Pointers may point to all types of data but not to functions.

Expressions are variable names and literals. Moreover, if e and i are expressions and n is a component name, array access $e[i]$, access to structure components $e.n$, dereferencing $*e$, and the 'address-of' operation $\&e$ are expressions. Additionally, C0 supports the usual unary and binary operators.

Finally, C0 supports statements for assignments, dynamic memory allocation, sequential composition, conditional and repeated execution, inline assembly, function calls and returns from functions.

Small-step Semantics. C0 programs are statically represented by the program environment Γ, which comprises a symbol table of global variables, a type-name environment, and a function table. The symbol table is a list of pairs of variable names and types. The type-name environment maps type names to types. The function table maps function names to functions, which are represented by a tuple consisting of (a) a symbol table for the function's parameters, (b) a symbol table for the stack variables, (c) the function's return type, and (d) a statement representing the function body.

The dynamically changing state s_{C0} of a C0 program in execution comprises:

- the remaining program $s_{\mathsf{C0}}.prog$, and
- the current state of the program variables $s_{\mathsf{C0}}.mem$.

The transition relation δ_{C0} of this semantics is deterministic, i. e., a partial function.

2.2.2 The Vamp Assembly Language

The DLX architecture [38] lays the foundation of the VAMP architecture which was initially presented in [57]. An implementation of the VAMP has been formally verified [13, 14]. Since then, the VAMP has been extended with address translation and support for I/O devices [24, 1, 77].

There are three models [4, 77] related to the VAMP architecture. The most concrete one is the gate-level implementation followed by the instruction set architecture (ISA) specification and the assembly-language specification. The adjacent models are related via simulation proofs, such that properties shown at the assembly level can eventually be transferred down to the gate-level.

In the context of this work, we only rely on the VAMP assembly language specification. Details on the gate-level implementation and its verification can be found in [77].

The VAMP assembly language is the target language of the verified C0 compiler and represents a convenient layer for the implementation and the verification of hardware-dependent programs. Its specification abstracts from certain aspects of the lower layers, which are irrelevant for these purposes. For instance, the VAMP assembly machine employs a linear memory model with a conventional memory semantics, i. e.,. it abstracts from memory-mapped device I/O and the paging mechanism of the processor. This abstraction is useful for assembly code executed in the untranslated system mode as well as in the user mode with a transparent handling of address translation and page faults.

Furthermore, the bit vectors from the ISA specification are superceded in the VAMP assembly machine (a) by integers for data, (b) by naturals for addresses, and (c) by a tailored abstract data type for instructions. While this representation is optimized for assembly programs working with integers, arguments regarding naturals and bit-vector operations require the conversion from / to integers. Thus, we define, for instance, the two functions to_int32 and to_nat32 converting between 32-bit integers and naturals.

Assembly Semantics. An assembly state s_{asm} is a record with the following components:

- two program counters $s_{\mathsf{asm}}.\mathsf{dpc}$ and $s_{\mathsf{asm}}.\mathsf{pcp}$ for implementing the delayed branch mechanism, which hold the byte addresses of the current and next instruction,
- general purpose and special-purpose register files $s_{\mathsf{asm}}.\mathsf{gprs}$ and $s_{\mathsf{asm}}.\mathsf{sprs}$ both holding lists of data, and
- main memory $s_{\mathsf{asm}}.\mathsf{mm}$, which is a map from word addresses to data.

A VAMP assembly configuration is called *valid*, if it fulfills certain basic well-formedness conditions: (a) the program counters must be 32-bit naturals, (b) register files must contain 32 registers, and (c) all registers and memory cells must be 32-bit integers [1]. Formally, the predicate is_ASMcore encapsulates these well-formedness conditions:

[1]Note that register $s_{\mathsf{asm}}.\mathsf{gprs}\ !\ 0$ is always 0 and can thus be represented as 32-bit integer.

$\mathsf{is\_ASMcore}\ s_{\mathsf{asm}} \equiv$
$0 \le s_{\mathsf{asm}}.\mathsf{dpc} < 2^{32} \wedge 0 \le s_{\mathsf{asm}}.\mathsf{pcp} < 2^{32} \wedge \mathsf{length}\ s_{\mathsf{asm}}.\mathsf{gprs} = 32 \wedge$
$\mathsf{length}\ s_{\mathsf{asm}}.\mathsf{sprs} = 32 \wedge (\forall i{\in}\{0{<}..{<}32\}.\ -2^{31} \le s_{\mathsf{asm}}.\mathsf{gprs}\ !\ i < 2^{31}) \wedge$
$(\forall i{\in}\mathsf{used\_sprs}.\ -2^{31} \le s_{\mathsf{asm}}.\mathsf{sprs}\ !\ i < 2^{31}) \wedge (\forall ad.\ -2^{31} \le s_{\mathsf{asm}}.\mathsf{mm}\ ad < 2^{31})$

Instructions are represented by an abstract data type and converted on instruction fetch from memory cells using the conversion function cell2instr. Thus, the function

$\mathsf{current\_instr}\ s_{\mathsf{asm}} \equiv \mathsf{cell2instr}\ (s_{\mathsf{asm}}.\mathsf{mm}\ (s_{\mathsf{asm}}.\mathsf{dpc}\ \mathsf{div}\ 4))$

denotes the instruction that is executed next in the assembly machine. Note that the byte address $s_{\mathsf{asm}}.\mathsf{dpc}$ is divided by 4 in order to get the word address of the current instruction in the main memory of s_{asm}.

The assembly semantics can equally be employed in user- and system mode. The mode is determined by the special-purpose register SPR_MODE. In user mode, it is illegal to access the special-purpose registers and solely the page-table length register SPR_PTL is relevant for us. It determines the size of the main memory in pages of 1024 words. For convenience, we encapsulate this fact in the function $\mathsf{size}_{\mathsf{asm}}$ and define:

$\mathsf{size}_{\mathsf{asm}}\ s_{\mathsf{asm}} \equiv \mathsf{to\_nat32}\ (s_{\mathsf{asm}}.\mathsf{sprs}\ !\ \mathsf{SPR\_PTL} + 1)$

The page table length is incremented by 1 because it is initially set to -1 which denotes that no virtual memory is allocated for the process.

A memory access beyond the specified size generates an exception in the real system and an illegal exception in the assembly semantics.

The VAMP assembly transition function δ_{asm} computes for a given assembly state s_{asm} the next state. Essentially, the transition is specified by a case distinction over $\mathsf{current\_instr}\ s_{\mathsf{asm}}$.

2.2.3 Isabelle/Simpl - A Verification Environment for C0

For the verification of C0, we use a general program-verification framework for sequential imperative programming languages: Isabelle/Simpl [68, 69]. It is built as a conservative extension on top of Isabelle/HOL. The key feature of Isabelle/Simpl is the notion of a Hoare-Triple:

$\Gamma \vdash P\ c\ Q$

In a procedure environment Γ, this statement claims that under the assertion P on the original program state, the assertion Q will hold after the execution of the code c given in the *Simpl* language. The assertions P and Q are simply sets of states. In principle, Isabelle/Simpl is polymorphic over the state space; we use records but hide the details by an Isabelle syntax, such that $\{\!|\sigma.\ {}^{\sigma}\mathsf{var} = 5|\!\}$ denotes the assertion that the value of program variable var in state σ is five. Whenever we implicitly refer to the state, the name will be decorated by the acute prefix $'$. For example, $'\mathsf{x}$ will refer to field x in the state record.

Expressions in Simpl are HOL expressions. In addition to the HOL operations, we have defined bitwise conjunction $\mathsf{x} \wedge_{\mathsf{u}} \mathsf{y}$ and disjunction $\mathsf{x} \vee_{\mathsf{u}} \mathsf{y}$ over natural numbers (unsigned integers) x and y. Both operators first convert the natural numbers x and y into bit vectors of the same length. Afterwards, the corresponding bitwise operation is applied to the particular bits resulting in a bit vector which is finally converted into a natural number again. While the conjunction as well as the disjunction work on arbitrary natural numbers, the result of a bitwise negation depends on the width of the data type. Thus, $\neg_{\mathsf{u}/32}\ \mathsf{x}$ denotes the bitwise negation of a 32-bit natural number x. As above, x is first converted into a bit vector of length 32. Afterwards, the particular bits are negated and the resulting bit vector is converted into a natural number again. In a similar way, we proceed with shift operations. With $\mathsf{x} \ll_{\mathsf{u}/32} \mathsf{y}$ we define a left shift of the 32-bit natural number x by y bits. Similarily, we define a right shift $x \gg_{\mathsf{u}/32} y$.

In contrast to expressions, statements are represented by an abstract datatype. The statement syntax is highly abstract, e. g., $\mathsf{Basic}\ f$ represents a state update using function f. In order to present programs in conventional terms, we employ Isabelle's powerful syntax translation machinery and denote a program variable by $'\mathsf{var}$, an assignment by $'\mathsf{var} :== 5$, a conditional by **IF** b **THEN** s_1 **ELSE** s_2 **FI**, a procedure call by $'\mathsf{var} :==$ **CALL** $\mathsf{update}(5)$, etc. The index $_{\mathsf{g}}$ is used to instruct the parser to generate guards that protect against runtime faults like overflows.

The procedure environment Γ is a partial function from procedure names to statements. These statements constitute the procedure body and are defined by the Isabelle/Simpl command **procedures**. The following command, for instance, defines the procedure update:

procedures update(var | res_nat) =
  $'\mathsf{res\_nat} :==_{\mathsf{g}} {'\mathsf{var}}$

It has one formal parameter called var and the result to return to the calling function is held in variable res_nat, i. e., the bar separates input and output parameters. When formally specifying the functionality of the procedure, we write $'\mathsf{res\_nat} :==$ **PROC** $\mathsf{update}('\mathsf{var})$ as shorthand for the code of procedure update.

The Hoare-Triple

$$\Gamma \vdash \{\!|\sigma.\ {}^{\sigma}\mathsf{var} = x|\!\}\ {'\mathsf{res\_nat}} :== \textbf{PROC}\ \mathsf{update}('\mathsf{var})\ \{\!|\tau.\ {}^{\tau}\mathsf{res\_nat} = x|\!\}$$

states that procedure update assigns the value of its argument to the return variable.

The framework includes a big-step semantics, a Hoare logic for partial as well as total correctness and an automated verification-condition generator for Simpl. Within this sequential core language, assembly fragments as well as the C fragment C0 are embedded. The embedding is based on a compiler converting C0-constructs in terms of operations provided by the small-step semantics of the VAMP machine. A correctness proof for this compiler, which links the small-step semantics to the Simpl big-step semantics, is also provided [3]. This correctness theorem about the embedding of C0 into Simpl allows for mapping low-level properties to more abstract ones formulated on the big-step semantics of C0. Thus, throughout this paper, we will present all algorithms in Simpl (so that we can rely on a uniform Isabelle/HOL foundation); note that this Simpl code is the result of an automatic translation from the C0 code that is actually compiled and runs on the machine.

We employ the HOL type system to model C0 programming language types. Isabelle's type inference then takes care of typing constraints that would otherwise have to be explicitly maintained in the assertions. Propositions that explicitly refer to the memory layout or to hardware device registers cannot be proven on the C0-Hoare-Logic level; in these situations, the verification necessarily descends down to the VAMP level. Our approach is to abstract the effect of those low-level computations into atomic *XCalls* (extended calls) in all our semantic layers. In particular, the state-space of C0 is augmented with an additional component that represents the state of the external component, e. g., a device. An XCall is a procedure call that performs a transition on this external state and communicates with C0 via parameter passing and return values. With this model, it is straightforward to integrate XCalls into the semantics and into Hoare logic reasoning. XCalls are typically implemented in assembly.

Code Verification in Isabelle/Simpl

The introduction presents a bird's eye view on the refinement, as depicted in Figure 1.1. In principle, we can reformulate the depicted claim in Isabelle/ Simpl as follows:

$$\Gamma \vdash \{\sigma.\ \mathsf{Abs}\ \sigma\ s\}\ \textbf{PROC}\ \mathsf{system\_step}()\ \{\tau.\ \mathsf{Abs}\ \tau\ (\delta\ s)\}$$

Assuming an abstraction relation Abs that holds for a concrete state σ and an abstract state s, the relation is preserved by the transitions system_step on the concrete level and δ on the abstract level. Just like the figure, this statement is an over-simplification. We postpone the details to Chapter 5.

Our approach to code verification combines refinement and code correctness, i. e., contracts are specified in terms of the abstract states. As a simple example, we assume a library function append, which appends an element to a singly-linked list. In the implementation, the list is represented as a pointer structure described by the variables head and next. In order to prove that a specific pointer structure in the state τ after a function invocation indeed denotes a list, it is useful to know that the original pointer structure in state σ already denoted a list. We encapsulate the list property in a predicate $\mathsf{inv}_{\mathsf{list}}$ over the pointer variables and formulate correctness of $\mathsf{append}(\mathsf{head}, \mathsf{elem})$ as follows:

$$\begin{aligned}&\Gamma \vdash \{\!\!\{\sigma.\ \mathsf{inv}_{\mathsf{list}}\ {}^{\sigma}\mathsf{head}\ {}^{\sigma}\mathsf{next} \wedge \mathsf{abs\_rel}_{\mathsf{list}}\ {}^{\sigma}\mathsf{head}\ {}^{\sigma}\mathsf{next}\ xs\}\!\!\}\ \mathbf{PROC}\ \mathsf{append}({}'\mathsf{head}, {}'\mathsf{elem})\\ &\quad \{\!\!\{\tau.\ \mathsf{inv}_{\mathsf{list}}\ {}^{\tau}\mathsf{head}\ {}^{\tau}\mathsf{next} \wedge \mathsf{abs\_rel}_{\mathsf{list}}\ {}^{\tau}\mathsf{head}\ {}^{\tau}\mathsf{next}\ (xs\ @\ [{}^{\sigma}\mathsf{elem}])\}\!\!\}\end{aligned}$$

i. e., we express the effect of the function in terms of its abstract representation, which is a concatenation of lists.

2.3 Devices

Devices may communicate with an external environment and the processor. The former is used to model non-determinism and communication; a network interface card, for example, sends and receives network packets. The processor accesses a device by reading or writing special addresses. The devices, in turn, can signal interrupts to the processor. So far, direct memory access (DMA) is not supported.

Device communication happens on many different layers throughout our system. In Verisoft, we established a uniform way of interacting with devices by developing a generic device framework featuring standardized transition functions for all layers. The detailed discussion of this model goes far beyond the scope of this work and it is not necessary for the further developing. Alkassar and Hillebrand [2] describe the model in detail.

In a nutshell, the device model is pseudo-parallel in the sense that we either do an internal step – the device consumes a processor input – or an external step – the device consumes an external input, but never both at the same time. We use the automaton

$$\mathcal{A}_{\mathsf{D}} = (\mathcal{S}_{\mathsf{D}}, \mathcal{S}_{\mathsf{D}}^{0}, \Sigma_{\mathsf{int}}, \Omega_{\mathsf{int}}, \omega_{\mathsf{D}}, \delta_{\mathsf{Dint}}, \Sigma_{\mathsf{ext}}, \Omega_{\mathsf{ext}}, \delta_{\mathsf{Dext}})$$

to describe the devices. The state $\mathcal{S}_{\mathsf{D}}$ of the device system subsumes the particular device configurations with $\mathcal{S}_{\mathsf{D}}^{0} \subset \mathcal{S}_{\mathsf{D}}$ representing the set of the initial configurations. Devices communicate with the kernel resp. the processor by using the alphabet Σ_{int} (from the device subsystem to the processor) and the alphabet Ω_{int} (from the processor to the devices). Our kernel uses memory-mapped I/O for device communication. Hence, the output alphabet Ω_{int} comprises read and write accesses to device addresses whereas the input alphabet Σ_{int} consists of interrupt lines and optionally incoming data. The kernel can access the device system by means of the output function ω_{D} which returns the interrupt vector of currently active interrupts. Kernel-initiated communication is performed by the transition function δ_{Dint}. Based on an input from the kernel, it does not only yield a successor device state, but also output to the kernel and, potentially, to the environment.

The communication with the environment is based on an *external interface* in order to receive keystrokes, for example, or send network packages.

It is defined by the alphabets Σ_{ext} and Ω_{ext}. Inputs from the environment consist of a device ID and an external device input. The external transition function δ_{Dext} takes this input and computes the next state. While internal steps may result in both external and internal output, external steps only produce external output.

2.4 Communicating Virtual Machines

Virtually every modern operating system kernel is devided in a hardware-dependent and a hardware-independent part. Usually, the former part is small but employs inline assembly code, while the latter one is larger in size but written in a high-level programming language. From the verification perspective, inline assembly code requires a logic with much more machine details than that of a well-designed high-level language. Hence, the partitioning regarding functionality is very useful for our intended verification as well.

Based on these observations, we encapsulate all hardware-specific low-level functionality, which is possibly using inline assembly, in the layer of *communicating virtual machines* (Cvm), which was first introduced in [34] and further developed in [65, 76]. We identify three major tasks for this abstraction layer: memory virtualization [72, 6], switching between different threads of execution [74], and data exchange [76]. For the hardware-independent part, Cvm provides the means for access and manipulation of user machines by so-called *primitives* [73]. Examples are cvm_copy, for copying data between two user machines, cvm_alloc for increasing the virtual memory space, and cvm_get_gpr for reading a general purpose register of a user machine. The Cvm primitives are simply functions that operate on Cvm data structures and might possibly contain inline assembly. From the programmer's view, we have thus established a solid framework for microkernel construction [40]. Programmers can implement microkernels using Cvm primitives and plain C0 without extra inline-assembler portions.

From the verification perspective, we established a semi-parallel model of computation that formalizes concurrent user machines interacting with a kernel. These user machines are abstract processors with virtual memory. The so-called *abstract kernel* is represented as an abstract C0 machine with typed memory. In addition to usual C0 functions, the kernel code might call Cvm primitives. Hence, the primitives must have been declared in the C0 machine. Moreover, a function called dispatcher_kernel must be defined, which Cvm will call when an interrupt occurs. In order to obtain an executable, we have to link the Cvm implementation with the abstract kernel [65]. We call the resulting executable the *concrete kernel*.

In the following, we will only briefly summarize the formal model of Cvm. Apart from that, we refer to the mentioned literature.

2.4.1 Cvm State

A Cvm state s_{CVM} is a record with the following components:

- the abstract kernel cvm_kernel,
- the virtual user processes cvm_up, and
- the state of the device system cvm_devs.

In Cvm, the abstract kernel cvm_kernel is modelled as a C0 machine. The component cvm_up contains information on the virtual machines of the user processes. We denote the number of user processes including the kernel that are allowed to run in our system by the constant PID_MAX = 128. For identifiers of user processes, we introduce the data type procnumT = $\{1..<\mathsf{PID\_MAX}\}$, whereas identifier 0 is used for the kernel. Cvm applies the Vamp assembly semantics to model the virtual user machines. Accordingly, component cvm_up.userprocesses provides a mapping between process identifiers and Vamp assembly states. The component cvm_up.currentp holds the identifier of the current process, which is determined by the process identifier $\lfloor pid \rfloor$, if process pid is running and $\perp$, if the kernel is running. Common to all processes is the interrupt mask cvm_up.statusreg which is represented as a 32-bit natural number.

Initial Cvm State. An initial Cvm state represents the situation after a reset and is described by the function init_cvm_sys. As the abstract kernel as well as the devices are parameters of the Cvm model, they have to be provided as input:

$$\begin{array}{l}\mathsf{init\_cvm\_sys}\ ak\ devs \equiv \\ \quad (\!|\mathsf{cvm\_kernel} = \mathsf{init\_cvm}\ ak,\ \mathsf{cvm\_up} = \mathsf{init\_cvmup},\ \mathsf{cvm\_dev} = \mathsf{init\_dev}\ devs|\!)\end{array}$$

In a nutshell, init_cvm initializes the local stack and the program rest of the abstract C0 kernel machine ak, whereas init_dev initializes the devices $devs$. Function init_cvmup is used to initialize the abstraction of the virtual user processes, i. e., the program counters are set to 0, and the memory as well as the general- and most of the special-purpose registers are filled with zeroes. However, the page table origin in register SPR_PTO is process-specific, whereas the page table length in register SPR_PTL is set to -1, which denotes that no virtual memory is allocated for the process so far. In addition, the SPR_MODE register is set to 1 intending that the machine runs in user mode.

So far, no process is running, i. e., the current process identifier is set to $\perp$. Finally, the status register enables the interrupts for illegal instructions, misalignment, page faults, traps, and the timer device.

For more formal details, we refer to [65, 76].

Valid Virtual Machines. In the remainder of this thesis and, in particular, in the correctness proof of our kernel, we rely on certain validity requirements for the virtual machines. Predicate is_valid_cvmup formally encapsulates these requirements:

$$\begin{aligned}
&\mathsf{is\_valid\_cvmup}\ up \equiv\\
&\quad \forall i.\ \mathsf{is\_ASMcore}\ (up.\mathsf{userprocesses}\ i) \wedge\\
&\qquad (up.\mathsf{userprocesses}\ i).\mathsf{sprs}\ !\ \mathsf{SPR\_MODE} \neq 0 \wedge\\
&\qquad -1 \leq (up.\mathsf{userprocesses}\ i).\mathsf{sprs}\ !\ \mathsf{SPR\_PTL} < \mathsf{to\_int32}\ \mathsf{TVM\_MAXPAGES}
\end{aligned}$$

First of all, each virtual machine represents a Vamp assembler machine in terms of is_ASMcore and runs in user mode. Furthermore, a virtual machine is either occupied with no memory, i. e., its page table length equals -1, or the number of pages is less than TVM_MAXPAGES.

2.4.2 Cvm Transitions

A Cvm transition δ_{CVM} takes a Cvm state s_{CVM} and an external device input din_{ext} as parameters, yielding either a next configuration $\lfloor s_{\mathsf{CVM}}' \rfloor$ or $\perp$, if a run-time error has occurred, and potentially a device output to the environment.

Depending on the external device input din_{ext} and the current-process identifier $s_{\mathsf{CVM}}.\mathsf{cvm\_up.currentp}$, the Cvm transition function δ_{CVM} distinguishes three cases[2]:

1. If $din_{\mathsf{ext}} \neq \perp$, a device step is performed. In this case, only the device component of s_{CVM} is updated with the new device configuration (obtained by δ_{Dext}). Additionally, a device output to the environment might be generated.

2. If there is no external device input and the current process identifier is $\perp$, the abstract kernel makes a step.

3. Otherwise, the current process makes a step.

In the latter case, Cvm distinguishes three sub-cases: (a) an user step without interrupts, (b) an user step with an interrupt that aborts the user execution (illegal instruction, misalignment, etc.), and (c) an user step with an interrupt, but one which allows the current process to take a step (external interrupts, trap, and overflow).

User steps without interrupts boil down to an application of the Vamp assembly transition function δ_{asm} to the virtual machine of the current process. Steps with interrupts result in an invocation of the abstract kernel.

[2] Note that, in this context, we only consider *live* executions meaning that the kernel, the user processes and the devices are executed infinitely often. Thus, it may not happen that, for instance, device steps might block the kernel forever.

Whether thereby the current process takes a step depends on the kind of interrupts. In case of a runtime error, i. e., the user execution is aborted, the virtual machine of the current process remains unchanged. Otherwise, the current process makes a step according to δ_{asm}.

The update of the virtual machine vm_{cp} of the current process in case of a user step determines function $\mathsf{user}_{\mathsf{step}}$:

$$\mathsf{user}_{\mathsf{step}}\ vm_{\mathsf{cp}} \equiv \textbf{if}\ \mathsf{user\_step\_progress}\ vm_{\mathsf{cp}}\ \textbf{then}\ \delta_{\mathsf{asm}}\ vm_{\mathsf{cp}}\ \textbf{else}\ vm_{\mathsf{cp}}$$

Predicate user_step_progress holds, as long as no runtime errors occur during the execution of the current instruction.

2.4.3 Cvm Primitives

Cvm primitives [73] provide mechanisms for process management, inter-process and device communication. The abstract kernel uses these primitives to implement a scheduler, interrupt delivery and kernel calls, allowing user processes to interact with each other and with devices.

While implementing the abstract Vamos kernel, we use the following set of Cvm primitives: cvm_reset and cvm_clone for process initialization, cvm_alloc for increasing and cvm_free for decreasing memory of an user process, cvm_copy to copy data from one process to another, cvm_get_gpr and cvm_set_gpr to read and write user process register, for device communication cvm_in_word, cvm_out_word and cvm_virt_io, cvm_set_mask for setting the Cvm interrupt mask, and cvm_load_os for loading the operating system into the memory.

2.4.4 Cvm in Isabelle/Simpl

The code correctness proof for the Vamos microkernel completely relies on the verification environment Isabelle/Simpl. Recall, however, that Vamos is implemented to run as Cvm's kernel machine. For this reason, we have to represent this framework in our programming model and describe, how we model the effects of Cvm in Simpl as seen by a Vamos programmer.

We do this by introducing an external state component cvmX subsuming the visible remnants of a Cvm state: (a) the processor abstraction cvm_up, and (b) the device subsystem cvm_devs. The abstract C0 kernel machine is missing, because it is instantiated with the concrete Vamos implementation.

Based on component cvmX, we are able to model the steps in Cvm before invoking and after leaving the Vamos kernel, i. e., the function dispatcher_kernel.

Figure 2.3 depicts the pseudo-code of a Cvm transition.

Formally, the component cvmX is represented as a tuple and we define the function cvm_ups cvmX in order to get the first component with the virtual processes and cvm_devs cvmX to get the second one with the devices.

Furthermore, we define function mca_nat *up devs stat*, which computes the vector of the exceptions that occur during the next step, under consideration of the virtual machine *vm* of the current process, the device system *devs* and the status register *stat*. Some possible exceptions, like traps, write an exception-data register; its content is computed by edata_nat *vm*.

After power-up, the processor generates a reset signal which, according to Figure 2.3, leads to the initialization of the CVM component. Apart from a reset, a transition consists of up to two phases: First, the current process executes one (assembly) step if existing.[3] Second, the CVM layer computes the vector of enabled interrupts and invokes the kernel's dispatcher_kernel function, if an enabled interrupt has been raised. There are two possible sources of interrupts: The current process may cause an exception, and external devices may raise their interrupt line. Interrupts are ignored when not enabled in the status register. The return value of dispatcher_kernel describes the process to be run next: If the value is a valid process number, this process is elected, otherwise, the system idles until the next device interrupt occurs.

We address the actual implementation of dispatcher_kernel resp. the VAMOS microkernel in Section 2.5.2.

Apart from modelling the CVM steps around the VAMOS invocation, there is another reason for the introduction of cvmX. Applying the concept of XCalls, it enables the possibility that low-level accesses of the VAMOS kernel, like setting register values, become visible in the code specification. This is crucial, as we do not only want to specify the C0 parts of the kernel but aim at an overall model of kernel executions.

In consequence, with the integration of CVM into Simpl, we provide an optimal background for both, proving the VAMOS implementation correct and combining CVM and VAMOS, in order to get an overall kernel correctness theorem.

2.5 Vamos Microkernel: Design and Implementation

The *kernel* of an operating system is the code that runs in the privileged mode of a processor, i. e., this and only this code has unrestricted access to all hardware resources. Traditionally, kernels provided an abstraction from the hardware processor and the external devices. When kernels grew in size over the years because of a rising variety of hardware, and especially external devices, the traditional systems were called *monolithic* and the idea of smaller *microkernels* was born.

The motivation for microkernels, however, is manifold. Microkernels be-

[3] Recall that we do not rely on the existence of an idle process. Hence, there might be no current process.

```
procedures system_step() =
 IF$_g$ 'reset THEN    (* CVM Reset *)
    'reset :==$_g$ False;
    'eca :==$_g$ 1;
    'edata :==$_g$ 0;
    'cvmX :==$_g$ (init_cvmup, init_dev (cvm_devs 'cvmX))
 ELSE
   'up :==$_g$ cvm_ups 'cvmX;
   IF$_g$ currentp 'up = $\bot$ THEN    (* CVM is idle *)
    'eca :==$_g$ mca_nat None (cvm_devs 'cvmX) (statusreg (cvm_ups 'cvmX));
    'edata :==$_g$ 0
   ELSE
    'proc :==$_g$ (userprocesses 'up) $\lceil$currentp 'up$\rceil$;
    'eca :==$_g$ mca_nat ($\lfloor$'proc$\rfloor$) (cvm_devs 'cvmX) (statusreg (cvm_ups 'cvmX));
    'edata :==$_g$ edata_nat 'proc;
    'cvmX :==$_g$ ('up (|userprocesses :=
                 (userprocesses 'up)
                   ($\lceil$currentp 'up$\rceil$ := user$_{step}$ (userprocesses 'up $\lceil$currentp 'up$\rceil$))|),
            cvm_devs 'cvmX)
   FI
 FI;
 IF$_g$ 'eca > 0 THEN   (* has an interrupt been raised? -- call dispatcher_kernel *)
   'cp :== CALL$_g$ dispatcher_kernel ('eca, 'edata );
   'cvmX :==$_g$ ( (cvm_ups 'cvmX)
                (|currentp := if 'cp $\in$ {1..<PID_MAX} then $\lfloor$Abs_procnumT 'cp$\rfloor$
                         else $\bot$|),
                  cvm_devs 'cvmX )
 FI
```

Figure 2.3: Simpl function system_step, which represents a combined step of CVM and VAMOS

came a popular research topic in the late 1980s together with the idea of multi-personality operating systems, which demanded a more general hardware abstraction than the traditional approach. This first generation of microkernels such as Mach [64] or IBM's Workplace OS [33] suffered from a poor performance. When Jochen Liedtke analyzed these systems [53], he pinned the problem down to a feature-overloaded mechanism for inter-process communication (IPC). Together with a light-weight, flexible IPC mechanism [51], he proposed the minimality of hardware abstractions [52] in the kernel and suggested that servers implement the traditional services of operating systems. Engler *et al.* [32] took the idea of minimality a step further and banned (nearly) all abstractions from the kernel. Instead of resource management, the kernel was restricted to resource protection. Abstractions were implemented in operating-system libraries and directly linked to the user processes.

Our microkernel design is not as minimal as Engler or Liedtke proposed. We host all functionality in the kernel that would be hard to verify if implemented in user processes. Obeying this principle, the memory management, support for finite IPC timeouts, and the scheduler live in our microkernel. Device drivers, which constitute the largest part of today's monolithic kernels, are implemented outside our microkernel.

An initial version of the VAMOS microkernel had been implemented by Stefan Maus and Dominik Rester under the supervision of Mauro Gargano. Over time, many students and staff improved and extended the kernel – its implementation as well as its design. In particular, Matthias Daum introduced the capability-like concept of process identifiers and implemented the overflow-safe management of timeouts and a generic debug library.

The next section describes the basic functionality of our kernel in more detail. The chapter concludes with the representation of the Vamos implementation in Isabelle/Simpl. As it is taken as basis for the later verification, we do not elaborate on the details of the C0 implementation. Anyways, both representations are quite similar.

2.5.1 Vamos Functionality

Our microkernel Vamos performs the following tasks: (a) enforcement of a minimal access control, (b) process management, (c) memory management, (d) priority-based round-robin scheduling, (e) support for user-mode device drivers, and (f) inter-process communication (IPC). Processes can control these tasks via the kernel's application binary interface (ABI). Table 2.1 lists the kernel calls that constitute the ABI.

Subsequently, we present the basic concepts incorporated in Vamos to achieve the desired functionality.

Access Control and Initial Process. A minimal access-control mechanism reserves most kernel calls for so-called *privileged processes*. Thus, only a privileged process can bring up new processes or kill existing ones, alter the memory consumption of processes, change their scheduling parameters, or control the registration of device drivers. Any process, however, might use the IPC mechanism.

When VAMOS boots, it launches one privileged single process, the *init process*. We presume that the process constitutes the user-mode parts of the operating system, for instance, the simple operating system (SOS) of Bogan [15]. It has to set up the required servers of the operating system, start and register the device drivers, and possibly implement a more sophisticated access-control mechanism. Non-privileged processes may then communicate with the privileged processes, in order to request kernel services on their behalf.

Table 2.1: Application binary interface of the VAMOS kernel

Kernel Call	**Description**
Access Control	
SET_PRIVILEGES ^{p}	add process to set of privileged processes
Process Management	
PROCESS_CREATE ^{p}	create a new process from a memory image
PROCESS_CLONE ^{p}	copy an already existing process
PROCESS_KILL ^{p}	kill a process
Memory Management	
MEMORY_ADD ^{p}	increase the memory amount of a process
MEMORY_FREE ^{p}	decrease the memory amount of a process
Scheduling Mechanism	
CHG_SCHED_PARAMS ^{p}	change scheduling parameters
Device Driver Support	
CHANGE_DRIVER ^{p}	(un)register a driver for a set of devices
ENABLE_INTERRUPTS ^{d}	re-enable a set of interrupts
DEV_READ ^{d} / DEV_WRITE ^{d}	communicate with a certain device
Inter-Process Communication	
IPC_SEND / IPC_RECEIVE	unidirectionally communicate with process
IPC_REQUEST	send and immediately wait for a reply
CHANGE_RIGHTS	manipulate IPC rights
READ_KERNEL_INFO	receive information from the kernel

^{p} call is reserved for privileged processes
^{d} call is reserved for device drivers

Handles and a Capability-like Access-Control Mechanism for IPC. Our system, i. e., the Vamos kernel, identifies the different user processes by unique numbers. These process numbers, in principle, would also be suitable for user processes to refer to each other. However, an identification based on the plain process numbers would permit the guessing of these numbers, such that processes might find (possibly) unauthorized communication channels. Furthermore, the reuse of process numbers of terminated processes is somehow tricky. Liedke, for instance, introduced a generation counter for process numbers but this counter might overflow and is not well suited in the context of formal verification. Thus, the Vamos kernel introduces a layer of indirection which allows processes to refer to each other only via process-local alias names, so-called *handles*. A handle can only be used in the local context of a process and has to be translated to the actual process number by the kernel. The translation, however, is attached to certain conditions. Furthermore, the concept of handles lays the foundation for a capability-like access-control mechanism for IPC. The latter is necessary as, in the context of IPC, the distinction between privileged and unprivileged processes is too coarse.

For this to work, the kernel maintains for each process x a list which we call the *handle database* of the process. A handle hn points to one entry e in this handle database which, in turn, stores information on the corresponding handle.

The handle hn is considered as valid, i. e., the process it refers to is known by x, if it is marked as *known* in the entry e. Only in this case, the process x may safely use the handle hn and the kernel translates it into a process number, if needed.

The handle databases, however, are dynamic datastructures. Thus, process x might, for instance, *release* the handle hn from its own database. While such operations are noncritical as they are initiated by the process itself, things are different, if a privileged process, for instance, selectively steals the handle hn from x's handle database or the process, hn refers to, has been terminated. Both actions result in the invalidation of handle hn in x's handle database, i. e., it is no longer marked as known. However, neither the stealing nor the process termination do necessarily go on with the knowledge of x. Thus, although hn is invalidated, the process x might continue to believe that hn is still valid. Normally, this misunderstanding will only be resolved, if x tries to use handle hn and the kernel responds with an according error message. While designing the Vamos kernel, however, we aimed to reduce such errors. For this purpose, we mark handles like hn as *stolen*. Based on that, the kernel tries to inform the processes about the existence of stolen handles by means of kernel notifications which are supplied with each IPC message. While the latter only inform about the existence of stolen handles, the Vamos call read_kernel_info provides a detailed listing of the stolen handles in the handle database of the calling process. In this way, the kernel can be sure that the owners of stolen handles are notified and does no longer mark the corresponding handles as stolen. Thus, they may be reused for other processes.

With the concept of handles we allow for a conceptional infinite name space of process identifiers and improve the reuse of such identifiers in comparison to Liedke's generation counter which might overflow. Moreover, the handles prevent from guessing any process numbers which may lead to unauthorized communication. The only drawback of this indirection is the extra maintenance effort in the VAMOS kernel. However, with a very basic handle database, we keep the costs as low as possible. There are only two special handles regarding the process identification. Handle HN_SELF enables a process to identify itself, e. g., if it wants to clone itself. Furthermore, handle HN_PARENT refers to the parent process. In all other cases, we use a one-to-one mapping between process numbers and handles.

Apart from the process identification, the concept of handles lays the foundation for a capability-like access-control mechanism for IPC. For this reason, each valid handle is associated with IPC rights controlling the communication with the corresponding process it refers to. We distinguish four different rights which are also part of the handle database entries. A process is cut off from the communication with another process, if none of the rights in the corresponding handle database entry is set. Apart from that, the *request* right is the weakest one. It only allows combined send and receive calls and forces, for instance, that a client waits for a server's response. The timeout of the waiting is infinite, as long as the client does not have the *finite* right which abolishes the limitation and enables the client to initiate a request with a finite receive timeout. The *send* right is the strongest one. It allows an unrestricted communication, i. e., no limitations regarding timeouts and the possibility to perform only a send without waiting for a response. However, unless the *multiple* right is not set, existing rights will be cleared after the first send operation. Receiving from a known process is always possible.

Inter-Process Communication. IPC is the only way how processes in VAMOS may communicate with each other. The communication is synchronous, i. e., if the IPC partner is not already waiting, the calling process is blocked until either the partner is ready or the timeout is exceeded. For sending and receiving, the VAMOS kernel provides the calls IPC_SEND and IPC_RECEIVE. In addition to that, the call IPC_REQUEST enables a combined send and receive with the same communication partner which is typically used by clients to place requests on some servers.

As described above, processes identify the desired communication partners by means of handles and the kernel checks whether the corresponding IPC rights are sufficient for the intended communication.

Apart from the ordinary receiving from another process, the Vamos call IPC_RECEIVE supports two further operation modes. The *open-receive* mode enables a process to be simultaneously available for more than one sender. A prominent application area of this mode is found with servers that rely on the possibility to receive requests from various clients. The actual request handling, however, is based on first-come, first-served. Nevertheless, a server may restrict the circle of clients by means of the IPC rights.

The remaining operating mode is tailor-made for device drivers. Ultimately, device drivers take a great interest in receiving external interrupts as fast as possible. In order to meet this desire, the call IPC_RECEIVE provides the possibility of a so-called *closed kernel-receive*. This option restricts the receiving to notifications from the Vamos kernel only. In doing so, the VAMOS kernel is qualified to deliver according notifications as soon as interrupts for the driver occur. A more detailed view on that provides Section 4.6.

With the basic functional principles at the back, we now take a look at the data that can be transferred via IPC. Data, in the context of IPC, is considered as message and comprises multiple parts. The most common one is the memory message. Specified by the sender, it describes a memory region that should be transferred to the receiver. For this to work, the receiver also has to specify a suitable memory region to buffer this message. Another part of an IPC message comprises rights. A sender may grant IPC rights to a receiver, on which the latter relies on in future communications with the sender. The receiver is informed about the new rights by the so-called return rights which describe the combination of the (possibly) already existing rights and the new ones. On the one hand, these return rights are stored in the receiver's handle database and, on the other one, they are made available for the receiver in an according result register. In addition to that, it is possible that the sender introduces new processes (together with associated rights) to the receiver. Thus, the operating system may, for instance, introduce a server to a newly created process and provide it with the corresponding rights for the communication. As above, an according handle together with the rights are stored in the handle database of the new process and published in according result registers.

Finally, an IPC message also delivers notifications from the kernel. Thus, as mentioned before, they inform the receiver on stolen handles or occurred interrupts the receiver acts as driver for.

Process Scheduling. The basic policy underlying the Vamos scheduler is round-robin process selection. This basic policy can be adjusted by two regulators: priorities and timeslices. Our scheduler supports three different priority levels. Only processes in the highest, non-empty priority class will be scheduled. Processes in a lower class wait until no processes are ready to compute in any higher priority class. Within one priority class, the timeslice determines how long a certain process may compute until it is preempted in favor of another process of the same priority class. Thus, timeslices determine the relative weight of process runtimes while priorities lead to the preemption of lower process classes.

Device Drivers. A *device driver* is a user process, which is designated for the communication with certain devices. Only if a process is registered as a driver for a particular device, it may place read or write requests from or to that device, respectively. Moreover, the device driver is notified of interrupts from that device.

2.5.2 Vamos in Isabelle/Simpl

In Section 2.4.4, we got a first impression on the implementation layer of the Vamos microkernel. The Cvm framework in Simpl provides a solid background to argue on a mixed language implementation, like we have with Cvm and Vamos. It allows the modelling of the low-level entities before and after the invocation of Vamos, as well as accesses to these entities from within Vamos.

Due to this, we are able to implement the Vamos microkernel in pure C0. Accesses to low-level entities, like process registers, are only provided by Cvm primitives. These primitives are implemented as XCalls and can be treated as normal C0 function calls. However, in terms of Isabelle/Simpl, they operate on the external state component cvmX according to the corresponding primitive semantics and communicate with the C0 implementation via parameter passing and return values.

In addition, the Vamos kernel maintains several datastructures in order to provide its intended functionality. These so-called *kernel datastructures*, together with the external component cvmX, form the Vamos state space which will be introduced subsequently. The section concludes with the representation of the top-level function of Vamos.

Vamos Implementation State

The Vamos implementation state in Isabelle/Simpl is a record whose individual elements are global and local variables from the Vamos implementation. In contrast to C0, Isabelle/Simpl does not support variables of structural types, like `structs`, but flattens them, i. e., for each individual field there is a separate component in the state space. In addition to that, each pointer variable or pointer structure field is replaced by a heap function in the state space. Thus, we have one heap variable f of type ref ⇒ value for each component f of type value of the `struct`. To clarify the matter, we consider a typical structure `list` to represent a linked list in the heap:

```
struct list {
  int data;
  list *next;
};
```

The structure contains two components: the data element `data` and a pointer `next` to the next node. Thus, we would get two heaps in the state space record: data of type ref ⇒ int and next of type ref ⇒ ref.

Against this background, we proceed with the different (global) datastructures in the VAMOS state. In order to provide its intended functionality, VAMOS maintains general as well as process-specific data. The former is used to accomplish tasks like process scheduling, interrupt delivery or memory management. An overview on the particular state components is given in Table 2.2. Process-specific data gives a more fine-grained view on the particular processes and is provided in so-called *process information blocks*. Implemented as a struct in C0, Simpl provides heap functions for the particular components, see Table 2.3. These heap functions are applied to *process pointers* in order to get the process-specific value of the corresponding component, like the state or priority of the process.

General Data. The process pointers are given by the array pib. It is of length PID_MAX and thus provides a unique pointer to each of the processes.

Table 2.2: General Datastructures in VAMOS

State Component	**Description**
Process Scheduling	
inactive_list :: ref	head of the inactive queue
wakeup_list :: ref	head of the wait queue
ready_lists :: ref list	array of heads of the particular ready queues
Time Management	
current_time :: nat	current time in clock ticks
next_timeout :: nat	next point in time when an timeout expires
Interrupt Management	
intmask :: nat	currently enabled interrupts
inthd :: nat	interrupts with registered handlers
intocc :: nat	occured, but not yet delivered interrupts
Miscellaneous	
pib :: ref list	array with all process pointers available
current_process :: ref	pointer to the currently running process
vamos_pages_used	total number of used memory pages
current_max_priority :: nat	current maximum priority

The indices are the according process identifiers.

All processes are organized in one of the scheduling lists. By means of these queues, the processes are classified into inactive, ready or waiting ones. The implementation enables the access to these queues by pointers to their heads. State components inactive_list denotes the head to the inactive list, whereas wakeup_list denotes the one to the wait list. Due to the three priority levels in VAMOS, the access to the according ready lists is provided by the array ready_lists. The current maximum priority is stored in component current_max_priority. The head of the ready list assigned with this priority determines the currently running process. Additionally, the pointer is explicitly stored in component current_process.

VAMOS measures the time in clock ticks which are actually interrupts from the external timer device[4]. Thus, component current_time holds the number of kernel entries as consequence of timer interrupts. The time might also influence pending IPC operations of waiting processes, if any timeouts expired. Component next_timeout specifies the minimum of the timeout values of waiting processes, i. e., the next point in time, when one of these timeouts expires.

VAMOS only accepts interrupts from enabled devices. An according mask is given by intmask, which actually represents the user-visible portion of CVM's status register. However, enabled interrupts are not necessarily handled by a driver. Those assigned to a driver are subsumed in inthd. Whenever interrupts occur, the VAMOS kernel tries to deliver them to the according drivers as fast as possible. However, it might happen that a driver is not ready to receive the interrupt notification. In this case, VAMOS stores these occurred but not yet delivered interrupts in intocc for a later delivery.

As VAMOS has to cope with restricted memory, it stores the number of used memory pages in vamos_pages_used. Based on this number, it decides whether new processes may be launched and whether existing processes may allocate new memory.

Process-specific Data. Together with the process creation, the process-specific data of the new process is set up.

As already said, the process-specific data is provided by heap functions which, applied to process pointers, return the desired information.

First of all, the process-specific data states some general properties of the process. Component pid denotes the unique identifier of the process and parent holds the identifier of the parent. Actually, the list membership determines the status of a process. However, especially for waiting processes, a more-fine grained notion of state is necessary and provided by component state. For inactive and ready processes it returns INACTIVE_STATE resp. READY_STATE, whereas it distinguishes three cases for waiting processes:

[4]We assume the existence and liveness of the timer device.

The state of a waiting process reflects the IPC operation it is currently waiting for. Thus, a pending IPC_SEND call leads to the state SEND_STATE, whereas SEND_RECEIVE_STATE denotes a pending IPC_REQUEST. If a process waits due to an IPC_RECEIVE call, its state is set to RECEIVE_STATE. The latter is also used to denote the receive phase of an IPC_REQUEST call.

Table 2.3: Process-specific Data in VAMOS

State Component	**Description**
General Information	
state :: ref ⇒ nat	state of a process
pid :: ref ⇒ nat	process identifier
parent :: ref ⇒ nat	process identifier of the parent
fip :: ref ⇒ nat	first invalid page
privileged :: ref ⇒ bool	privileged status
Scheduling Parameters	
priority :: ref ⇒ nat	priority of the process
timeslice :: ref ⇒ nat	how long will the process be active
consumed_time :: ref ⇒ nat	how long is the process already active
timeout :: ref ⇒ nat	when will the process be ready again
queue_next :: ref ⇒ ref	next pointer for the scheduling queue
queue_prev :: ref ⇒ ref	previous pointer for the scheduling queue
Handles and Rights	
handle_db :: ref ⇒ nat list	handle database
stolen_count :: ref ⇒ nat	number of currently stolen handles
Interrupt Handling	
reg_devices :: ref ⇒ nat	mask of interrupts handled by process
IPC Arguments	
ipc_pid :: ref ⇒ nat	process identifier of desired IPC partner
ipc_rights :: ref ⇒ nat	rights to grant to the receiver
ipc_snd_msg :: ref ⇒ nat	pointer to send message
ipc_snd_len :: ref ⇒ nat	length of send message
ipc_rcv_msg :: ref ⇒ nat	pointer to receive buffer
ipc_rcv_len :: ref ⇒ nat	length of receive buffer
ipc_add_pid :: ref ⇒ nat	process identifier of additional process
ipc_add_rights :: ref ⇒ nat	rights to assign with the additional process
ipc_snd_timeout :: ref ⇒ int	timeout for the send operation
ipc_rcv_timeout :: ref ⇒ int	timeout for the receive operation
send_queue :: ref	pointer to send queue of process
send_queue_next :: ref ⇒ ref	next pointer for send queue
send_queue_prev :: ref ⇒ ref	previous pointer for send queue

Whether a process is privileged determines flag privileged. Its current memory consumption is returned by fip denoting the index of the first invalid page.

Furthermore, each active process is involved in the priority-based round-robin scheduling. For this reason, Vamos stores the priority of a process in priority, the timeslice in timeslice and the already consumed time in consumed_time. Each process is classified in one of the scheduling lists. In order to identify its neighbours, each process is equipped with pointers to the previous and next element of the queue, it is currently contained in. The corresponding components are queue_next and queue_prev.

As already mentioned, processes identify each other by means of handles. These handles are process-local and stored in the component handle_db. The number of stolen handles in a handle database is given by stolen_count.

Function reg_device delivers the mask of interrupts that are handled by a process.

If a process performs an IPC operation, Vamos stores the according arguments as process-specific data. We do not dwell on the particular arguments, but refer to Table 2.3. However, in the context of IPC, Vamos also maintains so-call send lists. The send list of a process p contains all pointers to processes which are currently willing to send to p. Similar to the scheduling lists, head pointers send_queue provide the access to the send lists. The traversing is enabled by next and previous pointers, i. e., send_queue_next and send_queue_prev, respectively.

The top-level function of Vamos

Upon kernel entry, the Cvm routine calls the function dispatcher_kernel, which is the actual implementation of the Vamos kernel. As intended by the name, the Vamos kernel mainly acts as dispatcher handling the incoming interrupts that could not be taken care of by Cvm. It takes two parameters: The exception cause eca, which encodes the occurred interrupts, and the exception data edata, which contains the trap number, if a trap has occurred. Figure 2.4 shows the implementation of the function dispatcher_kernel of our microkernel.

The function is characterized by a number of case distinctions. We distinguish:

Initialization. After power-up, the processor generates a *reset* interrupt. In this case, the Cvm framework sets up its internal data structures and then passes the interrupt on to the dispatcher_kernel function. If called with the reset-interrupt bit set, dispatcher_kernel calls the function vamos_init, which initializes the data structures of Vamos.

Process Exceptions. The current process may cause a number of exceptions during its execution. From the kernel's perspective, there are

```
procedures dispatcher_kernel (eca, edata | res_nat) =
  IF$_g$ ('eca $\wedge_u$ 1) $\neq$ 0 THEN
    'dummy_i :== CALL$_g$ vamos_init()
  ELSE
    'old_cup :==$_g$ 'current_process;
    IF$_g$ ('eca $\wedge_u$ UEXCEPT_MASK) $\neq$ 0 THEN
      'dummy_i :== CALL$_g$ process_kill(HANDLE_SELF)
    ELSE
      IF$_g$ ('eca $\wedge_u$ EXCEPT_TRAP) $\neq$ 0 THEN
        'dummy_i :== CALL$_g$ handle_trap('edata)
      FI
    FI;
    IF$_g$ ('eca $\wedge_u$ DEVICE_TIMER_BIT) $\neq$ 0 THEN
      'dummy_i :== CALL$_g$ handle_timer('old_cup)
    FI;
    IF$_g$ ('eca $\wedge_u$ UEXT_INT_MASK) $\neq$ 0 THEN
      'dummy_i :== CALL$_g$ int_delivery('eca)
    FI
  FI;
  IF$_g$ 'current_process $\neq$ Null THEN
    'res_nat :==$_g$ 'current_process $\rightarrow$ 'pid
  ELSE
    'res_nat :==$_g$ 0
  FI
```

Figure 2.4: The kernel-dispatcher function of VAMOS

only two alternatives: a *fatal exception* like an illegal page fault or a *trap*. In the former case, there is no reasonable recover procedure, and thus, the process is simply killed by VAMOS. The semantics of a fatal exception is exactly the same as if the process had requested to be killed via the kernel call PROCESS_KILL. Hence, we reuse the function process_kill. In case of a trap, the function handle_trap is called. This function is essentially a huge case distinction over trap numbers.

Timer Interrupt. Independently from the process exceptions, we check for the timer interrupt, which is passed on to the function handle_timer. This function implements the scheduler.

Device Interrupts. If external devices have raised interrupts, the function int_delivery is invoked in order to disable the interrupts and then either immediately deliver the interrupts, if the corresponding device-driver processes are waiting, or buffer the interrupts for later delivery.

The function dispatcher_kernel determines its return value using the global variable current_process. If there is no current process, the pointer is Null and we return 0 (meaning "wait for interrupts"). Otherwise, we retrieve the process number of the current process from heap function pid. Once, the function dispatcher_kernel returns, the CVM framework transfers the CPU to the process identified by the return value.

Chapter 3

The Abstract System Layer

Contents

3.1 The Overall System - A Bird's Eye View

The abstract layer describes our system: on top of the VAMP processor runs the VAMOS kernel and communicates with external devices.

As Figure 3.1 depicts, we use a number of communicating automata for the specification. Automaton $\mathcal{A}_{\mathsf{V+D}}$ encapsulates the overall system and consolidates the device automaton $\mathcal{A}_{\mathsf{D}}$ and the VAMOS automaton $\mathcal{A}_{\mathsf{V}}$.

The former was already introduced in Section 2.3 and describes the behavior of the external devices like the keyboard, the timer or the network card. The devices may interact by an *external interface* with the environment in order to receive keystrokes, for example, or network packages. The alphabets Σ_{D} and Ω_{D} define this interface, which is adopted by the overall system (alphabets $\Sigma_{\mathsf{V+D}}$ and $\Omega_{\mathsf{V+D}}$).

The interface with the alphabets Σ_{V} (from the device subsystem to the processor) and Ω_{V} (from the processor to the devices) connects the devices

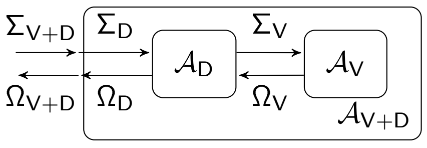

Figure 3.1: The input/output automata of the abstract system layer and their relationship

with the VAMOS automaton. Our kernel uses memory-mapped I/O for device communication. Hence, the output alphabet Ω_{V} comprises read and write accesses to device addresses, whereas the input alphabet Σ_{V} consists of interrupt lines and optionally incoming data.

Subsequently, we focus on $\mathcal{A}_{\mathsf{V}}$ and use it, together with $\mathcal{A}_{\mathsf{D}}$, to assemble $\mathcal{A}_{\mathsf{V+D}}$.

We specify VAMOS running on the VAMP by $\mathcal{A}_{\mathsf{V}} = (\mathcal{S}_{\mathsf{V}}, \mathcal{S}_{\mathsf{V}}^0, \Sigma_{\mathsf{V}}, \Omega_{\mathsf{V}}, \omega_{\mathsf{V}}, \delta_{\mathsf{V}})$ with the state space $\mathcal{S}_{\mathsf{V}}$, the set of initial states $\mathcal{S}_{\mathsf{V}}^0 \subset \mathcal{S}_{\mathsf{V}}$, the input alphabet Σ_{V}, the output alphabet Ω_{V}, the output function ω_{V}, and the transition function δ_{V}. Likewise, we define $\mathcal{A}_{\mathsf{V+D}}$.

The states $\mathcal{S}_{\mathsf{V+D}}$ of the overall system are pairs of a VAMOS state $s_{\mathsf{V}} \in \mathcal{S}_{\mathsf{V}}$ and a device-system state $s_{\mathsf{D}} \in \mathcal{S}_{\mathsf{D}}$.

The input alphabet $\Sigma_{\mathsf{V+D}}$ is used to determine the subsystem which takes the next step. It extends Σ_{D} (indicating a device step) by the value $\perp$ for a processor step. The output alphabets are the same, i. e., $\Omega_{\mathsf{V+D}} = \Omega_{\mathsf{D}}$. Note that the output of the device subsystem depends on the transition, i. e., δ_{Dint} returns a tupel (s_{D}, w) of a successor state s_{D} and an output w. Consequently, there is no separate output function for $\mathcal{A}_{\mathsf{V+D}}$.

Transitions $\delta_{\mathsf{V+D}}$ of the overall system inspect the input i and distinguish three cases:

1. An *external device transition* is performed, if the input $i \in \Sigma_{\mathsf{V+D}}$ contains an input for the device subsystem. The VAMOS state remains constant in this case.

2. A *local kernel transition* is performed, if the input is $\perp$ and the VAMOS output $\omega_{\mathsf{V}}\ s_{\mathsf{V}} = \mathsf{idle}_{\Omega\mathsf{V}}$. Formally, this transition is performed by δ_{V} without any device input.

3. A *kernel-device transition* is performed, if it is the processor's turn (overall input $\perp$) and the kernel requests a device communication, i. e., $\omega_{\mathsf{V}}\ s_{\mathsf{V}} = \mathsf{read}_{\Omega\mathsf{V}}$ *device port count* or $\omega_{\mathsf{V}}\ s_{\mathsf{V}} = \mathsf{write}_{\Omega\mathsf{V}}$ *device port data*. In this case, $\delta_{\mathsf{V+D}}$ passes the VAMOS output with the transition function δ_{Dint} to the device subsystem. In response, the device subsystem delivers an output which contains the read data, if requested. Using this data, $\delta_{\mathsf{V+D}}$ finally performs the VAMOS transition δ_{V}.

In the context of this thesis, we do not consider any autonomous device transitions but only those which are triggered by the kernel. Accordingly, the external input to the transition function $\delta_{\mathsf{V+D}}$ is set to $\perp$. The linchpin of both, local kernel and kernel-device transitions, is the VAMOS kernel. Before we give its formal specification, we first introduce some basic types used in the specification and introduce the model for user processes.

The complete formalization is available from the public Verisoft Repository [78] and directory `verification/spec/vamos/` contains the theory files related to the VAMOS kernel.

3.2 Abstract Types

While specifying the Vamos kernel, we make use of the advanced type system in Isabelle/HOL.

So-called type synonyms allow new names for an existing construction which mainly increases the readability of the theories. Furthermore, Isabelle insists that all terms and formulae must be well-typed. Thus, defining new types and datatypes prevents us, amongst others, from writing nonsense. Hence, we avoid by construction any type mismatches within the specification.

Process Numbers, Priorities, Devices and Ports. The easiest way to introduce new types is the type definition, where any non-empty subset of an existing type is turned into a new one. Vamos distinguishes between PID_MAX user processes classified into PRIOCNT different priority levels. Device communication is limited to DEV_COUNT devices which are identified by device numbers. The sets of process numbers and priorities as well as the set of device numbers are subsets of the natural numbers. Consequently, the corresponding types define three non-empty subsets of type nat.

As already introduced in Section 2.4.1, process identifiers are of type procnumT. Type prioT = {0..<PRIOCNT} subsumes the priorities and devnumT = {1..DEV_COUNT} defines the type of device numbers. Device interrupts are closely related to the devices and represented as set of device numbers. Formally, device interrupts are represented by the type intsT. Each device is equipped with PORT_COUNT ports represented by the type portT = {0..<PORT_COUNT}.

IPC Rights. In the context of IPC rights, we rather want to talk on a more abstract and intuitive level. For this reason, Isabelle provides the possibility to define datatypes. With it, an individual right is represented by the datatype rightT with the constructors:

- v_requestR – the right to send, but with the duty to immediately wait for the response of the receiver,
- v_sendR – the right to send to a process without waiting for the response,
- v_multipleR – the right to perform multiple IPC operations, and
- v_finiteR – the right to perform IPC operations with finite timeout

As rights usually come in sets, type rightsT is used as synonym for rightT set.

Memory Objects. Besides the transfer of rights, IPC also provides the possibility to send and receive data in form of memory objects. A memory object specifies a memory region in the virtual memory of a process, starting at a certain address and ranging over a certain length. Both parameters, the start address as well as the length of the memory object are specified by the process and might be erroneous.

The datatype memobjT represents a memory object and also reflects the possible errors:

- MObjUndefined – either the start address or the length are not word-aligned,
- MObjUnavailable – the specified memory region is not available in the memory of the process, and
- MObjSeq s – the content of the memory region as sequence s of integers

The first two constructors imply errors due to the specified values for the start address and the length. The former indicates mis-alignment whereas the second one indicates that the memory region is not completely located in the virtual memory of the process. A well-defined memory object gives the memory content as a sequence of integers. The start address as well as the length are not of interest any longer and are abstracted away. For a well-defined memory object, the accessor mObjSeq returns the sequence s of integers, i. e., $\mathsf{mObjSeq}\ (\mathsf{MObjSeq}\ s) = s$.

A memory object acting as buffer of an IPC receive operation, for instance, is defined in a similar way. For a well-defined memory object, we are not interested in its content but in its length. Consequently, datatype bufferT consists of the following constructors:

- BufUndefined – either the start address or the length are not word-aligned,
- BufUnavailable – the specified memory region is not available in the memory of the process, and
- BufLength *len* – the length *len* of the actual buffer

As before, the accessor bufLength returns the length *len* of a well-defined buffer, i. e., $\mathsf{bufLength}\ (\mathsf{BufLength}\ len) = len$

Kernel Notifications. If a process explicitly wants to receive any kernel notifications, it has to specify the type of notifications. Notifications are represented by the datatype kinfoT which currently comprises only constructor StolenHandles.

IPC Timeouts. Timeouts are represented by the existing type, natural numbers with infinity. Hence, the datatype timeoutT comes with two constructors: FIN n, for finite timeout values, and ∞ for infinity For abstract timeouts, we defined the operation $+=$, in order to increment a timeout by a natural number.

Process Handles. Process handles are integers but represented by the type handleT.

3.3 The Vamp Assembly Process Model

Our formalization is based on the observation that VAMOS interacts with processes only via a well-defined interface, which is the kernel ABI. Hence, we can encapsulate processes in a self-contained input-output automaton, thereby hiding the internal state and exposing only the interface. The tailor-made model for VAMP assembly processes defines an input-output automaton represented by the tupel:

$$\mathcal{A}_{\mathsf{proc}} = (\mathcal{S}_{\mathsf{proc}}, \mathsf{init}_{\mathsf{proc}}, \Sigma_{\mathsf{proc}}, \Omega_{\mathsf{proc}}, \omega_{\mathsf{proc}}, \mathsf{size}_{\mathsf{proc}}, \delta_{\mathsf{proc}})$$

As state space $\mathcal{S}_{\mathsf{proc}}$ for the assembly processes, we reuse the one of the VAMP assembly semantics (cf. Section 2.2.2). Similarily, we chose function $\mathsf{size}_{\mathsf{proc}}$ to be $\mathsf{size}_{\mathsf{asm}}$, in order to determine the current memory assumption of a process. More elaborate are the definitions of the other components which we present below. The complete formalization can be found in the theory file `vamosAsmProc.thy`.

Note that the VAMP assembly process model which has been defined in this work, is an instance of a generic process model which goes back to Daum and which is described in more detail in [25, 26]. From the kernel perspective, the VAMP assembly process model is rather a matter of taste than necessity because only assembly processes are involved in the interaction with the kernel. However, the VAMOS model serves as basis for more abstract models, which indeed rely on this generic framework, as described in [28].

The VAMP assembly process model also satisfies certain well-formedness constraints. Thus, the processes fulfill the requirements encapsulated in predicate is_valid_cvmup. Daum [25] showed that these constraints are preserved by process transitions under valid inputs. We do not dwell on this here, but refer to Section 5.2 which describes the implementation invariant. The latter plays an essential role regarding the functional verification of the VAMOS kernel and (among other things) takes up the requirements on the VAMP assembly processes in this context.

3.3.1 Process Initialization

The function $\mathsf{init}_{\mathsf{proc}}$ initializes an assembly machine with a memory image *img* and supplies the machine with *pgs* memory pages:

$$\begin{aligned}
&\mathsf{init}_{\mathsf{proc}}\ img\ pgs \equiv \\
&\quad (\!|\mathsf{dpc} = 0,\ \mathsf{pcp} = 4,\ \mathsf{gprs} = \mathsf{replicate}\ 32\ 0, \\
&\qquad \mathsf{sprs} = \mathsf{replicate}\ 32\ 0[\mathsf{SPR\_PTL} := \mathsf{to\_int32}\ pgs - 1,\ \mathsf{SPR\_MODE} := 1], \\
&\qquad \mathsf{mm} = \lambda ad.\ \mathbf{if}\ ad < \mathsf{length}\ img\ \mathbf{then}\ img\ !\ ad\ \mathbf{else}\ 0|\!)
\end{aligned}$$

In principle, we adopt the initialization of the virtual assembly processes as described by function init_cvmup in Section 2.4.1. However, the function $\mathsf{init}_{\mathsf{proc}}$ initializes a virtual assembly machine in the sense that it starts executing. Thus, the program counters dpc and pcp of the delayed branch mechanism are set to the first addresses, i. e., 0 and 4. Furthermore, starting at address 0, the memory is filled with the image *img* which is given as a sequence of integers. Memory cells with addresses greater than the length of *img* are initialized with zeros. In addition, the machine is provided with *pgs* memory pages. Accordingly, the page table has length $pgs - 1$. The remaining components are initialized as before. Setting register SPR_MODE to 1 implies that the machine runs in user mode. The remaining special and general purpose registers are filled with zeros.

After the initialization, the process starts its execution with the instruction at address dpc $(= 0)$, i. e., the instruction at the beginning of the memory image.

3.3.2 Process Interface

Output alphabet Ω_{proc} enumerates all possible kernel calls, a few error conditions, and the output ε_Ω denoting the intention to perform a process-local computation.

In contrast, the input alphabet Σ_{proc} contains all responses to kernel calls and error conditions, as well as the input ε_Σ for a process-local transition.

Formally, Ω_{proc} is defined as a datatype with the following constructors:

- Kernel Calls:
  - SET_PRIVILEGES *hn* – Privilege process (identified by handle) *hn*.
  - PROCESS_CREATE *tsl prio img pgs* – Create a new process of priority *prio* and with timeslice *tsl*. The process should be assigned with *pgs* virtual memory pages holding the initial memory image *img*.
  - PROCESS_CLONE *hn* – Clone process *hn*
  - PROCESS_KILL *hn* – Kill process *hn*
  - MEMORY_ADD *hn pgs* – Provide process *hn* with *pgs* additional memory pages

- MEMORY_FREE *hn pgs* – Release *pgs* pages in the virtual memory of process *hn*
- CHG_SCHED_PARAMS *hn tsl prio* – Change the scheduling parameters of process *hn*: the timeslice should be set to *tsl*, the priority to *prio*
- CHANGE_DRIVER *hn devs register* – Register (if $register = \mathsf{True}$) or deregister process *hn* as driver for the set *devs* of devices
- ENABLE_INTERRUPTS *devs* – Re-enable the interrupts of the devices *devs*
- DEV_READ *dev port buffer* – Read from port *port* of device *dev* and store the content in the memory buffer *buffer* of the calling process
- DEV_WRITE *dev port msg* – Write the memory object *msg* to the port *port* of device *dev*
- IPC_SEND hn_{rcv} $rights_{\mathsf{snd}}$ *msg* hn_{add} $rights_{\mathsf{add}}$ $timeout_{\mathsf{snd}}$ – Send a memory object *msg* together with the rights $rights_{\mathsf{snd}}$ to the sender to process hn_{rcv}. Furthermore, there is the opportunity to send an additional handle hn_{add} which can be provided with the rights $rights_{\mathsf{add}}$. The timeout of this operation is given by $timeout_{\mathsf{snd}}$.
- IPC_RECEIVE hn_{snd} *buffer* $timout_{\mathsf{rcv}}$ – Receive a message from process hn_{snd} and store it in the memory buffer *buffer* within the next $timout_{\mathsf{rcv}}$ clock ticks
- IPC_SEND_RECEIVE hn_{rcv} $rights_{\mathsf{snd}}$ *msg* hn_{add} $rights_{\mathsf{add}}$ $timeout_{\mathsf{snd}}$ *buffer* $timeout_{\mathsf{rcv}}$ – Send to and receive immediately afterwards from process hn_{rcv}.
- CHANGE_RIGHTS hn_{subj} hn_{obj} *grant rights* – Grant (if $grant = \mathsf{True}$) or revoke the rights *rights* to process hn_{obj} of process hn_{subj}
- READ_KERNEL_INFO *type* – request kernel notifications of type *type*

• Error conditions:

- UNDEFINED_TRAP – the current instruction is a trap instruction but specifies a wrong trap number
- RUNTIME_ERROR – the current instruction causes a runtime error, like an illegal page fault

• Process-local Transition:

- PROC_NO_OUTPUT– the current instruction causes no exception, i. e., a process-local transition is accomplished

Process inputs either comprise (a) kernel responses to previous kernel calls, (b) *kernel commands* in order to manipulate the process state, or (c) *empty messages* intending a process-local transition. Accordingly, the input alphabet Σ_{proc} contains the following constructors:

- Responses to successful kernel calls:
  - SUCCESS – response to a successful operation
  - SUCC_DEV_READ *data* – response to a successful read operation where *data* contains the data from the device
  - SUCC_RECEIVE hn_{snd} $reused_{\mathsf{snd}}$ $rights_{\mathsf{snd}}$ *msg* hn_{add} $reused_{\mathsf{add}}$ $rights_{\mathsf{add}}$ *stolen* $infy_{\mathsf{to}}$ *devs* – response to a successful receive operation in the context of either an IPC_RECEIVE or IPC_REQUEST call. The process handle hn_{snd} points to the sender and is reused for the sender iff $reused_{\mathsf{snd}} = \mathsf{True}$. Currently, the receiver associates the rights $rights_{\mathsf{snd}}$ with process hn_{snd}. The sender sent the sequence *msg* of words and the additional handle hn_{add}. The handle is reused iff $reused_{\mathsf{add}} = \mathsf{True}$. The receiver associates the rights $rights_{\mathsf{add}}$ with process hn_{add}. The remaining three parameters subsume the kernel notifications: *stolen* is set iff there are stolen handles in the receiver's handle database, $infy_{\mathsf{to}}$ is active iff the sender performs an IPC_REQUEST and waits with inifinite timeout for the response, and the set *devs* collects all occurred device interrupts handled by the receiver.
  - SUCC_NEW_PROCESS *hn* – response to a successful process creation. The new process can be identified by handle *hn*
  - SUCC_KINFO_STALEH *hns* – response to requested kernel notifications with the set *hns* of stolen handles
- Responses to unsuccessful kernel calls:
  - ERR_OUT_OF_MEM – the request to create or clone a process failed due to insufficient memory resources
  - ERR_OUT_OF_PIDS – the request to create or clone a process failed because the maximum number of processes is reached
  - ERR_PROCESS_NOT_READY – the request to change the memory amount of a process failed because the specified process is not ready for this
  - ERR_UNPRIVILEGED – the desired operation failed due to lacking privileges or insufficient IPC rights
  - ERR_INVALID_ARGS – the arguments specified with the call are invalid

  - ERR_INT_ALREADY_HANDLED – changing the registration of a driver failed because the specified interrupts are already handled
  - ERR_INVALID_HANDLE – the specified handle is not valid in the context of the caller
  - ERR_INVALID_SUBJ_HANDLE /ERR_INVALID_OBJ_HANDLE / ERR_SND_INVALID_HANDLE /ERR_RCV_INVALID_HANDLE – some kernel calls take more than one handle. In this case we distinguish the invalid handle by a more detailed error code.
  - ERR_SND_TIMEOUT /ERR_RCV_TIMEOUT – the timeout for the desired IPC operation expired
  - ERR_SND_BUFFER_OVFL – the sending of a message failed because the message was to large to fit into the receive buffer
  - ERR_SND_SEGV /ERR_RCV_SEGV – the specified message or buffer is not accessible in the memory

- Kernel Commands:
  - ADD_MEMORY *pgs* – increase the memory amount by *pgs* pages
  - FREE_MEMORY *pgs* – decrease the memory amount by *pgs* pages

- Process-Local Computation:
  - PROC_NO_OUTPUT – perform a process-local step

Later on in the VAMOS model, we use predicate $\mathsf{is\_error}$, in order to determine errors during the execution of kernel calls. It only applies for the error responses and delivers False for all other components of the alphabet Σ_{proc}.

3.3.3 Output Functions

The output functions permit VAMOS a limited and indirect access to the state of an assembler machine s_{proc}.

Memory Consumption. VAMOS applies the function $\mathsf{size}_{\mathsf{proc}}$, in order to determine the current memory amount. As we only consider VAMP assembly processes, we adopt function $\mathsf{size}_{\mathsf{asm}}$, i. e., $\mathsf{size}_{\mathsf{proc}} = \mathsf{size}_{\mathsf{asm}}$.

User-Generated Outputs. While $\mathsf{size}_{\mathsf{proc}}$ can be considered as rather *passive*, function ω_{proc} delivers the *active* process output to the kernel. It relies on exceptions, i. e., user-generated interrupts. Only two kinds of user-generated interrupts are passed on to the kernel: runtime-errors and traps. Both may arise while executing the current instruction. Runtime-errors subsume errors occuring during the execution of the current instruction, like illegal pagefaults. In most cases, they are traceable to erroneous programming and the VAMOS kernel responds by killing the process causing the error.

Traps, in turn, constitute a possibility to place a request to the kernel. A process may trigger a trap exception by means of a TRAP instruction defined within the VAMP assembly semantics. Together with the associated immediate constant *imm*, traps are mapped to VAMOS calls which provide the intended functionality. Instructions that neither cause runtime-error nor trap exceptions do not involve the kernel and are executed in the local context of the process.

In order to determine the different situations, our process model provides the output function ω_{proc}:

$$\begin{aligned}&\omega_{\mathsf{proc}}\ s_{\mathsf{proc}} = \\ &(\textbf{if}\ \mathsf{runtimeError}\ s_{\mathsf{proc}}\ \textbf{then}\ \textsc{runtime\_error} \\ &\ \textbf{else if}\ \exists imm.\ \mathsf{current\_instr}\ s_{\mathsf{proc}} = \textsc{trap}\ imm\ \textbf{then}\ \mathsf{trap\_dispatch}\ s_{\mathsf{proc}}\ \textbf{else}\ \varepsilon_{\Omega})\end{aligned}$$

Runtime errors are signaled by the output RUNTIME_ERROR. If the current instruction denotes a TRAP instructions, the VAMOS kernel is involved and function trap_dispatch determines the corresponding output. In all other cases, output ε_{Ω} implies that the process operates in its local context.

As the name suggests, trap_dispatch considers the current instruction of s_{proc} and realizes a case distinction over the values of *imm*.

Due to the fact that VAMOS provides 16 different VAMOS calls, the formal definition of trap_dispatch is quite substantial. Thus, in this section, we restrict ourselves to a couple of exemplary excerpts. The complete formal definition of function trap_dispatch is given in Section 8.1.

For a start, we chose the case $imm = 6$, which is affiliated with the VAMOS call SET_PRIVILEGES. According to the ABI, register 11 contains the handle to the process that should become privileged. Thus, from the definition of trap_dispatch, we can derive the following implication:

$$\begin{aligned}&\mathsf{current\_instr}\ s_{\mathsf{proc}} = \textsc{trap}\ 6 \Longrightarrow \\ &\mathsf{trap\_dispatch}\ s_{\mathsf{proc}} \equiv \textsc{set\_privileged}\ (s_{\mathsf{proc}}.\mathsf{gprs}\ !\ 11)\end{aligned}$$

The definition is pretty easy for this case and benefits from the fact that concrete as well as abstract handles are represented by 32-bit integer values.

Things get more elaborate, when more complex types are involved. As an example, we chose TRAP 15 which is mapped to the VAMOS call IPC_REQUEST:

$$\begin{aligned}&\mathsf{current\_instr}\ s_{\mathsf{proc}} = \textsc{trap}\ 15 \Longrightarrow \\ &\mathsf{trap\_dispatch}\ s_{\mathsf{proc}} \equiv \\ &\ \ \textbf{let}\ hn_{\mathsf{rcv}} = s_{\mathsf{proc}}.\mathsf{gprs}\ !\ 11;\ rights_{\mathsf{snd}} = s_{\mathsf{proc}}.\mathsf{gprs}\ !\ 12; \\ &\qquad msg_{\mathsf{ad}} = s_{\mathsf{proc}}.\mathsf{gprs}\ !\ 13;\ msg_{\mathsf{len}} = s_{\mathsf{proc}}.\mathsf{gprs}\ !\ 14; \\ &\qquad hn_{\mathsf{add}} = s_{\mathsf{proc}}.\mathsf{gprs}\ !\ 17;\ rights_{\mathsf{add}} = s_{\mathsf{proc}}.\mathsf{gprs}\ !\ 18; \\ &\qquad to_{\mathsf{snd}} = s_{\mathsf{proc}}.\mathsf{gprs}\ !\ 19;\ buf_{\mathsf{ad}} = s_{\mathsf{proc}}.\mathsf{gprs}\ !\ 15; \\ &\qquad buf_{\mathsf{len}} = s_{\mathsf{proc}}.\mathsf{gprs}\ !\ 16;\ to_{\mathsf{rcv}} = s_{\mathsf{proc}}.\mathsf{gprs}\ !\ 20\end{aligned}$$

```
  in IPC_REQUEST $hn_{\mathsf{rcv}}$ (num2rights $rights_{\mathsf{snd}}$)
      (build_memobj $s_{\mathsf{proc}}$ (to_nat32 $msg_{\mathsf{ad}}$) (to_nat32 $msg_{\mathsf{len}}$)) $hn_{\mathsf{add}}$
      (num2rights $rights_{\mathsf{add}}$) (int2timeout $to_{\mathsf{snd}}$)
      (build_buffer $s_{\mathsf{proc}}$ (to_nat32 $buf_{\mathsf{ad}}$) (to_nat32 $buf_{\mathsf{len}}$))
      (int2timeout $to_{\mathsf{rcv}}$)
```

Not only the sheer number of arguments constitutes the complexity of the definition, but also the various conversion functions.

As before, the ABI specifies the register locations of the particular arguments. For instance, register 11 contains the handle hn_{rcv} to the receiver and register 13 specifies the start address msg_{ad} of the message. The registers always deliver 32-bit integer values. This is fine regarding the handles, but requires conversions with other arguments.

The call IPC_REQUEST provides the opportunity that the sender grants rights to the receiver, such that the receiver later on, for instance, is allowed to send to the sender. These rights are encoded in the value of $rights_{\mathsf{snd}}$ located in register 12. Based on this value, function num2rights extracts the respective set of abstract rights:

```
num2rights $r \equiv$
  if $r < 0 \vee 15 < r$ then $\bot$
  else $\lfloor$(if $r$ mod 2 = 1 then {v_finiteR} else {}) $\cup$
         (if $r$ mod 4 div 2 = 1 then {v_multipleR} else {}) $\cup$
         (if $r$ mod 8 div 4 = 1 then {v_requestR} else {}) $\cup$
         (if $r$ div 8 = 1 then {v_sendR} else {})$\rfloor$
```

The numerical register value r actually represents a bit vector, whereas four bits suffice to encode arbitrary sets of rights. The least significant bit represents the right v_finiteR, followed by v_multipleR, v_requestR and v_sendR. A 4-bit vector can only encode numerical values ranging from 0 to 15. Thus, values r out of this range are considered as invalid and result in $\bot$. Otherwise, the various bits can be extracted by ordinary division (div) and modulo (mod) operations.

We proceed in the same way with extracting the rights to the (possibly) additional process from $rights_{\mathsf{add}}$.

The memory message is specified by the start address msg_{ad} and the length msg_{len}. Based on these values, function build_memobj generates the corresponding abstract memory object:

```
build_memobj $s_{\mathsf{proc}}$ $obj_{\mathsf{addr}}$ $obj_{\mathsf{len}}$ $\equiv$
  if $obj_{\mathsf{len}} = 0$ then MObjSeq []
  else if $obj_{\mathsf{addr}}$ mod 4 $\neq 0 \vee obj_{\mathsf{len}}$ mod 4 $\neq 0$ then MObjUndefined
      else if $\mathsf{size}_{\mathsf{asm}}\ s_{\mathsf{proc}} * \mathsf{PAGE\_SIZE} \leq obj_{\mathsf{addr}} + obj_{\mathsf{len}}$ then MObjUnavailable
          else MObjSeq (mem_part_access $s_{\mathsf{proc}}$.mm $obj_{\mathsf{addr}}$ $obj_{\mathsf{len}}$)
```

For a start address obj_{addr} and a length obj_{len}, it returns an empty memory object, i. e., MObjSeq [], if $obj_{\mathsf{len}} = 0$. The memory object is undefined, if obj_{addr} or obj_{len} are not word-aligned. If the last address ($obj_{\mathsf{addr}} + obj_{\mathsf{len}}$)

of the memory object is not available in the virtual memory, the object is considered as unavailable. A valid memory object contains a sequence of 32-bit integers of length obj_{len} starting from address obj_{addr}. The sequence is obtained by function mem_part_access.

It is almost the same with the receive buffer starting at addresss buf_{ad} and with length buf_{len}. The function build_buffer generates an abstract buffer, where obj_{addr} denotes the start address and obj_{len} the buffer length:

```
build_buffer $s_{\mathsf{proc}}$ $obj_{\mathsf{addr}}$ $obj_{\mathsf{len}}$ $\equiv$
  if $obj_{\mathsf{len}} = 0$ then BufLength 0
  else if $obj_{\mathsf{addr}}$ mod $4 \neq 0 \vee obj_{\mathsf{len}}$ mod $4 \neq 0$ then BufUndefined
       else if $\mathsf{size}_{\mathsf{asm}}\ s_{\mathsf{proc}} * \mathsf{PAGE\_SIZE} \leq obj_{\mathsf{addr}} + obj_{\mathsf{len}}$ then BufUnavailable
            else BufLength ($obj_{\mathsf{len}}$ div 4)
```

In case that $obj_{\mathsf{len}} = 0$, the resulting buffer has also length 0. The buffer is undefined, if either the start address or the length is not word-aligned. If the specified memory region is defined and available, the buffer is of length obj_{len} div 4. The division by 4 is due to the fact, that users specify the length of a memory region in bytes whereas the kernel uses words.

Converting the specified timeout values to_{snd} and to_{rcv} is straightforward, as the definition of int2timeout suggests:

int2timeout $r \equiv$ **if** $r < 0$ **then** ∞ **else** Fin (nat r)

For negative values r it returns ∞. Positive values are first converted to natural numbers before the result Fin (nat r) is returned.

There are other conversion functions used in the context of ω_{asm}. In this document, however, we only mention them and give a short description but leave out their exact formal definitions and refer to the according theory file in Isabelle/HOL instead.

The function reg2prio converts integer numbers n into abstract priorities. It delivers $\perp$, if n cannot be mapped to any value of prioT. Otherwise, reg2prio returns the corresponding abstract priority. It is used with the calls PROCESS_CREATE and CHG_SCHED_PARAMS.

Regarding the device communication, ω_{proc} applies the conversion functions reg2devnum and reg2port, both converting integer numbers to abstract device and port numbers, respectively. In addition, function reg2ints extracts a set of device interrupts out of an integer number.

Finally, we present a special case of ω_{proc}, namely, if the immediate constant *imm* specifies an undefined trap number. We use predicate undefined_trapnr to determine this situation and the according output is UNDEFINED_TRAP:

$[\![\mathsf{current\_instr}\ s_{\mathsf{proc}} = \mathrm{TRAP}\ i;\ \mathsf{undefined\_trapnr}\ i]\!]$
$\Longrightarrow \mathsf{trap\_dispatch}\ s_{\mathsf{proc}} = \mathrm{UNDEFINED\_TRAP}$

3.3.4 Process Transitions

The function δ_{proc} defines the transition of a VAMP assembly process automaton. As input, it takes the current state s_{proc} and an input i from the kernel. Finally, δ_{proc} acts as wrapper for function δ_{asm}, while taking the kernel input i into account. The kernel input follows the input alphabet Σ_{proc} and provides the basis for the particular case distinctions. As with function ω_{proc}, the definition of δ_{proc} is pretty substantial. For this purpose, we limit ourselves again to exemplary excerpts.

The easiest case treats the input ε_{Σ} from the kernel. It implies that the process has not initiated a kernel request before and that the execution of the current instruction happens in the local context of the process. The definition of δ_{proc} handles this case by a simple call of the assembly transition function δ_{asm}:

$$\delta_{\mathsf{proc}}\ \varepsilon_{\Sigma}\ s_{\mathsf{proc}} \equiv \delta_{\mathsf{asm}}\ s_{\mathsf{proc}}$$

VAMOS uses kernel commands to enforce changes on the process state. Commands are only used in the context of the memory management and allow the increasing or demand the decreasing of the memory amount. Both actions are encapsulated as commands, because they might be triggered by other processes.

Increasing the memory amount of a process is based on the kernel input ADD_MEMORY and the provided number *pgs* of pages:

```
$\delta_{\mathsf{proc}}$ (ADD_MEMORY $pgs$) $s_{\mathsf{proc}} \equiv s_{\mathsf{proc}}$
  $(\!|$sprs := $s_{\mathsf{proc}}$.sprs[SPR_PTL := $s_{\mathsf{proc}}$.sprs ! SPR_PTL + to_int32 $pgs$],
     mm := $\lambda i.$ if size$_{\mathsf{asm}}$ $s_{\mathsf{proc}} * 1024 \leq i <$ (size$_{\mathsf{asm}}$ $s_{\mathsf{proc}} + pgs) * 1024$ then 0
                  else $s_{\mathsf{proc}}$.mm $i|\!)$
```

Due to the additional memory pages, the page table length is incremented by *pgs* and again stored in register SPR_PTL. Furthermore, the additional memory region is initialized with zeros. Note that function $\mathsf{size}_{\mathsf{asm}}$ returns the size of the memory of process s_{proc} in pages of 1024 words.

Releasing memory pages only involves the decreasing of the page table length. Again, the number *pgs* of pages is included in the input message FREE_MEMORY:

```
$\delta_{\mathsf{proc}}$ (FREE_MEMORY $pgs$) $s_{\mathsf{proc}} \equiv s_{\mathsf{proc}}$
  $(\!|$sprs := $s_{\mathsf{proc}}$.sprs
     [SPR_PTL :=
        if $s_{\mathsf{proc}}$.sprs ! SPR_PTL < to_int32 $pgs$ then $-1$
        else $s_{\mathsf{proc}}$.sprs ! SPR_PTL $-$ to_int32 $pgs$]$|\!)$
```

If the number of released pages is greater than the actual number of occupied pages, the page table length is set to -1. Otherwise, the length is decreased by *pgs*.

The remaining kernel inputs can all be traced back to previous kernel requests. Thus, the process has initiated a kernel request by means of a TRAP instruction, the kernel has handled this trap and now returns its results by means of input *i* to the process. Independent from the result of the kernel computation, the TRAP instruction is executed in the context of δ_{asm}. The semantics of TRAP is the incrementation of the program counters as the following statement suggests:

$\mathsf{current\_instr}\ s_{\mathsf{proc}} = \text{TRAP}\ i \Longrightarrow \delta_{\mathsf{asm}}\ s_{\mathsf{proc}} \equiv \mathsf{incPcs}\ s_{\mathsf{proc}}$

with

$\mathsf{incPcs}\ s_{\mathsf{proc}} = s_{\mathsf{proc}}(\!|\mathsf{dpc} := s_{\mathsf{proc}}.\mathsf{pcp},\ \mathsf{pcp} := (s_{\mathsf{proc}}.\mathsf{pcp} + 4)\ \mathsf{mod}\ 2^{32}|\!)$

In addition to that, δ_{proc} also writes result registers and data into the process's virtual memory.

Minor effort is involved with error responses or 'simple' success messages. The former arise, for instance, if the kernel's memory amount does not suffice to create a new process. The kernel acknowledges this situation with ERR_OUT_OF_MEM and δ_{proc} updates the process state as follows:

$\delta_{\mathsf{proc}}\ \text{ERR\_OUT\_OF\_MEM}\ s_{\mathsf{proc}} \equiv \mathsf{incPcs}\ s_{\mathsf{proc}}(\!|\mathsf{gprs} := s_{\mathsf{proc}}.\mathsf{gprs}[22 := -2]|\!)$

Register 22 denotes the standard result register and is set to value -2, the respective integer value to error code ERR_OUT_OF_MEM. Other error codes are of course associated with different integer values.

The kernel message SUCC_NEW_PROCESS signals a successful process creation and is accompanied with the handle hn_{new} to the new process. The latter is written into the result register, such that from now on the process can use this handle to communicate with its child:

$\delta_{\mathsf{proc}}\ (\text{SUCC\_NEW\_PROCESS}\ hn_{\mathsf{new}})\ s_{\mathsf{proc}} \equiv$
$\quad \mathsf{incPcs}\ (s_{\mathsf{proc}}(\!|\mathsf{gprs} := s_{\mathsf{proc}}.\mathsf{gprs}[22 := hn_{\mathsf{new}}]|\!))$

Things get more elaborate, when one result register does not suffice or when abstract values have to be converted into register values. As examples, we consult the impact of responses to successful DEV_READ and IPC_RECEIVE calls.

In the former case, the response SUCC_DEV_READ also comprises device data *is* represented as a sequence of 32-bit integers. Depending on a single or burst read, the device data needs to be written into a register or the process's memory:

```
$\delta_{\mathsf{proc}}\ (\text{SUCC\_DEV\_READ}\ is)\ s_{\mathsf{proc}} \equiv$
  **let** $read_{\mathsf{single}} = \mathsf{instr\_mem\_read}\ s_{\mathsf{proc}}.\mathsf{mm}\ s_{\mathsf{proc}}.\mathsf{dpc} = \text{TRAP}\ 19$;
      $s_{\mathsf{single}} = s_{\mathsf{proc}}(\!|\mathsf{gprs} := s_{\mathsf{proc}}.\mathsf{gprs}[13 := \mathsf{hd}\ is]|\!)$;
      $buf_{\mathsf{addr}} = \mathsf{to\_nat32}\ (s_{\mathsf{proc}}.\mathsf{gprs}\ !\ 13)$;
      $s_{\mathsf{block}} = s_{\mathsf{proc}}(\!|\mathsf{mm} := \mathsf{mem\_part\_update}\ s_{\mathsf{proc}}.\mathsf{mm}\ buf_{\mathsf{addr}}\ is|\!)$
  **in if** $read_{\mathsf{single}}$ **then** $\mathsf{incPcs}\ (s_{\mathsf{single}}(\!|\mathsf{gprs} := s_{\mathsf{single}}.\mathsf{gprs}[22 := 0]|\!))$
     **else** $\mathsf{incPcs}\ (s_{\mathsf{block}}(\!|\mathsf{gprs} := s_{\mathsf{block}}.\mathsf{gprs}[22 := 0]|\!))$
```

The distinction between single and burst reads relies on the trap number. Instruction TRAP 19 implies a single read, TRAP 13 a burst read operation. In the former case, only the first word of the data sequence *is* is written into register 13.

While performing a burst read, register 13 contains the start address $\mathit{buf}_{\mathsf{addr}}$ of the buffer, the device data should be stored in. As the operation was acknowledged with a SUCC_DEV_READ message, it is ensured that the device data fits into the specified buffer. The actual data transfer is done by means of mem_part_update which simply takes the sequence *is* and writes it into the virtual memory, starting at address $\mathit{buf}_{\mathsf{addr}}$. In both cases, register 22 finally contains the value 0 (meaning "success") and the program counters are increased.

Even more complex is the impact of the response SUCC_RECEIVE to an IPC_RECEIVE call. It contains the message and handles as well as the rights to the sender and (possibly) to an additional process:

```
$\delta_{\mathsf{proc}}$ (SUCC_RECEIVE $rhn_{\mathsf{snd}}$ $reused_{\mathsf{snd}}$ $rrights_{\mathsf{snd}}$ $msg_{\mathsf{rcv}}$ $rhn_{\mathsf{add}}$ $reused_{\mathsf{add}}$ $rrights_{\mathsf{add}}$
        $stolen$ $to_{\mathsf{infty}}$ $irqs$)
 $s_{\mathsf{proc}} \equiv$
  let $\mathit{buf}_{\mathsf{addr}}$ = to_nat32 ($s_{\mathsf{proc}}$.gprs ! 15); $\mathit{rights}_{\mathsf{snd}}$ = rights2num $rrights_{\mathsf{snd}}$;
      $\mathit{rights}_{\mathsf{add}}$ =
        if $rhn_{\mathsf{add}}$ = HN_NONE then $s_{\mathsf{proc}}$.gprs ! 18 else rights2num $rrights_{\mathsf{add}}$;
      $\mathit{res}$ = combine_notifications $reused_{\mathsf{snd}}$ $reused_{\mathsf{add}}$ $stolen$ $to_{\mathsf{infty}}$ $irqs$
  in incPcs
       ($s_{\mathsf{proc}}$
        (|mm := mem_part_update $s_{\mathsf{proc}}$.mm $\mathit{buf}_{\mathsf{addr}}$ $msg_{\mathsf{rcv}}$,
          gprs := $s_{\mathsf{proc}}$.gprs
            [11 := $rhn_{\mathsf{snd}}$, 12 := $\mathit{rights}_{\mathsf{snd}}$, 16 := to_int32 (length $msg_{\mathsf{rcv}}$ * 4),
             17 := $rhn_{\mathsf{add}}$, 18 := $\mathit{rights}_{\mathsf{add}}$, 22 := $\mathit{res}$]|))
```

The message msg_{rcv} is given as a sequence of integers and copied, by means of function mem_part_update, into the memory of the process starting at the specified address $\mathit{buf}_{\mathsf{addr}}$ in register 15. The length of the sequence is written into register 16. Remember that users measure the length in bytes whereas the kernel uses words. Thus, we multiply the length with 4. Registers 11 and 17 finally contain the handles rhn_{snd} and rhn_{add} to the sender and to an additional process, respectively. Besides the process handles, the receiver obtains rights. Set $rrights_{\mathsf{snd}}$ contains the rights to the sender and $rrights_{\mathsf{add}}$ specifies the rights to the additional process. Function rights2num converts these abstract set representations into the corresponding numerical ones:

```
rights2num $rights$ $\equiv$
  (if v_finiteR $\in$ $rights$ then 1 else 0) +
  (if v_multipleR $\in$ $rights$ then 2 else 0) +
  (if v_requestR $\in$ $rights$ then 4 else 0) +
  (if v_sendR $\in$ $rights$ then 8 else 0)
```

For a well-defined set of rights *rights*, it determines the active rights and sums up the associated numerical values. In our case, the results are buffered in $\mathit{rights}_{\mathsf{snd}}$ and $\mathit{rights}_{\mathsf{add}}$ and finally stored in registers 12 and 18. However, if no additional process was specified, i. e., $rhn_{\mathsf{add}} = \mathsf{HN\_NONE}$, $\mathit{rights}_{\mathsf{add}}$ is set to the value of register 18, i. e., in this case the register remains unchanged.

Finally, the function combine_notifications takes all kinds of kernel notifications and generates a result value *res* which is stored in register 22:

$\mathsf{combine\_notifications}\ \mathit{reused}_{\mathsf{snd}}\ \mathit{reused}_{\mathsf{add}}\ \mathit{stolen}\ \mathit{to}_{\mathsf{infty}}\ \mathit{irqs} \equiv$
$(\textbf{if}\ \mathit{reused}_{\mathsf{snd}}\ \textbf{then}\ 1\ \textbf{else}\ 0) + (\textbf{if}\ \mathit{reused}_{\mathsf{add}}\ \textbf{then}\ 2\ \textbf{else}\ 0) +$
$(\textbf{if}\ \mathit{to}_{\mathsf{infty}}\ \textbf{then}\ 4\ \textbf{else}\ 0) +$
$(\textbf{if}\ \mathit{stolen}\ \textbf{then}\ 8\ \textbf{else}\ 0) +$
$(\sum i{\in}\mathit{irqs}.\ 2^{\mathsf{dev2nat}\ i})$

Chapter 4

The Vamos Model

Contents

This chapter deals with the details of the VAMOS automaton describing the VAMOS kernel and the user processes running on it. The devices are not included but associated by a communication interface.

As already introduced, the tupel $\mathcal{A}_\mathsf{V}$ describes the VAMOS automaton:

$$\mathcal{A}_\mathsf{V} = (\mathcal{S}_\mathsf{V}, \mathcal{S}^0_\mathsf{V}, \Sigma_\mathsf{V}, \Omega_\mathsf{V}, \omega_\mathsf{V}, \delta_\mathsf{V})$$

Subsequently, we give a precise description of the particular components starting with the VAMOS state space $\mathcal{S}_\mathsf{V}$. We continue with the communication between the kernel and the devices which is based on the interface established by the alphabets Σ_V and Ω_V and the output function ω_V. The chapter concludes with the transition function δ_V processing possible input from the devices and defining the interaction with the user processes.

4.1 The Vamos State Space

The VAMOS state space $\mathcal{S}_\mathsf{V}$ represents the abstract counterpart of the VAMOS implementation state, which was introduced in Section 2.5.2. For efficiency reasons, the latter maintains partly redundant datastructures. On the abstract level, we avoid these redundancies and reduce $\mathcal{S}_\mathsf{V}$ to the essential. Formally, a record describes the VAMOS state space and a state $s_\mathsf{V} \in \mathcal{S}_\mathsf{V}$ comprises the following components:

- the mapping of virtual assembly machines $s_{\mathsf{V}}.\mathsf{procs}$ of the user processes,
- the priority database $s_{\mathsf{V}}.\mathsf{priodb}$,
- the scheduling datastructures $s_{\mathsf{V}}.\mathsf{schedds}$,
- the datastructures $s_{\mathsf{V}}.\mathsf{rightsdb}$ containing information for the IPC rights management and the set of privileged processes,
- the send status database $s_{\mathsf{V}}.\mathsf{sndstatdb}$, and
- the datastructures $s_{\mathsf{V}}.\mathsf{devds}$ containing data for the device communication.

More details on the particular components present the following paragraphs. Afterwards we present the initial states represented by $\mathcal{S}^0_{\mathsf{V}} \subset \mathcal{S}_{\mathsf{V}}$.

4.1.1 Virtual User Machines

Regarding the user machines, the VAMOS model uses the same level of abstraction as the implementation: virtual assembly machines. VAMOS employs unique process numbers of type $\mathsf{procnumT}$ to identify the user processes. Accordingly, component $s_{\mathsf{V}}.\mathsf{procs}$ realizes a partial mapping between process numbers and virtual assembly machines:

$$s_{\mathsf{V}}.\mathsf{procs} \in \mathsf{procnumT} \rightharpoonup \mathcal{S}_{\mathsf{asm}}$$

The model only maintains the machines of active processes. Entries for inactive processes are set to $\perp$.

Apart from actually representing the virtual user machines, the state component $s_{\mathsf{V}}.\mathsf{procs}$ also implicitly carries system-relevant information. Thus, based on $s_{\mathsf{V}}.\mathsf{procs}$, VAMOS determines free resources regarding new processes and memory.

Availability of Processes. New processes are taken out of the set of inactive ones. As a consequence, the existence of inactive processes is a main prerequisite regarding the process creation. Based on the observation that entries in $s_{\mathsf{V}}.\mathsf{procs}$ for inactive processes are set to $\perp$, predicate $\mathsf{procs\_available}$ checks for available virtual machines that can be assigned to a new process:

$$\mathsf{procs\_available}\ s_{\mathsf{V}}.\mathsf{procs} \equiv \exists p.\ s_{\mathsf{V}}.\mathsf{procs}\ p = \perp$$

Availability of Memory. Another prerequisite concerns the availability of memory. Function $\mathsf{total\_mem}$ computes the current memory consumption of VAMOS, i. e., the memory amount of all active processes. Based on the output function $\mathsf{size}_{\mathsf{proc}}$ for user processes, it sums up the memory amounts of the active processes:

$$\begin{aligned}&\mathsf{total\_mem}\ s_{\mathsf{V}}.\mathsf{procs} \equiv\\ &\quad \mathsf{list\_sum}\ (\mathsf{map}\ (\lambda x.\ \mathbf{case}\ s_{\mathsf{V}}.\mathsf{procs}\ x\ \mathbf{of}\ \bot \Rightarrow 0 \mid \lfloor p \rfloor \Rightarrow \mathsf{size}_{\mathsf{proc}}\ p)\ [1..{<}\mathsf{PID\_MAX}])\end{aligned}$$

The total virtual memory amount of the VAMOS kernel is restricted to $\mathsf{TVM\_MAXPAGES}$ pages. Thus, if a new process should be equipped with n pages, the predicate $\mathsf{mem\_available}$ has to hold:

$$\mathsf{mem\_available}\ s_{\mathsf{V}}.\mathsf{procs}\ n \equiv \mathsf{total\_mem}\ s_{\mathsf{V}}.\mathsf{procs} + n < \mathsf{TVM\_MAXPAGES}$$

It determines the overall memory amount and checks whether this value together with the additional n pages does exceed the limit of $\mathsf{TVM\_MAXPAGES}$ pages.

4.1.2 Priorities

Similar to the implementation, the model also supports three different priority levels. Priority values are of type prioT and the mapping $s_{\mathsf{V}}.\mathsf{priodb}$ assigns such values to the particular processes:

$$s_{\mathsf{V}}.\mathsf{priodb} \in \mathsf{procnumT} \rightharpoonup \mathsf{prioT}$$

Again, process numbers are used to distinguish between the different processes and entries for inactive ones are set to $\bot$.

4.1.3 Scheduling Data Structures

The component $s_{\mathsf{V}}.\mathsf{schedds}$ represents the scheduling data structures. Defined as record of type $\mathsf{scheddsT}$ they provide global as well as process-specific scheduling data:

- the current time $s_{\mathsf{V}}.\mathsf{schedds.time} \in \mathbb{N}$,
- the ready queues $s_{\mathsf{V}}.\mathsf{schedds.ready} \in \mathsf{prioT} \rightharpoonup \mathsf{procnumT}^*$,
- the wait queue $s_{\mathsf{V}}.\mathsf{schedds.wait} \in \mathsf{procnumT}^*$
- the inactive queue $s_{\mathsf{V}}.\mathsf{schedds.inactive} \in \mathsf{procnumT}^*$, and
- the process-specific scheduling information

$$s_{\mathsf{V}}.\mathsf{schedds.procdb} \in \mathsf{procnumT} \rightharpoonup \mathsf{procdbT}.$$

The current time s_V.schedds.time is a counter for clock ticks, i. e., interrupts from the external timer device. To realize the scheduling, the VAMOS scheduler maintains different queues. They are represented as finite sequences of process numbers in the VAMOS specification. There is a ready queue s_V.schedds.ready *prio* of schedulable processes for each priority *prio*. Additionally, all processes currently not ready to be scheduled (because they are waiting for an IPC partner) are held in the queue s_V.schedds.wait. Inactive ones are stored in s_V.schedds.inactive which, at the same time, determines the order of the reuse of the process numbers.

Only active processes participate in the scheduling mechanism. The relevant information is collected in a record of type procdbT with the following components:

- the timeslice tsl $\in \mathbb{N}$,
- the amount of consumed time ctsl $\in \mathbb{N}$, and
- the absolute timeout timeout $\in$ timeoutT.

If a process is found to be computing when a timer interrupt raises, the component ctsl is increased until the process has finally run for tsl ticks. In this case, another process is scheduled. If a process calls the kernel for IPC and no partner is ready for communication, the absolute timeout timeout is computed from the current time and the relative timeout that has been specified with the call. Timeouts are defined as natural numbers with infinity which are represented by type timeoutT.

As usual, the mapping is not defined for inactive processes.

Current Process. In VAMOS, the current process is the first process in the highest, non-empty ready queue. If all ready queues are empty, the current process is undefined. Or more technically, we concatenate the ready queues from the highest to the lowest and specify the first process in this list as current:

```
v_cup schedds ≡
  case concat (map schedds.ready [high_prio, med_prio, low_prio]) of [] ⇒ ⊥
  | x · xs ⇒ ⌊x⌋
```

While specifying the VAMOS kernel we often have to wake up or put processes to sleep. Both actions only concern the scheduling datastructures and are encapsulated in dedicated functions.

Waking Up Processes. Usually, a waiting process x is part of the wait queue s_V.schedds.wait. After waking up, however, x is no longer located in this queue but appended to the ready queue of its priority. In addition, x gets back its full timeslice again, i. e., the consumed time is reset to 0.

All these actions are combined in function $\mathsf{v\_wkp\_updt\_schedds}$; it provides the possibility to wake up several processes simultaneously. Candidates are given as a list *wkp* which also defines the order of reintegration into the ready queues. In order to determine the process priorities the function takes the priority database *priodb* and updates the scheduling datastructures *schedds*:

$$\begin{array}{l}
\mathsf{v\_wkp\_updt\_schedds}\ \mathit{schedds}\ \mathit{wkp}\ \mathit{priodb} \equiv \mathit{schedds} \\
\quad (\!|\mathsf{ready} := \lambda \mathsf{p}.\ \mathit{schedds}.\mathsf{ready}\ \mathsf{p}\ @\ [\mathsf{x}{\in}\mathit{wkp}\ .\ \lceil \mathit{priodb}\ \mathsf{x}\rceil = \mathsf{p}], \\
\qquad \mathsf{wait} := [\mathsf{x}{\in}\mathit{schedds}.\mathsf{wait}\ .\ \neg\ \mathsf{x}\ \mathsf{mem}\ \mathit{wkp}], \\
\qquad \mathsf{procdb} := \lambda \mathsf{x}.\ \textbf{if}\ \mathsf{x} \in \mathsf{set}\ \mathit{wkp}\ \textbf{then}\ \lfloor \lceil \mathit{schedds}.\mathsf{procdb}\ \mathsf{x}\rceil (\!|\mathsf{ctsl} := 0|\!)\rfloor \\
\qquad\qquad\qquad\quad \textbf{else}\ \mathit{schedds}.\mathsf{procdb}\ \mathsf{x}|\!)
\end{array}$$

Putting Processes To Sleep. Putting processes to sleep is provided by the function $\mathsf{v\_ipc\_wait\_updt\_schedds}$. It is mainly used in the context of IPC, where processes are often forced to wait for an IPC partner. If a process p_{wait} of priority $\mathit{prio}_{\mathsf{wait}}$ has to wait, it is removed from the corresponding ready queue and appended to the wait queue. The duration of waiting depends on the relative timeout value $\mathit{to}_{\mathsf{wait}}$ which is added to the current time. The formal definition of these actions is given as follows:

$$\begin{array}{l}
\mathsf{v\_ipc\_wait\_updt\_schedds}\ \mathit{schedds}\ p_{\mathsf{wait}}\ \mathit{prio}_{\mathsf{wait}}\ \mathit{to}_{\mathsf{wait}} \equiv \mathit{schedds} \\
\quad (\!|\mathsf{ready} := \mathit{schedds}.\mathsf{ready}(\mathit{prio}_{\mathsf{wait}} := [\mathsf{x}{\in}\mathit{schedds}.\mathsf{ready}\ \mathit{prio}_{\mathsf{wait}}\ .\ \mathsf{x} \neq p_{\mathsf{wait}}]), \\
\qquad \mathsf{wait} := \mathit{schedds}.\mathsf{wait}\ @\ [p_{\mathsf{wait}}], \\
\qquad \mathsf{procdb} := \mathit{schedds}.\mathsf{procdb}(p_{\mathsf{wait}} \mapsto \lceil \mathit{schedds}.\mathsf{procdb}\ p_{\mathsf{wait}}\rceil \\
\qquad\quad (\!|\mathsf{timeout} := \mathit{to}_{\mathsf{wait}} \mathrel{+}= \mathit{schedds}.\mathsf{time}|\!))|\!)
\end{array}$$

4.1.4 Rights Datastructures

As in the implementation, the abstract rights datastructures are process-specific and hold information regarding the IPC rights management and the set of privileged processes:

$$s_{\mathsf{V}}.\mathsf{rightsdb} \in \mathsf{procnumT} \rightharpoonup \mathsf{rightsdataT}$$

The entries for active processes describe record values of type $\mathsf{rightsdataT}$ with the following components:

- the privileged flag $\mathsf{priv} \in \mathsf{bool}$,
- the process number $\mathsf{parent} \in \mathsf{procnumT}_{\perp}$ of the parent,
- the handle database $\mathsf{hdb} \in \mathsf{handleT} \rightharpoonup \mathsf{procnumT}$,
- the set of stolen handles $\mathsf{stolen} \in \mathsf{handleT\ set}$, and
- the rights database $\mathsf{rdb} \in \mathsf{procnumT}_{\perp} \rightharpoonup \mathsf{rightsT}$.

The privileged flag priv determines whether a process is privileged or not. Component parent denotes the identifier of the parent. For a process p, it delivers $\perp$, if p is either the initial process (the OS) or a direct child of the OS, or if the parent has been terminated in the meantime. The component hdb maintains the mapping between the process-local handles and the actual process numbers. If a handle is stolen, it is added to the set stolen. Component rdb finally determines the rights to other processes. Entry $\lceil s_V.\mathsf{rightsdb}\ p\rceil.\mathsf{rdb}\ q$ specifies, for instance, the rights of process p to a process q. However, if q does not denote a valid process number, i. e., $q = \perp$, the rights database returns $\perp$.

Privileged Processes. The privileged flag priv plays a decisive role in the Vamos specification later on and is therefore encapsulated in predicate privileged:

$\mathsf{privileged}\ s_V.\mathsf{rightsdb}\ p \equiv \lceil s_V.\mathsf{rightsdb}\ p\rceil.\mathsf{priv}$

Valid Handles. Predicate valid_handle signals a valid handle hn in the context of process p:

$\mathsf{valid\_handle}\ s_V.\mathsf{rightsdb}\ p\ hn \equiv \exists p_{\mathsf{hn}}.\ \lceil s_V.\mathsf{rightsdb}\ p\rceil.\mathsf{hdb}\ hn = \lfloor p_{\mathsf{hn}}\rfloor$

It only holds, if the access to the handle database of process p with handle hn returns a process number. If hn is invalid, the access results in $\perp$.

Converting Handles to Process Numbers. The opposite direction, i. e., determining the handle hn to process q in the handle database of process p, defines function to_hn:

$\mathsf{to\_hn}\ \mathit{rightsdb}\ p\ q \equiv \textbf{if}\ \lfloor q\rfloor = \lceil \mathit{rightsdb}\ p\rceil.\mathsf{parent}\ \textbf{then}\ \mathsf{HN\_PARENT}\ \textbf{else}\ \mathsf{int}\ q$

If q denotes the process number of p's parent, handle HN_PARENT is returned. Otherwise, to_hn realizes the one-to-one mapping between handles and process numbers.

Known Processes. The model also determines known processes by means of the handle database hdb. A process q is known by process p iff there exists a handle hn in the context of p pointing to q. The corresponding predicate is_known is defined as follows:

$\mathsf{is\_known}\ \mathit{rightsdb}\ p\ q \equiv \exists \mathit{hn}.\ \lceil \mathit{rightsdb}\ p\rceil.\mathsf{hdb}\ \mathit{hn} = \lfloor q\rfloor$

4.1.5 Send Status Database

The remaining component of the process state in the implementation defines the send status database

$$s_{\mathsf{V}}.\mathsf{sndstatdb} \in \mathsf{procnumT} \rightharpoonup \mathsf{bool}$$

Usually, the membership in one of the scheduling queues determines the current process status—except for one case: An IPC_REQUEST call has a send and a receive phase, which might both require waiting. The send status database $s_{\mathsf{V}}.\mathsf{sndstatdb}$ keeps track of the phase.

4.1.6 Device Data Structures

The state component $s_{\mathsf{V}}.\mathsf{devds}$ encapsulates the device datastructures which are represented as a record of type devdsT with the following components:

- the driver registry $\mathsf{driver} \in \mathsf{devnumT} \rightharpoonup \mathsf{procnumT}$,
- the currently enabled device interrupts $\mathsf{enabled} \in \mathsf{intsT}$, and
- the occured and saved device interrupts $\mathsf{saved} \in \mathsf{intsT}$.

The driver registry assigns device numbers with the corresponding process numbers of the drivers. For instance, if process p acts as driver for device d, $\lceil s_{\mathsf{V}}.\mathsf{devds}.\mathsf{driver}\ d \rceil = p$. If no driver is assigned with device d, the mapping delivers $\perp$.

The set of currently enabled interrupts is given by $s_{\mathsf{V}}.\mathsf{devds}.\mathsf{enabled}$.

It might happen that drivers are not ready to handle arising interrupts, for instance, when they are currently handling interrupts from other devices. In order to prevent any interrupt losses, VAMOS stores occurred but not yet handled interrupts in component saved for a later delivery.

4.1.7 Initial Vamos States

The initial VAMOS state (i. e., the state describing the system when it switches to the user mode for the first time after the reset signal) depends on the code image *OS_IMAGE* for the user-level operating system (OS). In the implementation, this image is stored at the swap harddisk, which is abstracted away in the CVM model. Furthermore, the initial size of the OS-process memory is configured in the implementation via the macro-constant *OS_PAGES*. This constant has to be smaller than $\mathsf{TVM\_MAXPAGES}$, the maximum virtual memory size supported by CVM. We represent the OS image as a list of integers and require that (a) the OS image is not too large with respect to *OS_PAGES* and (b) all list elements are valid integers with respect to the 32-bit target VAMP machine. Furthermore, VAMOS determines some constants regarding the OS: The operating system is identified by process number $\mathsf{OS\_PID}$ and classified in the priority class $\mathsf{OS\_PRIO}$. Furthermore, its timeslice is specified with $\mathsf{OS\_TSL}$.

Based on these observations function $\mathsf{initialConf}$ delivers the initial VAMOS state:

```
initialConf $OS\_IMAGE\ OS\_PAGES \equiv$
 let $os\_procdb$ = (|tsl = OS_TSL, ctsl = 0, timeout = $\infty$|);
     $init\_schedds$ =
       (|time = 0, ready = $\lambda pri$. if $pri$ = OS_PRIO then [OS_PID] else [],
         wait = [],
         inactive =
           [$x$∈map Abs_procnumT (rev [1..<PID_MAX]) . $x \neq$ OS_PID],
         procdb = [OS_PID $\mapsto$ $os\_procdb$]|);
     $os\_rightsdb$ =
       (|priv = True, hdb = [HN_SELF $\mapsto$ OS_PID], rdb = empty,
         stolen = {}, parent = $\bot$|);
     $init\_devds$ = (|driver = empty, enabled = {DEV_TIMER}, saved = {}|)
 in (|procs = [OS_PID $\mapsto$ $\mathsf{init}_{\mathsf{proc}}$ $OS\_IMAGE\ OS\_PAGES$],
      schedds = $init\_schedds$, priodb = [OS_PID $\mapsto$ OS_PRIO],
      sndstatdb = [OS_PID $\mapsto$ False],
      rightsdb = [OS_PID $\mapsto$ $os\_rightsdb$], devds = $init\_devds$|)
```

The only entry in the partial mapping of procs relates to OS_PID. Relying on the given code image OS_IMAGE together with the number OS_PAGES, function $\mathsf{init}_{\mathsf{proc}}$ (cf. Section 3.3) sets up the initial virtual machine for the OS.

The component time of the initial scheduling datastructures *init_schedds* starts counting at 0. As the only running process, the OS is put into the ready queue of priority OS_PRIO. The other ready queues as well as the wait queue are empty because all remaining processes are classified as inactive and reside in the queue inactive. The latter orders the processes by their process numbers where PID_MAX denotes the head and 1 the end of the queue. Certainly, process number OS_PID is left out.

The OS-specific scheduling information *os_procdb* pretends that the OS did not yet consumed any time of its timeslice OS_TSL and that there is no pending timeout denoted by ∞.

Similar to procs, the partial mappings priodb, sndstatdb and rightsdb only contain a single valid entry, namely, the one for OS_PID. While the priority of the OS is set to OS_PRIO, the entry in sndstatdb is initialized with False.

As the initial rights datastructures *os_rightsdb* suggest, the OS runs in privileged mode. The only valid entry in the handle database hdb describes the self-reference. The partial mapping of rdb is empty because no other processes yet exist and, thus, no rights need to be maintained. The set stolen is empty and component parent is set to $\bot$.

Finally, initialConf sets up the initial device datastructures *init_devds*. The only enabled interrupts are the ones from the timer device (DEV_TIMER) which are handled by VAMOS itself. So far, no drivers are assigned to any devices and no interrupts are saved for later delivery.

4.2 The Output Function

Similar to the process automata, the VAMOS automaton $\mathcal{A}_V$ possesses an output function ω_V, in order to generate the kernel's output to the devices. As only processes can initiate device interaction by invoking the corresponding VAMOS calls, ω_V reverts to the process outputs. In particular, the output of the VAMOS kernel in state s_V depends on the currently running process. The definition of ω_V points this out:

```
$\omega_V\ s_V \equiv$
  case $\mathsf{v\_cup}\ s_V.\mathsf{schedds}$ of $\bot \Rightarrow \mathsf{idle}_{\Omega V}$
  | $\lfloor p_{cp} \rfloor \Rightarrow \omega\mathsf{proc2devout}\ (\omega_{\mathsf{proc}}\ \lceil s_V.\mathsf{procs}\ p_{cp}\rceil)\ s_V.\mathsf{devds}\ p_{cp}$
```

A non-existent running process, i. e., $\mathsf{v\_cup}\ s_V.\mathsf{schedds} = \bot$, cannot activate any device interaction. Thus, ω_V returns $\mathsf{idle}_{\Omega V}$.

If a current process p_{cp} exists, VAMOS applies function $\omega\mathsf{proc2devout}$. For the output out_{cp} of p_{cp} and the device datastructures $devds$ it returns the according output to the devices:

```
$\omega\mathsf{proc2devout}\ out_{cp}\ devds\ p_{cp} \equiv$
  case $out_{cp}$ of
  DEV_READ $dev_{id}\ dev_{port}\ buffer \Rightarrow$
    if $\mathsf{is\_error}\ (\mathsf{vamos\_result\_dev\_read}\ devds\ p_{cp}\ dev_{id}\ dev_{port}\ buffer)$
    then $\mathsf{idle}_{\Omega V}$ else $\mathsf{read}_{\Omega V}\ \lceil dev_{id}\rceil\ \lceil dev_{port}\rceil\ (\mathsf{bufLength}\ buffer)$
  | DEV_WRITE $dev_{id}\ dev_{port}\ data \Rightarrow$
      if $\mathsf{is\_error}\ (\mathsf{vamos\_result\_dev\_write}\ devds\ p_{cp}\ dev_{id}\ dev_{port}\ data)$
      then $\mathsf{idle}_{\Omega V}$ else $\mathsf{write}_{\Omega V}\ \lceil dev_{id}\rceil\ \lceil dev_{port}\rceil\ (\mathsf{map\ to\_nat32}\ (\mathsf{mObjSeq}\ data))$
  | _ $\Rightarrow \mathsf{idle}_{\Omega V}$
```

All process outputs out_{cp} different from DEV_READ and DEV_WRITE have no bearing on the devices and thus result in the output $\mathsf{idle}_{\Omega V}$. However, as the kernel only forwards promising requests, DEV_READ and DEV_WRITE do not automatically cause kernel outputs. At first, it must be ensured that the corresponding kernel call will not result in any error message. For the DEV_READ call this happens with function vamos_result_dev_read and function vamos_result_dev_write performs this task for the call DEV_WRITE. For the proper definitions of both result functions we refer to the specifications of the just mentioned VAMOS calls (cf. Section 4.4.5). In a nutshell, if the kernel detects any errors while requesting any device, e. g., p_{cp} is not a registered driver, the result functions return the corresponding error code. Any error code satisfies predicate is_error and VAMOS represses the device interaction, i. e., the output denotes $\mathsf{idle}_{\Omega V}$. In case that VAMOS detects no errors, function $\omega\mathsf{proc2devout}$ takes the process output out_{cp} and converts it into output to the devices. An output of type $\mathsf{read}_{\Omega V}$ initiates a read request whereas $\mathsf{write}_{\Omega V}$ is used for writing.

4.3 The Transition Function

The intention of the VAMOS kernel is the handling and processing of (a) external device interrupts and data, and (b) user-generated interrupts. All actions are encapsulated in the transition function δ_V carrying the VAMOS automaton from one state over into an adjacent one. As the devices are not part of the VAMOS model, function δ_V gets additional external input. This input is given as a tuple consisting of a set of interrupts $irqs$ and device data w. Requests from the user processes can be derived from a VAMOS state s_V.

Accordingly, the definition of δ_V looks as follows:

```
$\delta_V\ (irqs,\ w)\ s_V \equiv$
 let $\mathit{handle\_trap}$ =
     $\lambda s.$ if $\mathsf{v\_cup}\ s.\mathsf{schedds} \neq \bot$ then $\mathsf{vamosDispatcher}\ s\ w$ else $s$;
   $\mathit{handle\_timer}$ =
     $\lambda s.$ if $\mathsf{DEV\_TIMER} \in irqs$
        then $\mathsf{vamosScheduler}\ s\ (\mathsf{v\_cup}\ s_V.\mathsf{schedds})$ else $s$;
   $\mathit{handle\_extint}$ =
     $\lambda s.\ \mathsf{vamosInterruptDelivery}\ s\ (irqs - \{\mathsf{DEV\_TIMER}\})$
 in $\mathit{handle\_extint}\ (\mathit{handle\_timer}\ (\mathit{handle\_trap}\ s_V))$
```

Within a transition, the VAMOS kernel traverses up to three phases:

1. If the current process $p_{\mathsf{cp}} = \lceil \mathsf{v\_cup}\ s_V.\mathsf{schedds} \rceil$ is defined, we call the VAMOS trap handler vamosDispatcher. It consults the output ω_{proc} $\lceil s_V.\mathsf{procs}\ p_{\mathsf{cp}} \rceil \in \Omega_{\mathsf{proc}}$ and computes the response according to the current VAMOS state and the (possible) device data w (cf. Section 4.4).

2. If the timer-interrupt line is raised, i. e., $\mathsf{DEV\_TIMER} \in irqs$, the timer-interrupt handler vamosScheduler is invoked (cf. Section 4.5).

3. Finally, VAMOS delivers the occurred interrupts to the according drivers. The interrupt delivery is defined by function vamosInterruptDelivery and introduced in Section 4.6.

The upcoming sections introduce the dedicated specification functions of the particular phases.

4.4 Vamos Trap Handler

The main purpose of the VAMOS trap handler is the handling of user-generated exceptions, like traps or runtime errors. A single run of the kernel only considers the exceptions of the currently running process p_{cp}. Hence, the invocation of function vamosDispatcher is bounded to the existence of p_{cp}. For a VAMOS state s_V, this process is given by $p_{\mathsf{cp}} = \lceil \mathsf{v\_cup}\ s_V.\mathsf{schedds} \rceil$. Handling traps leads to the invocation of the respective kernel calls which, in turn, might involve device interaction. As described in Section 4.3, the possible device data is given as input $data_{\mathsf{dev}}$ to vamosDispatcher and integrated into the subsequent operations which update the original state s_V:

```
$\mathsf{vamosDispatcher}\ s_V\ data_{\mathsf{dev}} \equiv$
 $\mathbf{let}\ p_{cp} = \lceil \mathsf{v\_cup}\ s_V.\mathsf{schedds}\rceil$
 $\mathbf{in\ case}\ \omega_{\mathsf{proc}}\ \lceil s_V.\mathsf{procs}\ p_{cp}\rceil\ \mathbf{of}$
    $\textsc{process\_create}\ tsl\ prio\ img\ pgs \Rightarrow$
      $\mathsf{vamos\_process\_create}\ s_V\ tsl\ prio\ img\ pgs$
    $|\ \textsc{process\_clone}\ hn \Rightarrow \mathsf{vamos\_process\_clone}\ s_V\ hn$
    $|\ \textsc{process\_kill}\ hn \Rightarrow \mathsf{vamos\_process\_kill}\ s_V\ hn$
    $|\ \textsc{chg\_sched\_params}\ hn\ tsl\ prio \Rightarrow$
       $\mathsf{vamos\_change\_sched\_param}\ s_V\ hn\ tsl\ prio$
    $|\ \textsc{set\_privileged}\ hn \Rightarrow \mathsf{vamos\_set\_privileges}\ s_V\ hn$
    $|\ \textsc{memory\_add}\ hn\ pages \Rightarrow \mathsf{vamos\_memory\_add}\ s_V\ hn\ pages$
    $|\ \textsc{memory\_free}\ hn\ pages \Rightarrow \mathsf{vamos\_memory\_free}\ s_V\ hn\ pages$
    $|\ \textsc{change\_driver}\ hn\ irqs\ register \Rightarrow$
       $\mathsf{vamos\_change\_driver}\ s_V\ hn\ irqs\ register$
    $|\ \textsc{enable\_interrupts}\ irqs \Rightarrow \mathsf{vamos\_enable\_interrupts}\ s_V\ irqs$
    $|\ \textsc{dev\_read}\ devid\ port\ buffer \Rightarrow$
       $\mathsf{vamos\_dev\_read}\ s_V\ devid\ port\ buffer\ data_{\mathsf{dev}}$
    $|\ \textsc{dev\_write}\ devid\ port\ data \Rightarrow \mathsf{vamos\_dev\_write}\ s_V\ devid\ port\ data$
    $|\ \textsc{ipc\_send}\ hn_{\mathsf{rcv}}\ rights_{\mathsf{snd}}\ msg\ hn_{\mathsf{add}}\ rights_{\mathsf{add}}\ to_{\mathsf{snd}} \Rightarrow$
       $\mathsf{vamos\_ipc\_send}\ s_V\ hn_{\mathsf{rcv}}\ rights_{\mathsf{snd}}\ msg\ hn_{\mathsf{add}}\ rights_{\mathsf{add}}\ to_{\mathsf{snd}}$
    $|\ \textsc{ipc\_receive}\ hn_{\mathsf{snd}}\ msg\ to_{\mathsf{rcv}} \Rightarrow \mathsf{vamos\_ipc\_receive}\ s_V\ hn_{\mathsf{snd}}\ msg\ to_{\mathsf{rcv}}$
    $|\ \textsc{ipc\_request}\ hn_{\mathsf{rcv}}\ rights_{\mathsf{snd}}\ msg\ hn_{\mathsf{add}}\ rights_{\mathsf{add}}\ to_{\mathsf{snd}}\ buffer\ to_{\mathsf{rcv}} \Rightarrow$
       $\mathsf{vamos\_ipc\_send\_receive}\ s_V\ hn_{\mathsf{rcv}}\ rights_{\mathsf{snd}}\ msg\ hn_{\mathsf{add}}\ rights_{\mathsf{add}}$
        $to_{\mathsf{snd}}\ buffer\ to_{\mathsf{rcv}}$
    $|\ \textsc{change\_rights}\ hn_{\mathsf{subj}}\ hn_{\mathsf{obj}}\ grant\ rights \Rightarrow$
       $\mathsf{vamos\_change\_rights}\ s_V\ hn_{\mathsf{subj}}\ hn_{\mathsf{obj}}\ grant\ rights$
    $|\ \textsc{read\_kernel\_info}\ note \Rightarrow \mathsf{vamos\_read\_kernel\_info}\ s_V\ note$
    $|\ \textsc{undefined\_trap} \Rightarrow s_V$
       $(\!|\mathsf{procs} := s_V.\mathsf{procs}(p_{cp} \mapsto \delta_{\mathsf{proc}}\ \textsc{err\_unprivileged}\ \lceil s_V.\mathsf{procs}\ p_{cp}\rceil)|\!)$
    $|\ \textsc{runtime\_error} \Rightarrow \mathsf{vamos\_process\_kill}\ s_V\ \mathsf{HN\_SELF}$
    $|\ \varepsilon_\Omega \Rightarrow s_V(\!|\mathsf{procs} := s_V.\mathsf{procs}(p_{cp} \mapsto \delta_{\mathsf{proc}}\ \varepsilon_\Sigma\ \lceil s_V.\mathsf{procs}\ p_{cp}\rceil)|\!)$
```

Linchpin of the definition of vamosDispatcher is the case distinction over the output $\omega_{\mathsf{proc}}\ \lceil s_V.\mathsf{procs}\ p_{cp}\rceil$ of the current process. Outputs like PROCESS_CREATE or MEMORY_ADD imply the invocation of VAMOS calls. Accordingly, vamosDispatcher describes the effects by the respective specification functions. The output UNDEFINED_TRAP implies that the current instruction of p_{cp} denotes an undefined TRAP instruction. The acknowledgement of VAMOS comprises the error message ERR_UNPRIVILEGED which is delivered to p_{cp} by function δ_{proc}. Runtime errors are signaled by the output RUNTIME_ERROR and usually serve as indication of malicious programming. The VAMOS kernel handles this kind of error in a pretty rigorous way: It simply terminates the causing process. In order to describe the effects of this extraordinary process termination, VAMOS diverts function vamos_process_kill from its intended use, namely describing the effects of the VAMOS call PROCESS_KILL. The invocation with handle HN_SELF, however, intends the self-destruction of process p_{cp}. More details on that presents Section 4.4.3.

However, not all steps of process p_{cp} involve the kernel. Local transitions are implied by the output ε_Ω and performed by δ_{proc} with input ε_Σ.

This section continues with the specifications of the various VAMOS calls. As a matter of clarity, we categorize them into the fields: Access Control, Memory Management, Process Management, Scheduling Mechanism, Device Driver Support, and Inter-Process Communication.

4.4.1 Access Control

The minimal access control in VAMOS reserves most kernel calls for so-called privileged processes. Consequently, passing on privileges in itself is subject to privileged processes. Furthermore, the process p_{cp} can only contract privileges out to processes which are valid in its context.

Function vamos_result_set_privileges checks for possible violations of these restrictions by computing the response to the caller p_{cp}:

```
vamos_result_set_privileges $rightsdb$ $p_{cp}$ $hn_{obj}$ $\equiv$
  if $\neg$ privileged $rightsdb$ $p_{cp}$ then ERR_UNPRIVILEGED
  else if $\neg$ valid_handle $rightsdb$ $p_{cp}$ $hn_{obj}$ then ERR_INVALID_HANDLE
       else SUCCESS
```

Whenever the caller p_{cp} is not privileged or the handle hn_{obj} to the object is not valid, p_{cp} will receive an error message. Otherwise, a success message is returned. In the success case, the privileged status of the object is additionally set active. Formally, function vamos_set_privileges describes the according state updates:

```
vamos_set_privileges $s_V$ $hn_{obj}$ $\equiv$
  let $p_{cp}$ = $\lceil$v_cup $s_V$.schedds$\rceil$; $p_{obj}$ = $\lceil s_V$.rightsdb $p_{cp}\rceil$.hdb $hn_{obj}$;
      $res$ = vamos_result_set_privileges $s_V$.rightsdb $p_{cp}$ $hn_{obj}$
  in if is_error $res$ then $s_V$(|procs := $s_V$.procs($p_{cp} \mapsto \delta_{proc}$ $res$ $\lceil s_V$.procs $p_{cp}\rceil$)|)
     else $s_V$(|procs := $s_V$.procs($p_{cp} \mapsto \delta_{proc}$ $res$ $\lceil s_V$.procs $p_{cp}\rceil$),
                rightsdb := $s_V$.rightsdb($\lceil p_{obj}\rceil \mapsto \lceil s_V$.rightsdb $\lceil p_{obj}\rceil\rceil$
                  (|priv := True|))|)
```

The calling process p_{cp} is obtained by function v_cup and p_{obj} denotes the process number of the object process. The latter results from the access to p_{cp}'s handle database with hn_{obj}. As p_{obj} is only relevant in the success case, this access is valid because hn_{obj} is valid. In all cases, *res* denotes the result of the operation and δ_{proc} delivers it to p_{cp}. Only if *res* does not signal an error, the privilege status of p_{obj} is set to active.

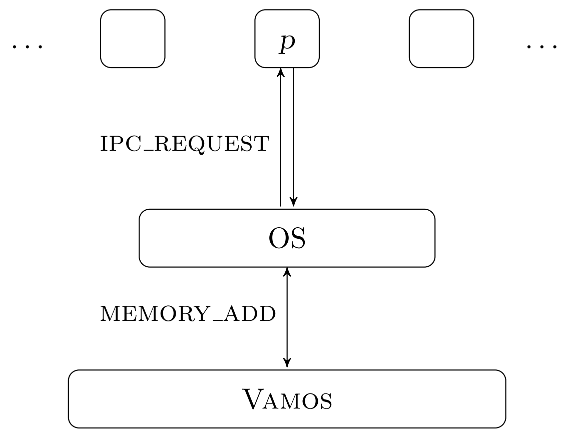

Figure 4.1: Memory Management in the Overall System

4.4.2 Memory Management

The memory management in VAMOS is restricted to the in- and decreasing of the memory amount of particular user processes. The former reflects the allocation, the latter the releasing of memory pages. For this purpose, VAMOS provides the calls MEMORY_ADD and MEMORY_FREE. By means of these calls, privileged processes are enabled to take care of their own but also to manipulate the memory amount of other processes.

In particular, for the latter case, it is essential that a process p affected by the memory manipulation is advised of this. An allocation of additional pages is of no avail unless p knows about it. Things are even worse, if the manipulation reduces the memory. Note that accesses to the released memory regions cause runtime errors which, in turn, lead to the termination of p.

In order to avoid such scenarios, VAMOS attaches certain conditions to the memory manipulations and thereby relies on the memory management in the overall system.

Memory Management in the Overall System. In the overall system as depicted in Figure 4.1, the privileged processes will constitute the OS itself or at least parts of the OS, like servers. Thus, the operating system (OS) runs as privileged process on top of the VAMOS kernel and interacts with the user processes. Requests to the OS or to the servers can be triggered by means of the VAMOS call IPC_REQUEST which is open to all user processes. Thus, if an unprivileged process p, for instance, wants to increase its memory amount, it utilizes IPC_REQUEST to place its demand and waits for a response of the OS. As privileged process, the OS forwards this request to the VAMOS kernel. If the request succeeds, VAMOS increases the memory amount of p and reports a SUCCESS message to the OS. Accordingly, a failure is acknowledged with an error message. In both cases, the OS transfers the kernel response to p which still waits in the receive phase of IPC_REQUEST.

Important in the context of the VAMOS kernel is the following observation: If the calling process p_{cp} wants to manipulate the memory amount of a different process p, this process p must reside in the wait state, i. e., in the wait queue.

Against that background, we proceed with the formal specification of the VAMOS calls MEMORY_ADD and MEMORY_FREE.

Increasing the Memory Amount

Increasing the memory amount of a process is rather straightforward and only affects the virtual machines. The calling process p_{cp} specifies a handle hn_{victim} to identify the process whose memory should be increased by pgs pages.

As usual, the invocation of a VAMOS call either succeeds or not. VAMOS uses function vamos_result_memory_add to compute the corresponding response to p_{cp}:

```
$\mathsf{vamos\_result\_memory\_add}\ uprocs\ rightsdb\ waiting_{\mathsf{victim}}\ p_{\mathsf{cp}}\ hn_{\mathsf{victim}}\ pgs \equiv$
  $\textbf{if}\ \neg\ \mathsf{privileged}\ rightsdb\ p_{\mathsf{cp}}\ \textbf{then}\ \text{ERR\_UNPRIVILEGED}$
  $\textbf{else if}\ \neg\ \mathsf{valid\_handle}\ rightsdb\ p_{\mathsf{cp}}\ hn_{\mathsf{victim}}\ \textbf{then}\ \text{ERR\_INVALID\_HANDLE}$
      $\textbf{else if}\ \neg\ \mathsf{mem\_available}\ uprocs\ pgs\ \textbf{then}\ \text{ERR\_OUT\_OF\_MEM}$
          $\textbf{else if}\ \neg\ waiting_{\mathsf{victim}} \wedge hn_{\mathsf{victim}} \neq \mathsf{HN\_SELF}$
              $\textbf{then}\ \text{ERR\_PROCESS\_NOT\_READY}\ \textbf{else}\ \text{SUCCESS}$
```

Checking the privileges of p_{cp}, the validity of hn_{victim} and the availability of further memory was introduced before. The last check, however, takes the aformentioned scenario into account. Everything is fine, if p_{cp} wants to increase its own memory amount, i. e., $hn_{\mathsf{victim}} = \mathsf{HN\_SELF}$. Otherwise, p_{cp} is only enabled to increase the memory amount of the victim, if this is waiting. The latter is denoted by the additional flag $waiting_{\mathsf{victim}}$.

Based on the result of vamos_result_memory_add, the overall specification function vamos_memory_add describes the effects of the memory increasing on a VAMOS state s_{V}:

```
$\mathsf{vamos\_memory\_add}\ s_{\mathsf{V}}\ hn_{\mathsf{victim}}\ pgs \equiv$
  $\textbf{let}\ p_{\mathsf{cp}} = \lceil \mathsf{v\_cup}\ s_{\mathsf{V}}.\mathsf{schedds}\rceil;\ p_{\mathsf{victim}} = \lceil\lceil s_{\mathsf{V}}.\mathsf{rightsdb}\ p_{\mathsf{cp}}\rceil.\mathsf{hdb}\ hn_{\mathsf{victim}}\rceil;$
     $waiting_{\mathsf{victim}} = p_{\mathsf{victim}} \in \mathsf{set}\ s_{\mathsf{V}}.\mathsf{schedds.wait};$
     $res = \mathsf{vamos\_result\_memory\_add}\ s_{\mathsf{V}}.\mathsf{procs}\ s_{\mathsf{V}}.\mathsf{rightsdb}\ waiting_{\mathsf{victim}}\ p_{\mathsf{cp}}$
            $hn_{\mathsf{victim}}\ pgs;$
     $procs_{\mathsf{add}} = s_{\mathsf{V}}.\mathsf{procs}(p_{\mathsf{victim}} \mapsto \delta_{\mathsf{proc}}\ (\text{ADD\_MEMORY}\ pgs)\ \lceil s_{\mathsf{V}}.\mathsf{procs}\ p_{\mathsf{victim}}\rceil)$
  $\textbf{in if}\ \mathsf{is\_error}\ res\ \textbf{then}\ s_{\mathsf{V}}(\!|\mathsf{procs} := s_{\mathsf{V}}.\mathsf{procs}(p_{\mathsf{cp}} \mapsto \delta_{\mathsf{proc}}\ res\ \lceil s_{\mathsf{V}}.\mathsf{procs}\ p_{\mathsf{cp}}\rceil)|\!)$
     $\textbf{else}\ s_{\mathsf{V}}(\!|\mathsf{procs} := procs_{\mathsf{add}}(p_{\mathsf{cp}} \mapsto \delta_{\mathsf{proc}}\ \text{SUCCESS}\ \lceil procs_{\mathsf{add}}\ p_{\mathsf{cp}}\rceil)|\!)$
```

In case of failure, only the error message is delivered to the calling process p_{cp}. For the success case, the definition obtains the process number

p_{victim} of the victim by accessing p_{cp}'s handle database with hn_{victim} and flag $waiting_{\mathsf{victim}}$ abbreviates the fact that p_{victim} is part of the wait queue. The intermediate update $procs_{\mathsf{add}}$ describes the allocation of pgs additional memory pages and is obtained by the application of function δ_{proc} with message ADD_MEMORY to the virtual machine of p_{victim}. Based on $procs_{\mathsf{add}}$, VAMOS finally delivers the SUCCESS message to process p_{cp}.

Note that the application of δ_{proc} with message ADD_MEMORY only increases the page table length but not the program counters. This is due to the fact, that p_{victim} is expected to be still in the receive phase of an IPC_REQUEST. The program counters are increased not until this call is completed, i. e., when the OS responds to the request. With the OS response, the VAMOS kernel sends a SUCC_RECEIVE message to p_{victim} which involves the increasing of the program counters.

Decreasing the Memory Amount

Similar to the increasing, the calling process p_{cp} specifies a handle hn_{victim} in order to identify the victim whose memory amount should be decreased by pgs pages. The result function vamos_result_memory_free adopts most error checks from above, as its definition illustrates:

```
vamos_result_memory_free $uprocs\ rightsdb\ waiting_{\mathsf{victim}}\ p_{\mathsf{cp}}\ hn_{\mathsf{victim}}\ pgs \equiv$
  let $p_{\mathsf{victim}} = \lceil rightsdb\ p_{\mathsf{cp}} \rceil.\mathsf{hdb}\ hn_{\mathsf{victim}}$
  in if $\neg$ privileged $rightsdb\ p_{\mathsf{cp}}$ then ERR_UNPRIVILEGED
     else if $\neg$ valid_handle $rightsdb\ p_{\mathsf{cp}}\ hn_{\mathsf{victim}}$ then ERR_INVALID_HANDLE
          else if $\mathsf{size}_{\mathsf{proc}}\ \lceil uprocs\ \lceil p_{\mathsf{victim}} \rceil \rceil \leq pgs$ then ERR_INVALID_ARGS
               else if $\neg\ waiting_{\mathsf{victim}} \wedge hn_{\mathsf{victim}} \neq \mathsf{HN\_SELF}$
                    then ERR_PROCESS_NOT_READY else SUCCESS
```

Being in the process of freeing memory makes the check for memory resources superfluous. Instead, vamos_result_memory_free introduces an opposite check. Releasing the whole memory of a process would result in its termination. As this is not the intended meaning of the FREE_MEMORY call, VAMOS restricts the number of released pages. Thus, pgs must be smaller than the number of pages the victim occupies. The latter is obtained by applying function $\mathsf{size}_{\mathsf{proc}}$ to the virtual machine of p_{victim}. The process number p_{victim} is taken from p_{cp}'s handle database.

In many cases, it is more effort to remove than to add something. With the decreasing of the memory amount it is exactly the same. While adding memory pages only affects the virtual machines, releasing memory pages might also affect pending IPC operations of the victim. The latter happens, if involved memory regions become unavailable. VAMOS determines this situation by predicate is_ipc_memory_err:

```
is_ipc_memory_err $freed_{\mathsf{victim}}\ sndstat_{\mathsf{victim}} \equiv$
  case $\omega_{\mathsf{proc}}\ freed_{\mathsf{victim}}$ of
  IPC_SEND $hn_{\mathsf{recv}}\ rights_{\mathsf{send}}\ msg\ hn_{\mathsf{add}}\ rights_{\mathsf{add}}\ timeout_{\mathsf{send}} \Rightarrow$
```

```
    $msg$ = MObjUnavailable
  | IPC_RECEIVE $hn_{\mathsf{send}}$ $buffer$ $timeout_{\mathsf{recv}}$ ⇒ $buffer$ = BufUnavailable
  | IPC_REQUEST $hn_{\mathsf{recv}}$ $rights_{\mathsf{send}}$ $msg$ $hn_{\mathsf{add}}$ $rights_{\mathsf{add}}$ $timeout_{\mathsf{send}}$ $buffer$
     $timeout_{\mathsf{recv}}$ ⇒
      $buffer$ = BufUnavailable ∨ ¬ $sndstat_{\mathsf{victim}}$ ∧ $msg$ = MObjUnavailable
  | _ ⇒ True
```

Let $freed_{\mathsf{victim}}$ determine the virtual machine of the victim after the memory release and $sndstat_{\mathsf{victim}}$ the victim's send status. In order to figure out any errors regarding the memory used in a (possible) IPC operation, we apply function ω_{proc} to $freed_{\mathsf{victim}}$. The predicate applies, if any specified message or buffer becomes unavailable by the memory release. In this case, VAMOS decreases the memory amount of the victim and additionally aborts the operation with an according error message.

The overall specification function vamos_memory_free encapsulates the relevant updates:

```
vamos_memory_free $s_{\mathsf{V}}$ $hn_{\mathsf{victim}}$ $pgs$ ≡
  let $p_{\mathsf{cp}}$ = ⌈v_cup $s_{\mathsf{V}}$.schedds⌉;
      $waiting_{\mathsf{victim}}$ =
        ⌈⌈$s_{\mathsf{V}}$.rightsdb $p_{\mathsf{cp}}$⌉.hdb $hn_{\mathsf{victim}}$⌉ ∈ set $s_{\mathsf{V}}$.schedds.wait;
      $res$ =
        vamos_result_memory_free $s_{\mathsf{V}}$.procs $s_{\mathsf{V}}$.rightsdb $waiting_{\mathsf{victim}}$
         $p_{\mathsf{cp}}$ $hn_{\mathsf{victim}}$ $pgs$
  in if is_error $res$ then $s_{\mathsf{V}}$(|procs := $s_{\mathsf{V}}$.procs($p_{\mathsf{cp}}$ ↦ $\delta_{\mathsf{proc}}$ $res$ ⌈$s_{\mathsf{V}}$.procs $p_{\mathsf{cp}}$⌉)|)
     else $s_{\mathsf{V}}$(|procs := v_memory_free_updt_procs $s_{\mathsf{V}}$.procs $s_{\mathsf{V}}$.priodb
                          $s_{\mathsf{V}}$.rightsdb $s_{\mathsf{V}}$.sndstatdb $waiting_{\mathsf{victim}}$ $p_{\mathsf{cp}}$ $hn_{\mathsf{victim}}$
                          $pgs$,
              schedds := v_memory_free_updt_schedds $s_{\mathsf{V}}$.schedds $s_{\mathsf{V}}$.procs
                            $s_{\mathsf{V}}$.priodb $s_{\mathsf{V}}$.rightsdb $s_{\mathsf{V}}$.sndstatdb $p_{\mathsf{cp}}$ $hn_{\mathsf{victim}}$
                            $pgs$,
              sndstatdb := v_memory_free_updt_sndstatdb $s_{\mathsf{V}}$.sndstatdb $s_{\mathsf{V}}$.procs
                              $s_{\mathsf{V}}$.rightsdb $waiting_{\mathsf{victim}}$ $p_{\mathsf{cp}}$ $hn_{\mathsf{victim}}$ $pgs$|)
```

In case of success, dedicated functions encapsulate the particular state updates. Otherwise, only the error message is delivered to the calling process p_{cp}.

Updating the Virtual Machines. The update of the virtual machines comprises the delivery of the success message to the calling process p_{cp}, on the one hand, and the decreasing of p_{victim}'s memory amount together with a (possible) error message, on the other one.

Formally, function v_memory_free_updt_procs describes the update:

```
v_memory_free_updt_procs $uprocs$ $priodb$ $rightsdb$ $sndstatdb$ $waiting_{\mathsf{victim}}$ $p_{\mathsf{cp}}$
 $hn_{\mathsf{victim}}$ $pgs$ ≡
  let $p_{\mathsf{victim}}$ = ⌈⌈$rightsdb$ $p_{\mathsf{cp}}$⌉.hdb $hn_{\mathsf{victim}}$⌉;
      $freed_{\mathsf{victim}}$ = $\delta_{\mathsf{proc}}$ (FREE_MEMORY $pgs$) ⌈$uprocs$ $p_{\mathsf{victim}}$⌉;
```

```
    $\mathit{error}_{\mathsf{victim}} =$
      $\mathit{waiting}_{\mathsf{victim}} \wedge \mathsf{is\_ipc\_memory\_err}\ \mathit{freed}_{\mathsf{victim}}\ \lceil \mathit{sndstatdb}\ p_{\mathsf{victim}} \rceil;$
    $\mathit{res}_{\mathsf{victim}} =$
      $\mathbf{if}\ \mathsf{vamosIsSending}\ \mathit{uprocs}\ \mathit{sndstatdb}\ p_{\mathsf{victim}}\ \mathbf{then}\ \text{ERR\_SND\_SEGV}$
      $\mathbf{else}\ \text{ERR\_RCV\_SEGV};$
    $\mathit{updt}_{\mathsf{victim}} = \mathbf{if}\ \mathit{error}_{\mathsf{victim}}\ \mathbf{then}\ \delta_{\mathsf{proc}}\ \mathit{res}_{\mathsf{victim}}\ \mathit{freed}_{\mathsf{victim}}\ \mathbf{else}\ \mathit{freed}_{\mathsf{victim}}$
 $\mathbf{in}\ \mathbf{if}\ p_{\mathsf{victim}} = p_{\mathsf{cp}}\ \mathbf{then}\ \mathit{uprocs}(p_{\mathsf{victim}} \mapsto \delta_{\mathsf{proc}}\ \text{SUCCESS}\ \mathit{updt}_{\mathsf{victim}})$
    $\mathbf{else}\ \mathit{uprocs}(p_{\mathsf{victim}} \mapsto \mathit{updt}_{\mathsf{victim}},\ p_{\mathsf{cp}} \mapsto \delta_{\mathsf{proc}}\ \text{SUCCESS}\ \lceil \mathit{uprocs}\ p_{\mathsf{cp}} \rceil)$
```

The definition introduces a couple of abbreviations. Process number p_{victim} of the victim is obtained by accessing p_{cp}'s handle database with hn_{victim}. The update of p_{victim}'s virtual machine comes in two steps: The first one $\mathit{freed}_{\mathsf{victim}}$ results from the application of δ_{proc} with input FREE_MEMORY and releases *pgs* memory pages. The second step leads to the final update $\mathit{updt}_{\mathsf{victim}}$ of p_{victim}'s virtual machine. If the releasing entails an IPC error, $\mathit{updt}_{\mathsf{victim}}$ additionally comprises the delivery of the according error message $\mathit{res}_{\mathsf{victim}}$. Otherwise, it is equal to $\mathit{freed}_{\mathsf{victim}}$. The flag $\mathit{error}_{\mathsf{victim}}$ signals the arising of an IPC error. It is active, if p_{victim} is waiting and predicate is_ipc_memory_err applies.

Based on these abbreviations, v_memory_free_updt_procs performs the actual update of the virtual machines *uprocs*. If $p_{\mathsf{victim}} = p_{\mathsf{cp}}$, only the success message is delivered. Note that in this case $\mathit{updt}_{\mathsf{victim}} = \mathit{freed}_{\mathsf{victim}}$, because p_{cp} is not waiting, i. e., $\neg\ \mathit{error}_{\mathsf{victim}}$. Otherwise, the entries for p_{victim} and p_{cp} are set seperately.

As with the memory increasing, the application of function δ_{proc} with the input FREE_MEMORY does not increase the program counters.

Updating the Scheduling Datastructures. The scheduling datastructures only need an update, if an IPC error occurs. In this case, the victim is put back into the ready state.

Function v_memory_free_updt_schedds gives the formal specification:

```
$\mathsf{v\_memory\_free\_updt\_schedds}\ \mathit{schedds}\ \mathit{uprocs}\ \mathit{priodb}\ \mathit{rightsdb}\ \mathit{sndstatdb}\ p_{\mathsf{cp}}\ hn_{\mathsf{victim}}$
 $\mathit{pgs} \equiv$
 $\mathbf{let}\ p_{\mathsf{victim}} = \lceil \lceil \mathit{rightsdb}\ p_{\mathsf{cp}} \rceil.\mathsf{hdb}\ hn_{\mathsf{victim}} \rceil;$
    $\mathit{freed}_{\mathsf{victim}} = \delta_{\mathsf{proc}}\ (\text{FREE\_MEMORY}\ \mathit{pgs})\ \lceil \mathit{uprocs}\ p_{\mathsf{victim}} \rceil;$
    $\mathit{waiting}_{\mathsf{victim}} = p_{\mathsf{victim}} \in \mathsf{set}\ \mathit{schedds}.\mathsf{wait};$
    $\mathit{error}_{\mathsf{victim}} =$
      $\mathit{waiting}_{\mathsf{victim}} \wedge \mathsf{is\_ipc\_memory\_err}\ \mathit{freed}_{\mathsf{victim}}\ \lceil \mathit{sndstatdb}\ p_{\mathsf{victim}} \rceil$
 $\mathbf{in}\ \mathbf{if}\ \mathit{error}_{\mathsf{victim}}\ \mathbf{then}\ \mathsf{v\_wkp\_updt\_schedds}\ \mathit{schedds}\ [p_{\mathsf{victim}}]\ \mathit{priodb}$
    $\mathbf{else}\ \mathit{schedds}$
```

As above, an active flag $\mathit{error}_{\mathsf{victim}}$ signals an IPC error. The awakening of p_{victim} describes function v_wkp_updt_schedds.

Updating the Send Status Database. As with the scheduling datastructures, the send status database is only updated, if an IPC error occurs.

If so, the entry of p_{victim} is reset to $\bot$.

Accordingly, the formal definition of v_memory_free_updt_sndstatdb is straightforward:

```
v_memory_free_updt_sndstatdb $sndstatdb\ uprocs\ rightsdb\ waiting_{\mathsf{victim}}\ p_{\mathsf{cp}}$
 $hn_{\mathsf{victim}}\ pgs \equiv$
  let $p_{\mathsf{victim}} = \lceil\lceil rightsdb\ p_{\mathsf{cp}}\rceil.\mathsf{hdb}\ hn_{\mathsf{victim}}\rceil$;
      $freed_{\mathsf{victim}} = \delta_{\mathsf{proc}}$ (FREE_MEMORY $pgs$) $\lceil uprocs\ p_{\mathsf{victim}}\rceil$;
      $error_{\mathsf{victim}} =$
        $waiting_{\mathsf{victim}} \wedge$ is_ipc_memory_err $freed_{\mathsf{victim}}\ \lceil sndstatdb\ p_{\mathsf{victim}}\rceil$
  in if $error_{\mathsf{victim}}$ then $sndstatdb(p_{\mathsf{victim}} \mapsto \mathsf{False})$ else $sndstatdb$
```

4.4.3 Process Management

The process management in VAMOS provides the possibility to dynamically create, clone and terminate processes. In the kernel's ABI, these operations are represented by the calls PROCESS_CREATE, PROCESS_CLONE and PROCESS_KILL. The two former calls are reserved for privileged processes only, whereas self-termination is allowed to all processes. Terminating other processes, however, requires again the privileged status.

Creating a New Process

If the calling process p_{cp} wants to create a new process, it has to specify several arguments. Regarding the scheduling, p_{cp} has to determine the priority *pri* and the timeslice *tsl* for the new process. In order to configure the initial memory, p_{cp} specifies the number *pgs* of memory pages that should be allocated for the new process and provides the initial memory image *img*.

The process creation only succeeds, if the parameters fulfill certain requirements. In addition, system-related sources of failure must be excluded. For this reason, we define function vamos_result_create determining the kernel's response to the calling process:

```
vamos_result_create $uprocs\ rightsdb\ p_{\mathsf{cp}}\ pri\ img\ pgs \equiv$
  if $\neg$ privileged $rightsdb\ p_{\mathsf{cp}}$ then ERR_UNPRIVILEGED
  else if $pri = \bot \vee$
          $img \in \{\mathsf{MObjUndefined}, \mathsf{MObjUnavailable}\} \vee$
          $(\exists wl.\ img = \mathsf{MObjSeq}\ wl \wedge pgs * \mathsf{PAGE\_SIZE} < \mathsf{length}\ wl) \vee$
          $pgs = 0 \vee \mathsf{TVM\_MAXPAGES} \leq pgs$
       then ERR_INVALID_ARGS
       else if $\neg$ procs_available $uprocs$ then ERR_OUT_OF_PIDS
          else if $\neg$ mem_available $uprocs$
                  (mem_img_length $img$ div PAGE_SIZE)
             then ERR_OUT_OF_MEM else SUCCESS
```

As long as the calling process p_{cp} is not privileged, the call is aborted with the error message ERR_UNPRIVILEGED. If p_{cp} is privileged, function vamos_result_create inspects the validity of the arguments. Process p_{cp} has invoked PROCESS_CREATE with invalid arguments, if either: (a) the priority is invalid, i. e., $pri = \bot$, (b) the memory image *img* is undefined or unavailable, (c) the memory image *img* indeed denotes a sequence *ws* of words but its length would exceed the assigned memory, or (d) the new process should be occupied with 0 or more than TVM_MAXPAGES memory pages. In all cases, VAMOS cancels the call and delivers the error message ERR_INVALID_ARGS. Finally, vamos_result_create checks the system resources. A process creation is not possible, if all processes are in use. In this case, VAMOS returns ERR_OUT_OF_PIDS to p_{cp}. The same applies for lacking memory resources which are acknowledged with error ERR_OUT_OF_MEM.

In the absence of any errors, function vamos_result_create returns SUCCESS and initiates thereby the updates of the kernel datastructures in the overall specification function vamos_process_create:

```
vamos_process_create $s_V$ $tsl$ $pri$ $img$ $pgs$ $\equiv$
  let $p_{cp} = \lceil$v_cup $s_V$.schedds$\rceil$; $p_{new}$ = hd $s_V$.schedds.inactive;
      $res$ = vamos_result_create $s_V$.procs $s_V$.rightsdb $p_{cp}$ $pri$ $img$ $pgs$
  in if is_error $res$ then $s_V$(|procs := $s_V$.procs($p_{cp} \mapsto \delta_{proc}$ $res$ $\lceil s_V$.procs $p_{cp}\rceil$)|)
     else $s_V$(|procs := v_create_updt_procs $s_V$.procs $s_V$.rightsdb $p_{cp}$ $p_{new}$
                          (init$_{proc}$ (mObjSeq $img$) $pgs$),
                schedds := v_create_updt_schedds $s_V$.schedds $p_{cp}$ $p_{new}$ $tsl$
                              $\lceil pri\rceil$,
                priodb := $s_V$.priodb($p_{new}$ := $pri$),
                sndstatdb := $s_V$.sndstatdb($p_{new} \mapsto$ False),
                rightsdb := v_create_updt_rightsdb $s_V$.rightsdb $p_{cp}$ $p_{new}$|)
```

Function v_cup delivers the calling process p_{cp}, whereas the head of the inactive list determines the process number p_{new} of the new process. Setting the priority and the send status is easy enough to be done directly. The entry p_{new} in the priority database is set to *pri* and the send status is set to False. Dedicated functions complete the updates of the remaining state components.

Updating the User Machines. As shown in the definition above, the update function v_create_updt_procs takes the initial machine $proc_{\mathsf{new}}$ for the new process p_{new} as input. It is based on the memory image *img* and computed by the initialization function $\mathsf{init}_{\mathsf{proc}}$ for user processes.

Apart from setting the entry p_{new} in the user machine mapping *uprocs* to $proc_{\mathsf{new}}$, the update also comprises the delivery of the success message to the caller p_{cp}:

```
v_create_updt_procs $uprocs$ $rightsdb$ $p_{cp}$ $p_{new}$ $proc_{new}$ $\equiv$
  let $hn_{new}$ = to_hn $rightsdb$ $p_{cp}$ $p_{new}$
  in $uprocs$($p_{cp} \mapsto \delta_{proc}$ (SUCC_NEW_PROCESS $hn_{new}$) $\lceil uprocs$ $p_{cp}\rceil$, $p_{new} \mapsto$
     $proc_{new}$)
```

The success message SUCC_NEW_PROCESS also contains the handle hn_{new} to the newly created process. Based on the process number p_{new}, it is computed by the function $\mathsf{to\_hn}$.

Updating the Scheduling Datastructures. The new process p_{new} needs to be integrated into the scheduling mechanism. Consequently, it is appended to the ready queue of priority *pri* and removed from the inactive queue. Regarding the process-specific scheduling data of p_{new}, VAMOS inherits the specified value *tsl* as new timeslice. By default, no time is yet consumed and the timeout is infinite. Function $\mathsf{v\_create\_updt\_schedds}$ formalizes this:

$$
\begin{array}{l}
\mathsf{v\_create\_updt\_schedds}\ \mathit{schedds}\ p_{\mathsf{cp}}\ p_{\mathsf{new}}\ \mathit{tsl}\ \mathit{pri} \equiv \mathit{schedds} \\
\quad (\!|\mathsf{ready} := \mathit{schedds}.\mathsf{ready}(\mathit{pri} := \mathit{schedds}.\mathsf{ready}\ \mathit{pri}\ @\ [p_{\mathsf{new}}]), \\
\qquad \mathsf{inactive} := [x{\in}\mathit{schedds}.\mathsf{inactive}\ .\ x \neq p_{\mathsf{new}}], \\
\qquad \mathsf{procdb} := \mathit{schedds}.\mathsf{procdb}(p_{\mathsf{new}} \mapsto (\!|\mathsf{tsl} = \mathit{tsl},\ \mathsf{ctsl} = 0,\ \mathsf{timeout} = \infty|\!))|\!)
\end{array}
$$

Updating the Rights Datastructures. The update of the rights datastructures splits up into two parts: (a) updating the rights datastructures of the calling process p_{cp}, and (b) initializing the ones of the new process p_{new}.

Formally, these parts are described by function $\mathsf{v\_create\_updt\_rightsdb}$:

$$
\begin{array}{l}
\mathsf{v\_create\_updt\_rightsdb}\ \mathit{rightsdb}\ p_{\mathsf{cp}}\ p_{\mathsf{new}} \equiv \\
\quad \mathbf{let}\ \mathit{hn}_{\mathsf{new}} = \mathsf{to\_hn}\ \mathit{rightsdb}\ p_{\mathsf{cp}}\ p_{\mathsf{new}}; \\
\qquad \mathit{rightsdb}_{\mathsf{cp}} = \lceil \mathit{rightsdb}\ p_{\mathsf{cp}} \rceil \\
\qquad\quad (\!|\mathsf{hdb} := \lceil \mathit{rightsdb}\ p_{\mathsf{cp}} \rceil.\mathsf{hdb}(\mathit{hn}_{\mathsf{new}} \mapsto p_{\mathsf{new}}), \\
\qquad\qquad \mathsf{rdb} := \lceil \mathit{rightsdb}\ p_{\mathsf{cp}} \rceil.\mathsf{rdb}(\lfloor p_{\mathsf{new}} \rfloor \mapsto \\
\qquad\qquad\quad \{\mathsf{v\_sendR},\ \mathsf{v\_requestR},\ \mathsf{v\_multipleR},\ \mathsf{v\_finiteR}\})|\!); \\
\qquad \mathit{rightsdb}_{\mathsf{new}} = \\
\qquad\quad (\!|\mathsf{priv} = \mathsf{False},\ \mathsf{hdb} = [\mathsf{HN\_PARENT} \mapsto p_{\mathsf{cp}},\ \mathsf{HN\_SELF} \mapsto p_{\mathsf{new}}], \\
\qquad\qquad \mathsf{rdb} = [\lfloor p_{\mathsf{cp}} \rfloor \mapsto \{\mathsf{v\_requestR}\}],\ \mathsf{stolen} = \{\},\ \mathsf{parent} = \lfloor p_{\mathsf{cp}} \rfloor|\!) \\
\quad \mathbf{in}\ \mathit{rightsdb}(p_{\mathsf{cp}} \mapsto \mathit{rightsdb}_{\mathsf{cp}},\ p_{\mathsf{new}} \mapsto \mathit{rightsdb}_{\mathsf{new}})
\end{array}
$$

The definition abbreviates the update of the caller's rights datastructures by $\mathit{rightsdb}_{\mathsf{cp}}$. Within $\mathit{rightsdb}_{\mathsf{cp}}$, handle $\mathit{hn}_{\mathsf{new}}$ refers to the new process p_{new}. It corresponds to the one in the success message and enables p_{cp} to identify the new process p_{new}. In addition, p_{cp} is fit out with all rights to p_{new}.

The initial rights datastructures $\mathit{rightsdb}_{\mathsf{new}}$ of p_{new} comprise no privileges, an empty set of stolen handles and p_{cp} as parent. The latter is reflected in the handle database by $\mathsf{HN\_PARENT}$ pointing to p_{cp}. Furthermore, $\mathsf{HN\_SELF}$ is assigned to p_{new}. The only right owned by p_{new} is the $\mathsf{v\_requestR}$ right to p_{cp}.

Cloning a Process

Cloning creates a new process that acts as duplicate of the original process. For this to work, the calling process p_{cp} has to specify a handle hn_{cl} to the process that should be cloned. Again, a function vamos_result_clone checks, whether the cloning will be successful and returns the response to p_{cp} if not:

```
$\mathsf{vamos\_result\_clone}\ uprocs\ rightsdb\ waiting_{\mathsf{cl}}\ p_{\mathsf{cp}}\ hn_{\mathsf{cl}} \equiv$
  let $p_{\mathsf{cl}} = \lceil\lceil rightsdb\ p_{\mathsf{cp}}\rceil.\mathsf{hdb}\ hn_{\mathsf{cl}}\rceil$
  in if $\neg\ \mathsf{privileged}\ rightsdb\ p_{\mathsf{cp}}$ then ERR_UNPRIVILEGED
     else if $\neg\ \mathsf{valid\_handle}\ rightsdb\ p_{\mathsf{cp}}\ hn_{\mathsf{cl}}$ then ERR_INVALID_HANDLE
          else if $\neg\ \mathsf{procs\_available}\ uprocs$ then ERR_OUT_OF_PIDS
               else if $\neg\ \mathsf{mem\_available}\ uprocs\ (\mathsf{size}_{\mathsf{proc}}\ \lceil uprocs\ p_{\mathsf{cl}}\rceil)$
                    then ERR_OUT_OF_MEM
                    else if $\neg\ waiting_{\mathsf{cl}} \wedge hn_{\mathsf{cl}} \neq \mathsf{HN\_SELF}$
                         then ERR_PROCESS_NOT_READY else SUCCESS
```

The response comprises a failure notice in case of (a) an unprivileged calling process p_{cp}, (b) an invalid handle hn_{cl}, (c) insufficient system-resources regarding available processes and memory, or (d) the process to be cloned is not p_{cp} and not waiting. As with the calls MEMORY_FREE and MEMORY_ADD, if the calling process p_{cp} wants to clone a process p_{cl} which is different from itself, i. e., $hn_{\mathsf{cl}} \neq \mathsf{HN\_SELF}$, this process p_{cl} must reside in the wait state, i. e., in the wait queue. The latter is denoted by the additional flag $waiting_{\mathsf{cl}}$.

As long as vamos_result_clone detects any errors, only the error message is passed on to the caller by means of function δ_{proc}. The remaining internal kernel datastructures remain untouched. Only the response SUCCESS enables the process cloning and the involved updates.

The overall formalization is given by function vamos_process_clone:

```
$\mathsf{vamos\_process\_clone}\ s_{\mathsf{V}}\ hn_{\mathsf{cl}} \equiv$
  let $p_{\mathsf{cp}} = \lceil \mathsf{v\_cup}\ s_{\mathsf{V}}.\mathsf{schedds}\rceil$; $p_{\mathsf{cl}} = \lceil\lceil s_{\mathsf{V}}.\mathsf{rightsdb}\ p_{\mathsf{cp}}\rceil.\mathsf{hdb}\ hn_{\mathsf{cl}}\rceil$;
      $p_{\mathsf{new}} = \mathsf{hd}\ s_{\mathsf{V}}.\mathsf{schedds.inactive}$;
      $res = \mathsf{vamos\_result\_clone}\ s_{\mathsf{V}}.\mathsf{procs}\ s_{\mathsf{V}}.\mathsf{rightsdb}\ (p_{\mathsf{cl}}\ \mathsf{mem}\ s_{\mathsf{V}}.\mathsf{schedds.wait})$
            $p_{\mathsf{cp}}\ hn_{\mathsf{cl}}$
  in if $\mathsf{is\_error}\ res$ then $s_{\mathsf{V}}(\!|\mathsf{procs} := s_{\mathsf{V}}.\mathsf{procs}(p_{\mathsf{cp}} \mapsto \delta_{\mathsf{proc}}\ res\ \lceil s_{\mathsf{V}}.\mathsf{procs}\ p_{\mathsf{cp}}\rceil)|\!)$
     else $s_{\mathsf{V}}(\!|\mathsf{procs} := \mathsf{v\_clone\_updt\_procs}\ s_{\mathsf{V}}.\mathsf{procs}\ s_{\mathsf{V}}.\mathsf{rightsdb}\ p_{\mathsf{cp}}\ p_{\mathsf{cl}}\ p_{\mathsf{new}}$,
                $\mathsf{schedds} := \mathsf{v\_clone\_updt\_schedds}\ s_{\mathsf{V}}.\mathsf{schedds}\ s_{\mathsf{V}}.\mathsf{procs}\ s_{\mathsf{V}}.\mathsf{sndstatdb}$
                          $s_{\mathsf{V}}.\mathsf{rightsdb}\ p_{\mathsf{cl}}\ p_{\mathsf{new}}\ \lceil s_{\mathsf{V}}.\mathsf{priodb}\ p_{\mathsf{cl}}\rceil$,
                $\mathsf{priodb} := s_{\mathsf{V}}.\mathsf{priodb}(p_{\mathsf{new}} := s_{\mathsf{V}}.\mathsf{priodb}\ p_{\mathsf{cl}})$,
                $\mathsf{sndstatdb} := s_{\mathsf{V}}.\mathsf{sndstatdb}(p_{\mathsf{new}} := s_{\mathsf{V}}.\mathsf{sndstatdb}\ p_{\mathsf{cl}})$,
                $\mathsf{rightsdb} := \mathsf{v\_clone\_updt\_rightsdb}\ s_{\mathsf{V}}.\mathsf{rightsdb}\ p_{\mathsf{cp}}\ p_{\mathsf{cl}}\ p_{\mathsf{new}}|\!)$
```

In case of success, the process number p_{cl} of the cloned process is obtained by accessing p_{cp}'s handle database with handle hn_{cl}. Again, the head of the inactive queue determines the process number p_{new} of the new process.

Similar as above, the updates of the priority and the send status databases are straightforward: VAMOS simply adopts the values of p_{cl} and assigns them to p_{new}. Dedicated functions describe the remaining ones.

Updating the User Machines. Creating and cloning of processes has a similar impact on the user machines. The calling process p_{cp} is informed about the successful operation and the new process p_{new} is assigned to a virtual assembly machine. In the context of cloning, however, Vamos does not associate p_{new} with an initial machine but with the one of the cloned process p_{cl}. The definition of v_clone_updt_procs reflects this:

```
v_clone_updt_procs $uprocs\ rightsdb\ p_{\mathsf{cp}}\ p_{\mathsf{cl}}\ p_{\mathsf{new}} \equiv$
  let $hn_{\mathsf{new}} = \mathsf{to\_hn}\ rightsdb\ p_{\mathsf{cp}}\ p_{\mathsf{new}}$
  in $\lambda x.$ if $x = p_{\mathsf{cp}}$ then $\lfloor \delta_{\mathsf{proc}}\ (\text{SUCC\_NEW\_PROCESS}\ hn_{\mathsf{new}})\ \lceil uprocs\ p_{\mathsf{cp}} \rceil \rfloor$
          else if $x = p_{\mathsf{new}} \wedge p_{\mathsf{cl}} = p_{\mathsf{cp}}$
               then $\lfloor \delta_{\mathsf{proc}}\ (\text{SUCC\_NEW\_PROCESS}\ \mathsf{HN\_NONE})\ \lceil uprocs\ p_{\mathsf{cp}} \rceil \rfloor$
               else if $x = p_{\mathsf{new}}$ then $uprocs\ p_{\mathsf{cl}}$ else $uprocs\ x$
```

Updating the virtual machine of p_{cp} is clear enough such that we directly move on to the one of process p_{new}. While assigning the virtual machine to p_{new}, function v_clone_updt_procs has to take into account that the process p_{cp} might have cloned itself, i. e., $p_{\mathsf{cl}} = p_{\mathsf{cp}}$. If so, p_{new} – as duplicate of p_{cp} – expects a kernel response and Vamos delivers it. In this case, however, the response does not contain a process handle but HN_NONE. In case that $p_{\mathsf{cl}} \neq p_{\mathsf{cp}}$, p_{new} is assigned to the virtual machine of p_{cl}. All other virtual machines remain untouched.

Updating the Scheduling Datastructures. The update of the scheduling datastructures entails the integration of p_{new} into the Vamos process scheduling. It is removed from the inactive queue and its consumed time is set to 0. Depending on the state of the cloned process p_{cl}, it is either appended to a ready or the wait queue.

The formal specification is given by function v_clone_updt_schedds:

```
v_clone_updt_schedds $schedds\ uprocs\ sndstatdb\ rightsdb\ p_{\mathsf{cl}}\ p_{\mathsf{new}}\ pri_{\mathsf{cl}} \equiv$
  let $ready_{\mathsf{cl}} = p_{\mathsf{cl}} \in \mathsf{set}\ (schedds.\mathsf{ready}\ pri_{\mathsf{cl}})$
  in $schedds$
     $(\!|\mathsf{ready} := schedds.\mathsf{ready}$
        $(pri_{\mathsf{cl}} :=$
          if $ready_{\mathsf{cl}}$ then $schedds.\mathsf{ready}\ pri_{\mathsf{cl}}\ @\ [p_{\mathsf{new}}]$
          else $schedds.\mathsf{ready}\ pri_{\mathsf{cl}}),$
       $\mathsf{inactive} := [x{\in}schedds.\mathsf{inactive}\ .\ x \neq p_{\mathsf{new}}],$
       $\mathsf{wait} :=$ if $ready_{\mathsf{cl}}$ then $schedds.\mathsf{wait}$ else $schedds.\mathsf{wait}\ @\ [p_{\mathsf{new}}],$
       $\mathsf{procdb} := schedds.\mathsf{procdb}(p_{\mathsf{new}} \mapsto \lceil schedds.\mathsf{procdb}\ p_{\mathsf{cl}} \rceil (\!|\mathsf{ctsl} := 0|\!))|\!)$
```

In the definition, flag $ready_{\mathsf{cl}}$ is active, if p_{cl} is member of the ready queue $schedds.\mathsf{ready}\ pri_{\mathsf{cl}}$, where pri_{cl} denotes its priority. As p_{new} will have the same state as p_{cl}, Vamos appends p_{new} to $schedds.\mathsf{ready}\ pri_{\mathsf{cl}}$, if $ready_{\mathsf{cl}}$ is active and to $schedds.\mathsf{wait}$, otherwise.

Updating the Rights Datastructures The update of the caller's rights datastructures $rightsdb_{\mathsf{cp}}$ happens in perfect analogy to the one within the process creation. Regarding the rights datastructures $rightsdb_{\mathsf{new}}$ of the new process, VAMOS mainly copies the ones of p_{cl}. Only the entry $\mathsf{HN\_SELF}$ in the handle database is reset to p_{new}:

```
$\mathsf{v\_clone\_updt\_rightsdb}\ \mathit{rightsdb}\ p_{\mathsf{cp}}\ p_{\mathsf{cl}}\ p_{\mathsf{new}} \equiv$
  $\mathbf{let}\ \mathit{hn}_{\mathsf{new}} = \mathsf{to\_hn}\ \mathit{rightsdb}\ p_{\mathsf{cp}}\ p_{\mathsf{new}};$
      $\mathit{rightsdb}_{\mathsf{cp}} = \lceil \mathit{rightsdb}\ p_{\mathsf{cp}} \rceil$
        $(\!|\mathsf{hdb} := \lceil \mathit{rightsdb}\ p_{\mathsf{cp}} \rceil.\mathsf{hdb}(\mathit{hn}_{\mathsf{new}} \mapsto p_{\mathsf{new}}),$
           $\mathsf{rdb} := \lceil \mathit{rightsdb}\ p_{\mathsf{cp}} \rceil.\mathsf{rdb}(\lfloor p_{\mathsf{new}} \rfloor \mapsto$
             $\{\mathsf{v\_sendR}, \mathsf{v\_requestR}, \mathsf{v\_multipleR}, \mathsf{v\_finiteR}\})|\!);$
      $\mathit{rightsdb}_{\mathsf{new}} = \lceil \mathit{rightsdb}\ p_{\mathsf{cl}} \rceil (\!|\mathsf{hdb} := \lceil \mathit{rightsdb}\ p_{\mathsf{cl}} \rceil.\mathsf{hdb}(\mathsf{HN\_SELF} \mapsto p_{\mathsf{new}})|\!)$
  $\mathbf{in}\ \mathit{rightsdb}(p_{\mathsf{cp}} \mapsto \mathit{rightsdb}_{\mathsf{cp}},\ p_{\mathsf{new}} \mapsto \mathit{rightsdb}_{\mathsf{new}})$
```

Killing a Process

Process termination in VAMOS is a pretty elaborate issue. The main steps on the way to terminate a process p_{victim} are (a) its removal from the scheduling queues, (b) its deregistration as device driver, (c) the invalidation of handles pointing to p_{victim}, and (d) the abortion of pending IPC operations involving it.

In particular, the two latter steps are the most complex ones. They might entail the delivery of notifications or error messages which comes along with the awakening of the processes concerned. In order to identify these situations later on in the formal specification, we introduce some auxiliary predicates.

A process p only receives a notification, if it is active and prepared for receiving, on the one hand, and if it knows the victim process p_{victim}, on the other one. Formally, predicate $\mathsf{knotify\_after\_kill}$ describes this situation:

```
$\mathsf{knotify\_after\_kill}\ \mathit{uprocs}\ \mathit{rightsdb}\ p_{\mathsf{victim}}\ p \equiv$
  $\mathbf{case}\ \mathit{uprocs}\ p\ \mathbf{of}\ \bot \Rightarrow \mathsf{False}$
  $|\ \lfloor x \rfloor \Rightarrow \mathbf{case}\ \omega_{\mathsf{proc}}\ x\ \mathbf{of}$
             $\textsc{ipc\_receive}\ \mathit{hn}_{\mathsf{send}}\ \mathit{msg}\ \mathit{timeout}_{\mathsf{recv}} \Rightarrow$
               $\mathit{hn}_{\mathsf{send}} \in \{\mathsf{HN\_NONE}, \mathsf{HN\_KERNEL}\} \wedge \mathsf{is\_known}\ \mathit{rightsdb}\ p\ p_{\mathsf{victim}}$
             $|\ \_ \Rightarrow \mathsf{False}$
```

Process p is prepared for receiving notifications, if IPC_RECEIVE describes the output and handle hn_{snd} either specifies an open receive ($\mathsf{HN\_NONE}$) or an explicit request to the kernel ($\mathsf{HN\_KERNEL}$). Whether p knows p_{victim} determines predicate $\mathsf{is\_known}$.

Another situation, why a process p receives a message, arises, if p resides in a pending IPC operation, where p_{victim} is involved as receiver, sender, or additional process. The predicate $\mathsf{killed\_assigned\_recv}$ describes this situation:

```
$\mathsf{killed\_assigned\_recv}\ \mathit{uprocs}\ \mathit{rightsdb}\ p_{\mathsf{victim}}\ p \equiv$
  $\mathbf{case}\ \mathit{uprocs}\ p\ \mathbf{of}\ \bot \Rightarrow \mathsf{False}$
  $|\ \lfloor x \rfloor \Rightarrow \mathbf{case}\ \omega_{\mathsf{proc}}\ x\ \mathbf{of}$
             $\textsc{ipc\_send}\ \mathit{hn}_{\mathsf{rcv}}\ \mathit{rights}_{\mathsf{send}}\ \mathit{msg}\ \mathit{hn}_{\mathsf{add}}\ \mathit{rights}_{\mathsf{add}}\ \mathit{timeout}_{\mathsf{send}} \Rightarrow$
```

```
      $\lceil rightsdb\ p\rceil.\mathsf{hdb}\ hn_{\mathsf{rcv}} = \lfloor p_{\mathsf{victim}}\rfloor$
    | $\textsc{ipc\_request}\ hn_{\mathsf{rcv}}\ rights_{\mathsf{send}}\ msg\ hn_{\mathsf{add}}\ rights_{\mathsf{add}}\ timeout_{\mathsf{send}}\ buffer$
      $timeout_{\mathsf{recv}} \Rightarrow$
       $\lceil rightsdb\ p\rceil.\mathsf{hdb}\ hn_{\mathsf{rcv}} = \lfloor p_{\mathsf{victim}}\rfloor$
    | $\_ \Rightarrow \mathsf{False}$
```

For an arbitrary process p, the predicate holds, if p entails three properties: (a) it is active, (b) the output denotes a send operation, i. e., IPC_SEND or IPC_REQUEST, and (c) the specified handle hn_{rcv} refers to p_{victim} in p's handle database. Similar predicates killed_assigned_snd and killed_assigned_add help to determine the remaining situations.

However, these situations only arise, if the process termination succeeds. Thus, as usual, we define a function vamos_result_kill that checks possible invocation errors. The only argument that the calling process p_{cp} has to specify is a handle hn_{victim} to the process that should be terminated. Based on it, the definition of function vamos_result_kill looks as follows:

```
$\mathsf{vamos\_result\_kill}\ rightsdb\ p_{\mathsf{cp}}\ hn_{\mathsf{victim}} \equiv$
  if $hn_{\mathsf{victim}} \neq \mathsf{HN\_SELF} \wedge \neg\ \mathsf{privileged}\ rightsdb\ p_{\mathsf{cp}}$ then ERR_UNPRIVILEGED
  else if $\neg\ \mathsf{valid\_handle}\ rightsdb\ p_{\mathsf{cp}}\ hn_{\mathsf{victim}}$ then ERR_INVALID_HANDLE
       else SUCCESS
```

A special case describes $hn_{\mathsf{victim}} = \mathsf{HN\_SELF}$. It either indicates the voluntary self-termination of p_{cp} or its forced termination by the VAMOS kernel. The latter happens in the context of the runtime-error handling (cf. Section 4.4). Anyways, in both cases, the operation is not attached to any conditions and always succeeds.

Things are different, if $hn_{\mathsf{victim}} \neq \mathsf{HN\_SELF}$, i. e., no self-termination is desired. In this case, SUCCESS is only returned, if p_{cp} owns privileges and handle hn_{victim} is valid. Lacking privileges result in the error message ERR_UNPRIVILEGED, an invalid handle in ERR_INVALID_HANDLE.

The actual delivery of the error message and the state update in the success case describes function vamos_process_kill:

```
$\mathsf{vamos\_process\_kill}\ s_{\mathsf{V}}\ hn_{\mathsf{victim}} \equiv$
  let $p_{\mathsf{cp}} = \lceil \mathsf{v\_cup}\ s_{\mathsf{V}}.\mathsf{schedds}\rceil;$
      $p_{\mathsf{victim}} = \lceil s_{\mathsf{V}}.\mathsf{rightsdb}\ p_{\mathsf{cp}}\rceil.\mathsf{hdb}\ hn_{\mathsf{victim}};$
      $res = \mathsf{vamos\_result\_kill}\ s_{\mathsf{V}}.\mathsf{rightsdb}\ p_{\mathsf{cp}}\ hn_{\mathsf{victim}}$
  in if $\mathsf{is\_error}\ res$ then $s_{\mathsf{V}}(\!|\mathsf{procs} := s_{\mathsf{V}}.\mathsf{procs}(p_{\mathsf{cp}} \mapsto \delta_{\mathsf{proc}}\ res\ \lceil s_{\mathsf{V}}.\mathsf{procs}\ p_{\mathsf{cp}}\rceil)|\!)$
     else $s_{\mathsf{V}}(\!|\mathsf{procs} := \mathsf{v\_kill\_updt\_procs}\ s_{\mathsf{V}}.\mathsf{procs}\ s_{\mathsf{V}}.\mathsf{rightsdb}\ s_{\mathsf{V}}.\mathsf{devds}$
                          $(\mathsf{set}\ s_{\mathsf{V}}.\mathsf{schedds.wait})\ p_{\mathsf{cp}}\ \lceil p_{\mathsf{victim}}\rceil,$
                $\mathsf{schedds} := \mathsf{v\_kill\_updt\_schedds}\ s_{\mathsf{V}}.\mathsf{schedds}\ s_{\mathsf{V}}.\mathsf{procs}\ s_{\mathsf{V}}.\mathsf{priodb}$
                             $s_{\mathsf{V}}.\mathsf{rightsdb}\ \lceil p_{\mathsf{victim}}\rceil,$
                $\mathsf{priodb} := s_{\mathsf{V}}.\mathsf{priodb}(\lceil p_{\mathsf{victim}}\rceil := \bot),$
                $\mathsf{sndstatdb} := \mathsf{v\_kill\_updt\_sndstatdb}\ s_{\mathsf{V}}.\mathsf{sndstatdb}\ s_{\mathsf{V}}.\mathsf{procs}$
                               $s_{\mathsf{V}}.\mathsf{rightsdb}\ \lceil p_{\mathsf{victim}}\rceil,$
                $\mathsf{rightsdb} := \mathsf{v\_kill\_updt\_rightsdb}\ s_{\mathsf{V}}.\mathsf{rightsdb}\ \lceil p_{\mathsf{victim}}\rceil,$
                $\mathsf{devds} := \mathsf{v\_kill\_updt\_devds}\ s_{\mathsf{V}}.\mathsf{devds}\ \lceil p_{\mathsf{victim}}\rceil|\!)$
```

In case of success, the access to p_{cp}'s handle database with handle hn_{victim} is valid and delivers the process number p_{victim}. Setting the entry of p_{victim} in the priority database to $\perp$ is done directly. Dedicated update functions describe the effects on the remaining state components.

Updating the Virtual Machines. The update of the virtual machines affects the entries for the calling process p_{cp}, the terminated process p_{victim} and the processes which are either informed about the abortion of their pending IPC operation or the loss of handles in their handle database.

Function v_kill_updt_procs formally defines the update:

```
$\mathsf{v\_kill\_updt\_procs}\ uprocs\ rightsdb\ devds\ waitQ\ p_{\mathsf{cp}}\ p_{\mathsf{victim}} \equiv$
  $\textbf{let}\ rightsdb_{\mathsf{new}} = \mathsf{v\_kill\_updt\_rightsdb}\ rightsdb\ p_{\mathsf{victim}};$
      $knote = \mathsf{deliverNotifications}\ rightsdb_{\mathsf{new}}\ devds\ devds.\mathsf{saved}$
  $\textbf{in}\ \lambda x.\ \textbf{if}\ x = p_{\mathsf{victim}}\ \textbf{then}\ \perp$
          $\textbf{else if}\ x = p_{\mathsf{cp}}\ \textbf{then}\ \lfloor \delta_{\mathsf{proc}}\ \textsc{success}\ \lceil uprocs\ p_{\mathsf{cp}} \rceil \rfloor$
              $\textbf{else if}\ x \in waitQ \wedge \mathsf{killed\_assigned\_recv}\ uprocs\ rightsdb\ p_{\mathsf{victim}}\ x$
                  $\textbf{then}\ \lfloor \delta_{\mathsf{proc}}\ \textsc{err\_snd\_invalid\_handle}\ \lceil uprocs\ x \rceil \rfloor$
                  $\textbf{else if}\ x \in waitQ \wedge$
                          $\mathsf{killed\_assigned\_add}\ uprocs\ rightsdb\ p_{\mathsf{victim}}\ x$
                      $\textbf{then}\ \lfloor \delta_{\mathsf{proc}}\ \textsc{err\_invalid\_handle}\ \lceil uprocs\ x \rceil \rfloor$
                      $\textbf{else if}\ x \in waitQ \wedge$
                              $\mathsf{killed\_assigned\_snd}\ uprocs\ rightsdb\ p_{\mathsf{victim}}\ x$
                          $\textbf{then}\ \lfloor \delta_{\mathsf{proc}}\ \textsc{err\_rcv\_invalid\_handle}\ \lceil uprocs\ x \rceil \rfloor$
                          $\textbf{else if}\ x \in waitQ \wedge$
                                  $\mathsf{knotify\_after\_kill}\ uprocs\ rightsdb\ p_{\mathsf{victim}}$
                                   $x$
                              $\textbf{then}\ \lfloor \delta_{\mathsf{proc}}\ (knote\ x)\ \lceil uprocs\ x \rceil \rfloor\ \textbf{else}\ uprocs\ x$
```

Due to the successful termination, process p_{victim} is no longer associated with a virtual machine and the caller p_{cp} gets a SUCCESS message. Aborting the pending IPC operations of waiting processes x involves an error message passing. The actual message is determined by means of the predicates introduced above. Thus, x receives ERR_SND_INVALID_HANDLE, if predicate killed_assigned_recv applies. Finally, VAMOS delivers kernel notifications to all processes x that fulfill predicate knotify_after_kill. The virtual machines of the remaining processes remain untouched.

Kernel notifications are assembled by function deliverNotifications which relies on the updated rights datastructures given by v_kill_updt_rightsdb. As we will see later on, it provides the set of stolen handles of a process x after the termination of process p_{victim} containing the handle hn that pointed to p_{victim}.

Based on the updated rights datastructures *rightsdb*, the device datastructures *devds* and the saved interrupts *irqs*, the notification for process x contains the following information:

```
deliverNotifications $rightsdb\ devds\ irqs\ x \equiv$
  let $irqs_{\mathsf{handled}} = \{i \in irqs.\ devds.\mathsf{driver}\ i = \lfloor x \rfloor\}$;
      $stolen_{\mathsf{hns}}$ = case $rightsdb\ x$ of $\perp \Rightarrow \mathsf{False} \mid \lfloor a \rfloor \Rightarrow a.\mathsf{stolen} \neq \{\}$
  in SUCC_RECEIVE HN_KERNEL False {} [] HN_NONE False {} $stolen_{\mathsf{hns}}$ False
      $irqs_{\mathsf{handled}}$
```

In the definition, $irqs_{\mathsf{handled}}$ abbreviates the set of interrupts $i \in irqs$ handled by x. Flag $stolen_{\mathsf{hns}}$ is active, if the entry of x in the rights datastructures $rightsdb$ comprises a non-empty set stolen. Both $irqs_{\mathsf{handled}}$ and $stolen_{\mathsf{hns}}$ represent the main content of the kernel notification to x. Handle HN_KERNEL identifies the kernel as sender, whereas the remaining parts do not communicate any relevant data.

Updating the Scheduling Datastructures. The udpate of the scheduling datastructures comprises the awakening of all waiting processes p that either receive a kernel notification or an error message. Furthermore, the terminated process p_{victim} is taken out of the VAMOS process scheduling, i. e., it is removed from the scheduling queues (wait or ready queues) and put into the inactive queue. Moreover, the process-specific scheduling information is deleted.

Function v_kill_updt_schedds gives the formal specification of all these actions:

```
v_kill_updt_schedds $schedds\ uprocs\ priodb\ rightsdb\ p_{\mathsf{victim}} \equiv$
  let $schedds_{\mathsf{tmp}}$ =
        v_wkp_updt_schedds $schedds$
         (filter (v_kill_wkp $uprocs\ rightsdb\ p_{\mathsf{victim}}$) $schedds.\mathsf{wait}$) $priodb$
  in $schedds_{\mathsf{tmp}}$
     (|inactive := $schedds_{\mathsf{tmp}}.\mathsf{inactive}$ @ [$p_{\mathsf{victim}}$],
       procdb := $schedds_{\mathsf{tmp}}.\mathsf{procdb}(p_{\mathsf{victim}} := \perp)$,
       ready := $\lambda p.\ [x \in schedds_{\mathsf{tmp}}.\mathsf{ready}\ p\ .\ x \neq p_{\mathsf{victim}}]$,
       wait := $[x \in schedds_{\mathsf{tmp}}.\mathsf{wait}\ .\ x \neq p_{\mathsf{victim}}]$|)
```

In a first step, v_kill_updt_schedds realizes the awakening of all waiting processes p that fulfill predicate v_kill_wkp and stores this intermediate result in $schedds_{\mathsf{tmp}}$. The predicate itself realizes a disjunction of the predicates introduced above:

```
v_kill_wkp $uprocs\ rightsdb\ p_{\mathsf{victim}}\ p \equiv$
  killed_assigned_recv $uprocs\ rightsdb\ p_{\mathsf{victim}}\ p\ \vee$
  killed_assigned_add $uprocs\ rightsdb\ p_{\mathsf{victim}}\ p\ \vee$
  killed_assigned_snd $uprocs\ rightsdb\ p_{\mathsf{victim}}\ p\ \vee$
  knotify_after_kill $uprocs\ rightsdb\ p_{\mathsf{victim}}\ p$
```

Based on $schedds_{\mathsf{tmp}}$, p_{victim} is appended to the inactive queue and the entry for p_{victim} in the process-specific scheduling information is set to $\perp$. The act of removing p_{victim} from the scheduling queues is quite brute force. VAMOS does not distinguish whether p_{victim} previously resided in the wait or the ready queue but simply removes it from all of these queues.

Updating the Rights Datastructures. After the termination, VAMOS does no longer hold any information on p_{victim}. Accordingly, the entry for p_{victim} in the rights database is set to $\bot$. Furthermore, all entries regarding p_{victim} in the handle and rights databases of processes p are set to $\bot$. In addition, handles hn refering to p_{victim} are added to the according sets of stolen handles.

The formal semantics encapsulates function v_kill_updt_rightsdb:

```
v_kill_updt_rightsdb $rightsdb$ $p_{\mathsf{victim}}$ $\equiv$
  $\lambda p.$ **if** $p = p_{\mathsf{victim}}$ **then** $\bot$
      **else case** $rightsdb\ p$ **of** $\bot \Rightarrow \bot$
            | $\lfloor db \rfloor \Rightarrow$
              $\lfloor db(\!|$hdb := $\lambda hn.$ **if** $db$.hdb $hn = \lfloor p_{\mathsf{victim}} \rfloor$ **then** $\bot$ **else** $db$.hdb $hn$,
                    rdb := $db$.rdb$(\lfloor p_{\mathsf{victim}} \rfloor := \bot)$,
                    stolen := $db$.stolen $\cup \{hn.\ db$.hdb $hn = \lfloor p_{\mathsf{victim}} \rfloor\}|\!)\rfloor$
```

Note that VAMOS does not allow multiple handles on a single process, i. e., $\{hn.\ db.\mathsf{hdb}\ hn = \lfloor p_{\mathsf{victim}} \rfloor\}$ is a singleton or empty.

Updating the Send Status Database. Updating the send status database is rather straightforward. The entry of p_{victim} is set to $\bot$. Entries of awaken processes (those for which predicate v_kill_wkp holds) are set to False because they no longer perform any IPC operation. The remaining entries stay untouched.

Formally, function v_kill_updt_sndstatdb describes the update:

```
v_kill_updt_sndstatdb $sndstatdb$ $uprocs$ $rightsdb$ $p_{\mathsf{victim}}$ $\equiv$
  $\lambda p.$ **if** $p = p_{\mathsf{victim}}$ **then** $\bot$
      **else if** v_kill_wkp $uprocs$ $rightsdb$ $p_{\mathsf{victim}}$ $p$ **then** $\lfloor$False$\rfloor$ **else** $sndstatdb\ p$
```

Updating the Device Datastructures. Terminating a process might also shut down a driver. This reflects the update of the device datastructures described by function v_kill_updt_devds:

```
v_kill_updt_devds $devds$ $p_{\mathsf{victim}}$ $\equiv$
  **let** $irqs = \{i.\ devds$.driver $i = \lfloor p_{\mathsf{victim}} \rfloor\}$
  **in** $devds$
     $(\!|$driver := $\lambda i.$ **if** $i \in irqs$ **then** $\bot$ **else** $devds$.driver $i$,
       enabled := $devds$.enabled $-$ $irqs$, saved := $devds$.saved $-$ $irqs|\!)$
```

The definition first determines the set $irqs$ of interrupts served by process p_{victim}. Based on this set, the actual update of the device datastructures $devds$ is performed. All interrupts $i \in irqs$ are no longer handled or assigned to any driver, i. e., the entry i in driver is set to $\bot$. As long as no other device driver is assigned, the interrupts are disabled and, hence, subtracted from the set enabled. Finally, VAMOS removes interrupts associated to the driver p_{victim} from the set saved because a delivery is no longer possible.

4.4.4 Scheduling Mechanism

The process scheduling in VAMOS highly depends on the so-called scheduling parameters which comprise the priority and the timeslice of a process. Both values are initially determined during the process creation (cf. Section 4.4.3) and play, as we will see in Section 4.5, an essential role within the decision of the VAMOS scheduler.

The VAMOS call CHG_SCHED_PARAMS provides the possibility to dynamically change these values. As arguments, the calling process p_{cp} has to specify a handle hn_{victim} which identifies the process whose scheduling parameters should be changed to the new timeslice tsl_{new} and priority $prio_{\mathsf{new}}$.

The call only succeeds, if p_{cp} is privileged, hn_{victim} is valid, and $prio_{\mathsf{new}}$ denotes a valid priority, i. e., $prio_{\mathsf{new}} \neq \bot$.

Function vamos_result_change_sched_param checks for possible errors and delivers the corresponding response to p_{cp}:

```
vamos_result_change_sched_param $rightsdb\ p_{\mathsf{cp}}\ hn_{\mathsf{victim}}\ prio_{\mathsf{new}} \equiv$
  if $\neg$ privileged $rightsdb\ p_{\mathsf{cp}}$ then ERR_UNPRIVILEGED
  else if $\neg$ valid_handle $rightsdb\ p_{\mathsf{cp}}\ hn_{\mathsf{victim}}$ then ERR_INVALID_HANDLE
       else if $prio_{\mathsf{new}} = \bot$ then ERR_INVALID_ARGS else SUCCESS
```

The response of vamos_result_change_sched_param is used in the overall specification function vamos_change_sched_param, in order to determine between success and failure:

```
vamos_change_sched_param $s_V\ hn_{\mathsf{victim}}\ tsl_{\mathsf{new}}\ prio_{\mathsf{new}} \equiv$
  let $p_{cp} = \lceil$v_cup $s_V$.schedds$\rceil$; $p_{victim} = \lceil\lceil s_V$.rightsdb $p_{cp}\rceil$.hdb $hn_{\mathsf{victim}}\rceil$;
      $res$ = vamos_result_change_sched_param $s_V$.rightsdb $p_{cp}\ hn_{\mathsf{victim}}\ prio_{\mathsf{new}}$
  in if is_error $res$ then $s_V(\!|$procs := $s_V$.procs$(p_{cp} \mapsto \delta_{proc}\ res\ \lceil s_V$.procs $p_{cp}\rceil)|\!)$
     else $s_V(\!|$procs := $s_V$.procs$(p_{cp} \mapsto \delta_{proc}$ SUCCESS $\lceil s_V$.procs $p_{cp}\rceil)$,
                schedds := v_chng_sched_param_updt_schedds $s_V$.schedds $s_V$.priodb
                                $s_V$.rightsdb $p_{cp}\ hn_{\mathsf{victim}}\ tsl_{\mathsf{new}}\ \lceil prio_{\mathsf{new}}\rceil$,
                priodb := $s_V$.priodb$(p_{victim} := prio_{\mathsf{new}})|\!)$
```

If the call fails, the only update concerns the error message passing to p_{cp}. Otherwise, the message turns into SUCCESS and the scheduling parameters are changed. The process number p_{victim} of the victim is obtained by accessing p_{cp}'s handle database with hn_{victim}. Based on p_{victim}, VAMOS sets the entry in the priority database to $prio_{\mathsf{new}}$ and describes the update of the scheduling datastructures by means of v_chng_sched_param_updt_schedds:

```
v_chng_sched_param_updt_schedds $schedds\ priodb\ rightsdb\ p_{\mathsf{cp}}\ hn_{\mathsf{victim}}$
 $tsl_{\mathsf{new}}\ prio_{\mathsf{new}} \equiv$
  let $p_{victim} = \lceil\lceil rightsdb\ p_{\mathsf{cp}}\rceil$.hdb $hn_{\mathsf{victim}}\rceil$;
      $prio_{old} = \lceil priodb\ p_{victim}\rceil$
  in if $p_{victim} \in$ set $schedds$.wait $\vee\ prio_{\mathsf{new}} = prio_{old}$
     then $schedds$
          $(\!|$procdb := $schedds$.procdb$(p_{victim} \mapsto \lceil schedds$.procdb $p_{victim}\rceil$
             $(\!|$tsl := $tsl_{\mathsf{new}}|\!))|\!)$
```

```
else $schedds$
    $(\!|\mathsf{ready} := schedds.\mathsf{ready}$
      $(prio_{\mathsf{new}} := schedds.\mathsf{ready}\ prio_{\mathsf{new}}\ @\ [p_{\mathsf{victim}}],$
       $prio_{\mathsf{old}} := [x{\in}schedds.\mathsf{ready}\ prio_{\mathsf{old}}\ .\ x \neq p_{\mathsf{victim}}]),$
      $\mathsf{procdb} := schedds.\mathsf{procdb}(p_{\mathsf{victim}} \mapsto \lceil schedds.\mathsf{procdb}\ p_{\mathsf{victim}}\rceil$
       $(\!|\mathsf{tsl} := tsl_{\mathsf{new}}, \mathsf{ctsl} := 0|\!))|\!)$
```

In addition to p_{victim}, the definition also relies on the old priority $prio_{\mathsf{old}}$ of p_{victim}. The first case deals with the situation that either p_{victim} is waiting or the priority of p_{victim} does not change, i. e., $prio_{\mathsf{new}} = prio_{\mathsf{old}}$. Both results in the same update, namely, that only the timeslice of p_{victim} is set to tsl_{new}. Due to the rearrangement of the ready queues, the second case is a bit more elaborate. VAMOS removes p_{victim} from the ready queue of priority $prio_{\mathsf{old}}$ and appends it to the one of priority $prio_{\mathsf{new}}$. For this reason, its consumed time ctsl is reset to 0. Again, the timeslice is set to tsl_{new}.

4.4.5 Device Driver Support

The VAMOS kernel itself only handles the external interrupts from the timer device. For the remaining external interrupts, VAMOS shifts the responsibility to so-called device drivers. Device drivers in VAMOS are dedicated user processes that handle the interrupts from and interact with the external devices. Nevertheless, the VAMOS kernel still acts as interface between these drivers and the devices. On the one hand, it receives the interrupts and passes them on to the associated drivers and, on the other hand, it forwards the requests of the drivers to the devices.

A user process may only act as driver, if it was previously registered by means of the VAMOS call CHANGE_DRIVER. Regarding a dynamic assignment of drivers and devices, the call additionally allows to unregister drivers.

As Section 4.6 illustrates, VAMOS uses IPC for the interrupt delivery. Along with the actual delivery, VAMOS also disables the respective interrupts. An interrupt i remains disabled as long as its driver d completed the handling. Not until then d re-enables i by means of the VAMOS call ENABLE_INTERRUPTS.

Among other duties, the main task of a driver is reading from and writing to a device. Thus, VAMOS provides the corresponding calls DEV_READ and DEV_WRITE.

Based on this brief introduction, the section proceeds with the formal specifications of the device-related VAMOS calls.

Assignment of Device Drivers

The VAMOS call CHANGE_DRIVER enables the dynamic assignment of device drivers. Primarily, it allows the registration and deregistration of drivers but also provides the opportunity to dynamically change the set of interrupts handled by a certain driver.

Accordingly, a calling process p_{cp} has to specify the handle hn_{driver} to the desired driver and the set *irqs* of interrupts, the driver should be registered for or deregistered from. The latter is determined by a flag *register*.

Function vamos_result_change_driver decides on success or failure:

```
vamos_result_change_driver $rightsdb\ devds\ p_{\mathsf{cp}}\ hn_{\mathsf{driver}}\ irqs \equiv$
  if $\neg$ privileged $rightsdb\ p_{\mathsf{cp}}$ then ERR_UNPRIVILEGED
  else if $\neg$ valid_handle $rightsdb\ p_{\mathsf{cp}}\ hn_{\mathsf{driver}}$ then ERR_INVALID_HANDLE
       else if $irqs = \bot \vee \mathsf{DEV\_TIMER} \in \lceil irqs \rceil$ then ERR_INVALID_ARGS
            else if $\exists i \in \lceil irqs \rceil.$
                          $devds.\mathsf{driver}\ i \neq \bot\ \wedge$
                          $devds.\mathsf{driver}\ i \neq \lceil rightsdb\ p_{\mathsf{cp}} \rceil.\mathsf{hdb}\ hn_{\mathsf{driver}}$
                 then ERR_INT_ALREADY_HANDLED else SUCCESS
```

The call CHANGE_DRIVER only succeeds, if the calling process p_{cp} is privileged and provides a valid handle hn_{driver}. Furthermore, the set *irqs* must neither be empty nor contain the interrupt for the timer device. Apart from this, VAMOS does not allow more than one driver for each interrupt. In consequence of that, VAMOS responses with the error message ERR_INT_ALREADY_HANDLED, if an interrupt i out of *irqs* is already assigned to a driver which does not correspond to the one chosen by p_{cp}.

The overall function vamos_change_driver uses the response from the function above and specifies the corresponding update of a VAMOS state s_{V}:

```
vamos_change_driver $s_{\mathsf{V}}\ hn_{\mathsf{driver}}\ irqs\ register \equiv$
  let $p_{\mathsf{cp}} = \lceil \mathsf{v\_cup}\ s_{\mathsf{V}}.\mathsf{schedds} \rceil;$
      $res = \mathsf{vamos\_result\_change\_driver}\ s_{\mathsf{V}}.\mathsf{rightsdb}\ s_{\mathsf{V}}.\mathsf{devds}\ p_{\mathsf{cp}}\ hn_{\mathsf{driver}}\ irqs$
  in if is_error $res$ then $s_{\mathsf{V}}(\!|\mathsf{procs} := s_{\mathsf{V}}.\mathsf{procs}(p_{\mathsf{cp}} \mapsto \delta_{\mathsf{proc}}\ res\ \lceil s_{\mathsf{V}}.\mathsf{procs}\ p_{\mathsf{cp}} \rceil)|\!)$
     else $s_{\mathsf{V}}(\!|\mathsf{procs} := s_{\mathsf{V}}.\mathsf{procs}(p_{\mathsf{cp}} \mapsto \delta_{\mathsf{proc}}\ \mathrm{SUCCESS}\ \lceil s_{\mathsf{V}}.\mathsf{procs}\ p_{\mathsf{cp}} \rceil),$
               $\mathsf{devds} := \mathsf{v\_chg\_drv\_updt\_devds}\ s_{\mathsf{V}}.\mathsf{devds}\ s_{\mathsf{V}}.\mathsf{rightsdb}\ p_{\mathsf{cp}}\ hn_{\mathsf{driver}}\ irqs$
                    $register|\!)$
```

Delivering the result message to the caller p_{cp} goes on as usual. The remaining changes only affect the device datastructures and are described by the dedicated update function v_chg_drv_updt_devds:

```
v_chg_drv_updt_devds $devds\ rightsdb\ p_{\mathsf{cp}}\ hn_{\mathsf{driver}}\ irqs\ register \equiv$
  let $p_{\mathsf{driver}} = \lceil rightsdb\ p_{\mathsf{cp}} \rceil.\mathsf{hdb}\ hn_{\mathsf{driver}}$
  in if $register$
     then $devds$
          $(\!|\mathsf{driver} := \lambda i.$ if $i \in \lceil irqs \rceil$ then $p_{\mathsf{driver}}$ else $devds.\mathsf{driver}\ i,$
            $\mathsf{enabled} := devds.\mathsf{enabled} \cup \lceil irqs \rceil|\!)$
     else $devds$
          $(\!|\mathsf{driver} := \lambda i.$ if $i \in \lceil irqs \rceil$ then $\bot$ else $devds.\mathsf{driver}\ i,$
            $\mathsf{saved} := devds.\mathsf{saved} - \lceil irqs \rceil,\ \mathsf{enabled} := devds.\mathsf{enabled} - \lceil irqs \rceil|\!)$
```

The process number p_{driver} of the driver is obtained by accessing p_{cp}'s handle database with hn_{driver}. Two cases are distinguished. In the first one, p_{driver} should be registered as driver for the interrupts *irqs*. Accordingly, for each interrupt $i \in \lceil irqs \rceil$, the entry in the driver registry is set to p_{driver}. Adding the newly assigned interrupts to the set enabled turns the interrupt delivery on.

The second case deals with the deregistration. According entries in driver are set to $\perp$ and the interrupts are taken out of the sets saved and enabled.

Enabling Interrupts

Specifying the call ENABLE_INTERRUPTS is straightforward. The calling process p_{cp} usually acts as driver and specifies a set *irqs* that should be re-enabled. Function vamos_result_enable_interrupts determines whether this request succeeds or fails:

```
vamos_result_enable_interrupts $devds$ $p_{cp}$ $irqs$ $\equiv$
  if $\exists i \in \lceil irqs \rceil.\ devds.\mathsf{driver}\ i \neq \lfloor p_{cp} \rfloor \vee irqs = \perp$ then ERR_UNPRIVILEGED
  else SUCCESS
```

The latter happens, if either the interrupts are given in an invalid format or at least one of the specified interrupts does not belong to the caller p_{cp}.

The overall specification function vamos_enable_interrupts reflects the low complexity of ENABLE_INTERRUPTS:

```
vamos_enable_interrupts $s_V$ $irqs$ $\equiv$
  let $p_{cp} = \lceil \mathsf{v\_cup}\ s_V.\mathsf{schedds} \rceil$;
      $res = \mathsf{vamos\_result\_enable\_interrupts}\ s_V.\mathsf{devds}\ p_{cp}\ irqs$
  in if is_error $res$ then $s_V(\!| \mathsf{procs} := s_V.\mathsf{procs}(p_{cp} \mapsto \delta_{\mathsf{proc}}\ res\ \lceil s_V.\mathsf{procs}\ p_{cp} \rceil) |\!)$
     else $s_V(\!| \mathsf{procs} := s_V.\mathsf{procs}(p_{cp} \mapsto \delta_{\mathsf{proc}}\ \mathrm{SUCCESS}\ \lceil s_V.\mathsf{procs}\ p_{cp} \rceil),$
                 $\mathsf{devds} := s_V.\mathsf{devds}(\!| \mathsf{enabled} := s_V.\mathsf{devds.enabled} \cup \lceil irqs \rceil |\!) |\!)$
```

As usual, VAMOS either reports success or failure to the caller p_{cp}. In the former case, the interrupts $\lceil irqs \rceil$ are additionally added to the set of enabled interrupts.

Reading from a Device

The VAMOS call DEV_READ enables a driver to read from a device. Thus, as calling process p_{cp}, the driver has to specify a device number dev_{id}, a port number dev_{port} and a memory region *buffer* in its virtual memory that should be used to store the device data.

Specifying invalid device and port numbers are one source of failure. Another one is the memory region. To be used as buffer, it has to be defined and available in the virtual memory of p_{cp}. Furthermore, VAMOS only enables a read operation, if p_{cp} is registered as driver for dev_{id}.

Formally, function vamos_result_dev_read checks for these errors and delivers an according response:

```
vamos_result_dev_read $devds$ $p_{cp}$ $dev_{id}$ $dev_{port}$ $buffer$ $\equiv$
  if $dev_{id} = \perp \vee dev_{port} = \perp \vee buffer \in \{\mathsf{BufUndefined}, \mathsf{BufUnavailable}\}$
  then ERR_INVALID_ARGS
  else if $devds$.driver $\lceil dev_{id} \rceil \neq \lfloor p_{cp} \rfloor$ then ERR_UNPRIVILEGED else SUCCESS
```

Range violations as well as an invalid buffer lead to ERR_INVALID_ARGS. The attempt to read from device dev_{id} without being registered as driver is rejected with the error message ERR_UNPRIVILEGED. Otherwise, the call succeeds and SUCCESS is returned.

The overall specification function vamos_dev_read does not describe the acquisition of the device data but only its delivery to the driver. As described in Section 3.1, the actual interaction between the device dev_{id} and the VAMOS kernel takes place in the overall transition function δ_{V+D}, where a succeeding interaction results in the device data dev_{data}. Function δ_{V+D} then uses this data with the VAMOS transition function δ_V which, in turn, passes dev_{data} on to the VAMOS trap handler, where it finally arrives at vamos_dev_read (cf. Section 4.3 and Section 4.4).

Accordingly, vamos_dev_read takes – apart from the VAMOS state s_V and the parameters specified by the calling process p_{cp} – the device data dev_{data} as input:

```
vamos_dev_read $s_V$ $dev_{id}$ $dev_{port}$ $buffer$ $dev_{data}$ $\equiv$
  let $p_{cp}$ = $\lceil$v_cup $s_V$.schedds$\rceil$;
      $res$ = vamos_result_dev_read $s_V$.devds $p_{cp}$ $dev_{id}$ $dev_{port}$ $buffer$;
      $data$ = mifo_norm (bufLength $buffer$) $dev_{data}$
  in if is_error $res$ then $s_V$(|procs := $s_V$.procs($p_{cp} \mapsto \delta_{proc}$ $res$ $\lceil s_V$.procs $p_{cp}\rceil$)|)
     else $s_V$(|procs := $s_V$.procs($p_{cp} \mapsto \delta_{proc}$ (SUCC_DEV_READ $data$) $\lceil s_V$.procs $p_{cp}\rceil$)|)
```

If DEV_READ fails, the cause of fault is delivered to process p_{cp}. Otherwise, process p_{cp} is provided with the device data in form of message SUCC_DEV_READ. The message, however, comprises the normalized device data *data*, which is obtained by function mifo_norm. In a nutshell, function mifo_norm positions dev_{data} at the beginning of the buffer and, if necessary, adds zeroes, if the buffer is bigger than the actual data portion.

Writing to a Device

Write accesses to a device are provided with the VAMOS call DEV_WRITE. A calling process p_{cp} has to specify the target by means of a device number dev_{id} and a port number dev_{port}. The data that should be written to the device is specified by a memory region *msg*.

Writing to and reading from a device are attached with similar conditions, as the definition of vamos_result_dev_write reflects:

```
vamos_result_dev_write $devds$ $p_{cp}$ $dev_{id}$ $dev_{port}$ $msg$ $\equiv$
  if $dev_{id} = \perp \lor dev_{port} = \perp \lor msg \in$ {MObjUndefined, MObjUnavailable}
  then ERR_INVALID_ARGS
  else if $devds$.driver $\lceil dev_{id} \rceil \neq \lfloor p_{cp} \rfloor$ then ERR_UNPRIVILEGED else SUCCESS
```

Instead of the buffer, in this context the message msg has to be defined and available in the virtual memory of p_{cp}.

The only issue regarding the VAMOS state update is the result delivery. The actual write access to device dev_{id} is again handled outside by $\delta_{\mathsf{V+D}}$.

Accordingly, the definition of vamos_dev_write is quite simple:

```
vamos_dev_write $s_V$ $dev_{id}$ $dev_{port}$ $msg$ $\equiv$
  let $p_{cp} = \lceil \text{v\_cup } s_V.\text{schedds} \rceil$;
      $res = \text{vamos\_result\_dev\_write } s_V.\text{devds } p_{cp}\ dev_{id}\ dev_{port}\ msg$
  in $s_V(|\text{procs} := s_V.\text{procs}(p_{cp} \mapsto \delta_{proc}\ res\ \lceil s_V.\text{procs } p_{cp} \rceil)|)$
```

4.4.6 Inter-Process Communication

As the name suggests, IPC gives the user processes the opportunity to communicate with each other. For sending and receiving, the VAMOS kernel provides the calls IPC_SEND and IPC_RECEIVE. In addition to that, the call IPC_REQUEST enables a combined send and receive with the same communication partner.

Usually, IPC calls traverse several phases. The invocation phase checks the various IPC arguments for their validity. After successfully passing this phase, the operation enters the pre-transmission phase. It searches for possible communication partners and proceeds into the actual transmission, if a rendez-vous situation can be established. If no suitable partner is available, it prepares the calling process for waiting or aborts the call, if the caller specified an immediate timeout. Both, the invocation as well as the pre-transmission phase are call-specific, whereas the transmission has the same semantics for each call.

Accordingly, we first specify the effects of the transmission phase and embed it afterwards in the particular specifications of the IPC calls.

This section concludes with the specification of the call CHANGE_RIGHTS. It does not enable any communication but the administration of the IPC rights.

IPC transmission

The transmission phase finalizes a succeeding IPC operation and relies on a rendez-vous situation between a sender p_{snd} and a receiver p_{rcv}. During this phase, the IPC message of p_{snd} is transferred to p_{rcv} and the VAMOS kernel updates its datastructures, accordingly. As the transmission follows the invocation and pre-transmission phase, we assume the validity of the involved arguments.

The IPC message itself comprises multiple parts. The most common one is the memory message msg_{snd}. It is specified by the sender p_{snd} and describes the memory region that is transferred to the receiver. The receiver, in turn, specifies a memory region buf_{rcv} to store msg_{snd}.

The sender p_{snd} may also grant rights $rights_{\mathsf{snd}}$ to the receiver p_{rcv}. These rights are combined with the (possibly) existing ones of p_{rcv} to p_{snd} and stored in the corresponding entry in p_{rcv}'s rights database. The updated entry denotes the so-called return rights $rrights_{\mathsf{snd}}$ and is advertised to p_{rcv} as part of the result message SUCC_RECEIVE.

Furthermore, p_{snd} is allowed to introduce an additional process p_{add} to p_{rcv}. For this purpose, p_{snd} specifies a handle hn_{add} and rights $rights_{\mathsf{add}}$. As hn_{add} is only valid in p_{snd}'s context, the kernel translates it and provides an according return handle rhn_{add}. In the handle database, VAMOS connects rhn_{add} with p_{add} and, as above, updates the according entry in the rights database. Again, the handle rhn_{add} and the return rights $rrights_{\mathsf{add}}$ are advertised by means of the result message to p_{rcv}.

In addition, the SUCC_RECEIVE message to p_{rcv} also carries notifications from the kernel. Thus, p_{rcv} is informed about stolen or reused handles and occurred interrupts, if acting as driver.

The transmission phase involves various updates on the VAMOS state which are, as usual, encapsulated in dedicated functions.

Updating the Rights Datastructures. The most complex component update in the scope of an IPC transmission is the one for the rights datastructures *rightsdb*. Updating the rights datastructures regarding the sender p_{snd} is rather straightforward and encapsulated in function snd_rightsdb:

```
snd_rightsdb $rightsdb\ p_{\mathsf{snd}}\ p_{\mathsf{rcv}} \equiv$
  let $rights_{\mathsf{rcv}}$ =
        if $\lceil\lceil rightsdb\ p_{\mathsf{snd}}\rceil.\mathsf{rdb}\ p_{\mathsf{rcv}}\rceil \cap \{\mathsf{v\_multipleR}\} = \{\}$ then $\lfloor\{\}\rfloor$
        else $\lceil rightsdb\ p_{\mathsf{snd}}\rceil.\mathsf{rdb}\ p_{\mathsf{rcv}}$
  in $\lceil rightsdb\ p_{\mathsf{snd}}\rceil(|\mathsf{rdb} := \lceil rightsdb\ p_{\mathsf{snd}}\rceil.\mathsf{rdb}(p_{\mathsf{rcv}} := rights_{\mathsf{rcv}})|)$
```

The sender p_{snd} only keeps the rights to p_{rcv}, if it is equipped with the right v_multipleR. Otherwise, VAMOS revokes all rights.

Much more effort is involved with the updates regarding the database of the receiver which mainly rely on the return handles and rights. We use auxiliary functions to determine the particular values.

Function rhn_snd computes the return handle rhn_{snd} to the sender:

```
rhn_snd $rightsdb\ p_{\mathsf{rcv}}\ p_{\mathsf{snd}} \equiv$
  if $\lfloor p_{\mathsf{snd}}\rfloor = \lceil rightsdb\ \lceil p_{\mathsf{rcv}}\rceil\rceil.\mathsf{parent}$ then HN_PARENT else int $p_{\mathsf{snd}}$
```

If the sender p_{snd} is p_{rcv}'s parent, the function returns HN_PARENT. Otherwise, p_{snd} is converted into a handle, i. e., transformed into an integer number. The return rights $rrights_{\mathsf{snd}}$ to the sender result from function rrights_snd:

```
rrights_snd $rightsdb\ p_{\mathsf{rcv}}\ p_{\mathsf{snd}}\ rights_{\mathsf{snd}} \equiv$
  if is_known $rightsdb\ \lceil p_{\mathsf{rcv}}\rceil\ p_{\mathsf{snd}}$
  then $\lfloor\lceil\lceil rightsdb\ \lceil p_{\mathsf{rcv}}\rceil\rceil.\mathsf{rdb}\ \lfloor p_{\mathsf{snd}}\rfloor\rceil \cup \lceil rights_{\mathsf{snd}}\rceil\rfloor$ else $rights_{\mathsf{snd}}$
```

If the sender is already known to the receiver, the return rights describe the combination of the already existing ones with those of $\mathit{rights}_{\mathsf{snd}}$. Otherwise, $\mathit{rights}_{\mathsf{snd}}$ denotes the initial rights to the sender.

The updates of p_{rcv}'s entries in the rights datastructures $\mathsf{rightsdb}$ regarding the sender are encapsulated in function $\mathsf{rcv\_rightsdb\_snd}$:

$$\begin{array}{l}
\mathsf{rcv\_rightsdb\_snd}\ \mathit{rightsdb}\ p_{\mathsf{rcv}}\ p_{\mathsf{snd}}\ \mathit{rhn}_{\mathsf{snd}}\ \mathit{rrights}_{\mathsf{snd}} \equiv \\
\mathit{rightsdb}(\lceil p_{\mathsf{rcv}}\rceil \mapsto \lceil \mathit{rightsdb}\ \lceil p_{\mathsf{rcv}}\rceil\rceil \\
(\!|\mathsf{hdb} := \lceil \mathit{rightsdb}\ \lceil p_{\mathsf{rcv}}\rceil\rceil.\mathsf{hdb}(\mathit{rhn}_{\mathsf{snd}} \mapsto p_{\mathsf{snd}}), \\
\quad \mathsf{rdb} := \lceil \mathit{rightsdb}\ \lceil p_{\mathsf{rcv}}\rceil\rceil.\mathsf{rdb}(\lfloor p_{\mathsf{snd}}\rfloor := \mathit{rrights}_{\mathsf{snd}}), \\
\quad \mathsf{stolen} := \lceil \mathit{rightsdb}\ \lceil p_{\mathsf{rcv}}\rceil\rceil.\mathsf{stolen} - \{\mathit{rhn}_{\mathsf{snd}}\}|\!))
\end{array}$$

Within the handle database of p_{rcv}, $\mathit{rhn}_{\mathsf{snd}}$ is connected with the process number p_{snd} and $\mathit{rrights}_{\mathsf{snd}}$ is stored in the according entry in the rights database. Finally, $\mathit{rhn}_{\mathsf{snd}}$ is removed from the set of stolen handles. This operation has no effect as long as $\mathit{rhn}_{\mathsf{snd}}$ has not been marked as stolen. Otherwise, it is now reused and no longer stolen. Nevertheless, later on we will see, that p_{rcv} is notified about this replacement by marking $\mathit{rhn}_{\mathsf{snd}}$ as reused in the result message.

More complex are the computations of the return handle $\mathit{rhn}_{\mathsf{add}}$ and the return rights $\mathit{rrights}_{\mathsf{add}}$, because we have to take into account that no additional process has been announced at all or that the process corresponds with the receiver. This leads to the following definition of the auxiliary function $\mathsf{rhn\_add}$ delivering $\mathit{rhn}_{\mathsf{add}}$:

```
$\mathsf{rhn\_add}\ \mathit{rightsdb}\ p_{\mathsf{rcv}}\ p_{\mathsf{snd}}\ p_{\mathsf{add}}\ \mathit{hn}_{\mathsf{add}} \equiv$
  **if** $\mathit{hn}_{\mathsf{add}} = \mathsf{HN\_NONE}$ **then** $\mathsf{HN\_NONE}$
  **else if** $p_{\mathsf{add}} = p_{\mathsf{rcv}}$ **then** $\mathsf{HN\_SELF}$
       **else if** $p_{\mathsf{add}} = \lceil \mathit{rightsdb}\ \lceil p_{\mathsf{rcv}}\rceil\rceil.\mathsf{parent}$ **then** $\mathsf{HN\_PARENT}$ **else** $\mathsf{int}\ \lceil p_{\mathsf{add}}\rceil$
```

In case that no additional process has been announced by the sender, i. e., $\mathit{hn}_{\mathsf{add}} = \mathsf{HN\_NONE}$, the return handle $\mathit{rhn}_{\mathsf{add}}$ is also $\mathsf{HN\_NONE}$. Handle $\mathsf{HN\_SELF}$ is returned, if the additional process p_{add} corresponds with the receiver. The return handle denotes $\mathsf{HN\_PARENT}$, if p_{add} denotes the receiver's parent. Otherwise, p_{add} is converted into a handle.

The return rights $\mathit{rrights}_{\mathsf{add}}$ to the additional process result from function $\mathsf{rrights\_add}$:

```
$\mathsf{rrights\_add}\ \mathit{rightsdb}\ p_{\mathsf{rcv}}\ p_{\mathsf{snd}}\ p_{\mathsf{add}}\ \mathit{hn}_{\mathsf{add}}\ \mathit{rights}_{\mathsf{add}} \equiv$
  **if** $\mathit{hn}_{\mathsf{add}} = \mathsf{HN\_NONE}$ **then** $\bot$
  **else if** $p_{\mathsf{add}} = p_{\mathsf{rcv}}$ **then** $\lceil \mathit{rightsdb}\ \lceil p_{\mathsf{rcv}}\rceil\rceil.\mathsf{rdb}\ p_{\mathsf{add}}$
       **else if** $\mathsf{is\_known}\ \mathit{rightsdb}\ \lceil p_{\mathsf{rcv}}\rceil\ \lceil p_{\mathsf{add}}\rceil$
            **then** $\lfloor\lceil\lceil \mathit{rightsdb}\ \lceil p_{\mathsf{rcv}}\rceil\rceil.\mathsf{rdb}\ p_{\mathsf{add}}\rceil \cup \lceil \mathit{rights}_{\mathsf{add}}\rceil\rfloor$ **else** $\mathit{rights}_{\mathsf{add}}$
```

The return rights $\mathit{rrights}_{\mathsf{add}}$ denote $\bot$, if no additional process was specified, i. e., $\mathit{hn}_{\mathsf{add}} = \mathsf{HN\_NONE}$. If the additional process is equal to the receiver, they denote the corresponding entry in the rights database. In all other cases, they describe the combination of already existing rights with $rights_{\mathsf{add}}$, if the additional process was known, and are equal to $rights_{\mathsf{add}}$, otherwise.

The updates of p_{rcv}'s entries in the rights datastructures $\mathsf{rightsdb}$ regarding the additional process are similar to the ones regarding the sender, as the definition of function $\mathsf{rcv\_rightsdb\_add}$ suggests:

$$
\begin{aligned}
&\mathsf{rcv\_rightsdb\_add}\ rightsdb\ p_{\mathsf{rcv}}\ p_{\mathsf{snd}}\ p_{\mathsf{add}}\ rhn_{\mathsf{add}}\ rrights_{\mathsf{add}} \equiv rightsdb(\lceil p_{\mathsf{rcv}}\rceil \mapsto\\
&\quad \lceil rightsdb\ \lceil p_{\mathsf{rcv}}\rceil\rceil\\
&\quad (\!|\mathsf{hdb} := \lceil rightsdb\ \lceil p_{\mathsf{rcv}}\rceil\rceil.\mathsf{hdb}(rhn_{\mathsf{add}} := p_{\mathsf{add}}),\\
&\qquad \mathsf{rdb} := \lceil rightsdb\ \lceil p_{\mathsf{rcv}}\rceil\rceil.\mathsf{rdb}(p_{\mathsf{add}} := rrights_{\mathsf{add}}),\\
&\qquad \mathsf{stolen} := \lceil rightsdb\ \lceil p_{\mathsf{rcv}}\rceil\rceil.\mathsf{stolen} - \{rhn_{\mathsf{add}}\}|\!))
\end{aligned}
$$

Function $\mathsf{rcv\_rightsdb}$ combines the updates of the receiver's rights datastructures:

$$
\begin{aligned}
&\mathsf{rcv\_rightsdb}\ rightsdb\ p_{\mathsf{rcv}}\ p_{\mathsf{snd}}\ rights_{\mathsf{snd}}\ hn_{\mathsf{add}}\ rights_{\mathsf{add}} \equiv\\
&\quad \mathbf{let}\ \mathit{rhn}_{\mathsf{snd}} = \mathsf{rhn\_snd}\ rightsdb\ p_{\mathsf{rcv}}\ p_{\mathsf{snd}};\\
&\qquad \mathit{rrights}_{\mathsf{snd}} = \mathsf{rrights\_snd}\ rightsdb\ p_{\mathsf{rcv}}\ p_{\mathsf{snd}}\ rights_{\mathsf{snd}};\\
&\qquad \mathit{rightsdb}' = \mathsf{rcv\_rightsdb\_snd}\ rightsdb\ p_{\mathsf{rcv}}\ p_{\mathsf{snd}}\ \mathit{rhn}_{\mathsf{snd}}\ \mathit{rrights}_{\mathsf{snd}};\\
&\qquad \mathit{p}_{\mathsf{add}} = \lceil rightsdb\ p_{\mathsf{snd}}\rceil.\mathsf{hdb}\ hn_{\mathsf{add}};\\
&\qquad \mathit{rhn}_{\mathsf{add}} = \mathsf{rhn\_add}\ rightsdb\ p_{\mathsf{rcv}}\ p_{\mathsf{snd}}\ \mathit{p}_{\mathsf{add}}\ hn_{\mathsf{add}};\\
&\qquad \mathit{rrights}_{\mathsf{add}} = \mathsf{rrights\_add}\ \mathit{rightsdb}'\ p_{\mathsf{rcv}}\ p_{\mathsf{snd}}\ \mathit{p}_{\mathsf{add}}\ hn_{\mathsf{add}}\ rights_{\mathsf{add}};\\
&\qquad \mathit{rightsdb}'' = \mathsf{rcv\_rightsdb\_add}\ \mathit{rightsdb}'\ p_{\mathsf{rcv}}\ p_{\mathsf{snd}}\ \mathit{p}_{\mathsf{add}}\ \mathit{rhn}_{\mathsf{add}}\ \mathit{rrights}_{\mathsf{add}}\\
&\quad \mathbf{in}\ \lceil \mathit{rightsdb}''\ \lceil p_{\mathsf{rcv}}\rceil\rceil
\end{aligned}
$$

Based on the original rights datastructures *rightsdb*, the definition first computes the return handle $\mathit{rhn}_{\mathsf{snd}}$ and rights $\mathit{rrights}_{\mathsf{snd}}$ to the sender. Both are provided as input to function $\mathsf{rcv\_rightsdb\_snd}$ which computes the intermediate update $\mathit{rightsdb}'$. Afterwards, the definition computes the return handle $\mathit{rhn}_{\mathsf{add}}$ to the additional process $\mathit{p}_{\mathsf{add}}$. Note that the latter is only defined, if $hn_{\mathsf{add}} \neq \mathsf{HN\_NONE}$. The computation of the return rights $\mathit{rrights}_{\mathsf{add}}$ is based on the rights datastructures $\mathit{rightsdb}'$ and performed by function $\mathsf{rrights\_add}$. The rights datastructures $\mathit{rightsdb}'$ also serve as basis for function $\mathsf{rcv\_rightsdb\_add}$ which computes the updates of p_{rcv}'s rights database regarding the additional process. The result is stored in $\mathit{rightsdb}''$. Finally, function $\mathsf{rcv\_rightsdb}$ returns the entry of the receiver in $\mathit{rightsdb}''$.

The combination of the functions $\mathsf{snd\_rightsdb}$ and $\mathsf{rcv\_rightsdb}$ delivers the overall update of the rights datastructures:

$$
\begin{aligned}
&\mathsf{v\_ipc\_trans\_updt\_rightsdb}\ rightsdb\ p_{\mathsf{snd}}\ hn_{\mathsf{rcv}}\ hn_{\mathsf{add}}\ rights_{\mathsf{snd}}\ rights_{\mathsf{add}} \equiv\\
&\quad \mathbf{let}\ \mathit{p}_{\mathsf{rcv}} = \lceil rightsdb\ p_{\mathsf{snd}}\rceil.\mathsf{hdb}\ hn_{\mathsf{rcv}};\\
&\qquad \mathit{rightsdb}_{\mathsf{rcv}} = \mathsf{rcv\_rightsdb}\ rightsdb\ \mathit{p}_{\mathsf{rcv}}\ p_{\mathsf{snd}}\ rights_{\mathsf{snd}}\ hn_{\mathsf{add}}\ rights_{\mathsf{add}};\\
&\qquad \mathit{rightsdb}_{\mathsf{snd}} = \mathsf{snd\_rightsdb}\ rightsdb\ p_{\mathsf{snd}}\ \mathit{p}_{\mathsf{rcv}}\\
&\quad \mathbf{in}\ rightsdb(\lceil \mathit{p}_{\mathsf{rcv}}\rceil \mapsto \mathit{rightsdb}_{\mathsf{rcv}},\ p_{\mathsf{snd}} \mapsto \mathit{rightsdb}_{\mathsf{snd}})
\end{aligned}
$$

Updating the Virtual Machines. The update of the virtual machines mainly comprises the result messages to the sender and the receiver. However, the former only gets a SUCCESS message, if the sending was not part of an IPC_REQUEST call. If so, the intended operation is not completed because the receive phase will still follow.

In contrast, receiving always terminates an IPC operation. Consequently, the receiver p_{rcv} is notified about the successful transmission by means of a SUCC_RECEIVE message. Main content of the message are the return handles together with the associated rights and the actual memory message from the sender. In addition, p_{rcv} is provided with some administrative information, for instance, whether the returned handles have previously been used for other processes or any interrupts occurred or, if the handle database contains any invalid handles.

Function v_ipc_trans_updt_procs encapsulates the formal description of the message delivery:

```
v_ipc_trans_updt_procs $uprocs\ rightsdb\ devds\ p_{\mathsf{snd}}\ req\ hn_{\mathsf{rcv}}\ rights_{\mathsf{snd}}$
 $msg\ hn_{\mathsf{add}}\ rights_{\mathsf{add}} \equiv$
  let $p_{\mathsf{rcv}} = \lceil rightsdb\ p_{\mathsf{snd}}\rceil.\mathsf{hdb}\ hn_{\mathsf{rcv}};$
      $p_{\mathsf{add}} = \lceil rightsdb\ p_{\mathsf{snd}}\rceil.\mathsf{hdb}\ hn_{\mathsf{add}};$
      $rhn_{\mathsf{snd}} = \mathsf{to\_hn}\ rightsdb\ \lceil p_{\mathsf{rcv}}\rceil\ p_{\mathsf{snd}};$
      $reused_{\mathsf{snd}} = rhn_{\mathsf{snd}} \in \lceil rightsdb\ \lceil p_{\mathsf{rcv}}\rceil\rceil.\mathsf{stolen};$
      $rhn_{\mathsf{add}} = \mathsf{rhn\_add}\ rightsdb\ p_{\mathsf{rcv}}\ p_{\mathsf{snd}}\ p_{\mathsf{add}}\ hn_{\mathsf{add}};$
      $reused_{\mathsf{add}} = rhn_{\mathsf{add}} \neq rhn_{\mathsf{snd}} \wedge rhn_{\mathsf{add}} \in \lceil rightsdb\ \lceil p_{\mathsf{rcv}}\rceil\rceil.\mathsf{stolen};$
      $rightsdb_{\mathsf{post}} =$
        v_ipc_trans_updt_rightsdb $rightsdb\ p_{\mathsf{snd}}\ hn_{\mathsf{rcv}}\ hn_{\mathsf{add}}\ rights_{\mathsf{snd}}$
         $rights_{\mathsf{add}};$
      $rrights_{\mathsf{snd}} = \lceil\lceil rightsdb_{\mathsf{post}}\ \lceil p_{\mathsf{rcv}}\rceil\rceil.\mathsf{rdb}\ \lfloor p_{\mathsf{snd}}\rfloor\rceil;$
      $rrights_{\mathsf{add}} =$
        if $p_{\mathsf{add}} = p_{\mathsf{rcv}}$ then $\{\}$ else $\lceil\lceil rightsdb_{\mathsf{post}}\ \lceil p_{\mathsf{rcv}}\rceil\rceil.\mathsf{rdb}\ p_{\mathsf{add}}\rceil;$
      $stolen = \lceil rightsdb_{\mathsf{post}}\ \lceil p_{\mathsf{rcv}}\rceil\rceil.\mathsf{stolen} \neq \{\};$
      $irqs = \{i \in devds.\mathsf{saved}.\ devds.\mathsf{driver}\ i = p_{\mathsf{rcv}}\};$
      $to_{\mathsf{inf}} = \mathsf{timeoutInfinite}\ uprocs\ p_{\mathsf{snd}}$
  in $\lambda x.$ if $x = p_{\mathsf{snd}} \wedge \neg\ req$ then $\lfloor \delta_{\mathsf{proc}}\ \textsc{success}\ \lceil uprocs\ x\rceil\rfloor$
           else if $x = \lceil p_{\mathsf{rcv}}\rceil$
                then $\lfloor \delta_{\mathsf{proc}}$ (SUCC_RECEIVE $rhn_{\mathsf{snd}}\ reused_{\mathsf{snd}}\ rrights_{\mathsf{snd}}$
                                (mObjSeq $msg$) $rhn_{\mathsf{add}}\ reused_{\mathsf{add}}\ rrights_{\mathsf{add}}$
                                $stolen\ to_{\mathsf{inf}}\ irqs$)
                        $\lceil uprocs\ x\rceil\rfloor$
                else $uprocs\ x$
```

An additional input parameter req determines whether the sender p_{snd} performs an IPC_REQUEST. Depending on this, its virtual machine is either provided with a SUCCESS message or remains unchanged.

Assembling the SUCC_RECEIVE message for p_{rcv} describes the remaining parts of the definition. The return handles rhn_{snd} and rhn_{add} are computed as before and marked as reused, if they were previously contained in the stolen set of p_{rcv}. We use the flags $reused_{\mathsf{snd}}$ and $reused_{\mathsf{add}}$ to signal a re-use, whereas the latter is only active, if rhn_{add} is additionally different from rhn_{snd}. The returned rights rely on the updated rights datastructures $rightsdb_{\mathsf{post}}$ which result from the previously introduced function

$\mathsf{v\_ipc\_trans\_updt\_rightsdb}$. Actually, the rights $\mathit{rrights}_{\mathsf{snd}}$ and $\mathit{rrights}_{\mathsf{add}}$ correspond to the entries in the rights database. The latter, however, describes the empty set, if the additional process is the receiver itself. Invalid handles in the updated handle database of p_{rcv} are signaled by the flag *stolen*. It is active, if the set of stolen handles is not empty. The state component saved stores all occurred but not yet handled external device interrupts. If p_{rcv} acts as driver for any of these interrupts, VAMOS uses the SUCC_RECEIVE message for the interrupt delivery. For this purpose, set *irqs* subsumes all interrupts which are handled by p_{rcv}.

With p_{rcv} acting as server, it might be interesting to know, whether the transmission was part of an IPC_REQUEST initiated by p_{snd} and, in particular, whether the upcoming receive phase is performed with an infinite timeout. If so, flag $\mathit{to}_{\mathsf{inf}}$ is active, which is formally determined by predicate timeoutInfinite:

$\mathsf{timeoutInfinite}\ \mathit{uprocs}\ p \equiv$
$\quad \exists \mathit{rcv\_hn}\ \mathit{snd\_rights}\ \mathit{msg}\ \mathit{add\_hn}\ \mathit{add\_rights}\ \mathit{snd\_timeout}\ \mathit{buffer}.$
$\qquad \omega_{\mathsf{proc}}\ \lceil \mathit{uprocs}\ p \rceil =$
$\qquad \text{IPC\_REQUEST}\ \mathit{rcv\_hn}\ \mathit{snd\_rights}\ \mathit{msg}\ \mathit{add\_hn}\ \mathit{add\_rights}\ \mathit{snd\_timeout}\ \mathit{buffer}\ \infty$

Updating the Send Status Database. The update of the send status database is straightforward. From the receiver's point of view the IPC operation is completed. Consequently, the send status is set to False. After sending, the status of the sender depends on its actual operation. If the send phase was part of an IPC_REQUEST, the send status is active, i. e., set to True. Otherwise, the status is inactive. VAMOS determines both situation by means of the flag *req*.

$\mathsf{v\_ipc\_trans\_updt\_sndstatdb}\ \mathit{sndstatdb}\ p_{\mathsf{snd}}\ p_{\mathsf{rcv}}\ \mathit{req} \equiv \mathit{sndstatdb}(p_{\mathsf{snd}} \mapsto \mathit{req},\ p_{\mathsf{rcv}} \mapsto \mathsf{False})$

Updating the Device Datastructures. As seen before, the message SUCC_RECEIVE is used to deliver occurred interrupts to p_{rcv}. Hence, from the kernel's point of view there is no need to store them any longer. Thus, VAMOS removes the delivered interrupts from the set of saved interrupts.

The function v_ipc_trans_updt_devds specifies the according effects on the device datastructures:

$\mathsf{v\_ipc\_trans\_updt\_devds}\ \mathit{devds}\ p_{\mathsf{rcv}} \equiv$
$\quad \textbf{let}\ \mathit{irqs}_{\mathsf{del}} = \{i.\ \mathit{devds}.\mathsf{driver}\ i = \lfloor p_{\mathsf{rcv}} \rfloor\}\ \textbf{in}\ \mathit{devds}(\!|\mathsf{saved} := \mathit{devds}.\mathsf{saved} - \mathit{irqs}_{\mathsf{del}}|\!)$

IPC Send

In order to invoke the VAMOS call IPC_SEND, the (expected) sender p_{snd} has to specify (a) the handle $\mathit{hn}_{\mathsf{rcv}}$ to the receiver together with the rights

$\mathit{rights}_{\mathsf{snd}}$ that should be granted, (b) the handle $\mathit{hn}_{\mathsf{add}}$ to a (potential) additional process associated with the rights $\mathit{rights}_{\mathsf{add}}$ for the receiver, (c) the memory message $\mathit{msg}_{\mathsf{snd}}$, and (d) the timeout $\mathit{to}_{\mathsf{snd}}$.

Based on these values, VAMOS performs error checks which reflect the different phases of IPC_SEND. We first introduce these checks in the following paragraphs before we proceed with the overall specification.

Invocation. In order to pass the invocation phase, the input arguments specified by the sender p_{snd}, have to fulfill certain requirements. These requirements bear on the correct specification of the arguments but also take the required IPC rights into account.

Formally, function ipc_send_invoc_err describes the error checks:

```
$\mathsf{ipc\_send\_invoc\_err}\ \mathit{rightsdb}\ p_{\mathsf{snd}}\ \mathit{hn}_{\mathsf{rcv}}\ \mathit{rights}_{\mathsf{snd}}\ \mathit{msg}\ \mathit{hn}_{\mathsf{add}}\ \mathit{rights}_{\mathsf{add}} \equiv$
  if $\mathit{hn}_{\mathsf{rcv}} = \mathsf{HN\_SELF} \lor \neg\ \mathsf{valid\_handle}\ \mathit{rightsdb}\ p_{\mathsf{snd}}\ \mathit{hn}_{\mathsf{rcv}}$
  then ERR_SND_INVALID_HANDLE
  else if $\mathit{hn}_{\mathsf{add}} \neq \mathsf{HN\_NONE} \land \neg\ \mathsf{valid\_handle}\ \mathit{rightsdb}\ p_{\mathsf{snd}}\ \mathit{hn}_{\mathsf{add}}$
       then ERR_INVALID_HANDLE
       else if $\neg\ \mathsf{valid\_snd\_args}\ \mathit{rights}_{\mathsf{snd}}\ \mathit{rights}_{\mathsf{add}}\ \mathit{hn}_{\mathsf{add}}\ \mathit{msg}$
            then ERR_INVALID_ARGS
            else if $\mathit{msg} = \mathsf{MObjUnavailable}$ then ERR_SND_SEGV
                 else if $\neg\ \mathsf{allowed\_snd\_op}\ \mathit{rightsdb}\ p_{\mathsf{snd}}\ \mathit{hn}_{\mathsf{rcv}}\ \mathit{rights}_{\mathsf{add}}$
                      then ERR_UNPRIVILEGED else SUCCESS
```

VAMOS rejects the receiver handle $\mathit{hn}_{\mathsf{rcv}}$ with ERR_SND_INVALID_HANDLE, if it intends a self-reference or if it is invalid.

Error ERR_INVALID_HANDLE is returned, as soon as p_{snd} wants to publish an additional process, i. e., $\mathit{hn}_{\mathsf{add}} \neq \mathsf{HN\_NONE}$, but $\mathit{hn}_{\mathsf{add}}$ is not valid.

The requirements for the specified rights and the memory message encapsulates predicate valid_snd_args:

```
$\mathsf{valid\_snd\_args}\ \mathit{rights}_{\mathsf{snd}}\ \mathit{rights}_{\mathsf{add}}\ \mathit{hn}_{\mathsf{add}}\ \mathit{msg} \equiv$
  $\mathit{rights}_{\mathsf{snd}} \neq \bot \land \mathit{rights}_{\mathsf{add}} \neq \bot \land \mathit{msg} \neq \mathsf{MObjUndefined}$
```

It demands valid rights, i. e., $\mathit{rights}_{\mathsf{snd}}$ and $\mathit{rights}_{\mathsf{add}}$ are not equal to $\bot$, and a defined memory message, i. e., $\mathit{msg}_{\mathsf{snd}} \neq \mathsf{MObjUndefined}$. Any violations lead to the error value ERR_INVALID_ARGS.

An unavailable memory message is acknowledged with the error message ERR_SND_SEGV, whereas ERR_UNPRIVILEGED indicates insufficient rights. The latter is checked by means of predicate allowed_snd_op:

```
$\mathsf{allowed\_snd\_op}\ \mathit{rightsdb}\ p_{\mathsf{snd}}\ \mathit{hn}_{\mathsf{rcv}}\ \mathit{rights}_{\mathsf{add}} \equiv$
  let $p_{\mathsf{rcv}} = \lceil \mathit{rightsdb}\ p_{\mathsf{snd}} \rceil.\mathsf{hdb}\ \mathit{hn}_{\mathsf{rcv}}$
  in $\mathsf{v\_sendR} \in \lceil\lceil \mathit{rightsdb}\ p_{\mathsf{snd}} \rceil.\mathsf{rdb}\ p_{\mathsf{rcv}} \rceil \land$
     $(\mathit{rights}_{\mathsf{add}} = \lfloor\{\}\rfloor \lor \mathsf{privileged}\ \mathit{rightsdb}\ p_{\mathsf{snd}})$
```

Two prerequisites have to be fulfilled by the sender: (a) it needs the send right v_sendR to the receiver, and (b) it needs privileges to assign a non-empty set $\mathit{rights}_{\mathsf{add}}$ of rights to the additional process.

The invocation phase succeeds, if function ipc_send_invoc_err returns SUCCESS.

Pre-Transmission. After successfully passing the invocation phase, the IPC_SEND operation enters the pre-transmission phase. The aim of this phase is to determine whether the desired communication partner is ready to receive and, at the same time, meets all requirements for the operation, i. e., that the specified buffer buf_{rcv} is big enough for msg_{snd}.

VAMOS uses the predicate ipc_send_rendez_vous to determine the readiness of the receiver:

```
ipc_send_rendez_vous $uprocs$ $schedds$ $sndstatusdb$ $rightsdb$ $p_{\mathsf{snd}}$ $hn_{\mathsf{rcv}}$ $\equiv$
  let $p_{\mathsf{rcv}} = \lceil\lceil rightsdb\ p_{\mathsf{snd}}\rceil.\mathsf{hdb}\ hn_{\mathsf{rcv}}\rceil$
  in $p_{\mathsf{rcv}} \in \mathsf{set}\ schedds.\mathsf{wait} \wedge$
     (case $\omega_{\mathsf{proc}}\ \lceil uprocs\ p_{\mathsf{rcv}}\rceil$ of
      IPC_RECEIVE $hn_{\mathsf{snd}}$ $buffer$ $to_{\mathsf{rcv}}$ $\Rightarrow$
        $\lceil rightsdb\ p_{\mathsf{rcv}}\rceil.\mathsf{hdb}\ hn_{\mathsf{snd}} = \lfloor p_{\mathsf{snd}}\rfloor \vee hn_{\mathsf{snd}} = \mathsf{HN\_NONE}$
      | IPC_REQUEST $hn_{\mathsf{rcv}}$ $rights_{\mathsf{snd}}$ $msg$ $hn_{\mathsf{add}}$ $rights_{\mathsf{add}}$ $to_{\mathsf{snd}}$ $buffer$ $to_{\mathsf{rcv}}$ $\Rightarrow$
          $\lceil sndstatusdb\ p_{\mathsf{rcv}}\rceil \wedge \lceil rightsdb\ p_{\mathsf{rcv}}\rceil.\mathsf{hdb}\ hn_{\mathsf{rcv}} = \lfloor p_{\mathsf{snd}}\rfloor$
      | _ $\Rightarrow$ False)
```

Due to the validity of hn_{rcv}, the process number p_{rcv} of the receiver can be taken from the handle database of p_{snd}. The predicate only applies, if p_{rcv} is waiting while looking forward to a receive operation with p_{snd} involved. The former is denoted by p_{rcv}'s membership in the wait queue $schedds.\mathsf{wait}$, whereas the latter is determined by the process output. The will to receive is signaled by the outputs IPC_RECEIVE and IPC_REQUEST. In the former case, p_{rcv} is ready to receive from p_{snd}, if the handle hn_{snd} either specifies an open-receive, i. e., $hn_{\mathsf{snd}} = \mathsf{HN\_NONE}$, or points to p_{snd}. With IPC_REQUEST no open-receives are possible. Thus, hn_{snd} has to point to p_{snd}. In addition, the send phase has to be completed, which is indicated by an active send status.

Note that the calls IPC_RECEIVE and IPC_REQUEST already passed the invocation phase, before they have been forced to wait. Thus, demanding the wait state of p_{rcv} ensures, that the call arguments are valid and can safely be used.

Predicate ipc_send_buffer_ovfl determines, whether the receive buffer buf_{rcv} is big enough for msg_{snd}. Relying on the validity of msg_{snd} as well as buf_{rcv}, it simply compares their sizes:

```
ipc_send_buffer_ovfl $uprocs$ $rightsdb$ $p_{\mathsf{snd}}$ $hn_{\mathsf{rcv}}$ $msg_{\mathsf{snd}}$ $\equiv$
  let $p_{\mathsf{rcv}} = \lceil\lceil rightsdb\ p_{\mathsf{snd}}\rceil.\mathsf{hdb}\ hn_{\mathsf{rcv}}\rceil$; $proc_{\mathsf{rcv}} = \lceil uprocs\ p_{\mathsf{rcv}}\rceil$
  in $\mathsf{get\_buflen}\ (\omega_{\mathsf{proc}}\ proc_{\mathsf{rcv}}) < \mathsf{length}\ (\mathsf{mObjSeq}\ msg_{\mathsf{snd}})$
```

Both are compatible, if the length of msg_{snd} is smaller than the one of buf_{rcv}. Otherwise, a buffer overflow occurs.

Overall Specification. VAMOS distinguishes the different phases of the VAMOS call IPC_SEND by means of the output of function ipc_send_err:

```
ipc_send_err $uprocs$ $schedds$ $sndstatusdb$ $rightsdb$ $hn_{rcv}$ $rights_{snd}$ $msg$ $hn_{add}$
 $rights_{add}$ $to_{snd}$ $\equiv$
  let $p_{snd} = \lceil$v_cup $schedds\rceil$;
      $res$ = ipc_send_invoc_err $rightsdb$ $p_{snd}$ $hn_{rcv}$ $rights_{snd}$ $msg$ $hn_{add}$ $rights_{add}$
  in if is_error $res$ then $res$
     else if ipc_send_rendez_vous $uprocs$ $schedds$ $sndstatusdb$ $rightsdb$ $p_{snd}$ $hn_{rcv}$
          then if ipc_send_buffer_ovfl $uprocs$ $rightsdb$ $p_{snd}$ $hn_{rcv}$ $msg$
               then ERR_SND_BUFFER_OVFL else SUCCESS
          else if $to_{snd} = 0$ then ERR_SND_TIMEOUT else $\varepsilon_\Sigma$
```

Based on the current process acting as sender p_{snd} and the result *res* from the function ipc_send_invoc_err, the definition reflects the different phases. The invocation phase is represented by function ipc_send_invoc_err and its result *res*, respectively. If any invocation error occurred, predicate is_error holds and *res* is returned. The pre-transmission phase is represented by the predicates ipc_send_rendez_vous and ipc_send_buffer_ovfl. If the former holds, it depends on the latter whether the transmission starts or fails. A buffer overflow aborts the operation with the error message ERR_SND_BUFFER_OVFL whereas SUCCESS initiates an immediate transmission. The timeout to_{snd} comes into play, if no rendez-vous is detected. An immediate timeout, i. e., $to_{snd} = 0$, results in the error ERR_SND_TIMEOUT, whereas $res = \varepsilon_\Sigma$ indicates the will to wait.

The overall specification is given by function vamos_ipc_send:

```
vamos_ipc_send $s_V$ $hn_{rcv}$ $rights_{snd}$ $msg$ $hn_{add}$ $rights_{add}$ $to_{snd}$ $\equiv$
  let $uprocs = s_V$.procs; $schedds = s_V$.schedds; $priodb = s_V$.priodb;
      $sndstatdb = s_V$.sndstatdb; $rightsdb = s_V$.rightsdb;
      $devds = s_V$.devds;
      $res$ =
        ipc_send_err $uprocs$ $schedds$ $sndstatdb$ $rightsdb$ $hn_{rcv}$ $rights_{snd}$
         $msg$ $hn_{add}$ $rights_{add}$ $to_{snd}$;
      $p_{snd} = \lceil$v_cup $schedds\rceil$; $p_{rcv} = \lceil\lceil rightsdb\ p_{snd}\rceil$.hdb $hn_{rcv}\rceil$
  in if is_error $res$ then $s_V(\!|$procs := $uprocs(p_{snd} \mapsto \delta_{proc}\ res\ \lceil uprocs\ p_{snd}\rceil)|\!)$
     else if $res = \varepsilon_\Sigma$
          then $s_V(\!|$schedds := v_ipc_wait_updt_schedds $schedds$ $p_{snd}$
                                  $\lceil priodb\ p_{snd}\rceil$ $to_{snd}|\!)$
          else $s_V(\!|$procs := v_ipc_trans_updt_procs $uprocs$ $rightsdb$ $devds$
                             $p_{snd}$ False $hn_{rcv}$ $rights_{snd}$ $msg$ $hn_{add}$ $rights_{add}$,
                    schedds := v_wkp_updt_schedds $schedds$ $[p_{rcv}]$ $priodb$,
                    sndstatdb := v_ipc_trans_updt_sndstatdb $sndstatdb$ $p_{snd}$ $p_{rcv}$
                                 False,
                    rightsdb := v_ipc_trans_updt_rightsdb $rightsdb$ $p_{snd}$ $hn_{rcv}$
                                $hn_{add}$ $rights_{snd}$ $rights_{add}$,
                    devds := v_ipc_trans_updt_devds $devds$ $p_{rcv}|\!)$
```

The process number p_{snd} of the sender is given by function v_cup. Accessing p_{snd}'s handle database with hn_{rcv} delivers the process number p_{rcv} of the receiver and res holds the result of function $\mathsf{ipc\_send\_err}$.

The latter forms the basis for the following case distinctions. As usual, if res denotes an error, VAMOS delivers the according error message to the calling process, i. e., p_{snd}. If the return value of function $\mathsf{ipc\_send\_err}$ denotes ε_{Σ}, the sender is put into the wait state. Accordingly, function $\mathsf{v\_ipc\_wait\_updt\_schedds}$ describes the changes on the scheduling datastructures. The remaining case covers the transmission which is described by the dedicated functions.

IPC Receive

In order to invoke the VAMOS call IPC_RECEIVE, the (expected) receiver p_{rcv} has to specify (a) a handle hn_{snd}, (b) a buffer buf_{rcv}, and (c) the timeout to_{rcv}.

In Section 2.5.1, we already mentioned that the VAMOS call IPC_RECEIVE can be operated in different modes. Thereby, handle hn_{snd} plays a special role as it choses between the different operating modes. The first mode reflects the ordinary communication between processes. As invoking process, the receiver specifies the desired sender by means of the process handle hn_{snd} and commits itself to this particular process as only possible sender. Apart from that, IPC_RECEIVE can also be operated as an open-receive which is made for servers that rely on the possibility to receive requests from various clients. This option is enabled by setting hn_{snd} to $\mathsf{HN\_NONE}$ and allows to be simultaneously available for more than one sender. The remaining operating mode is tailor-made for device drivers which take a great interest in receiving external interrupts as fast as possible. In order to meet this desire, the call IPC_RECEIVE provides the possibility for a closed-receive from the kernel. This option is selected by setting hn_{snd} to $\mathsf{HN\_KERNEL}$ and restricts the receiving to notifications from the VAMOS kernel only. In doing so, the VAMOS kernel is qualified to deliver according notifications as soon as interrupts for the driver occur. A more detailed view on that provides Section 4.6.

The specified arguments are checked for their validity in the invocation phase. The detection of errors leads to the abortion of the call and p_{rcv} gets an according error message. Otherwise, the following pre-transmission phase looks for an appropriate sender. The pre-transmission phase largely depends on the operating mode of IPC_RECEIVE. Especially in the case of an open-receive, the determination of a sender could affect various processes. As a consequence, the updates of the particular state components get more involved which is why they are encapsulated in dedicated functions.

The outline of the remainder of this section looks as follows. We first introduce the checks regarding invocation errors. After that, we describe the determination of a sender and associated effects which we formally encapsulate into functions. Finally, we combine everything in order to get the overall specification of the call IPC_RECEIVE.

Invocation. Due to the small number of arguments, checking for invocation errors is quite straightforward. Formally, function ipc_rcv_invoc_err describes the error checks:

```
$\mathsf{ipc\_rcv\_invoc\_err}\ \mathit{rightsdb}\ p_{\mathsf{rcv}}\ hn_{\mathsf{snd}}\ \mathit{buffer} \equiv$
  **if** $hn_{\mathsf{snd}} = \mathsf{HN\_SELF}\ \lor$
     $hn_{\mathsf{snd}} \notin \{\mathsf{HN\_NONE}, \mathsf{HN\_KERNEL}\} \land \neg\ \mathsf{valid\_handle}\ \mathit{rightsdb}\ p_{\mathsf{rcv}}\ hn_{\mathsf{snd}}$
  **then** ERR_RCV_INVALID_HANDLE
  **else if** $\mathit{buffer} = \mathsf{BufUndefined}$ **then** ERR_INVALID_ARGS
        **else if** $\mathit{buffer} = \mathsf{BufUnavailable}$ **then** ERR_RCV_SEGV **else** SUCCESS
```

The receiver p_{rcv} is not allowed to receive from itself, and, as long as it does neither perform an open-receive nor a closed-receive from the kernel, the specified handle hn_{snd} has to be valid. Otherwise, error code ERR_RCV_INVALID_HANDLE is returned. Furthermore, the buffer has to be defined and available. Violating the former condition leads to the error message ERR_INVALID_ARGS while an unavailable buffer leads to ERR_RCV_SEGV. Every output from ipc_rcv_invoc_err not equal to SUCCESS leads to an abortion of the operation accompanied by an error notification to the originating process. The output SUCCESS, however, heralds the start of the pre-transmission phase.

Pre-Transmission. The simplest case depicts a closed-receive from the kernel. Except for p_{rcv}, no other process is involved and the VAMOS kernel solely checks whether there are any notifications for p_{rcv}. If so, VAMOS delivers an according message to p_{rcv} and thereby completes the call. Otherwise, either p_{rcv} waits or the call is aborted due to a timeout error.

Determining the communication partner, in case of a closed-receive from another process, is similar as with IPC_SEND. With the validity of handle hn_{snd}, the desired sender p_{snd} can be identified by accessing the handle database of p_{rcv} with hn_{snd}. By means of predicate v_is_sending_to, VAMOS further checks whether p_{snd} is willing to send to p_{rcv}:

```
$\mathsf{v\_is\_sending\_to}\ \mathit{uprocs}\ \mathit{sndstatdb}\ \mathit{rightsdb}\ p_{\mathsf{snd}}\ p_{\mathsf{rcv}} \equiv$
  **case** $\omega_{\mathsf{proc}}\ \lceil \mathit{uprocs}\ p_{\mathsf{snd}} \rceil$ **of**
  IPC_SEND $hn_{\mathsf{rcv}}\ \mathit{rights}_{\mathsf{snd}}\ \mathit{msg}\ hn_{\mathsf{add}}\ \mathit{rights}_{\mathsf{add}}\ \mathit{to}_{\mathsf{snd}} \Rightarrow$
    $\lceil \mathit{rightsdb}\ p_{\mathsf{snd}} \rceil.\mathsf{hdb}\ hn_{\mathsf{rcv}} = \lfloor p_{\mathsf{rcv}} \rfloor$
  | IPC_REQUEST $hn_{\mathsf{rcv}}\ \mathit{rights}_{\mathsf{snd}}\ \mathit{msg}\ hn_{\mathsf{add}}\ \mathit{rights}_{\mathsf{add}}\ \mathit{to}_{\mathsf{snd}}\ \mathit{buffer}\ \mathit{to}_{\mathsf{rcv}} \Rightarrow$
     $\lceil \mathit{rightsdb}\ p_{\mathsf{snd}} \rceil.\mathsf{hdb}\ hn_{\mathsf{rcv}} = \lfloor p_{\mathsf{rcv}} \rfloor \land \mathit{sndstatdb}\ p_{\mathsf{snd}} = \lfloor \mathsf{False} \rfloor$
  | _ $\Rightarrow \mathsf{False}$
```

Basically, that is the case, if the output denotes IPC_SEND or IPC_REQUEST and the handle hn_{snd} points to p_{rcv}. While a combined send and receive, however, the send phase may not yet be completed, i. e., the send status has to be inactive. A transmission comes about, if the message $\mathit{msg}_{\mathsf{snd}}$ of p_{snd}

fits into the receive buffer buf_{rcv}. Otherwise, p_{rcv} is put into the wait state or the call is aborted with a timeout message to p_{rcv}.

Things get more elaborate, if p_{rcv} performs an open-receive. Thereby, the communication is no longer restricted to the kernel or one particular process, but various processes come into consideration as potential sender. Namely, all those which are waiting for a send operation with p_{rcv} as receiver. These processes are subsumed in a so-called send queue sq_{rcv}, which is specific to p_{rcv}. The send queue sq_{rcv} results from the wait queue and is generated by function v_gen_sq:

```
v_gen_sq $uprocs\ schedds\ sndstatdb\ rightsdb\ p_{\mathsf{rcv}} \equiv$
  $[p{\in}schedds.$wait . v_is_sending_to $uprocs\ sndstatdb\ rightsdb\ p\ p_{\mathsf{rcv}}]$
```

The resulting queue contains all waiting processes p that fulfill the predicate v_is_sending_to, where p_{rcv} is assigned as receiver. Note that sq_{rcv} preserves the chronological order of the wait queue and thus enables the request handling on the basis of first-come, first-serve. Following the chronological order, the head p_{hd} of sq_{rcv} ought to be the actual sender. However, VAMOS first checks whether p_{hd}'s message fits into buf_{rcv}. If not, p_{hd} gets the error message ERR_SND_BUFFER_OVFL and VAMOS continues traversing sq_{rcv}, unless it finds an appropriate sender (with compatible message size) or no process remains.

Against this background, we define function ipc_rcv_split_sq, which takes all contingencies regarding the determination of a sender into account:

```
ipc_rcv_split_sq $uprocs\ rightsdb_{\mathsf{rcv}}\ sq_{\mathsf{rcv}}\ hn_{\mathsf{snd}}\ buf_{\mathsf{rcv}} \equiv$
  let $sq_{\mathsf{eff}}$ =
        if $hn_{\mathsf{snd}}$ = HN_KERNEL then []
        else if $hn_{\mathsf{snd}}$ = HN_NONE then $sq_{\mathsf{rcv}}$
            else filter (op = $\lceil rightsdb_{\mathsf{rcv}}.$hdb $hn_{\mathsf{snd}}\rceil$) $sq_{\mathsf{rcv}}$;
      $buf\_ovfl = \lambda p.$ bufLength $buf_{\mathsf{rcv}}$ < get_msglen ($\omega_{\mathsf{proc}}\ \lceil uprocs\ p\rceil$)
  in (takeWhile $buf\_ovfl\ sq_{\mathsf{eff}}$, dropWhile $buf\_ovfl\ sq_{\mathsf{eff}}$)
```

Based on the queue sq_{rcv} with the processes willing to send to p_{rcv}, the function first computes an effective send queue sq_{eff}. This queue is empty, if p_{rcv} performs a closed-receive from the kernel, i. e., hn_{snd} = HN_KERNEL, and equivalent to sq_{rcv} while an open-receive, i. e., hn_{snd} = HN_NONE. In case of a closed-receive, the process number of the desired sender is obtained by accessing p_{rcv}'s handle database with hn_{snd}. If this process number is contained in sq_{rcv}, it describes the only element of sq_{eff}. Otherwise, sq_{eff} is empty.

With the effective send queue sq_{eff} at hand, VAMOS continues with checking the compatibility of the send messages and the receive buffer buf_{rcv}. As a consequence, sq_{eff} is split into two queues. The former is the longest prefix of sq_{eff} with oversized messages, the latter is the remaining queue after stripping the prefix. In doing so, bufLength determines the size of buf_{rcv} and get_msglen the ones of the send messages.

In case that the remaining queue is not empty, its head determines the actual sender for the transmission. Otherwise, no transmission takes place but p_{rcv} could at least receive kernel notifications. Whether notifications are delivered decides predicate knotify which is also used in the context of a closed-receive from the kernel:

```
knotify $rightsdb\ devds\ hn_{\mathsf{snd}}\ p_{\mathsf{rcv}} \equiv$
 $hn_{\mathsf{snd}} \in \{\mathsf{HN\_NONE}, \mathsf{HN\_KERNEL}\} \wedge$
 $\neg\ (\lceil rightsdb\ p_{\mathsf{rcv}}\rceil.\mathsf{stolen} = \{\} \wedge$
     $\{i \in devds.\mathsf{saved}.\ devds.\mathsf{driver}\ i = \lfloor p_{\mathsf{rcv}}\rfloor\} = \{\})$
```

In general, a receiver p_{rcv} is ready to receive kernel notifications while performing an open-receive or a closed-receive from the kernel. The VAMOS kernel, however, only delivers notifications, if there is a reason, i. e., if there are invalid handles in p_{rcv}'s handle database or pending interrupts with p_{rcv} as registered driver. Regarding the definition of knotify this means, that VAMOS only sends a notification, if either the set of stolen handle of p_{rcv} is not empty or set saved contains interrupts handled by p_{rcv}.

Updating the User Processes. The update of the user processes is fairly elaborate and comprises several case distinctions. Common to all cases is, that processes causing buffer overflows are provided with an according error message. Apart from that, VAMOS distinguishes two main cases. The first one describes the situation that no suitable sender was found. Based on this, VAMOS determines whether it is possible that notification can be delivered to p_{rcv}. If so, VAMOS assembles an according message and sends it to p_{rcv}. In case that no notifications can be delivered, the timeout to_{rcv} decides whether p_{rcv} receives a timeout error. The effects of a transmission are described by function v_ipc_trans_updt_procs.

Formally, function v_ipc_rcv_updt_procs combines all these case distinctions:

```
v_ipc_rcv_updt_procs $uprocs\ rightsdb\ devds\ sq_{\mathsf{rcv}}\ p_{\mathsf{rcv}}\ hn_{\mathsf{snd}}\ buffer$
 $to_{\mathsf{rcv}} \equiv$
 let $(sendQ_{\mathsf{bovfl}}, sendQ_{\mathsf{snd}}) =$
       ipc_rcv_split_sq $uprocs\ \lceil rightsdb\ p_{\mathsf{rcv}}\rceil\ sq_{\mathsf{rcv}}\ hn_{\mathsf{snd}}\ buffer$;
     $uprocs_{\mathsf{bovfl}} =$
       $\lambda p.$ if $p \in$ set $sendQ_{\mathsf{bovfl}}$
            then $\lfloor \delta_{\mathsf{proc}}\ \mathrm{ERR\_SND\_BUFFER\_OVFL}\ \lceil uprocs\ p\rceil\rfloor$ else $uprocs\ p$;
     $irqs = \{i \in devds.\mathsf{saved}.\ devds.\mathsf{driver}\ i = \lfloor p_{\mathsf{rcv}}\rfloor\}$;
     $stolen = \lceil rightsdb\ p_{\mathsf{rcv}}\rceil.\mathsf{stolen} \neq \{\}$;
     $knot =$
      SUCC_RECEIVE HN_KERNEL False {} [] HN_NONE False {} $stolen$ False
       $irqs$
 in case $sendQ_{\mathsf{snd}}$ of
    [] ⇒ if knotify $rightsdb\ devds\ hn_{\mathsf{snd}}\ p_{\mathsf{rcv}}$
         then $uprocs_{\mathsf{bovfl}}(p_{\mathsf{rcv}} \mapsto \delta_{\mathsf{proc}}\ knot\ \lceil uprocs\ p_{\mathsf{rcv}}\rceil)$
```

```
        else if $to_{\mathsf{rcv}} = 0$
            then $\mathit{uprocs}_{\mathsf{bovfl}}(p_{\mathsf{rcv}} \mapsto \delta_{\mathsf{proc}}$ ERR_RCV_TIMEOUT $\lceil uprocs\ p_{\mathsf{rcv}}\rceil)$
            else $\mathit{uprocs}_{\mathsf{bovfl}}$
  | $p_{\mathsf{snd}} \cdot ps \Rightarrow$
     let $req$ = is_send_receive $(\omega_{\mathsf{proc}}\ \lceil uprocs\ p_{\mathsf{snd}}\rceil)$;
         $(hn_{\mathsf{rcv}}, rights_{\mathsf{snd}}, msg, hn_{\mathsf{add}}, rights_{\mathsf{add}}) =$
           get_sender_args $(\omega_{\mathsf{proc}}\ \lceil uprocs\ p_{\mathsf{snd}}\rceil)$
     in v_ipc_trans_updt_procs $\mathit{uprocs}_{\mathsf{bovfl}}$ $rightsdb$ $devds$ $p_{\mathsf{snd}}$ $req$
         $hn_{\mathsf{rcv}}$ $rights_{\mathsf{snd}}$ $msg$ $hn_{\mathsf{add}}$ $rights_{\mathsf{add}}$
```

Based on the send queue sq_{rcv}, function ipc_rcv_split_sq is used to subsume the processes causing buffer overflows in $\mathit{sendQ}_{\mathsf{bovfl}}$ whereas $\mathit{sendQ}_{\mathsf{snd}}$ denotes the remainder of sq_{rcv}. The intermediate update $\mathit{uprocs}_{\mathsf{bovfl}}$ of the virtual machines describes the delivery of message ERR_SND_BUFFER_OVFL to all processes in $\mathit{sendQ}_{\mathsf{bovfl}}$. Relying on this update, we proceed with the remaining case distinctions.

An empty queue $\mathit{sendQ}_{\mathsf{snd}}$ denotes the fact that no appropriate sender is ready. However, if predicate knotify applies, it is at least possible to deliver any notifications to p_{rcv}. The according message is generated by function deliverNotifications and abbreviated by *knot*. If the delivery of notifcations is not possible, p_{rcv} either receives the error message ERR_RCV_TIMEOUT or the virtual machines $\mathit{uprocs}_{\mathsf{bovfl}}$ remain untouched.

The sender p_{snd}, in case of a transmission, is given by the head of $\mathit{sendQ}_{\mathsf{snd}}$. Function get_sender_args delivers the according arguments and the predicate is_send_receive determines, whether the sender p_{snd} performs an IPC_REQUEST call.

Based on these values, the function v_ipc_trans_updt_procs performs the relevant updates.

Updating the Rights Datastructures. Taking the preceding considerations into account, the update of the rigths datastructures is straightforward. The function v_ipc_trans_updt_rightsdb delivers the update, if a transition takes place. Otherwise, the datastructures stay untouched:

```
v_ipc_rcv_updt_rightsdb $rightsdb$ $uprocs$ $sendQ$ $p_{\mathsf{rcv}}$ $hn_{\mathsf{snd}}$ $buffer \equiv$
  case snd (ipc_rcv_split_sq $uprocs$ $\lceil rightsdb\ p_{\mathsf{rcv}}\rceil$ $sendQ$ $hn_{\mathsf{snd}}$ $buffer$) of
  [] $\Rightarrow rightsdb$
  | $p \cdot ps \Rightarrow$
     let $(hn_{\mathsf{rcv}}, rights_{\mathsf{snd}}, msg, hn_{\mathsf{add}}, rights_{\mathsf{add}}) =$
           get_sender_args $(\omega_{\mathsf{proc}}\ \lceil uprocs\ p\rceil)$
     in v_ipc_trans_updt_rightsdb $rightsdb$ $p$ $hn_{\mathsf{rcv}}$ $hn_{\mathsf{add}}$ $rights_{\mathsf{snd}}$ $rights_{\mathsf{add}}$
```

A transmission only takes place, if the the second component of the tuple returned by ipc_rcv_split_sq is not empty. The head is chosen as sender and function get_sender_args determines its specified arguments and passes them on to the transmission function.

Updating the Scheduling Datastructures. The update of the scheduling datastructures follows the same structure as the one for the virtual machines. First of all, processes causing buffer overflows are waken up. Apart from that, VAMOS again distinguishes whether a transmission takes place or not. If not, p_{rcv} is put into the wait state, as long as no kernel notifications can be delivered and no timeout occurrs. After a transmission initiated by an IPC_RECEIVE, the sender p_{snd} is usually put back to the ready state. However, this is only true, if p_{snd} does not perform an IPC_REQUEST. Otherwise, it is forced to wait again for a response from p_{rcv} and only its timeout is recomputed.

Formally, VAMOS uses function v_ipc_rcv_updt_schedds to describe the relevant updates:

```
$\mathsf{v\_ipc\_rcv\_updt\_schedds}\ schedds\ uprocs\ rightsdb\ priodb\ sndstatusdb$
 $devds\ sendQ\ p_{\mathsf{rcv}}\ hn_{\mathsf{snd}}\ buffer\ to_{\mathsf{rcv}} \equiv$
  $\mathbf{let}\ (sendQ_{\mathsf{bovfl}}, sendQ_{\mathsf{snd}}) =$
        $\mathsf{ipc\_rcv\_split\_sq}\ uprocs\ \lceil rightsdb\ p_{\mathsf{rcv}}\rceil\ sendQ\ hn_{\mathsf{snd}}\ buffer;$
      $schedds_{\mathsf{bovfl}} = \mathsf{v\_wkp\_updt\_schedds}\ schedds\ sendQ_{\mathsf{bovfl}}\ priodb$
  $\mathbf{in\ case}\ sendQ_{\mathsf{snd}}\ \mathbf{of}$
     $[\,] \Rightarrow \mathbf{if}\ \neg\ \mathsf{knotify}\ rightsdb\ devds\ hn_{\mathsf{snd}}\ p_{\mathsf{rcv}} \wedge to_{\mathsf{rcv}} \neq 0$
          $\mathbf{then}\ \mathsf{v\_ipc\_wait\_updt\_schedds}\ schedds_{\mathsf{bovfl}}\ p_{\mathsf{rcv}}\ \lceil priodb\ p_{\mathsf{rcv}}\rceil$
                $to_{\mathsf{rcv}}$
          $\mathbf{else}\ schedds_{\mathsf{bovfl}}$
     $|\ p_{\mathsf{snd}} \cdot ps \Rightarrow$
        $\mathbf{if}\ \mathsf{is\_send\_receive}\ (\omega_{\mathsf{proc}}\ \lceil uprocs\ p_{\mathsf{snd}}\rceil)$
        $\mathbf{then}\ schedds_{\mathsf{bovfl}}$
              $(|\mathsf{procdb} := schedds_{\mathsf{bovfl}}.\mathsf{procdb}(p_{\mathsf{snd}} \mapsto$
                 $\lceil schedds_{\mathsf{bovfl}}.\mathsf{procdb}\ p_{\mathsf{snd}}\rceil$
                 $(|\mathsf{timeout} := \mathsf{compute\_new\_timeout}\ uprocs\ schedds_{\mathsf{bovfl}}\ p_{\mathsf{snd}}|))|)$
        $\mathbf{else}\ \mathsf{v\_wkp\_updt\_schedds}\ schedds_{\mathsf{bovfl}}\ [p_{\mathsf{snd}}]\ priodb$
```

Again, the split send queue $(sendQ_{\mathsf{bovfl}}, sendQ_{\mathsf{snd}})$ is the linchpin. The awakening of the processes in $sendQ_{\mathsf{bovfl}}$ results in the intermediate update $schedds_{\mathsf{bovfl}}$ which lays the foundation for the remaining steps.

If no sender is involved, i. e., $sendQ_{\mathsf{snd}}$ is empty, the receiver either waits or gets a timeout error. The former happens, if the timeout value to_{rcv} does not intend an immediate timeout and no delivery of kernel notifications is possible. Accordingly, the pre-defined function v_ipc_wait_updt_schedds describes the semantic effects. Otherwise, the scheduling datastructures $schedds_{\mathsf{bovfl}}$ stay unchanged.

A transmission takes place, if $sendQ_{\mathsf{snd}}$ is not empty. As long as p_{snd} does not perform an IPC_REQUEST, it is put back into the ready state which is described by function v_wkp_updt_schedds. If the sending was part of an IPC_REQUEST, p_{snd} switches over into the receive phase, whereas function compute_new_timeout determines the new timeout value:

```
compute_new_timeout $uprocs\ schedds\ p \equiv$
  case $\omega_{\mathsf{proc}}\ \lceil uprocs\ p\rceil$ of
  IPC_REQUEST $hn_{\mathsf{rcv}}\ rights_{\mathsf{snd}}\ msg\ hn_{\mathsf{add}}\ rights_{\mathsf{add}}\ to_{\mathsf{snd}}\ buffer\ to_{\mathsf{rcv}} \Rightarrow$
    $to_{\mathsf{rcv}}$ += $schedds$.time
  | _ $\Rightarrow \lceil schedds$.procdb $p\rceil$.timeout
```

The specified receive timeout to_{rcv} is added to the current time.

Updating the Send Status Database. The send status database is only updated, if the receiver's send queue sq_{rcv} contains a suitable sender. In this case, the pre-defined transmission function v_ipc_trans_updt_sndstatdb handles the update. Otherwise, the send status database remains unchanged. Function v_ipc_rcv_updt_sndstatdb encapsulates the semantic effects:

```
v_ipc_rcv_updt_sndstatdb $sndstatdb\ uprocs\ rightsdb\ sendQ\ p_{\mathsf{rcv}}\ hn_{\mathsf{snd}}$
 $buffer \equiv$
  case snd (ipc_rcv_split_sq $uprocs\ \lceil rightsdb\ p_{\mathsf{rcv}}\rceil\ sendQ\ hn_{\mathsf{snd}}$
               $buffer$) of
  [] $\Rightarrow sndstatdb$
  | $p \cdot ps \Rightarrow$
     v_ipc_trans_updt_sndstatdb $sndstatdb\ p\ p_{\mathsf{rcv}}$
      (is_send_receive ($\omega_{\mathsf{proc}}\ \lceil uprocs\ p\rceil$))
```

Updating the Device Datastructures. A transmission as well as kernel notifications involve the delivery of interrupts to the receiver. Thus, in both situations, an update of the device datastructures is to be expected. Function v_ipc_rcv_updt_devds formally defines this update while relying on function v_ipc_trans_updt_devds:

```
v_ipc_rcv_updt_devds $devds\ rightsdb\ uprocs\ sendQ\ p_{\mathsf{rcv}}\ hn_{\mathsf{snd}}\ buffer \equiv$
  if snd (ipc_rcv_split_sq $uprocs\ \lceil rightsdb\ p_{\mathsf{rcv}}\rceil\ sendQ\ hn_{\mathsf{snd}}\ buffer) \neq [\,] \vee$
     knotify $rightsdb\ devds\ hn_{\mathsf{snd}}\ p_{\mathsf{rcv}}$
  then v_ipc_trans_updt_devds $devds\ p_{\mathsf{rcv}}$ else $devds$
```

Overall Specification. The abstract function vamos_ipc_receive puts everything together and describes the overall specification of the Vamos call IPC_RECEIVE:

```
vamos_ipc_receive $s_{\mathsf{V}}\ hn_{\mathsf{snd}}\ buffer\ to_{\mathsf{rcv}} \equiv$
  let $procs = s_{\mathsf{V}}$.procs; $schedds = s_{\mathsf{V}}$.schedds; $priodb = s_{\mathsf{V}}$.priodb;
      $sndstatdb = s_{\mathsf{V}}$.sndstatdb; $rightsdb = s_{\mathsf{V}}$.rightsdb;
      $devds = s_{\mathsf{V}}$.devds; $p_{\mathsf{rcv}} = \lceil$v_cup $schedds\rceil$;
      $res$ = ipc_rcv_invoc_err $rightsdb\ p_{\mathsf{rcv}}\ hn_{\mathsf{snd}}\ buffer$;
      $sq_{\mathsf{rcv}}$ = v_gen_sq $procs\ schedds\ sndstatdb\ rightsdb\ p_{\mathsf{rcv}}$
  in if is_error $res$ then $s_{\mathsf{V}}(\!|$procs := $procs(p_{\mathsf{rcv}} \mapsto \delta_{\mathsf{proc}}\ res\ \lceil procs\ p_{\mathsf{rcv}}\rceil)|\!)$
     else $s_{\mathsf{V}}(\!|$procs := v_ipc_rcv_updt_procs $procs\ rightsdb\ devds\ sq_{\mathsf{rcv}}\ p_{\mathsf{rcv}}$
                         $hn_{\mathsf{snd}}\ buffer\ to_{\mathsf{rcv}}$,
               schedds := v_ipc_rcv_updt_schedds $schedds\ procs\ rightsdb$
```

```
                    $\mathit{priodb}\ \mathit{sndstatdb}\ \mathit{devds}\ \mathit{sq}_{\mathsf{rcv}}\ p_{\mathsf{rcv}}\ hn_{\mathsf{snd}}\ \mathit{buffer}$
                    $to_{\mathsf{rcv}}$,
    sndstatdb := v_ipc_rcv_updt_sndstatdb $\mathit{sndstatdb}\ \mathit{procs}\ \mathit{rightsdb}$
                    $\mathit{sq}_{\mathsf{rcv}}\ p_{\mathsf{rcv}}\ hn_{\mathsf{snd}}\ \mathit{buffer}$,
    rightsdb := v_ipc_rcv_updt_rightsdb $\mathit{rightsdb}\ \mathit{procs}\ \mathit{sq}_{\mathsf{rcv}}\ p_{\mathsf{rcv}}$
                    $hn_{\mathsf{snd}}\ \mathit{buffer}$,
    devds := v_ipc_rcv_updt_devds $\mathit{devds}\ \mathit{rightsdb}\ \mathit{procs}\ \mathit{sq}_{\mathsf{rcv}}$
                    $p_{\mathsf{rcv}}\ hn_{\mathsf{snd}}\ \mathit{buffer}$|)
```

IPC Request

The semantics of the IPC_REQUEST call is pretty similar to the one of IPC_SEND. They only differ in two points: the invocation errors and the update of the scheduling datastructures in case of a transmission.

Due to the additional call arguments the check for invocation errors has to be extended, as the definition of function ipc_sr_invoc_err illustrates:

```
ipc_sr_invoc_err $\mathit{rightsdb}\ p_{\mathsf{snd}}\ hn_{\mathsf{rcv}}\ \mathit{rights}_{\mathsf{snd}}\ \mathit{msg}\ hn_{\mathsf{add}}\ \mathit{rights}_{\mathsf{add}}\ \mathit{buffer}$
 $to_{\mathsf{rcv}} \equiv$
  if $hn_{\mathsf{rcv}} = \mathsf{HN\_SELF} \lor \neg$ valid_handle $\mathit{rightsdb}\ p_{\mathsf{snd}}\ hn_{\mathsf{rcv}}$
  then ERR_SND_INVALID_HANDLE
  else if $hn_{\mathsf{add}} \neq \mathsf{HN\_NONE} \land \neg$ valid_handle $\mathit{rightsdb}\ p_{\mathsf{snd}}\ hn_{\mathsf{add}}$
      then ERR_INVALID_HANDLE
      else if $\neg$ valid_sr_args $\mathit{rights}_{\mathsf{snd}}\ \mathit{rights}_{\mathsf{add}}\ \mathit{msg}\ \mathit{buffer}\ hn_{\mathsf{add}}\ to_{\mathsf{rcv}}$
          then ERR_INVALID_ARGS
          else if $\mathit{msg}$ = MObjUnavailable then ERR_SND_SEGV
              else if $\mathit{buffer}$ = BufUnavailable then ERR_RCV_SEGV
                  else if $\neg$ allowed_sr_op $\mathit{rightsdb}\ p_{\mathsf{snd}}\ hn_{\mathsf{rcv}}\ \mathit{rights}_{\mathsf{add}}$
                          $to_{\mathsf{rcv}}$
                      then ERR_UNPRIVILEGED else $\varepsilon_{\Sigma}$
```

Regarding the handles hn_{rcv} and hn_{add} the same conditions apply as within IPC_SEND. The predicate valid_sr_args is based on the former predicate valid_snd_args and additionally checks for a defined buffer and a finite receive timeout:

```
valid_sr_args $\mathit{rights}_{\mathsf{snd}}\ \mathit{rights}_{\mathsf{add}}\ \mathit{msg}\ \mathit{buffer}\ hn_{\mathsf{add}}\ to_{\mathsf{rcv}} \equiv$
  valid_snd_args $\mathit{rights}_{\mathsf{snd}}\ \mathit{rights}_{\mathsf{add}}\ hn_{\mathsf{add}}\ \mathit{msg} \land$
  $\mathit{buffer} \neq \mathsf{BufUndefined} \land to_{\mathsf{rcv}} \neq 0$
```

Besides an available message, an IPC_REQUEST call also requires an available buffer. The sender is only allowed to perform the combined send and receive operation, if predicate allowed_sr_op holds:

```
allowed_sr_op $\mathit{rightsdb}\ p_{\mathsf{snd}}\ hn_{\mathsf{rcv}}\ \mathit{rights}_{\mathsf{add}}\ to_{\mathsf{rcv}} \equiv$
  let $p_{\mathsf{rcv}} = \lceil \mathit{rightsdb}\ p_{\mathsf{snd}} \rceil.\mathsf{hdb}\ hn_{\mathsf{rcv}}$; $\mathit{rights}_{\mathsf{curr}} = \lceil \lceil \mathit{rightsdb}\ p_{\mathsf{snd}} \rceil.\mathsf{rdb}\ p_{\mathsf{rcv}} \rceil$
  in $\mathit{rights}_{\mathsf{curr}} \cap \{\mathsf{v\_sendR}, \mathsf{v\_requestR}\} \neq \{\} \land$
      (privileged $\mathit{rightsdb}\ p_{\mathsf{snd}} \lor \mathit{rights}_{\mathsf{add}} = \lfloor\{\}\rfloor) \land$
      $(to_{\mathsf{rcv}} = \infty \lor \mathit{rights}_{\mathsf{curr}} \cap \{\mathsf{v\_sendR}, \mathsf{v\_finiteR}\} \neq \{\})$
```

Three prerequisites have to be fulfilled by the sender: (a) it needs the send or the request right to the receiver, (b) it needs privileges to assign a non-empty set $rights_{\mathsf{add}}$ of rights to the additional process, and (c) the timeout value to_{rcv} is either infinite or it has the permission to send with a finite timeout. The latter is fulfilled, if one of the rights v_sendR or v_finiteR is owned. In an analagous way as with ipc_send_err, we use ipc_sr_invoc_err to define the predicate ipc_sr_err.

The second difference occurs while updating the scheduling datastructures in case of a transmission. In contrast to IPC_SEND, a combined send and receive operation is not completed after the send phase. The sender rather switches its role and is forced to wait for a response. This interferes with the update of the scheduling datastructures, which is performed by means of function v_ipc_wkp_wait_updt_schedds. The receiver is waken up and the sender is forced to wait. Furthermore, its new timeout value is set according to to_{rcv}.

Taking all the differences into account, the formal specification looks as follows:

```
vamos_ipc_send_receive $s_V$ $hn_{rcv}$ $rights_{snd}$ $msg$ $hn_{add}$ $rights_{add}$ $to_{snd}$
 $buffer$ $to_{rcv}$ ≡
  let $procs$ = $s_V$.procs; $schedds$ = $s_V$.schedds; $priodb$ = $s_V$.priodb;
      $sndstatdb$ = $s_V$.sndstatdb; $rightsdb$ = $s_V$.rightsdb;
      $devds$ = $s_V$.devds; $p_{snd}$ = ⌈v_cup $schedds$⌉;
      $p_{rcv}$ = ⌈⌈$rightsdb$ $p_{snd}$⌉.hdb $hn_{rcv}$⌉;
      $res$ =
        ipc_sr_err $procs$ $schedds$ $sndstatdb$ $rightsdb$ $hn_{rcv}$ $rights_{snd}$
         $msg$ $hn_{add}$ $rights_{add}$ $to_{snd}$ $buffer$ $to_{rcv}$
  in if is_error $res$ then $s_V$(|procs := $procs$($p_{snd}$ ↦ $\delta_{proc}$ $res$ ⌈$procs$ $p_{snd}$⌉)|)
     else if $res$ = $\varepsilon_\Sigma$
          then $s_V$(|schedds := v_ipc_wait_updt_schedds $schedds$ $p_{snd}$
                                  ⌈$priodb$ $p_{snd}$⌉ $to_{snd}$|)
          else $s_V$(|procs := v_ipc_trans_updt_procs $procs$ $rightsdb$ $devds$ $p_{snd}$
                             True $hn_{rcv}$ $rights_{snd}$ $msg$ $hn_{add}$ $rights_{add}$,
                  schedds := v_ipc_wkp_wait_updt_schedds $schedds$ $p_{rcv}$ $p_{snd}$
                              $priodb$ $to_{rcv}$,
                  sndstatdb := v_ipc_trans_updt_sndstatdb $sndstatdb$ $p_{snd}$ $p_{rcv}$
                                True,
                  rightsdb := v_ipc_trans_updt_rightsdb $rightsdb$ $p_{snd}$ $hn_{rcv}$
                              $hn_{add}$ $rights_{snd}$ $rights_{add}$,
                  devds := v_ipc_trans_updt_devds $devds$ $p_{rcv}$|)
```

Changing IPC Rights

We have just seen, that IPC enables the sender to provide the receiver of the IPC message with additional rights. In this context, VAMOS synchronously updates the handle and rights databases and informs the receiver on these changes by means of the result message. However, IPC only allows to grant rights but not to revoke them. In addition to that, it should be possible to reset entries in the handle and rights databases. This is necessary in order to clean up the databases, on the one hand, and to invalidate handles, on the other one. All this functionality is encapsulated in the VAMOS call CHANGE_RIGHTS. For this to work, a calling process p_{cp} has to specify the following parameters: (a) the handle hn_{subj} to the subject whose databases should be changed, (b) the handle hn_{obj} to the object to which the new rights should apply, (c) the flag *grant* determining whether the rights are granted or revoked, and (d) the description $rights_{\mathsf{obj}}$ how to change the rights.

As always, user-specified parameters might be erroneous and therefore have to pass the validity checks. The latter are encapsulated in function vamos_result_change_rights:

```
vamos_result_change_rights $rightsdb\ p_{\mathsf{cp}}\ rights_{\mathsf{obj}}\ hn_{\mathsf{subj}}\ hn_{\mathsf{obj}} \equiv$
  if $rights_{\mathsf{obj}} = \bot$ then ERR_INVALID_ARGS
  else if $\neg$ valid_handle $rightsdb\ p_{\mathsf{cp}}\ hn_{\mathsf{subj}}$
       then ERR_INVALID_SUBJ_HANDLE
       else if $\neg$ valid_handle $rightsdb\ p_{\mathsf{cp}}\ hn_{\mathsf{obj}}$
            then ERR_INVALID_OBJ_HANDLE
            else if $\neg$ legal_op $rightsdb\ p_{\mathsf{cp}}\ hn_{\mathsf{subj}}\ hn_{\mathsf{obj}}\ rights_{\mathsf{obj}}$
                 then ERR_UNPRIVILEGED
                 else if $\neg$ known_obj $rightsdb\ p_{\mathsf{cp}}\ hn_{\mathsf{subj}}\ hn_{\mathsf{obj}}$
                      then ERR_INVALID_HANDLE else SUCCESS
```

First of all, the specified rights $rights_{\mathsf{obj}}$ must follow the given rights format. Otherwise, $rights_{\mathsf{obj}} = \bot$ and error ERR_INVALID_ARGS is triggered. Quite apart from the fact that the handles hn_{subj} and hn_{obj} have to be valid in the context of p_{cp}, the operation has to be legal and the object needs to be known by the subject. The former is checked by predicate legal_op:

```
legal_op $rightsdb\ p_{\mathsf{cp}}\ hn_{\mathsf{subj}}\ hn_{\mathsf{obj}}\ rights \equiv$
  privileged $rightsdb\ p_{\mathsf{cp}} \vee$
  $hn_{\mathsf{subj}} = \mathsf{HN\_SELF} \wedge rights = \lfloor\{\}\rfloor \vee hn_{\mathsf{obj}} = \mathsf{HN\_SELF}$
```

Privileged processes are allowed to do any changes on the rights datastructures. Without privileges, p_{cp} is only allowed to either update its own databases or entries in others, as long as they refer to itself. The calling process p_{cp} expresses the will to clean up its own handle database by setting hn_{subj} to HN_SELF. As this operation does not involve the granting or revoking of rights, it is additionally required that $rights = \lfloor\{\}\rfloor$. In case that $hn_{\mathsf{subj}} \neq \mathsf{HN\_SELF}$, the handle to the object must denote HN_SELF. This ensures that only entries refering to p_{cp} are changed.

In order to change the entries in the subject's rights datastructures, the object even needs to be known by the subject. Predicate known_obj formally encapsulates this prerequisite:

```
known_obj $rightsdb\ p_{\mathsf{cp}}\ hn_{\mathsf{subj}}\ hn_{\mathsf{obj}} \equiv$
  $hn_{\mathsf{subj}} \neq hn_{\mathsf{obj}} \wedge$
```

```
(let $p_{\mathsf{subj}} = \lceil\lceil rightsdb\ p_{\mathsf{cp}}\rceil.\mathsf{hdb}\ hn_{\mathsf{subj}}\rceil$
 in $\exists hn.\ \lceil rightsdb\ p_{\mathsf{subj}}\rceil.\mathsf{hdb}\ hn = \lceil rightsdb\ p_{\mathsf{cp}}\rceil.\mathsf{hdb}\ hn_{\mathsf{obj}})$
```

The notion of *known* processes is mainly used in the context of IPC. As a process might not communicate with itself, we regard a process as unknown to itself. Moreover, it does not make any sense that a process changes the rights to itself. The first conjunction $hn_{\mathsf{subj}} \neq hn_{\mathsf{obj}}$ formalizes this. The second conjunction ensures, that there exists a handle in the subject's handle database pointing to the object. Note that p_{subj} is well-defined because the validity of hn_{subj} was previously checked. The same applies for the handle database access with hn_{obj}.

Based on the result of vamos_result_change_rights, the overall specification function vamos_change_rights describes the effects on a VAMOS state:

```
vamos_change_rights $s_V$ $hn_{subj}$ $hn_{obj}$ $grant$ $rights$ $\equiv$
  let $p_{cp} = \lceil$v_cup $s_V$.schedds$\rceil$;
      $waiting_{subj}$ =
        $\lceil\lceil s_V$.rightsdb $p_{cp}\rceil$.hdb $hn_{subj}\rceil$ mem $s_V$.schedds.wait;
      $res$ =
        vamos_result_change_rights $s_V$.rightsdb $p_{cp}$ $rights$ $hn_{subj}$ $hn_{obj}$
  in if is_error $res$ then $s_V$(|procs := $s_V$.procs($p_{cp} \mapsto \delta_{proc}$ $res$ $\lceil s_V$.procs $p_{cp}\rceil$)|)
     else $s_V$(|procs := v_chg_rights_updt_procs $s_V$.procs $s_V$.rightsdb
                       $waiting_{subj}$ $p_{cp}$ $hn_{subj}$ $hn_{obj}$ $grant$ $rights$,
               schedds := v_chg_rights_updt_schedds $s_V$.schedds $s_V$.procs
                          $s_V$.priodb $s_V$.rightsdb $p_{cp}$ $hn_{subj}$ $hn_{obj}$ $grant$
                          $rights$,
               rightsdb := v_chg_rights_updt_rightsdb $s_V$.rightsdb $s_V$.procs $p_{cp}$
                           $hn_{subj}$ $hn_{obj}$ $grant$ $rights$,
               sndstatdb := v_chg_rights_updt_sndstatdb $s_V$.sndstatdb $s_V$.procs
                            $s_V$.rightsdb $waiting_{subj}$ $p_{cp}$ $hn_{subj}$ $hn_{obj}$ $grant$
                            $rights$|)
```

The error case only involves the error message passing to the calling process p_{cp}. As the manipulation of rights might abort pending IPC operations of the subject, the according state updates are not only restricted to the rights datastructures but also involve the virtual machines and the send status databases.

Function res_if_pending_ipc has the decision on the abortion of an IPC operation:

```
res_if_pending_ipc $uprocs$ $rightsdb$ $p_{subj}$ $p_{obj}$ $grant$ $rights$ $\equiv$
  let $hdb_{subj} = \lceil rightsdb\ p_{subj}\rceil$.hdb; $rdb_{subj} = \lceil rightsdb\ p_{subj}\rceil$.rdb
  in case $\omega_{proc}$ $\lceil uprocs\ p_{subj}\rceil$ of
    IPC_SEND $hn_{rcv}$ $rights_{snd}$ $msg$ $hn_{add}$ $rights_{add}$ $to_{snd}$ $\Rightarrow$
      if $hdb_{subj}$ $hn_{rcv} = \lfloor p_{obj}\rfloor \wedge rights = \{\}$ then ERR_SND_INVALID_HANDLE
      else if $hdb_{subj}$ $hn_{add} = \lfloor p_{obj}\rfloor \wedge rights = \{\}$ then ERR_INVALID_HANDLE
           else if $\neg$ $grant$ $\wedge$
                   $hdb_{subj}$ $hn_{rcv} = \lfloor p_{obj}\rfloor \wedge$
```

```
                $\mathsf{v\_sendR} \notin \lceil \mathit{rdb}_{\mathsf{subj}}\ \lfloor p_{\mathsf{obj}} \rfloor \rceil - \mathit{rights}$
            then ERR_UNPRIVILEGED else $\varepsilon_\Sigma$
  | IPC_RECEIVE $\mathit{hn}_{\mathsf{snd}}$ $\mathit{buffer}$ $\mathit{to}_{\mathsf{rcv}}$ $\Rightarrow$
      if $\mathit{hdb}_{\mathsf{subj}}\ \mathit{hn}_{\mathsf{snd}} = \lfloor p_{\mathsf{obj}} \rfloor \wedge \mathit{rights} = \{\}$
      then ERR_RCV_INVALID_HANDLE
      else if $\mathit{hn}_{\mathsf{snd}} \in \{\mathsf{HN\_NONE}, \mathsf{HN\_KERNEL}\} \wedge$
              $\mathit{rights} = \{\} \wedge p_{\mathsf{subj}} \neq p_{\mathsf{obj}} \wedge \mathsf{is\_known}\ \mathit{rightsdb}\ p_{\mathsf{subj}}\ p_{\mathsf{obj}}$
          then SUCC_RECEIVE HN_KERNEL False {} [] HN_NONE False {} True
              False {}
          else $\varepsilon_\Sigma$
  | IPC_REQUEST $\mathit{hn}_{\mathsf{rcv}}$ $\mathit{rights}_{\mathsf{snd}}$ $\mathit{msg}$ $\mathit{hn}_{\mathsf{add}}$ $\mathit{rights}_{\mathsf{add}}$ $\mathit{to}_{\mathsf{snd}}$ $\mathit{buffer}$ $\mathit{to}_{\mathsf{rcv}}$ $\Rightarrow$
      if $\mathit{hdb}_{\mathsf{subj}}\ \mathit{hn}_{\mathsf{rcv}} = \lfloor p_{\mathsf{obj}} \rfloor \wedge \mathit{rights} = \{\}$
      then ERR_SND_INVALID_HANDLE
      else if $\mathit{hdb}_{\mathsf{subj}}\ \mathit{hn}_{\mathsf{add}} = \lfloor p_{\mathsf{obj}} \rfloor \wedge \mathit{rights} = \{\}$
          then ERR_INVALID_HANDLE
          else if $\neg\ \mathit{grant} \wedge$
                  $\mathit{hdb}_{\mathsf{subj}}\ \mathit{hn}_{\mathsf{rcv}} = \lfloor p_{\mathsf{obj}} \rfloor \wedge$
                  $\mathsf{revoked\_req\_rights}\ \mathit{hdb}_{\mathsf{subj}}\ \mathit{rdb}_{\mathsf{subj}}\ \mathit{rights}\ p_{\mathsf{obj}}\ \mathit{hn}_{\mathsf{rcv}}$
                   $\mathit{to}_{\mathsf{rcv}}$
              then ERR_UNPRIVILEGED else $\varepsilon_\Sigma$
  | _ $\Rightarrow \varepsilon_\Sigma$
```

The definition assumes that p_{subj} denotes the process number of the subject and p_{obj} the one of the object. Furthermore, it abbreviates the subject's handle database with $\mathit{hdb}_{\mathsf{subj}}$ and the rights database with $\mathit{rdb}_{\mathsf{subj}}$. Relying on that, it inspects the ouput of p_{subj}. The function returns ε_Σ, as long as either the output does not denote an IPC operation or the intended IPC operation is not affected by the rights manipulation. Otherwise, it returns the corresponding error code.

An IPC_SEND operation is aborted, if the handles $\mathit{hn}_{\mathsf{rcv}}$ or $\mathit{hn}_{\mathsf{add}}$ become invalid. The invalidation of the handles to p_{obj} is indicated by $\mathit{rights} = \{\}$. Accordingly, handle $\mathit{hn}_{\mathsf{rcv}}$ becomes invalid, if it points to p_{obj}. The resulting error message is ERR_SND_INVALID_HANDLE. Similarily, result ERR_INVALID_HANDLE is returned, if the handle $\mathit{hn}_{\mathsf{add}}$ points to p_{obj}. Besides the handle invalidation, the revocation of rights may also influence the send operation. It is implied by an inactive flag *grant* and aborts the operation, if p_{obj} is assigned as receiver and p_{subj} loses the v_sendR right to p_{obj}. Due to this, p_{subj} is no longer allowed to send to p_{obj} and obtains the error message ERR_UNPRIVILEGED. In a similar way, VAMOS proceeds, if p_{subj} performs an IPC_REQUEST. In this case, however, things are more elaborate regarding the revocation of rights. VAMOS considers p_{subj} as unprivileged, if predicate revoked_req_rights holds:

```
revoked_req_rights $\mathit{hdb}_{\mathsf{subj}}\ \mathit{rdb}_{\mathsf{subj}}\ \mathit{rights}\ p_{\mathsf{obj}}\ \mathit{hn}_{\mathsf{rcv}}\ \mathit{to}_{\mathsf{rcv}} \equiv$
  if $\mathit{hdb}_{\mathsf{subj}}\ \mathit{hn}_{\mathsf{rcv}} = \lfloor p_{\mathsf{obj}} \rfloor \wedge$
     $((\lceil \mathit{rdb}_{\mathsf{subj}}\ \lfloor p_{\mathsf{obj}} \rfloor \rceil - \mathit{rights}) \cap \{\mathsf{v\_sendR}, \mathsf{v\_requestR}\} = \{\} \vee$
      $(\lceil \mathit{rdb}_{\mathsf{subj}}\ \lfloor p_{\mathsf{obj}} \rfloor \rceil - \mathit{rights}) \cap \{\mathsf{v\_sendR}, \mathsf{v\_finiteR}\} = \{\} \wedge \mathit{to}_{\mathsf{rcv}} \neq \infty)$
  then True else False
```

First of all, the involved receiver must again correspond to p_{obj}. The process p_{subj} does no longer hold sufficient rights for the operation, if after the revocation (a) it neither owns the v_sendR nor the v_requestR right, or (b) it neither owns the v_sendR nor the v_finiteR right and the specified timeout for the receive part is not infinite.

An IPC_RECEIVE operation of p_{subj} is only affected by a process invalidation. The obvious case covers the correspondence of the invalidated process p_{obj} with the intended sender. Error message ERR_RCV_INVALID_HANDLE denotes this. However, IPC_RECEIVE could also be terminated with a kernel notification. Usually, a process invalidation comes along with moving the corresponding handle into the stolen set. As seen before, a non-empty stolen set triggers a kernel notification, if the receiver performs either an open-receive or a closed-receive from the kernel. However, the invalidation of p_{obj} affects p_{subj} only, if p_{obj} was previously known by p_{subj}. Furthermore, a notification is only passed on to p_{subj}, if it is different from p_{obj} because a process cannot be invalidated in its own context.

With these preliminary remarks at hand, we proceed with the particular updates of the state components.

Updating the User Processes The update of the user processes passes a SUCCESS message on to the calling process p_{cp}. In addition, if p_{subj} was waiting in a pending IPC operation that has been aborted, it gets the corresponding error notification.

The function v_chg_rights_updt_procs delivers the formal specification:

```
v_chg_rights_updt_procs $uprocs\ rightsdb\ waiting_{\mathsf{subj}}\ p_{\mathsf{cp}}\ hn_{\mathsf{subj}}\ hn_{\mathsf{obj}}\ grant$
 $rights \equiv$
  let $p_{\mathsf{subj}} = \lceil\lceil rightsdb\ p_{\mathsf{cp}}\rceil.\mathsf{hdb}\ hn_{\mathsf{subj}}\rceil$; $p_{\mathsf{obj}} = \lceil\lceil rightsdb\ p_{\mathsf{cp}}\rceil.\mathsf{hdb}\ hn_{\mathsf{obj}}\rceil$;
      $res_{\mathsf{ipc}} = \mathsf{res\_if\_pending\_ipc}\ uprocs\ rightsdb\ p_{\mathsf{subj}}\ p_{\mathsf{obj}}\ grant\ \lceil rights\rceil$
  in $\lambda x.$ if $x = p_{\mathsf{cp}}$ then $\lfloor\delta_{\mathsf{proc}}\ \text{SUCCESS}\ \lceil uprocs\ x\rceil\rfloor$
          else if $x = p_{\mathsf{subj}} \wedge res_{\mathsf{ipc}} \neq \varepsilon_{\Sigma} \wedge waiting_{\mathsf{subj}}$
                then $\lfloor\delta_{\mathsf{proc}}\ res_{\mathsf{ipc}}\ \lceil uprocs\ x\rceil\rfloor$ else $uprocs\ x$
```

The flag $waiting_{\mathsf{subj}}$ indicates whether p_{subj} is in the wait queue.

Updating the Scheduling Datastructures The scheduling datastructures are only affected by an aborted IPC operation. In this case, p_{subj} is waken up. The pre-defined function v_wkp_updt_schedds describes the semantic effect.

Function v_chg_rights_updt_schedds encapsulates the overall update:

```
v_chg_rights_updt_schedds $schedds\ uprocs\ priodb\ rightsdb\ p_{\mathsf{cp}}\ hn_{\mathsf{subj}}\ hn_{\mathsf{obj}}\ grant$
 $rights \equiv$
  let $p_{\mathsf{subj}} = \lceil\lceil rightsdb\ p_{\mathsf{cp}}\rceil.\mathsf{hdb}\ hn_{\mathsf{subj}}\rceil$; $p_{\mathsf{obj}} = \lceil\lceil rightsdb\ p_{\mathsf{cp}}\rceil.\mathsf{hdb}\ hn_{\mathsf{obj}}\rceil$;
      $res_{\mathsf{ipc}} = \mathsf{res\_if\_pending\_ipc}\ uprocs\ rightsdb\ p_{\mathsf{subj}}\ p_{\mathsf{obj}}\ grant\ \lceil rights\rceil$;
      $waiting_{\mathsf{subj}} = p_{\mathsf{subj}}\ \mathsf{mem}\ schedds.\mathsf{wait}$
```

```
**in if** $\mathit{res}_{\mathsf{ipc}} \neq \varepsilon_\Sigma \wedge \mathit{waiting}_{\mathsf{subj}}$ **then** $\mathsf{v\_wkp\_updt\_schedds}\ schedds\ [p_{\mathsf{subj}}]\ priodb$
    **else** $schedds$
```

Updating the Rights Datastructures The update of the rights datastructures distinguishes three cases: (a) the process invalidation or deletion, (b) the rights granting, and (c) the rights revocation.

The function v_chg_rights_updt_rightsdb encapsulates the corresponding updates:

```
$\mathsf{v\_chg\_rights\_updt\_rightsdb}\ rightsdb\ uprocs\ p_{\mathsf{cp}}\ hn_{\mathsf{subj}}\ hn_{\mathsf{obj}}\ grant\ rights \equiv$
  let $p_{\mathsf{subj}} = \lceil\lceil rightsdb\ p_{\mathsf{cp}}\rceil.\mathsf{hdb}\ hn_{\mathsf{subj}}\rceil;\ p_{\mathsf{obj}} = \lceil\lceil rightsdb\ p_{\mathsf{cp}}\rceil.\mathsf{hdb}\ hn_{\mathsf{obj}}\rceil;$
      $hdb_{\mathsf{subj}} = \lceil rightsdb\ p_{\mathsf{subj}}\rceil.\mathsf{hdb};\ rdb_{\mathsf{subj}} = \lceil rightsdb\ p_{\mathsf{subj}}\rceil.\mathsf{rdb};$
      $stolen_{\mathsf{subj}} = \lceil rightsdb\ p_{\mathsf{subj}}\rceil.\mathsf{stolen}$
  in $rightsdb(p_{\mathsf{subj}} \mapsto$
     if $\lceil rights\rceil = \{\}$
     then $\lceil rightsdb\ p_{\mathsf{subj}}\rceil$
            $(\!|\mathsf{hdb} := \lambda hn.$ if $hdb_{\mathsf{subj}}\ hn = \lfloor p_{\mathsf{obj}}\rfloor$ then $\bot$ else $hdb_{\mathsf{subj}}\ hn,$
              $\mathsf{rdb} := rdb_{\mathsf{subj}}(\lfloor p_{\mathsf{obj}}\rfloor := \bot),$
              $\mathsf{stolen} := stolen_{\mathsf{subj}} \cup \{hn.\ hn_{\mathsf{subj}} \neq \mathsf{HN\_SELF} \wedge hdb_{\mathsf{subj}}\ hn = \lfloor p_{\mathsf{obj}}\rfloor\}|\!)$
     else if $grant$
          then $\lceil rightsdb\ p_{\mathsf{subj}}\rceil$
                 $(\!|\mathsf{rdb} := rdb_{\mathsf{subj}}(\lfloor p_{\mathsf{obj}}\rfloor \mapsto \lceil rdb_{\mathsf{subj}}\ \lfloor p_{\mathsf{obj}}\rfloor\rceil \cup \lceil rights\rceil)|\!)$
          else $\lceil rightsdb\ p_{\mathsf{subj}}\rceil$
                 $(\!|\mathsf{rdb} := rdb_{\mathsf{subj}}(\lfloor p_{\mathsf{obj}}\rfloor \mapsto \lceil rdb_{\mathsf{subj}}\ \lfloor p_{\mathsf{obj}}\rfloor\rceil - \lceil rights\rceil)|\!))$
```

Accesses to the handle database of the calling process p_{cp} with the handles hn_{subj} and hn_{obj} deliver the process numbers of the subject p_{subj} and the object p_{obj}. The subject's handle and rights databases are abbreviated with hdb_{subj} and rdb_{subj}. For the set of stolen handles we use $stolen_{\mathsf{subj}}$.

Object p_{obj} is invalidated resp. deleted in the context of p_{subj}, if the parameter *rights* represents an empty set. The indices of handles in hdb_{subj} pointing to p_{obj} are set to $\bot$, as well as the entry of p_{obj} in the rights database rdb_{subj}. Additionally, the handles pointing to p_{obj} are inserted into the stolen set $stolen_{\mathsf{subj}}$, as long as it does not conform with HN_SELF. Note that due to the uniqueness of handles only one handle in hdb_{subj} points to p_{obj}.

Granting and revoking of rights is straightforward. The specified rights *rights* are either added to or subtracted from the former rights.

Updating the Send Status Database Updating the send status database is straightforward. If an IPC operation of p_{subj} is aborted, the entry of p_{subj} is set to False. Otherwise, it remains untouched:

```
$\mathsf{v\_chg\_rights\_updt\_sndstatdb}\ sndstatdb\ uprocs\ rightsdb\ waiting_{\mathsf{subj}}\ p_{\mathsf{cp}}\ hn_{\mathsf{subj}}$
 $hn_{\mathsf{obj}}\ grant\ rights \equiv$
  let $p_{\mathsf{subj}} = \lceil\lceil rightsdb\ p_{\mathsf{cp}}\rceil.\mathsf{hdb}\ hn_{\mathsf{subj}}\rceil;\ p_{\mathsf{obj}} = \lceil\lceil rightsdb\ p_{\mathsf{cp}}\rceil.\mathsf{hdb}\ hn_{\mathsf{obj}}\rceil;$
      $res_{\mathsf{ipc}} = \mathsf{res\_if\_pending\_ipc}\ uprocs\ rightsdb\ p_{\mathsf{subj}}\ p_{\mathsf{obj}}\ grant\ \lceil rights\rceil$
  in if $res_{\mathsf{ipc}} \neq \varepsilon_\Sigma \wedge waiting_{\mathsf{subj}}$ then $sndstatdb(p_{\mathsf{subj}} \mapsto \mathsf{False})$ else $sndstatdb$
```

4.4.7 Read Kernel Information

The reading of kernel information is provided by the call READ_KERNEL_INFO.

When the kernel detects a condition or event, like a stale handle or an external interrupt, it sends a notification with the next receive operation. However, the number of the notification bits is very limited and there might be much more useful information. Once a user process was notified about a certain notification type, it can ask for more information by READ_KERNEL_INFO. Currently, there is only one notification type supported: The stale-handle notification. However, we can use this mechanism as well, if we would like to carry more information on external interrupts, for instance, the keypress timestamps for the T-Systems project [47].

Function vamos_read_kernel_info specifies the semantic effects of reading kernel information:

```
$\mathsf{vamos\_read\_kernel\_info}\ s_V\ note \equiv$
  $\mathbf{let}\ p_{cp} = \lceil \mathsf{v\_cup}\ s_V.\mathsf{schedds} \rceil;\ stolen = \lceil s_V.\mathsf{rightsdb}\ p_{cp} \rceil.\mathsf{stolen};$
      $res = \mathbf{if}\ note = \bot\ \mathbf{then}\ \text{ERR\_INVALID\_ARGS}\ \mathbf{else}\ \text{SUCCESS}$
  $\mathbf{in}\ \mathbf{if}\ \mathsf{is\_error}\ res\ \mathbf{then}\ s_V(\!|\mathsf{procs} := s_V.\mathsf{procs}(p_{cp} \mapsto \delta_{proc}\ res\ \lceil s_V.\mathsf{procs}\ p_{cp} \rceil)|\!)$
      $\mathbf{else}\ s_V(\!|\mathsf{procs} := s_V.\mathsf{procs}(p_{cp} \mapsto$
                $\delta_{proc}\ (\text{SUCC\_KINFO\_STALEH}\ stolen)\ \lceil s_V.\mathsf{procs}\ p_{cp} \rceil),$
                $\mathsf{rightsdb} := s_V.\mathsf{rightsdb}(p_{cp} \mapsto \lceil s_V.\mathsf{rightsdb}\ p_{cp} \rceil(\!|\mathsf{stolen} := \{\}|\!))|\!)$
```

The calling process p_{cp} gets an ERR_INVALID_ARGS message, if the notification was not well-defined. Otherwise, p_{cp} is informed about all stolen handles in its handle database by means of a SUCC_KINFO_STALEH message. After receiving the message, p_{cp} knows about the stolen handles and could initiate further steps. In any case, the set of stolen handles is empty, afterwards.

4.5 Vamos Scheduler

On the abstract level, the VAMOS scheduler is part of the VAMOS transition function δ_V and copes with the handling of timer-interrupts. The function handleTimer, which delivers the specification of the VAMOS scheduler, accomplishes three steps:

```
$\mathsf{handleTimer}\ s_V\ p_{cup} \equiv$
  $\mathsf{charge}\ (\mathsf{checkTimeouts}\ (s_V(\!|\mathsf{schedds} := s_V.\mathsf{schedds}(\!|\mathsf{time} := s_V.\mathsf{schedds.time} + 1|\!)|\!)))$
   $p_{cup}$
```

In the first step, the current time time is incremented. Afterwards, the scheduler checks by means of function checkTimeouts for expired timeouts. Processes with expired timeouts are informed about this circumstance by means of an error message and set back to the ready state. Finally, the computing time of the current time is charged from its timeslice. This charging is formally described by the function charge.

Note that the current process p_{cup} is provided as a parameter because in the first phase of a VAMOS transition, the queues might change. Hence, we may not assume that v_cup s_{V}.schedds has been computing in the last step. Instead, the current process is stored in p_{cup} at kernel entry and provided as input to handleTimer.

Subsequently, we introduce the definitions of checkTimeouts and charge and, thus, complete the formal specification of the VAMOS scheduler.

Checking for Elapsed Timeouts. As shown in the definition of handleTimer, each timer-interrupt involves the incrementation of the current time. The task of the scheduler is to check whether the timeout of an arbitrary process p has thereby expired. If so, the scheduler sends a corresponding error message and puts p back into the ready state. Formally, this task is described by function checkTimeouts:

```
checkTimeouts $s_V \equiv s_V$
  (|procs := $\lambda p$. if expiredTimeout $s_V$.schedds $p$
                  then if vamosIsSending $s_V$.procs $s_V$.sndstatdb $p$
                        then $\lfloor \delta_{\mathsf{proc}}$ ERR_SND_TIMEOUT $\lceil s_V$.procs $p \rceil \rfloor$
                        else $\lfloor \delta_{\mathsf{proc}}$ ERR_RCV_TIMEOUT $\lceil s_V$.procs $p \rceil \rfloor$
                  else $s_V$.procs $p$,
   schedds := v_wkp_updt_schedds $s_V$.schedds
                  (filter (expiredTimeout $s_V$.schedds) $s_V$.schedds.wait)
                  $s_V$.priodb,
   sndstatdb := $\lambda p$. if expiredTimeout $s_V$.schedds $p$ then $\lfloor$False$\rfloor$
                        else $s_V$.sndstatdb $p$|)
```

A prominent role plays the predicate expiredTimeout. Relying on scheduling datastructures *schedds*, it figures out whether the timeout for an arbitrary process p has expired:

```
expiredTimeout schedds $p \equiv$
  $p \in$ set schedds.wait $\wedge$
  ($\exists$data.
    schedds.procdb $p = \lfloor data \rfloor \wedge (\exists t.\ data$.timeout = Fin $t \wedge t \leq$ schedds.time))
```

As the first conditions reflects, timeouts in VAMOS only apply to waiting processes p. Actually, the existence of the process-specific scheduling information *data* is implicitly given because p is not inactive. However, the predicate explicitly demands its existence and the last part of the conjunction establishes the basic message: Timeout t of p has expired, if t denotes a finite value Fin t and t is smaller than or equal to the current time denoted by *schedds*.time.

Relying on predicate expiredTimeout, the actual state updates appear as follows. All processes p with expired timeouts get an according error message. Thereby, VAMOS distinguishes whether a process p has been waiting in the send or the receive phase of an IPC operation. The former applies, if the previously defined predicate vamosIsSending is true for p. Setting a process p back into the ready state involves to dequeue it from the wait queue and to append it again on the ready queue of p's priortiy. All these actions only affect the scheduling datastructures and are encapsulated in the previously defined function v_wkp_updt_schedds. The list of processes that should be woken up comprises all processes out of s_V.schedds.wait which fulfill predicate expiredTimeout. In addition, the send status of these processes is reset to False.

Charging the Computing Time of the Current Process. The Vamos kernel concedes to each process a certain computing time of tsl clock ticks, whereas clock ticks are a measurment for occurred timer interrupts. Already consumed clock ticks are recorded in the component ctsl. Thus, if the current process p_{cup} is found to be computing when a timer interrupt raises, the scheduler increases ctsl of p_{cup}. This is repeated as long as process p_{cup} is either displaced by a higher-prioritized process or the value of ctsl has reached tsl. Latter involves a break of p_{cup}'s computation and a new process is scheduled. This means for p_{cup}, that its consumed time ctsl is reset and that p_{cup} is put to the end of its ready queue.

Function charge gives the formal specification:

```
charge $s_V$ $p_{\mathsf{cup}}$ $\equiv$
  let $\mathit{procdb}_{\mathsf{cup}} = s_V.\mathsf{schedds.procdb}\ \lceil p_{\mathsf{cup}}\rceil$;
      $\mathit{ready}_{\mathsf{cup}} = s_V.\mathsf{schedds.ready}\ \lceil s_V.\mathsf{priodb}\ \lceil p_{\mathsf{cup}}\rceil\rceil$
  in $s_V$(|schedds := if $p_{\mathsf{cup}} = \bot \vee \lceil p_{\mathsf{cup}}\rceil \notin \mathsf{set}\ \mathit{ready}_{\mathsf{cup}}$ then $s_V$.schedds
                       else if $\lceil \mathit{procdb}_{\mathsf{cup}}\rceil.\mathsf{tsl} \leq \lceil \mathit{procdb}_{\mathsf{cup}}\rceil.\mathsf{ctsl}$
                            then $s_V$.schedds
                                 (|ready := $s_V$.schedds.ready
                                    ($\lceil s_V.\mathsf{priodb}\ \lceil p_{\mathsf{cup}}\rceil\rceil$ :=
                                       $[p{\in}\mathit{ready}_{\mathsf{cup}}\ .\ p \neq \lceil p_{\mathsf{cup}}\rceil]$ @ $[\lceil p_{\mathsf{cup}}\rceil]$),
                                   procdb := $s_V$.schedds.procdb($\lceil p_{\mathsf{cup}}\rceil \mapsto \lceil \mathit{procdb}_{\mathsf{cup}}\rceil$
                                    (|ctsl := 0|))|)
                            else $s_V$.schedds
                                 (|procdb := $s_V$.schedds.procdb($\lceil p_{\mathsf{cup}}\rceil \mapsto \lceil \mathit{procdb}_{\mathsf{cup}}\rceil$
                                    (|ctsl := $\lceil \mathit{procdb}_{\mathsf{cup}}\rceil$.ctsl + 1|))|)|)
```

All in all, function charge distinguishes two situations: The first one reflects that either no current process p_{cup} existed at kernel entry or that p_{cup} has been rescheduled in the meantime. As Vamos accepts the rescheduling of p_{cup} as 'payment', both cases do not involve any changes on s_V.schedds.

The second situation reflects the actual charging of the current process p_{cup} as described above. In this context, the definition of charge abbreviates the process-specific scheduling datastructures of p_{cup} with $\mathit{procdb}_{\mathsf{cup}}$ and the ready queue of p_{cup}'s priority with $\mathit{ready}_{\mathsf{cup}}$. Two cases are considered.

The first case deals with a computing current process p_{cup} whose timeslice tsl is exhausted. The result is that the consumed time is reset to 0 and p_{cup} is appended to the end of $\mathit{ready}_{\mathsf{cup}}$. For this purpose, it is first completely removed and then appended at the end, again.

If the timeslice is not exhausted, p_{cup} is allowed to continue its computation but the consumed time ctsl is increased.

4.6 Vamos Interrupt Delivery

In the third phase of a VAMOS transition δ_{V}, the kernel delivers the interrupts that have occurred. The set *irqs* of interrupts that have been active at kernel entry are provided as input parameter to δ_{V}. The transition function, in turn, passes them on to function vamosInterruptDelivery which specifies the interrupt delivery:

```
$\mathsf{vamosInterruptDelivery}\ s_{\mathsf{V}}\ irqs \equiv$
  $\mathbf{let}\ \mathit{drivers}_{\mathsf{tbn}} =$
        $\mathsf{filter}\ (\mathsf{notifyHandler}\ s_{\mathsf{V}}.\mathsf{procs}\ s_{\mathsf{V}}.\mathsf{devds}\ irqs)\ s_{\mathsf{V}}.\mathsf{schedds.wait};$
      $\mathit{irq}_{\mathsf{pend}} = \{i \in irqs.\ \exists x.\ x \notin \mathsf{set}\ \mathit{drivers}_{\mathsf{tbn}} \wedge s_{\mathsf{V}}.\mathsf{devds.driver}\ i = \lfloor x \rfloor\}$
  $\mathbf{in}\ s_{\mathsf{V}}(\!|\mathsf{procs} := \lambda x.\ \mathbf{if}\ x \in \mathsf{set}\ \mathit{drivers}_{\mathsf{tbn}}$
                          $\mathbf{then}\ \lfloor \delta_{\mathsf{proc}}\ (\mathsf{deliverNotifications}\ s_{\mathsf{V}}.\mathsf{rightsdb}\ s_{\mathsf{V}}.\mathsf{devds}\ irqs\ x)$
                                  $\lceil s_{\mathsf{V}}.\mathsf{procs}\ x \rceil \rfloor$
                          $\mathbf{else}\ s_{\mathsf{V}}.\mathsf{procs}\ x,$
          $\mathsf{schedds} := \mathsf{v\_wkp\_updt\_schedds}\ s_{\mathsf{V}}.\mathsf{schedds}\ \mathit{drivers}_{\mathsf{tbn}}\ s_{\mathsf{V}}.\mathsf{priodb},$
          $\mathsf{devds} := s_{\mathsf{V}}.\mathsf{devds}$
            $(\!|\mathsf{enabled} := s_{\mathsf{V}}.\mathsf{devds.enabled} - irqs,\ \mathsf{saved} := s_{\mathsf{V}}.\mathsf{devds.saved} \cup \mathit{irq}_{\mathsf{pend}}|\!)|\!)$
```

Predicate notifyHandler is the basis of the definition. Relying on virtual machines *uprocs*, device datastructures *devds* and the set *irqs* of active interrupts, it determines whether an arbitrary process p is registered as handler for at least one interrupt $i \in irqs$ and, in addition, ready to accept possible interrupts for the handling:

```
$\mathsf{notifyHandler}\ uprocs\ devds\ irqs\ p \equiv$
  $\exists \mathit{proc}.$
    $uprocs\ p = \lfloor \mathit{proc} \rfloor \wedge$
    $(\exists \mathit{hn}_{\mathsf{snd}}\ \mathit{buffer}\ \mathit{to}_{\mathsf{rcv}}.$
      $\omega_{\mathsf{asm}}\ \mathit{proc} = \textsc{ipc\_receive}\ \mathit{hn}_{\mathsf{snd}}\ \mathit{buffer}\ \mathit{to}_{\mathsf{rcv}} \wedge$
      $\mathit{hn}_{\mathsf{snd}} \in \{\mathsf{HN\_NONE}, \mathsf{HN\_KERNEL}\} \wedge (\exists i \in irqs.\ devds.\mathsf{driver}\ i = \lfloor p \rfloor))$
```

Whether a process, in principle, is ready to receive any interrupts, depends on its current state. For this purpose, notifyHandler inspects the output of p. Therefore, the basic requirement is that process p is assigned with a virtual machine *proc*, i. e., p is not inactive. Applying the output function ω_{asm} to *proc* delivers p's current output. Regarding the reception of interrupts, this output must denote IPC_RECEIVE. Furthermore, it must specify an open receive or a closed receive to the kernel. Both cases are distinguished by handle hn_{snd}. Former is denoted by HN_NONE, latter by HN_KERNEL. Process p is only notified about interrupts, if it is a registered handler for at least one of the interrupts $i \in irqs$, i. e., $devds.\mathsf{driver}\ i = \lfloor p \rfloor$.

Based on notifyHandler the definition of vamosInterruptDelivery abbreviates the list of processes to be notified with $\mathit{handlers}_{\mathsf{tbn}}$. It contains all waiting processes p which fulfill predicate notifyHandler. In addition, $\mathit{irq}_{\mathsf{pend}}$ denotes the set of still pending interrupts, i. e., those interrupts whose handlers are not ready to receive.

According to that, the update of state s_{V} is performed as follows. Each notified process p receives a notification from the VAMOS kernel. This notification is generated by function deliverNotifications, which was already introduced in Section 4.4.3. After the interrupt delivery, the notified handlers are set back to the ready state. Corresponding effects describes the pre-defined function v_wkp_updt_schedds. Finally, the device datastructures are updated. As all interrupts *irqs* have been captured by the VAMOS kernel, they are disabled until their definite handling. Interrupts $\mathit{irq}_{\mathsf{pend}}$, that have not been delivered in this kernel run, remain pending and are added to the set saved.

Chapter 5

Vamos Correctness

Contents

In the last chapter, we introduced the abstract model of the VAMOS kernel. It constitutes a compact and easily accessible representation of the VAMOS functionality, without going into any implementation details. Thus, the VAMOS model can be taken as basis for higher-level models and further verification attempts. For instance, Bogan [15] relies on VAMOS while specifying the operating system in the academic system and Daum [25] showed the fairness of the VAMOS scheduler.

However, we certainly have to ensure that the VAMOS model is a true abstraction of the VAMOS implementation. For this reason, we aim at a formal simulation proof between implementation and specification by induction. As Figure 5.1 suggests, our induction start is based on the fact that the state σ_1 after the first system_step at a reset can be abstracted via an abstrac-

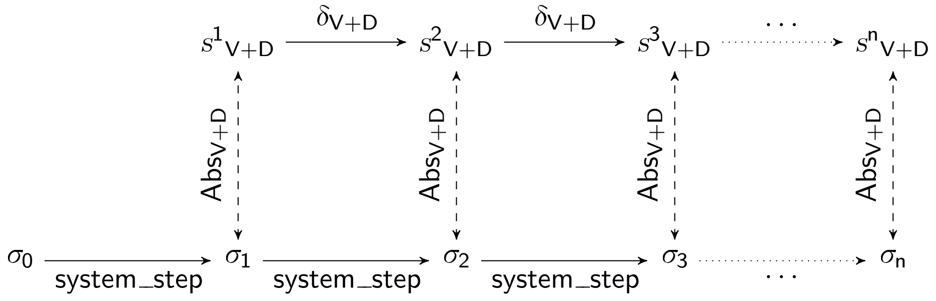
$$\begin{tikzcd}
 & s^1_{\mathsf{V+D}} \arrow[r, "\delta_{\mathsf{V+D}}"] & s^2_{\mathsf{V+D}} \arrow[r, "\delta_{\mathsf{V+D}}"] & s^3_{\mathsf{V+D}} \arrow[r, dotted, "\cdots"] & s^n_{\mathsf{V+D}} \\
\sigma_0 \arrow[r, "\mathsf{system\_step}"'] & \sigma_1 \arrow[u, dashed, leftrightarrow, "\mathsf{Abs_{V+D}}"] \arrow[r, "\mathsf{system\_step}"'] & \sigma_2 \arrow[u, dashed, leftrightarrow, "\mathsf{Abs_{V+D}}"] \arrow[r, "\mathsf{system\_step}"'] & \sigma_3 \arrow[u, dashed, leftrightarrow, "\mathsf{Abs_{V+D}}"] \arrow[r, dotted, "\cdots"'] & \sigma_n \arrow[u, dashed, leftrightarrow, "\mathsf{Abs_{V+D}}"]
\end{tikzcd}$$

Figure 5.1: Simulation between implementation and model

tion relation $\mathsf{Abs}_{\mathsf{V+D}}$ to an initial state ${s^1}_{\mathsf{V+D}}$ of the VAMOS model. The induction step formalizes that the semantic effects of the kernel execution can be expressed by the transition function $\delta_{\mathsf{V+D}}$ of the model and that the abstraction relation is preserved.

It turned out, that the induction step also requires various properties to hold during the kernel executions. We combine these properties in the so-called implementation invariant.

In the remainder of this chapter, we define the abstraction relation and state the implementation invariant. Based on these prerequisites, the chapter finally concludes with the formulation of the simulation theorem. The correctness of this theorem is based on the implementation correctness of all functions applied in the scope of system_step. We have not shown the implementation correctness of all functions. However, the functions which have not been proven so far are specified and completely integrated into the overall proof. The details on the already accomplished correctness proofs and the integration of the function specifications are postponed to the next chapter.

The formal definitions of the abstraction relation and the implementation invariant as well as the functional correctness proofs are given in the repository directory `llverification/vamos`.

5.1 Data Abstraction

Data abstraction, in general, enables a (for certain purposes) more manageable representation of datastructures. Take, for instance, the scheduling queues in the VAMOS model. Represented as queues of process numbers, they deliver the relevant information: which process is at which position in which queue. The fact that the concrete counterparts are implemented as doubly-linked lists is of no interest anymore and thus abstracted away. However, it must be ensured that there is a relation between the representation of the scheduling lists in the implementation and the queues in the model.

This is where the abstraction relations come into play connecting the datastructures of an implementation state σ with their abstract counterparts of a model state $s_{\mathsf{V+D}}$. In the following, we present the abstractions for the particular VAMOS state components.

While abstracting the datastructures, we have to take into account that the model identifies processes by means of abstract process numbers, whereas the implementation uses pointers. Function to_ptr defines a mapping from abstract process numbers to process pointers:

$\mathsf{to\_ptr}\ \sigma\ p \equiv {}^{\sigma}\mathsf{pib}\ !\ p$

By design, accessing pib with the process number p delivers the correrpsonding process pointer.

5.1.1 Abstracting the User Processes

The relation $\mathsf{rel\_procs_v}$ connects the virtual user-machines of the implementation with those of the model. Both, implementation and model, use the same granularity of virtual machines. However, some differences must be pointed out.

First, no virtual machine but $\bot$ is assigned to inactive processes. Second, the virtual machine states of waiting processes differ. We address this issue in more detail in Section 6.1. In a nutshell, a process only has to wait while performing an IPC operation with no suitable communication partner. An IPC operation, in turn, is triggered by a TRAP exception. In the implementation, this kind of exception arises while executing a TRAP instruction which happens within function $\mathsf{user_{step}}$ before calling the VAMOS kernel (cf. Section 2.4.4). Thus, the process first runs into the exception and then waits for its handling by the VAMOS kernel. The semantic effects of $\mathsf{user_{step}}$ in case of a TRAP instruction without runtime errors boils down to the incrementation of the program counters, as the following implication states:

$$[\![\mathsf{current\_instr}\ vm = \mathrm{TRAP}\ i;\ \mathsf{user\_step\_progress}\ vm]\!] \Longrightarrow \mathsf{user_{step}}\ vm = \mathsf{incPcs}\ vm$$

The VAMOS model, in contrast, handles exceptions in advance. It determines the corresponding process output by means of the function ω_{proc} which inspects the next instruction of the current process before the actual execution. Moreover, it executes the instruction not until the exception handling is completed. Accordingly, the virtual machines of waiting processes in the implementation are ahead of their abstract counterparts.

Incorporating these observations leads to the formal definition of the abstraction relation $\mathsf{rel\_procs_v}$:

```
$\mathsf{rel\_procs_v}\ \sigma\ inactQ\ waitQ\ uprocs \equiv$
  $\forall x.$ if $x \in \mathsf{set}\ inactQ$ then $uprocs\ x = \bot$
      else if $x \in \mathsf{set}\ waitQ$
          then $\mathsf{incPcs}\ \lceil uprocs\ x\rceil = (\mathsf{cvm\_ups}\ {}^{\sigma}\mathsf{cvmX}).\mathsf{userprocesses}\ x\ \wedge$
               $uprocs\ x \neq \bot$
          else $uprocs\ x = \lfloor(\mathsf{cvm\_ups}\ {}^{\sigma}\mathsf{cvmX}).\mathsf{userprocesses}\ x\rfloor$
```

The implementation state is given by σ, *inactQ* and *waitQ* denote the inactive and wait queue, and *uprocs* represents the virtual user machines in the model. Note that it would have been possible to derive the process states from the implementation state σ, as well. However, using the abstract queues is a bit more comfortable. Their abstraction is covered in Section 5.1.2.

For any process x residing in *inactQ*, the entry in *uprocs* is set to $\bot$. If x is waiting, the corresponding virtual machine *uprocs* x is the one that after the incrementation of the process counters corresponds to the machine $(\mathsf{cvm\_ups}\ \mathsf{cvmX}).\mathsf{userprocesses}\ x$ in the implementation. In all other cases, the virtual machine states are identical.

5.1.2 Abstracting the Scheduling Datastructures

The VAMOS state component schedds contains general as well as process-specific scheduling datastructures.

Abstracting the Scheduling Queues. A large part of the general data concerns the scheduling queues which are represented as doubly-linked lists in the implementation. The implementation identifies such a list with a pointer *head* to the first element and uses the next pointers *next* and previous pointers *prev* to traverse the list. In a first step, these lists are abstracted to HOL lists of references. Thereby, predicate Path *head next e Ps* tests whether the list *Ps* can be constructed starting with element *head* and collecting further elements by recursively applying function *next* until element *e* is reached. The formal definition of Path is:

$\mathsf{Path}\ x\ h\ y\ [] = (x = y)$
$\mathsf{Path}\ x\ h\ y\ (p \cdot ps) = (x = p \wedge x \neq \mathsf{Null} \wedge \mathsf{Path}\ (h\ x)\ h\ y\ ps)$

Doubly-linked lists are nothing but two singly-linked lists traversed in opposite directions. Accordingly, we use predicate dList for the abstraction, where *lst* denotes the last element of the list:

$\mathsf{dList}\ head\ next\ prev\ lst\ Ps \equiv$
$\quad \mathsf{Path}\ head\ next\ \mathsf{Null}\ Ps \wedge \mathsf{Path}\ lst\ prev\ \mathsf{Null}\ (\mathsf{rev}\ Ps)$

The model, however, uses abstract queues of process numbers. We base our abstraction of these process queues on predicate Queue:

$\mathsf{Queue}\ head\ next\ prev\ pid\ Ps \equiv$
$\quad \exists Rs\ lst.\ \mathsf{dList}\ head\ next\ prev\ lst\ Rs \wedge \mathsf{map}\ (\lambda x.\ pid\ x)\ Rs = Ps$

The first conjunction of Queue specifies a doubly-linked list *Rs* of references using predicate dList. The second conjunction states that the queue *Ps* of process numbers can be constructed from list *Rs* by applying the function *pid* to each element.

Using this predicate, we abstract the inactive list from the implementation variables inactive_list, queue_next, queue_prev, and pid. In an analogous way, we can abstract the wait and ready queues. The pointer to the wait list is given by wait_list, whereas those of the ready lists are contained in the array ready_lists.

Abstracting the Time and Process-specific Data. Abstracting the remaining parts – the time and the process-specific scheduling data – is more interesting, because, on the abstract level, we specify points in time as unbounded natural numbers, while the implementation uses unsigned 32-bit integers to represent those times.

Thus, the specification reflects the abstract idea of a monotonicly increasing time while the implementation employs the fact that only a finite range of time points is relevant at a certain execution step. Consequently, there is a growing offset between the implementation time and the abstract time. A natural number n defines the offset of the implementation variable current_time and the component time in the scheduling data structures of the model.

The same number n is used with function $\mathsf{rel\_procdb_v}$, which abstracts the process-specific scheduling data $procdb_\mathsf{p}$ of an active process p:

$$\begin{array}{l}
\mathsf{rel\_procdb_v}\ \sigma\ procdb_\mathsf{p}\ waitQ\ p\ n \equiv \\
\quad (p \in \mathsf{set}\ waitQ \longrightarrow \\
\quad\ \ procdb_\mathsf{p}.\mathsf{timeout} = \\
\quad\ \ (\textbf{if}\ {}^{\sigma}\mathsf{timeout}\ (\mathsf{to\_ptr}\ \sigma\ p) = \mathsf{INFINITE\_TIMEOUT}\ \textbf{then}\ \infty \\
\quad\ \ \ \ \textbf{else}\ \mathsf{Fin}\ ({}^{\sigma}\mathsf{timeout}\ (\mathsf{to\_ptr}\ \sigma\ p) + n))) \wedge \\
\quad procdb_\mathsf{p}.\mathsf{tsl} = {}^{\sigma}\mathsf{timeslice}\ (\mathsf{to\_ptr}\ \sigma\ p) \wedge \\
\quad procdb_\mathsf{p}.\mathsf{ctsl} = {}^{\sigma}\mathsf{consumed\_time}\ (\mathsf{to\_ptr}\ \sigma\ p)
\end{array}$$

If p is waiting, n describes the offset between the (absolute) timeouts. The implementation stores timeouts as 32-bit naturals, where the maximal value INFINITE_TIMEOUT represents an infinite timeout. Accordingly, if the heap function timeout delivers the maximal value for process p, the abstract timeout is set to ∞. Otherwise, the timeout is finite and denotes the value in the implementation incremented by n. The values for the timeslice tsl and the consumed time ctsl are directly taken from the implementation. The former is obtained by the heap function timeslice, the latter by consumed_time.

Summing Up. Based on these prerequisites, the definition of the abstraction relation $\mathsf{rel\_schedds_v}$ looks as follows:

$$\begin{array}{l}
\mathsf{rel\_schedds_v}\ \sigma\ schedds \equiv \\
\quad (\forall p.\ \mathsf{Queue}\ ({}^{\sigma}\mathsf{ready\_lists}\ !\ p)\ {}^{\sigma}\mathsf{queue\_next}\ {}^{\sigma}\mathsf{queue\_prev}\ {}^{\sigma}\mathsf{pid} \\
\qquad\qquad (schedds.\mathsf{ready}\ p)) \wedge \\
\quad \mathsf{Queue}\ {}^{\sigma}\mathsf{wakeup\_list}\ {}^{\sigma}\mathsf{queue\_next}\ {}^{\sigma}\mathsf{queue\_prev}\ {}^{\sigma}\mathsf{pid}\ schedds.\mathsf{wait} \wedge \\
\quad \mathsf{Queue}\ {}^{\sigma}\mathsf{inactive\_list}\ {}^{\sigma}\mathsf{queue\_next}\ {}^{\sigma}\mathsf{queue\_prev}\ {}^{\sigma}\mathsf{pid}\ schedds.\mathsf{inactive} \wedge \\
\quad (\exists n.\ schedds.\mathsf{time} = {}^{\sigma}\mathsf{current\_time} + n \wedge \\
\qquad (\forall p.\ \textbf{if}\ p \in \mathsf{set}\ schedds.\mathsf{inactive}\ \textbf{then}\ schedds.\mathsf{procdb}\ p = \bot \\
\qquad\qquad \textbf{else}\ schedds.\mathsf{procdb}\ p \neq \bot \wedge \\
\qquad\qquad\qquad \mathsf{rel\_procdb_v}\ \sigma\ \lceil schedds.\mathsf{procdb}\ p\rceil\ schedds.\mathsf{wait}\ p\ n))
\end{array}$$

5.1.3 Abstracting the Priorities

Both, the implementation and the model, distinguish three priority levels. The implementation maintains the process priorities in the heap function priority, whereas the model stores them in the state component priodb. The abstraction is established by $\mathsf{rel\_priodb_v}$:

$$\begin{array}{l}
\mathsf{rel\_priodb_v}\ \sigma\ priodb\ inactQ \equiv \\
\quad priodb = (\lambda p.\ \textbf{if}\ p \in \mathsf{set}\ inactQ\ \textbf{then}\ \bot\ \textbf{else}\ \lfloor {}^{\sigma}\mathsf{priority}\ (\mathsf{to\_ptr}\ \sigma\ p)\rfloor)
\end{array}$$

No priority but $\bot$ is assigned to inactive processes. For the remaining ones, we adopt the priority levels from the implementation.

5.1.4 Abstracting the Send Status Database

The VAMOS call IPC_REQUEST comprises a send as well as a receive operation. Process states in the implementation help to distinguish between both phases, whereas RECEIVE_STATE signals the successful completion of the send phase. The model abstracts from the particular process states and determines the current status on the basis of the queue memberships. As both phases of IPC_REQUEST might involve waiting, it is not possible to distinguish the situation in terms of the wait queue membership. For this purpose, the VAMOS model maintains a send status database sndstatdb. The abstraction is established by predicate $\mathsf{rel\_sndstatdb_v}$:

```
$\mathsf{rel\_sndstatdb_v}\ \sigma\ uprocs\ sndstatdb \equiv$
  $sndstatdb =$
  $(\lambda p.$ if $^{\sigma}\mathsf{state}\ (\mathsf{to\_ptr}\ \sigma\ p) = \mathsf{INACTIVE\_STATE}$ then $\bot$
       else if $\exists rcv\_hn\ snd\_rights\ msg\ add\_hn\ add\_rights\ snd\_timeout\ buffer$
                 $rcv\_timeout.$
                 $\omega_{\mathsf{proc}}\ \lceil uprocs\ p\rceil =$
                 $\mathrm{IPC\_REQUEST}\ rcv\_hn\ snd\_rights\ msg\ add\_hn\ add\_rights\ snd\_timeout$
                  $buffer\ rcv\_timeout\ \wedge$
                 $^{\sigma}\mathsf{state}\ (\mathsf{to\_ptr}\ \sigma\ p) = \mathsf{RECEIVE\_STATE}$
          then $\lfloor\mathsf{True}\rfloor$ else $\lfloor\mathsf{False}\rfloor)$
```

Inactive processes are neither in the send nor the receive phase and thus the send status denotes $\bot$. Apart from that, only processes p with an output denoting IPC_REQUEST come into consideration. The database entry for p is only True, if the corresponding process state is set to RECEIVE_STATE. Otherwise, it is False.

5.1.5 Abstracting the Rights Datastructures

Rights datastructures are process-specific and provide data concerning the privilege level, handles and IPC rights as well as the process parent. In the implementation, heap function privileged provides the privileged level of a process, parent denotes its parent and handle_db contains the handle databases. The latter combines information on the process-local handles and the assigned IPC rights.

Preliminaries. Subsequently, we use auxiliary functions to extract the information provided by the handle databases.

A handle hn is considered as known by a process p, if the KNOWN bit is active. Predicate knownBit denotes this fact:

$\mathsf{knownBit}\ \sigma\ p\ hn \equiv (^{\sigma}\mathsf{handle\_db}\ p\ !\ hn\ \wedge_{\mathsf{u}}\ \mathsf{KNOWN}) \neq 0$

Handles may also be marked as stolen by means of an active STOLEN bit. Similar as above, predicate stolenBit encapsulates this bit:

$\mathsf{stolenBit}\ \sigma\ p\ hn \equiv ({}^{\sigma}\mathsf{handle\_db}\ p\ !\ hn \wedge_{\mathsf{u}} \mathsf{STOLEN}) \neq 0$

Each handle hn in the handle database of a process p is associated with IPC rights. These rights are encoded in the four least significant bits of the corresponding entry. Function ipcRights returns the numerical value of the rights which process p owns to the process identified by handle hn:

$\mathsf{ipcRights}\ s\ p\ hn \equiv {}^{s}\mathsf{handle\_db}\ p\ !\ hn \wedge_{\mathsf{u}} 15$

The model maintains the rights datastructures in the Vamos state component rightsdb. In the following paragraphs, we define the functions which provide the abstractions for the sub-components of rightsdb.

Abstracting the Handle Database. Function abs_hdb abstracts the concrete information in the handle database of a process p and returns the corresponding abstract handle database hdb:

```
abs_hdb $\sigma$ $p$ $\equiv$
  $\lambda hn.$ **if** $hn = \mathsf{HN\_SELF}$ **then** $\lfloor {}^{\sigma}\mathsf{pid}\ p \rfloor$
        **else if** $hn = \mathsf{HN\_NONE} \vee hn = \mathsf{to\_int32}\ ({}^{\sigma}\mathsf{parent}\ p)$ **then** $\perp$
             **else if** $hn = \mathsf{HN\_PARENT} \wedge$
                    $({}^{\sigma}\mathsf{handle\_db}\ p\ !\ {}^{\sigma}\mathsf{parent}\ p \wedge_{\mathsf{u}} \mathsf{KNOWN}) \neq 0 \wedge$
                    $0 < {}^{\sigma}\mathsf{parent}\ p < \mathsf{PID\_MAX}$
                 **then** $\lfloor {}^{\sigma}\mathsf{parent}\ p \rfloor$
                 **else if** $({}^{\sigma}\mathsf{handle\_db}\ p\ !\ \mathsf{to\_nat32}\ hn \wedge_{\mathsf{u}} \mathsf{KNOWN}) \neq 0 \wedge$
                        $0 < \mathsf{to\_nat32}\ hn < \mathsf{PID\_MAX}$
                      **then** $\lfloor \mathsf{to\_nat32}\ hn \rfloor$ **else** $\perp$
```

Handle HN_SELF enables a process to determine its own identifier pid p. Handle HN_NONE is not associated to any process and delivers $\perp$. The same applies for accesses with to_int32 (parent p) which are considered as invalid. The only way to obtain the identifier of the parent, is accessing the handle database with handle HN_PARENT. However, the access only returns the corresponding process number of the parent, if parent p holds a valid process-identifier and the corresponding entry in the handle database is marked as known. Accessing the handle database with any handle hn delivers the corresponding process number $\lfloor \mathsf{to\_nat32}\ hn \rfloor$, as long as hn is in-range, i. e., represents a valid process number, and the known bit is active. Otherwise, the entry of hn is set to $\perp$.

Abstracting the Rights Database. The rights database rdb of a process realizes a mapping from process numbers to rights and is obtained by the abstraction function abs_rdb. As above, function abs_rdb takes a process pointer p as input in order to determine the data from handle_db:

```
abs_rdb $\sigma$ $p$ $\equiv$
  option_case $\bot$
   ($\lambda x$. **if** knownBit $\sigma$ $p$ $x$ **then** num2rights (to_int32 (ipcRights $\sigma$ $p$ $x$)) **else** $\bot$)
```

Accesses with invalid process numbers $\bot$, return $\bot$. Furthermore, rdb only stores information on known processes x. For this purpose, abs_rdb takes the concrete rights obtained by ipcRights and converts them into the abstract ones by means of the already introduced conversion function num2rights. Note that the numerical value of the concrete rights is represented as natural number, whereas num2rights expects integer values. Thus, we apply to_int32 to convert between both representations.

The entries of unknown processes are set to $\bot$.

Abstracting the Stolen Handles. Instead of explicitly marking the handles as stolen, the model subsumes all stolen handles in the handle database of a process in the set stolen. Accordingly, the task of function abs_stolen is to collect all stolen handles in the concrete handle database of a process p and put them into a set:

```
abs_stolen $\sigma$ $p$ $\equiv$
  **let** $stolen_{\mathsf{tmp}}$ =
        to_int32 ` $\{x \in \{0<..<\text{PID\_MAX}\}.\ x \neq {}^{\sigma}\text{parent}\ p \wedge \text{stolenBit}\ \sigma\ p\ x\}$
  **in if** $^{\sigma}$parent $p$ < PID_MAX $\wedge$ stolenBit $\sigma$ $p$ ($^{\sigma}$parent $p$)
     **then** {HN_PARENT} $\cup$ $stolen_{\mathsf{tmp}}$ **else** $stolen_{\mathsf{tmp}}$
```

In search of stolen handles, function abs_stolen inspects all entries of p's handle database. The indices $0 < x <$ PID_MAX which fulfill predicate stolenBit, are converted into handles (by means of to_int32) and stored in the set $stolen_{\mathsf{tmp}}$. However, while collecting stolen handles, the special meaning of HN_PARENT becomes apparent again. As parent p does not represent the actual handle to the parent process, it is excluded from $stolen_{\mathsf{tmp}}$. Instead, handle HN_PARENT is added to the set $stolen_{\mathsf{tmp}}$, if parent p is a valid process-identifier and fulfills predicate stolenBit.

Summing Up. Based on these prerequisites, the definition of the abstraction relation $\text{rel\_rightsdb}_{\mathsf{v}}$ looks as follows:

```
rel_rightsdb$_v$ $\sigma$ $inactQ$ $rightsdb$ $\equiv$
  $rightsdb$ =
  ($\lambda p$. **if** $p \in$ set $inactQ$ **then** $\bot$
       **else** $\lfloor$(|priv = $^{\sigma}$privileged (to_ptr $\sigma$ $p$), hdb = abs_hdb $\sigma$ (to_ptr $\sigma$ $p$),
              rdb = abs_rdb $\sigma$ (to_ptr $\sigma$ $p$),
              stolen = abs_stolen $\sigma$ (to_ptr $\sigma$ $p$),
              parent =
                **if** 0 < $^{\sigma}$parent (to_ptr $\sigma$ $p$) < PID_MAX
                **then** $\lfloor$$^{\sigma}$parent (to_ptr $\sigma$ $p$)$\rfloor$ **else** $\bot$|)$\rfloor$)
```

The model only stores data for active processes p. Entries for inactive ones are set to $\perp$. For active processes p, the privilege status is directly adopted from the implementation. Thereby, function $\mathsf{to\_ptr}$ is used to determine the corresponding process pointer of p. Together with the abstraction functions $\mathsf{abs\_hdb}$ and $\mathsf{abs\_rdb}$, this process pointer is also used in order to get the handle and rights databases hdb and rdb of process p. The model component parent is set to $\perp$, if the return value of the heap function parent is no valid process-identifier. Otherwise, this value is taken as abstract process number.

5.1.6 Abstracting the Device Datastructures

One of VAMOS's responsibilities is the interrupt delivery, i. e., accepting the interrupts from the external devices and forwarding them to the corresponding drivers. For this purpose, all relevant data is stored in the device datastructures. The implementation stores process-specific device data in the heap function $\mathsf{reg\_devices}$, and occurred and enabled interrupts in the global variables intocc and $\mathsf{intmask}$. The latter actually represents the user-visible parts of the CVM status register ($\mathsf{cvm\_ups\ cvmX}$).$\mathsf{statusreg}$. The double book-keeping is necessary, because the VAMOS implementation has no means to read out the status register of CVM[1].

In the implementation, all the device-relevant data is encoded in 32-bit natural numbers. These natural numbers, however, actually represent bit masks. For this reason, we use function $\mathsf{to\_bv32}$ to convert the natural numbers into their bit vector representation. Based on these bit vectors, the relevant device data is delivered by the entries with indices representing the device numbers. For instance, if an interrupt of device d has occured, the according bit $\mathsf{to\_bv32\ intocc\ !\ d}$ is active.

During the abstraction, we have to take into account that the implementation represents the device numbers as 32-bit natural numbers, whereas the model abstracts device numbers. Function $\mathsf{dev2nat}$ converts the abstract device numbers to the concrete ones.

With it, the definition of $\mathsf{rel\_devds_v}$ is given as follows:

$$
\begin{array}{l}
\mathsf{rel\_devds_v}\ \sigma\ \mathit{devds} \equiv \\
\quad (\forall d\ p.\ \mathit{devds}.\mathsf{driver}\ d = \\
\qquad\quad (\textbf{if}\ p \in \mathsf{set}\ {}^{\sigma}\mathsf{pib} \wedge \mathsf{to\_bv32}\ ({}^{\sigma}\mathsf{reg\_devices}\ p)\ !\ \mathsf{dev2nat}\ d = \mathbf{1} \\
\qquad\quad\ \ \textbf{then}\ \lfloor {}^{\sigma}\mathsf{pid}\ p \rfloor\ \textbf{else}\ \perp)) \wedge \\
\quad \mathit{devds}.\mathsf{enabled} = \{d.\ \mathsf{to\_bv32}\ (\mathsf{cvm\_ups}\ {}^{\sigma}\mathsf{cvmX}).\mathsf{statusreg}\ !\ \mathsf{dev2nat}\ d = \mathbf{1}\} \wedge \\
\quad \mathit{devds}.\mathsf{saved} = \{d.\ \mathsf{to\_bv32}\ {}^{\sigma}\mathsf{intocc}\ !\ \mathsf{dev2nat}\ d = \mathbf{1}\}
\end{array}
$$

The three conjunctions reflect the structure of the VAMOS state component devds. Sub-component driver maps device numbers to process numbers. In general, $\mathit{devds}.\mathsf{driver}\ d = \lfloor q \rfloor$ denotes the fact, that device d is handled by

[1]Note that there is no CVM primitive that delivers the value of the status register of CVM.

driver q. If d is not handled by any driver, its entry is set to $\perp$. In order to establish a relation to the implementation, we take an arbitrary process pointer p and check whether it is registered as driver for d. A driver is found, if bit dev2nat d of the vector Bind p. to_bv32 (reg_devices p) is active. In this case, the entry for d in driver is set to the process number pid p. If no such process pointer p can be found, the entry is set to $\perp$. Note that, by design, there exists at most one such p which is registered as driver for d.

The set enabled contains all device numbers whose interrupts are enabled. A relation to the implementation is established, if each device number d out of enabled is also enabled in the CVM status register, i. e., to_bv32 (cvm_ups cvmX).statusreg ! dev2nat d $= \mathbf{1}$.

Based on intocc, the abstraction proceeds in a similar way, in order to obtain the set saved of device numbers with raised but not yet delivered interrupts.

5.1.7 The Vamos Abstraction Relation

The overall VAMOS abstraction relation relies on the abstractions for the particular VAMOS state components described above:

$$
\begin{aligned}
&\mathsf{Abs}_{\mathsf{V+D}}\ \sigma\ (s_{\mathsf{V}}, s_{\mathsf{D}}) \equiv \\
&\quad \mathsf{cvm\_devs}\ {}^{\sigma}\mathsf{cvmX} = s_{\mathsf{D}}\ \wedge \\
&\quad \mathsf{rel\_procs_v}\ \sigma\ s_{\mathsf{V}}.\mathsf{schedds.inactive}\ s_{\mathsf{V}}.\mathsf{schedds.wait}\ s_{\mathsf{V}}.\mathsf{procs}\ \wedge \\
&\quad \mathsf{rel\_schedds_v}\ \sigma\ s_{\mathsf{V}}.\mathsf{schedds}\ \wedge \\
&\quad \mathsf{rel\_priodb_v}\ \sigma\ s_{\mathsf{V}}.\mathsf{priodb}\ s_{\mathsf{V}}.\mathsf{schedds.inactive}\ \wedge \\
&\quad \mathsf{rel\_sndstatdb_v}\ \sigma\ s_{\mathsf{V}}.\mathsf{procs}\ s_{\mathsf{V}}.\mathsf{sndstatdb}\ \wedge \\
&\quad \mathsf{rel\_rightsdb_v}\ \sigma\ s_{\mathsf{V}}.\mathsf{schedds.inactive}\ s_{\mathsf{V}}.\mathsf{rightsdb}\ \wedge \\
&\quad \mathsf{rel\_devds_v}\ \sigma\ s_{\mathsf{V}}.\mathsf{devds}\ \wedge\ (\mathsf{cvm\_ups}\ {}^{\sigma}\mathsf{cvmX}).\mathsf{currentp} = \mathsf{v\_cup}\ s_{\mathsf{V}}.\mathsf{schedds}
\end{aligned}
$$

The first conjunction is an equality for the device subsystem states because both, the implementation and the model, share the same device representation. For the remaining state components, we use the abstraction functions introduced above. In addition, the last part of the conjunction enforces the equivalence between the current process identifier currentp of CVM and the one v_cup of the VAMOS model.

The theory file `vamosAbstractions.thy` contains the complete formal definitions of the VAMOS abstraction relation.

Summary. A key feature of the VAMOS abstraction relation is that it allows the usage of infinite datatypes in the abstract model, while the implementation is bound to the 32-bit representation of numbers. Thus, on the abstract level, we specify points in time as unbounded natural numbers, while the implementation uses unsigned 32-bit integers to represent those times. With the abstract idea of a monotonicly increasing time, we further avoid the handling of possible overflows as it is required in the implementation (cf. Section 6.3). Consequently, the abstraction relation has to deal with an growing offset between the implementation time and the abstract time.

Another crucial point is the abstraction of the user processes. Although both, the implementation and the model, use the same granularity of virtual machines, the abstraction is not as simple as one might suppose. The reason for this are the differing virtual machines of waiting processes and (possibly) the one of the current process. In retrospect, this fact seems to be rather natural. However, we only became aware of these differing machines, when we considered the entire system, i. e., the steps before and after the actual VAMOS execution.

In contrast, abstracting the remaining kernel datastructures is rather straightforward.

5.2 Implementation Invariant

Each kernel execution is constrained by certain properties and requirements. They are encapsulated in the implementation invariant which is established at the initialization and preserved by the kernel executions and user steps. The main ingredients are:

- value bounds: The target 32-bit VAMP architectur as well as the design of the VAMOS kernel itself lead to various bounds.
- validity and consistency requirements: We have to ensure that the design guidelines are not violated. Thus, a handle, for instance, must not be simultaneously marked as stolen and known. Furthermore, for efficiency reasons, the implementation maintains redundant datastructures. For instance, the process priorities that actually could be derived from the ready list memberships. Certainly, the redundant information must not be contradicting.
- miscellaneous requirements: Mainly used in the context of IPC and waiting processes, they relate concrete with abstract properties, e. g., that a process with state SEND_STATE in the implementation produces the output IPC_SEND in the model.

We formulate the requirements of the two former categories solely over the implementation states, whereas the latter also involves abstract model states. It definitely would have been possible to formulate these properties over the implementation states, as well. The implementation invariant, however, is a proof tool and in the proof, we are often confronted with a mixture of concrete and abstract statements. Thus, it turned out to be reasonable to adapt this combination for stating certain properties.

The following sections introduce various predicates encapsulating the particular properties. Later on, these predicates are combined to the overall implementation invariant.

5.2.1 Value Bounds

The target Vamp architecture and the design of the Vamos kernel itself involve various bounds. Those which affect the variables of the implementation state σ are subsumed in predicate $\mathsf{valueBounds_v}$. The formal definition of the predicate is quite uniform and lengthy. For this reason, we merely present representative excerpts of the definition to convey the general nature of the value ranges.

Recall that many variables of the Vamos implementation state represent natural and integer numbers. The limiting factor, however, is the 32-bit Vamp architecture, the Vamos kernel runs on. Thus, natural numbers x can only describe values in the range $0 \leq x < 2^{32}$, and the values of integer numbers y are limited to $-2^{31} \leq y \leq 2^{31} - 1$.

Furthermore, our system relies on a restricted amount of memory and the Vamos kernel may only allocate a total of TVM_MAXPAGES memory pages. Accordingly, TVM_MAXPAGES is the upper bound for the value of the global variable vamos_pages_used, which incidentally also applies for the entries in the heap function fip denoting the number of memory pages occupied by an individual process.

The bounds are not only induced by the target architecture but also by the kernel design itself. For instance, our system only allows process-identifiers in the range {1<..<PID_MAX}. Accordingly, the heap function pid may only return values that fit into this range. By means of process pointers, Vamos enables the access to the corresponding process information blocks. The respective pointers are contained in the array pib which is of length PID_MAX, accordingly. Limiting the number of priority levels to PRIOCNT also has an impact. The entries in the heap function priority range between 0 and PRIOCNT, and the value current_max_prio of the current maximum priority is always smaller than PRIOCNT.

5.2.2 Validity and Consistency Requirements

The validity and consistency requirements ensure the conformance with the design guidelines and the consistency between redundant datastructures.

Requirements for the Current Process

The implemenation identifies the current process by the process pointer current_process. No current process is apparent, if current_process = Null. Otherwise, current_process denotes a process pointer out of pib. In addition, its state has to be READY_STATE.

Formally, predicate $\mathsf{validCup_v}$ encapsulates these requirements:

$$\begin{aligned}&\mathsf{validCup_v}\ \sigma \equiv\\ &\quad {}^{\sigma}\mathsf{current\_process} = \mathsf{Null} \lor\\ &\quad {}^{\sigma}\mathsf{current\_process} \in \mathsf{set}\ {}^{\sigma}\mathsf{pib} \land {}^{\sigma}\mathsf{state}\ {}^{\sigma}\mathsf{current\_process} = \mathsf{READY\_STATE}\end{aligned}$$

Requirements for the Virtual Machines

By means of the Cvm framework and, in particular, by means of the external state component cvmX, we are able to argue about low-level entities, like process registers or memory. These entities are not a direct part of the Vamos implementation but have an indirect impact on it, and thus have to fulfill certain requirements.

Formally, predicate $\mathsf{validVMs_v}$ encapsulates all demanded properties:

$$
\begin{array}{l}
\mathsf{validVMs_v}\ \sigma \equiv \\
\quad \mathsf{is\_valid\_cvmup}\ (\mathsf{cvm\_ups}\ {}^{\sigma}\mathsf{cvmX}) \wedge \\
\quad (\forall p{\in}\{x \in \mathsf{set}\ {}^{\sigma}\mathsf{pib}.\ {}^{\sigma}\mathsf{state}\ x \neq \mathsf{INACTIVE\_STATE}\}. \\
\qquad {}^{\sigma}\mathsf{fip}\ p = \mathsf{size_{asm}}\ ((\mathsf{cvm\_ups}\ {}^{\sigma}\mathsf{cvmX}).\mathsf{userprocesses}\ ({}^{\sigma}\mathsf{pid}\ p)) \wedge \\
\qquad 0 < {}^{\sigma}\mathsf{fip}\ p) \wedge \\
\quad {}^{\sigma}\mathsf{vamos\_pages\_used} = \\
\quad (\sum i{\in}\{0{<}..{<}\mathsf{PID\_MAX}\}.\ \mathsf{size_{asm}}\ ((\mathsf{cvm\_ups}\ {}^{\sigma}\mathsf{cvmX}).\mathsf{userprocesses}\ i))
\end{array}
$$

An essential part of the definition is predicate is_valid_cvmup ensuring the validity of the virtual user machines, as introduced in Section 2.4. The remaining requirements are due to the double book-keeping regarding the memory management. Data regarding the memory consumption can actually be derived from the virtual machines. Due to the limited access, however, the implementation maintains the heap function fip which stores the memory amount of the particular processes. Some kernel actions, like the process creation, require the total memory amount of all processes. For efficiency reasons, the kernel maintains the global variable vamos_pages_used which denotes the total amount and prevents the kernel from explicitly summing up the individual process amounts.

Nevertheless, these datastructures have to reflect the conditions given by the machines, where function $\mathsf{size_{asm}}$ returns the current memory amount of a machine. Thus, for an active process p, the value of fip p corresponds to the result of $\mathsf{size_{asm}}$ applied to the corresponding virtual machine. As active processes occupy at least one memory page, this value has to be greater than 0.

In addition, the total memory amount results from summing up the individual memory amounts of all user machines.

Unique Process Identification

We already mentioned that the array pib contains the pointers to the process information blocks. Each of these pointers is exclusively assigned to one process, i. e., the elements of pib have to be distinct. In addition to that, no Null pointers are allowed as array entries. The process pointers enable the access to the process information blocks which contain (amongst others) the process-identifiers pid. By design, a process-identifier corresponds to the index i of the corresponding pointer in pib, i. e., pid (pib ! i) = i. In addition, process-identifiers are unique, i. e., two pointers p and q are equal iff the application of pid returns the same values.

The predicate $\mathsf{uniqueProcId_v}$ formally states these properties:

$$
\begin{array}{l}
\mathsf{uniqueProcId_v}\ \sigma \equiv \\
\quad \mathsf{distinct}\ {}^{\sigma}\mathsf{pib} \land \\
\quad (\forall i{<}\mathsf{length}\ {}^{\sigma}\mathsf{pib}.\ {}^{\sigma}\mathsf{pib}\ !\ i \neq \mathsf{Null} \land {}^{\sigma}\mathsf{pid}\ ({}^{\sigma}\mathsf{pib}\ !\ i) = i) \land \\
\quad (\forall x{\in}\mathsf{set}\ {}^{\sigma}\mathsf{pib}.\ \forall y{\in}\mathsf{set}\ {}^{\sigma}\mathsf{pib}.\ ({}^{\sigma}\mathsf{pid}\ x = {}^{\sigma}\mathsf{pid}\ y) = (x = y))
\end{array}
$$

Requirements for the Rights Datastructures

The rights datastructures in the VAMOS implementation are process-specific and maintained by the heap function handle_db. For any process p, this function delivers an array of size PID_MAX. The array indices are connected to process handles. As already pointed out in Section 5.1.5, the array entries store natural numbers which encode the IPC rights associated to the handle and also whether the handle is known or stolen.

This information has to follow certain requirements which we define subsequently. Thereby, we mainly concentrate on the information regarding active processes. The implemenation considers a process as active, if the respective pointer p is out of pib and its status is different from INACTIVE_STATE. In order to distinguish between known and stolen handles, we rely on the predicates knownBit and stolenBit.

Special Handle Database Entries. Predicate specialHdbEntries deals with special entries in the handle databases:

$$
\begin{array}{l}
\mathsf{specialHdbEntries}\ \sigma \equiv \\
\quad \mathbf{let}\ \mathit{active} = \{x \in \mathsf{set}\ {}^{\sigma}\mathsf{pib}.\ {}^{\sigma}\mathsf{state}\ x \neq \mathsf{INACTIVE\_STATE}\}; \\
\qquad\quad \mathit{inactive} = \{x \in \mathsf{set}\ {}^{\sigma}\mathsf{pib}.\ {}^{\sigma}\mathsf{state}\ x = \mathsf{INACTIVE\_STATE}\} \\
\quad \mathbf{in}\ \forall p{\in}\mathit{active}. \\
\qquad\qquad {}^{\sigma}\mathsf{handle\_db}\ p\ !\ 0 = 0 \land \\
\qquad\qquad {}^{\sigma}\mathsf{handle\_db}\ p\ !\ {}^{\sigma}\mathsf{pid}\ p = 0 \land \\
\qquad\qquad (\forall q{\in}\mathit{inactive}.\ {}^{\sigma}\mathsf{handle\_db}\ p\ !\ {}^{\sigma}\mathsf{pid}\ q = 0 \lor \mathsf{stolenBit}\ \sigma\ p\ ({}^{\sigma}\mathsf{pid}\ q))
\end{array}
$$

There are two fixed entries in each handle database of an active process p storing 0 by default, i. e., no information. First, the entry with index 0, because no process with identifier 0 exists in VAMOS. Second, the entry with index pid p, because VAMOS does not allow processes to send to or receive from themselves. Entries for inactive processes q are either 0 or consist of an active stolen bit. The latter occurs, if q was known by p before its termination and p is not yet notified about this termination.

Consistent Counting of Stolen Handles. In order to figure out the current number of stolen handles in the handle database of an arbitrary process, the VAMOS kernel maintains the heap function stolen_count. Efficiency is the only reason for this function, because this information could also be derived from the particular handle databases. Predicate validStolen-Count ensures that the information in stolen_count does not contradict with the situation in the handle databases:

$$\begin{aligned}
&\mathsf{validStolenCount}\ \sigma \equiv\\
&\quad \textbf{let}\ \mathit{active} = \{x \in \mathsf{set}\ {}^{\sigma}\mathsf{pib}.\ {}^{\sigma}\mathsf{state}\ x \neq \mathsf{INACTIVE\_STATE}\}\\
&\quad \textbf{in}\ \forall p{\in}\mathit{active}.\ {}^{\sigma}\mathsf{stolen\_count}\ p = \mathsf{card}\ \{x \in \{0{<}..{<}\mathsf{PID\_MAX}\}.\ \mathsf{stolenBit}\ \sigma\ p\ x\}
\end{aligned}$$

For an active process p, the set $\{x \in \{0{<}..{<}\mathsf{PID\_MAX}\}.\ \mathsf{stolenBit}\ s\ p\ x\}$ subsumes the stolen handles. The cardinality of this set must be equal to the value stored in stolen_count p.

Properties on Known and Stolen Handles. As long as a process is known, i. e., the known bit is raised, it is not inactive. Furthermore, process handles are not simultaneously known and stolen.

Predicate propKnownStolen encapsulates these properties:

$$\begin{aligned}
&\mathsf{propKnownStolen}\ \sigma \equiv\\
&\quad (\forall p{\in}\mathsf{set}\ {}^{\sigma}\mathsf{pib}.\\
&\qquad \forall q{\in}\mathsf{set}\ {}^{\sigma}\mathsf{pib}.\ \mathsf{knownBit}\ \sigma\ p\ ({}^{\sigma}\mathsf{pid}\ q) \longrightarrow {}^{\sigma}\mathsf{state}\ q \neq \mathsf{INACTIVE\_STATE}) \land\\
&\quad (\forall p{\in}\mathsf{set}\ {}^{\sigma}\mathsf{pib}.\\
&\qquad \forall q{<}\mathsf{length}\ ({}^{\sigma}\mathsf{handle\_db}\ p).\\
&\qquad\quad (\mathsf{stolenBit}\ \sigma\ p\ q \longrightarrow \neg\ \mathsf{knownBit}\ \sigma\ p\ q) \land\\
&\qquad\quad (\mathsf{knownBit}\ \sigma\ p\ q \longrightarrow \neg\ \mathsf{stolenBit}\ \sigma\ p\ q))
\end{aligned}$$

Summing Up. The predicate $\mathsf{validRights}_{\mathsf{v}}$ combines the requirements for the rights datastructures:

$$\mathsf{validRights}_{\mathsf{v}}\ \sigma \equiv \mathsf{specialHdbEntries}\ \sigma \land \mathsf{validStolenCount}\ \sigma \land \mathsf{propKnownStolen}\ \sigma$$

Requirements for the Lists

The VAMOS microkernel uses queues for process scheduling, on the one hand, and matching communication partners for IPC, on the other one.

During the abstraction in Section 5.1.7, we claimed that the scheduling queues are implemented as doubly-linked lists. This assumption, however, must be discharged by the implementation invariant, as well as the requirement that only process pointers are contained in these lists. In addition to that, the invariant also establishes the connection between the process states and the list memberships.

More involved are the requirements for the send lists. Apart from the aforementioned properties they must also satisfy certain consistency requirements as they actually represent parts of the wait list.

The requirements for the list represent a conjunction of various sub-predicates for the particular lists.

Valid Inactive List. Predicate $\mathsf{validInact}_\mathsf{v}$ encapsulates the requirements for the inactive list:

$$
\begin{array}{l}
\mathsf{validInact_v}\ \sigma \equiv \\
\quad \exists \mathit{Inact}\ \mathit{lst}. \\
\quad\quad \mathsf{dList}\ {}^\sigma\mathsf{inactive\_list}\ {}^\sigma\mathsf{queue\_next}\ {}^\sigma\mathsf{queue\_prev}\ \mathit{lst}\ \mathit{Inact} \wedge \\
\quad\quad (\forall p.\ p \in \mathsf{set}\ \mathit{Inact} \longrightarrow p \in \mathsf{set}\ {}^\sigma\mathsf{pib}) \wedge \\
\quad\quad (\forall p.\ p \in \mathsf{set}\ {}^\sigma\mathsf{pib} \longrightarrow (p \in \mathsf{set}\ \mathit{Inact}) = ({}^\sigma\mathsf{state}\ p = \mathsf{INACTIVE\_STATE}))
\end{array}
$$

It states that the inactive list is implemented as a doubly-linked list with inactive_list pointing to the head, and queue_next and queue_prev denoting the next and previous pointers. Furthermore, all elements are process pointers out of pib. Finally, a process is in the inactive list iff its state is INACTIVE_STATE.

Valid Wait List. The predicate $\mathsf{validWkp}_\mathsf{v}$ is similar to $\mathsf{validInact}_\mathsf{v}$ and encapsulates the properties of the wait list. As the definition suggests, only the process states are adjusted:

$$
\begin{array}{l}
\mathsf{validWkp_v}\ \sigma \equiv \\
\quad \exists \mathit{Wkp}\ \mathit{lst}. \\
\quad\quad \mathsf{dList}\ {}^\sigma\mathsf{wakeup\_list}\ {}^\sigma\mathsf{queue\_next}\ {}^\sigma\mathsf{queue\_prev}\ \mathit{lst}\ \mathit{Wkp} \wedge \\
\quad\quad (\forall p.\ p \in \mathsf{set}\ \mathit{Wkp} \longrightarrow p \in \mathsf{set}\ {}^\sigma\mathsf{pib}) \wedge \\
\quad\quad (\forall p.\ p \in \mathsf{set}\ {}^\sigma\mathsf{pib} \longrightarrow \\
\quad\quad\quad\quad (p \in \mathsf{set}\ \mathit{Wkp}) = \\
\quad\quad\quad\quad ({}^\sigma\mathsf{state}\ p \in \{\mathsf{SEND\_STATE},\ \mathsf{RECEIVE\_STATE},\ \mathsf{SEND\_RECEIVE\_STATE}\}))
\end{array}
$$

Valid Ready Lists. The quintessence of both predicates above is also found in the predicate $\mathsf{validRdys}_\mathsf{v}$ which states the validity requirements for the ready lists. In addition, the different priority classes as well as the current process are taken into account: Let p denote a process and Rdy the ready list of priority i. Process p is in the ready list Rdy iff its state is READY_STATE and, in addition, its priority is i. Furthermore, p is in Rdy and simultaneously denotes the current process iff it is the head of Rdy and the priority i is equal to the current maximum priority current_max_prio:

$$
\begin{array}{l}
\mathsf{validRdys_v}\ \sigma \equiv \\
\quad \forall i {<} \mathsf{length}\ {}^\sigma\mathsf{ready\_lists}. \\
\quad\quad \exists \mathit{Rdy}\ \mathit{lst}. \\
\quad\quad\quad \mathsf{dList}\ ({}^\sigma\mathsf{ready\_lists}\ !\ i)\ {}^\sigma\mathsf{queue\_next}\ {}^\sigma\mathsf{queue\_prev}\ \mathit{lst}\ \mathit{Rdy} \wedge \\
\quad\quad\quad (\forall p.\ p \in \mathsf{set}\ \mathit{Rdy} \longrightarrow p \in \mathsf{set}\ {}^\sigma\mathsf{pib}) \wedge \\
\quad\quad\quad (\forall p.\ p \in \mathsf{set}\ {}^\sigma\mathsf{pib} \longrightarrow \\
\quad\quad\quad\quad\quad (p \in \mathsf{set}\ \mathit{Rdy}) = ({}^\sigma\mathsf{state}\ p = \mathsf{READY\_STATE} \wedge {}^\sigma\mathsf{priority}\ p = i) \wedge \\
\quad\quad\quad\quad\quad (p \in \mathsf{set}\ \mathit{Rdy} \wedge p = {}^\sigma\mathsf{current\_process}) = \\
\quad\quad\quad\quad\quad (p = {}^\sigma\mathsf{ready\_lists}\ !\ i \wedge i = {}^\sigma\mathsf{current\_max\_prio}))
\end{array}
$$

Valid Send Lists. In order to determine potential communication partners for IPC-receive operations, the VAMOS implementation organizes so-called send lists. Send lists exist solely for performance reasons, because they can actually be derived from the wait list. Hence, these lists have to fulfill – apart from the usual properties – certain consistency requirements.

The predicate SqsInWkp formalizes the forementioned property that the send lists are contained in the wait list:

```
SqsInWkp $\sigma \equiv$
  $\exists$Wkp lst.
   dList $^\sigma$wakeup_list $^\sigma$queue_next $^\sigma$queue_prev lst Wkp $\wedge$
   ($\forall p \in$set $^\sigma$pib.
      let Sq = [x$\in$Wkp .
                  $^\sigma$state x $\in$ {SEND_STATE, SEND_RECEIVE_STATE} $\wedge$
                  $^\sigma$ipc_pid x = $^\sigma$pid p]
      in $\exists$lst. dList ($^\sigma$send_queue p) $^\sigma$send_queue_next $^\sigma$send_queue_prev lst
               Sq)
```

Starting point is the existence of the wait list *Wkp*. The send list *Sq* of an arbitrary process *p* is derived from *Wkp* by filtering all processes *x* with state SEND_STATE or SEND_RECEIVE_STATE and which specified *p* as communication partner, i. e., ipc_pid x = pid p. In addition, *Sq* must represent a doubly-linked list.

The predicate distinctSqs ensures that the send lists of two different processes p and q have no elements in common, i. e., processes are not allowed to send to more than one process at the same time:

```
distinctSqs $\sigma \equiv$
  $\forall p \in$set $^\sigma$pib.
    $\forall q \in$set $^\sigma$pib.
      $p \neq q \longrightarrow$
      ($\exists$Sq Sq2 sq_lst sq_lst2.
         dList ($^\sigma$send_queue p) $^\sigma$send_queue_next $^\sigma$send_queue_prev sq_lst Sq $\wedge$
         dList ($^\sigma$send_queue q) $^\sigma$send_queue_next $^\sigma$send_queue_prev sq_lst2 Sq2 $\wedge$
         set Sq $\cap$ set Sq2 = {})
```

Only processes currently performing a send operation may reside in send lists. A process is not part of a send list, as long as the according next and previous pointers in its process control block are Null. Thus, if the state of a process *p* neither denotes SEND_STATE nor SEND_RECEIVE_STATE, it is not in a send list. Predicate onlySendersInSqs formalizes this requirement:

```
onlySendersInSqs $\sigma \equiv$
  $\forall p \in$set $^\sigma$pib.
    $^\sigma$state p $\neq$ SEND_STATE $\wedge$ $^\sigma$state p $\neq$ SEND_RECEIVE_STATE $\longrightarrow$
    $^\sigma$send_queue_next p = Null $\wedge$ $^\sigma$send_queue_prev p = Null
```

The predicate openRcvSqs states that waiting in case of an open-receive implies an empty send list:

$$
\begin{array}{l}
\mathsf{openRcvSqs}\ \sigma \equiv \\
\quad \forall p{\in}\mathsf{set}\ {}^{\sigma}\mathsf{pib}. \\
\qquad {}^{\sigma}\mathsf{state}\ p = \mathsf{RECEIVE\_STATE} \wedge {}^{\sigma}\mathsf{ipc\_pid}\ p = 0 \longrightarrow {}^{\sigma}\mathsf{send\_queue}\ p = \mathsf{Null}
\end{array}
$$

A process p waits while performing an open receive operation, if its state is set to RECEIVE_STATE and $\mathsf{ipc\_pid}\ p = 0$. The empty send list is denoted by $\mathsf{send\_queue}\ p = \mathsf{Null}$.

Finally, predicate $\mathsf{validSqs_v}$ combines all the requirements for the send lists:

$$\mathsf{validSqs_v}\ \sigma \equiv \mathsf{SqsInWkp}\ \sigma \wedge \mathsf{distinctSqs}\ \sigma \wedge \mathsf{onlySendersInSqs}\ \sigma \wedge \mathsf{openRcvSqs}\ \sigma$$

Summing Up. The overall predicate $\mathsf{validLists_v}$ formulates the requirements for the lists in the VAMOS implementation:

$$\mathsf{validLists_v}\ \sigma \equiv \mathsf{validRdys_v}\ \sigma \wedge \mathsf{validWkp_v}\ \sigma \wedge \mathsf{validInact_v}\ \sigma \wedge \mathsf{validSqs_v}\ \sigma$$

Requirements for the Device Datastructures

In the following, we discuss the particular aspects of requirements for the device datastructures. Again, we first define sub-predicates and combine them, later on.

Interrupt Mask Format. VAMOS stores device relevant data in 32-bit natural numbers. However, as already mentioned, we rather argue on bitvectors or interrupt masks. Again, function to_bv32 is used to convert the 32-bit natural numbers into bitvectors.

External interrupt handling in VAMOS is restricted to the user-visible devices. Starting with device number DEVFIRST, a total of DEVCOUNT devices is handled. Referring to the bitvectors, the relevant data is contained in the bitfields reaching from DEVFIRST to DEVFIRST + DEVCOUNT − 1.

The predicate intFormats is based on these observations:

$$
\begin{array}{l}
\mathsf{intFormats}\ \sigma \equiv \\
\quad \mathbf{let}\ \mathit{validIntFormat} = \\
\qquad\quad \lambda x.\ \forall i{<}\mathsf{length}\ (\mathsf{to\_bv32}\ x). \\
\qquad\qquad\quad i < \mathsf{DEVFIRST} \vee \mathsf{DEVFIRST} + \mathsf{DEVCOUNT} \le i \longrightarrow \\
\qquad\qquad\quad \mathsf{to\_bv32}\ x\ !\ i = \mathbf{0} \\
\quad \mathbf{in}\ \mathit{validIntFormat}\ {}^{\sigma}\mathsf{inthd}\ \wedge \\
\qquad \mathit{validIntFormat}\ {}^{\sigma}\mathsf{intocc}\ \wedge \\
\qquad (\forall p.\ p \in \mathsf{set}\ {}^{\sigma}\mathsf{pib} \longrightarrow \mathit{validIntFormat}\ ({}^{\sigma}\mathsf{reg\_devices}\ p))
\end{array}
$$

The auxiliary function *validIntFormat* ensures, that only the bits of user-visible devices may be active and is applied to all device-relevant data.

No Inactive Interrupt Handlers. Naturally, inactive processes cannot handle any device interrupts. However, this property has to be fixed with predicate inactNoHd:

$\mathsf{inactNoHd}\ \sigma \equiv$
$\quad \forall p{\in}\mathsf{set}\ {}^{\sigma}\mathsf{pib}.\ {}^{\sigma}\mathsf{state}\ p = \mathsf{INACTIVE\_STATE} \longrightarrow {}^{\sigma}\mathsf{reg\_devices}\ p = 0$

Unique Device Registration. The Vamos implementation does not allow multiple handlers for one device interrupt. Thus, if arbitrary processes p and q handle the same device i, they have to be equal.

Predicate uniqueDevReg formalizes this requirement:

$$\begin{array}{l}
\mathsf{uniqueDevReg}\ \sigma \equiv\\
\quad \forall p{\in}\mathsf{set}\ {}^{\sigma}\mathsf{pib}.\\
\quad\quad \forall q{\in}\mathsf{set}\ {}^{\sigma}\mathsf{pib}.\\
\quad\quad\quad \forall i{<}32.\\
\quad\quad\quad\quad \mathsf{to\_bv32}\ ({}^{\sigma}\mathsf{reg\_devices}\ p)\ !\ i = \mathbf{1} \wedge\\
\quad\quad\quad\quad \mathsf{to\_bv32}\ ({}^{\sigma}\mathsf{reg\_devices}\ q)\ !\ i = \mathbf{1} \longrightarrow\\
\quad\quad\quad\quad p = q
\end{array}$$

Handled and Registered Interrupts. For any handled interrupt i there exists a driver. Accordingly, each interrupt registered to a driver is also marked as handled.

The consistency between the variables reg_devices and inthd states predicate intHdregDev:

$$\begin{array}{l}
\mathsf{intHdregDev}\ \sigma \equiv\\
\quad (\forall i{<}32.\ \mathsf{to\_bv32}\ {}^{\sigma}\mathsf{inthd}\ !\ i = \mathbf{1} \longrightarrow\\
\quad\quad\quad\quad (\exists! p.\ \mathsf{to\_bv32}\ ({}^{\sigma}\mathsf{reg\_devices}\ p)\ !\ i = \mathbf{1})) \wedge\\
\quad (\forall p{\in}\mathsf{set}\ {}^{\sigma}\mathsf{pib}.\\
\quad\quad \forall i{<}32.\ \mathsf{to\_bv32}\ ({}^{\sigma}\mathsf{reg\_devices}\ p)\ !\ i = \mathbf{1} \longrightarrow \mathsf{to\_bv32}\ {}^{\sigma}\mathsf{inthd}\ !\ i = \mathbf{1})
\end{array}$$

Only Handled Interrupts Occur. Vamos stores only occurrences of handled interrupts.

Predicate intOccHd encapsulates this fact:

$$\mathsf{intOccHd}\ \sigma \equiv \forall i{<}32.\ \mathsf{to\_bv32}\ {}^{\sigma}\mathsf{intocc}\ !\ i = \mathbf{1} \longrightarrow \mathsf{to\_bv32}\ {}^{\sigma}\mathsf{inthd}\ !\ i = \mathbf{1}$$

Summing Up. Predicate $\mathsf{validDevds_v}$ constitutes the combination of the above properties:

$$\begin{array}{l}
\mathsf{validDevds_v}\ \sigma \equiv\\
\quad \mathsf{intFormats}\ \sigma \wedge \mathsf{inactNoHd}\ \sigma \wedge \mathsf{uniqueDevReg}\ \sigma \wedge \mathsf{intHdregDev}\ \sigma \wedge \mathsf{intOccHd}\ \sigma
\end{array}$$

5.2.3 Miscellaneous Requirements

The implementation of the Vamos kernel contains a number of global variables for which there are no analogous components in the abstract model state. However, these variables have precise meanings and their values can be expressed as formulas over the abstract Vamos state. Apart from these general requirements, we also demand certain properties on waiting processes and IPC operations.

Subsequently, we define the corresponding predicates.

General Requirements.

The VAMOS model uses the identifier v_cup to determine the currently running process. If no process is running, it denotes $\bot$, otherwise, the corresponding abstract process number. In contrast, the implementation uses the pointer current_process to identify the currently running process. If no such process exists, the pointer denotes Null. Thus, for consistency reasons, the abstract identifier v_cup delivers $\bot$ iff current_process denotes Null.

The global implementation variable current_max_prio also does not have a direct counterpart in a VAMOS state space. However, its value has to correspond to the highest priority of the non-empty ready queues in the model.

In addition to that, we have to fix that the interrupts of the timer device are always enabled. Interrupts of the swap device, in contrast, are not visible by the kernel anymore and thus disabled.

The predicate generalProps formally encapsulates these properties:

$\mathsf{generalProps}\ \sigma\ s_V \equiv$
$\quad(\mathsf{v\_cup}\ s_V.\mathsf{schedds} = \bot) = ({}^{\sigma}\mathsf{current\_process} = \mathsf{Null}) \wedge$
$\quad{}^{\sigma}\mathsf{current\_max\_prio} =$
$\quad\mathsf{Max}\ (\{i.\ i < \mathsf{PRIOCNT} \wedge s_V.\mathsf{schedds.ready}\ i \neq []\} \cup \{0\}) \wedge$
$\quad\mathsf{DEV\_TIMER} \in s_V.\mathsf{devds.enabled} \wedge \mathsf{DEV\_SWAP} \notin s_V.\mathsf{devds.enabled}$

Note that the current maximum priority is set to 0, if no process is ready.

Properties of Waiting Processes

Apart from waiting, the respective processes have to fulfill certain properties which are formalized in predicate waitProps:

$\mathsf{waitProps}\ \sigma\ s_V \equiv$
$\quad(\forall p{\in}\mathsf{set}\ s_V.\mathsf{schedds.wait}.\ \exists imm.\ \mathsf{current\_instr}\ \lceil s_V.\mathsf{procs}\ p\rceil = \textsc{trap}\ imm) \wedge$
$\quad(\forall p{\in}\mathsf{set}\ s_V.\mathsf{schedds.wait}.\ {}^{\sigma}\mathsf{next\_timeout} \leq {}^{\sigma}\mathsf{timeout}\ ({}^{\sigma}\mathsf{pib}\ !\ p)) \wedge$
$\quad({}^{\sigma}\mathsf{current\_time} < {}^{\sigma}\mathsf{next\_timeout} \longrightarrow$
$\quad\ (\forall p{\in}\mathsf{set}\ s_V.\mathsf{schedds.wait}.\ \neg\ \mathsf{expiredTimeout}\ s_V.\mathsf{schedds}\ p))$

Waiting processes p are contained in queue s_V.schedds.wait. In VAMOS, processes p only wait for pending IPC operations. IPC operations, in turn, are initiated by TRAP instructions. Thus, the first conjunction constitutes that the current instruction of a waiting process p denotes a TRAP. IPC operations are usually bounded by timeouts. In the implementation, the actual timeouts of processes are stored in the heap function timeout. The minimum of these values is stored in the global variable next_timeout. It determines the point in time, when the next timeout expires. Keeping the consistency between the heap function timeout and variable next_timeout requires, that each entry in timeout has to be greater than or equal to next_timeout.

Finally, the wait queue does not contain processes with expired timeouts as long as the time current_time is smaller than the value of next_timeout.

Properties of Inter-Process Communications

We already mentioned, that the implementation distinguishes the IPC operations by means of the process states, whereas the model uses the process outputs. Certainly, we have to establish a relation between these both representations.

In addition to that, pending IPC operations already passed the invocation phase. Thus, the arguments specified by the corresponding process can be considered as valid.

The predicate ipcProps fixes these properties:

```
ipcProps $\sigma$ $s_V$ $\equiv$
  $\forall p \in$set $^\sigma$pib.
    ($^\sigma$state $p$ = SEND_STATE $\longrightarrow$
     consisStateOutSEND $\sigma$ $s_V$ $p$ $\wedge$ ipcArgsSEND $\sigma$ $s_V$ $p$) $\wedge$
    ($^\sigma$state $p$ = SEND_RECEIVE_STATE $\longrightarrow$
     consisStateOutSENDRCV $\sigma$ $s_V$ $p$ $\wedge$ ipcArgsSENDRCV $\sigma$ $s_V$ $p$) $\wedge$
    ($^\sigma$state $p$ = RECEIVE_STATE $\longrightarrow$
     consisStateOutRCV $\sigma$ $s_V$ $p$ $\wedge$ ipcArgsRCV $\sigma$ $s_V$ $p$) $\wedge$
    ($^\sigma$state $p \neq$ INACTIVE_STATE $\longrightarrow$ consisStateOutGeneral $\sigma$ $s_V$ $p$)
```

The definition is based on arbitrary process pointers p and distinguishes the particular operations by means of the concrete process states. The requirements for the particular operations are two-fold. In case of an IPC_SEND operation, for instance, predicate consisStateOutSEND fixes the abstract process output, whereas predicate ipcArgsSEND states the validity requirements for the according arguments. Similarily, we proceed in the other cases. The respective predicates are defined below.

IPC States and Process Outputs. For a process p residing in the state SEND_STATE, predicate consisStateOutSEND encapsulates the requirement that the corresponding process output denotes IPC_SEND:

```
consisStateOutSEND $\sigma$ $s_V$ $p$ $\equiv$
  **case** $\omega_{\mathsf{proc}}$ $\lceil s_V$.procs ($^\sigma$pid $p$)$\rceil$ **of**
  IPC_SEND $hn_{\mathsf{rcv}}$ $rights_{\mathsf{snd}}$ $msg$ $hn_{\mathsf{add}}$ $rights_{\mathsf{add}}$ $to_{\mathsf{snd}}$ $\Rightarrow$ True | _ $\Rightarrow$ False
```

In a similar way, we define predicate consisStateOutSENDRCV for processes in state SEND_RECEIVE_STATE. However, in case of an ordinary receive operation, we have to take two situations into account, as the definition of predicate consisStateOutRCV suggests:

```
consisStateOutRCV $\sigma$ $s_V$ $p \equiv$
  case $\omega_{proc}$ $\lceil s_V$.procs ($^{\sigma}$pid $p$)$\rceil$ of IPC_RECEIVE $hn_{snd}$ $buf$ $to_{rcv} \Rightarrow$ True
  | IPC_REQUEST $hn_{rcv}$ $rights_{snd}$ $msg$ $hn_{add}$ $rights_{add}$ $to_{snd}$ $buf$ $to_{rcv} \Rightarrow$
      $\lceil s_V$.sndstatdb ($^{\sigma}$pid $p$)$\rceil \wedge p \neq {}^{\sigma}$current_process
  | _ $\Rightarrow$ False
```

The state RECEIVE_STATE may denote an IPC_RECEIVE call but also the receive phase of an IPC_REQUEST operation. Predicate consisStateOutRCV takes both possibilities into account and holds, if the process output is either IPC_RECEIVE or IPC_REQUEST. In addition, however, the latter case requires an active send status and the process must be different from the currently running process.

Furthermore, the predicate consisStateOutGeneral encapsulates some general properties regarding the output of an active process p:

```
consisStateOutGeneral $\sigma$ $s_V$ $p \equiv$
  case $\omega_{proc}$ $\lceil s_V$.procs ($^{\sigma}$pid $p$)$\rceil$ of
  IPC_RECEIVE $hn_{snd}$ $buf$ $to_{rcv} \Rightarrow$
    $^{\sigma}$state $p \neq$ RECEIVE_STATE $\longrightarrow$ $^{\sigma}$pid $p \notin$ set $s_V$.schedds.wait
  | IPC_REQUEST $hn_{rcv}$ $rights_{snd}$ $msg$ $hn_{add}$ $rights_{add}$ $to_{snd}$ $buf$ $to_{rcv} \Rightarrow$
      $\lceil s_V$.sndstatdb ($^{\sigma}$pid $p$)$\rceil$ = ($^{\sigma}$state $p$ = RECEIVE_STATE)
  | _ $\Rightarrow$ True
```

As long as the implementation state of a process, performing an IPC_RECEIVE call, is not set to RECEIVE_STATE, the process is not part of the wait queue. Furthermore, the send status of a process performing an IPC_REQUEST call is active iff its state in the implementation denotes RECEIVE_STATE.

IPC Registers and Arguments Validity. Each IPC operation comes along with various input arguments determining, for instance, the desired communication partner or the message to be sent. Before the initiation of an IPC operation, the calling process has to specify these arguments in dedicated registers. The VAMOS kernel takes these values out of the registers and performs several validity checks. Erroneously specified or invalid arguments lead to the abortion of the operation. In case of no errors, the arguments are stored in dedicated components of the process control block of the calling process. The state of the process is set according to the desired operation and the pre-transmission phase is entered. In the correctness proof, however, we have to retain the situation that the arguments – stored in the process information blocks of processes in IPC states – fulfill certain properties.

The particular definitions of the predicates above are quite substantial and elaborate but all follow the same structure. Hence, we only cite parts of the definition of ipcArgsSEND as an example.

An implemenation state σ, an abstract VAMOS state s_V and a process pointer p identifying a process performing an IPC_SEND operation are the basis:

$$
\begin{array}{l}
\mathsf{ipcArgsSEND}\ \sigma\ s_{\mathsf{V}}\ p \Longrightarrow \\
\textbf{let}\ \mathit{hn}_{\mathsf{rcv}} = \lceil s_{\mathsf{V}}.\mathsf{procs}\ ({}^{\sigma}\mathsf{pid}\ p)\rceil.\mathsf{gprs}\ !\ 11; \\
\quad \mathit{rights}_{\mathsf{snd}} = \mathsf{to\_nat32}\ (\lceil s_{\mathsf{V}}.\mathsf{procs}\ ({}^{\sigma}\mathsf{pid}\ p)\rceil.\mathsf{gprs}\ !\ 12); \\
\quad \mathit{msg}_{\mathsf{ptr}} = \mathsf{to\_nat32}\ (\lceil s_{\mathsf{V}}.\mathsf{procs}\ ({}^{\sigma}\mathsf{pid}\ p)\rceil.\mathsf{gprs}\ !\ 13); \\
\quad \mathit{msg}_{\mathsf{len}} = \mathsf{to\_nat32}\ (\lceil s_{\mathsf{V}}.\mathsf{procs}\ ({}^{\sigma}\mathsf{pid}\ p)\rceil.\mathsf{gprs}\ !\ 14) \\
\textbf{in}\ \mathit{hn}_{\mathsf{rcv}} \notin \{\mathsf{HN\_SELF}, \mathsf{HN\_NONE}\} \wedge \\
\quad \lfloor {}^{\sigma}\mathsf{ipc\_pid}\ p \rfloor = \lceil s_{\mathsf{V}}.\mathsf{rightsdb}\ ({}^{\sigma}\mathsf{pid}\ p)\rceil.\mathsf{hdb}\ \mathit{hn}_{\mathsf{rcv}} \wedge \\
\quad {}^{\sigma}\mathsf{ipc\_rights}\ p = \mathit{rights}_{\mathsf{snd}} \wedge \\
\quad ({}^{\sigma}\mathsf{ipc\_rights}\ p \wedge_{\mathsf{u}} \neg_{\mathsf{u/32}}\ 15) = 0 \wedge \\
\quad {}^{\sigma}\mathsf{ipc\_snd\_msg}\ p = \mathit{msg}_{\mathsf{ptr}} \wedge {}^{\sigma}\mathsf{ipc\_snd\_len}\ p = \mathit{msg}_{\mathsf{len}} \wedge \mathsf{validMsg}\ \sigma\ p
\end{array}
$$

The definition first introduces various abbreviations for the values in the source registers. Register 11 stores the handle $\mathit{hn}_{\mathsf{rcv}}$ to the desired receiver and register 12 contains the rights $\mathit{rights}_{\mathsf{snd}}$ that should be granted to it. Furthermore, process p specifies the start address $\mathit{msg}_{\mathsf{ptr}}$ of the send message in register 13, followed by its length $\mathit{msg}_{\mathsf{len}}$ in register 14.

Based on these prerequisites, predicate ipcArgsSEND claims (apart from others) the following properties: The handle $\mathit{hn}_{\mathsf{rcv}}$ is valid, i. e., it is neither HN_SELF nor HN_NONE and it points to the desired IPC partner. In the implementation, the process-identifier of the latter is stored in ipc_pid p. Accordingly, an access with $\mathit{hn}_{\mathsf{rcv}}$ to the abstract handle database of p results in its abstracted process number. The rights $\mathit{rights}_{\mathsf{snd}}$, as well as the start address $\mathit{msg}_{\mathsf{ptr}}$ and the length $\mathit{msg}_{\mathsf{len}}$ of the send message are stored in the according components of the process information block. In addition, they fulfill the validity requirements. The rights follow the right format, i. e., $(\mathsf{ipc\_rights\ p} \wedge_{\mathsf{u}} \neg_{\mathsf{u/32}}\ 15) \neq 0$. Note that ipc_rights p is a natural numbers actually encoding a bit vector. Within this bit vector, it is only allowed to use the four least significant bits to express the desired combination of rights. In order to meet the correct format, the remaining bits must be **0**. Transferred to the natural number representation, the operation above reflects this requirement.

Furthermore, the send message is defined and available in the virtual memory of p. Predicate validMsg encapsulates this property:

$$
\begin{array}{l}
\mathsf{validMsg}\ \sigma\ p \equiv \\
\ ({}^{\sigma}\mathsf{ipc\_snd\_len}\ p \wedge_{\mathsf{u}} 3) = 0 \wedge \\
\ ({}^{\sigma}\mathsf{ipc\_snd\_len}\ p = 0 \vee \\
\ \ ({}^{\sigma}\mathsf{ipc\_snd\_msg}\ p \wedge_{\mathsf{u}} 3) = 0 \wedge \\
\ \ {}^{\sigma}\mathsf{ipc\_snd\_msg}\ p < {}^{\sigma}\mathsf{fip}\ p \ll_{\mathsf{u/32}} 12 \wedge \\
\ \ {}^{\sigma}\mathsf{ipc\_snd\_len}\ p + {}^{\sigma}\mathsf{ipc\_snd\_msg}\ p < {}^{\sigma}\mathsf{fip}\ p \ll_{\mathsf{u/32}} 12)
\end{array}
$$

The message length is a multiple of the word length (measured in bytes), and the start address is word-aligned as long as the length is not 0. In case of the latter, the message resides within the boundaries of the virtual memory. The value fip p denotes the first invalid memory page, whereas each page consists of PAGE_SIZE $= 2^{12}$ words. We obtain the first invalid address by shifting fip p by 12 to the left. Accordingly, the start address as well as the end address of the message are smaller than the first invalid address.

The definitions of the predicates ipcArgsSENDRCV and ipcArgsRCV are quite similar. The former is based on ipcArgsSEND, but additionally demands some requirements regarding the receive phase which can also be found in the latter.

5.2.4 Summing Up

The predicate implInv combines the properties and requirements from above and establishes the Vamos implementation invariant:

$\mathsf{implInv}\ s\ s_{\mathsf{V}} \equiv$
$\mathsf{valueBounds_v}\ s \wedge \mathsf{validVMs_v}\ s \wedge \mathsf{validCup_v}\ s \wedge \mathsf{validRights_v}\ s \wedge \mathsf{uniqueProcId_v}\ s \wedge$
$\mathsf{validLists_v}\ s \wedge \mathsf{validDevds_v}\ s \wedge \mathsf{generalProps}\ s\ s_{\mathsf{V}} \wedge \mathsf{waitProps}\ s\ s_{\mathsf{V}} \wedge$
$\mathsf{ipcProps}\ s\ s_{\mathsf{V}}$

The theory file `vamosAbstractions.thy` contains the complete formal definitions of the Vamos implementation invariant.

5.3 The Simulation Theorem

The overall correctness statement – we are aiming at in this thesis – applies to a complete kernel step, i. e., the Vamos kernel embedded into the Cvm framework. An illustration of this statement was already given in Figure 5.1. The proof of this statement shows the simulation between the implementation and the model. We express this statement as a Hoare triple.

Theorem 5.3.1 (Kernel Correctness) *This theorem actually represents two cases: a reset and a normal system step. A reset is signaled by an active* reset *and causes the initialization of the datastructures leading to the state τ. On the abstract level, the initialization of the* Vamos *kernel is described by the function* initialConf*. This initial state together with the devices s_{D}' and the implementation state τ fulfill the abstraction relation and the implementation invariant.*

Apart from the reset case, let σ be the implementation state before and τ the one after the application of function system_step*. Furthermore, (s_{V}, s_{D}) denotes an abstract state and (s_{V}', s_{D}') its adjacent one after applying $\delta_{\mathsf{V+D}}$. Whenever the abstraction relation $\mathsf{Abs}_{\mathsf{V+D}}$ holds between σ and (s_{V}, s_{D}) and* implInv *holds before the invocation of* system_step*, $\mathsf{Abs}_{\mathsf{V+D}}$ holds between τ and (s_{V}', s_{D}') and* implInv *is preserved.*

In short, the transition function $\delta_{\mathsf{V+D}}$ models the system execution system_step.

```
$\Gamma \vdash_t \{\sigma.\ {}^{\sigma}\mathsf{reset} \vee \mathsf{Abs}_{\mathsf{V+D}}\ \sigma\ (s_{\mathsf{V}}, s_{\mathsf{D}}) \wedge \mathsf{implInv}\ \sigma\ s_{\mathsf{V}}\}$ PROC system_step()
    $\{\tau.$ let $(s_{\mathsf{V}}', s_{\mathsf{D}}') =$
            if ${}^{\sigma}\mathsf{reset}$
            then (initialConf (getOSimg $s_{\mathsf{D}}$) OS_PAGES,
                  init_dev (cvm_devs ${}^{\sigma}\mathsf{cvmX}$))
            else fst ($\delta_{\mathsf{V+D}}\ \perp\ (s_{\mathsf{V}}, s_{\mathsf{D}})$)
       in $\mathsf{Abs}_{\mathsf{V+D}}\ \tau\ (s_{\mathsf{V}}', s_{\mathsf{D}}') \wedge \mathsf{implInv}\ \tau\ s_{\mathsf{V}}'\}$
```

Recall that the transition function $\delta_{\mathsf{V+D}}$ uses its input parameter to determine whether the kernel or solely the device subsystem (e. g., by receiving a network package) advance during a transition of the overall system. If this input is not $\perp$, the kernel remains constant. Knowing that the abstraction relation of the kernel data structures and the one of the device subsystem are independent, we may consequently disregard changes that are limited to the device subsystem in the consideration of the kernel simulation. While we are interested in the kernel-device interaction, we disregard any external device input from the environment and the according output to it. As a consequence, this input is fixed at $\perp$ in the theorem above. Furthermore, the transition function $\delta_{\mathsf{V+D}}$ returns a pair of the updated state and the external output. We are, however, only interested in the updated state and thus take the first component of this pair, i. e., $(s_{\mathsf{V}}', s_{\mathsf{D}}') = \mathsf{fst}\ (\delta_{\mathsf{V+D}}\ \perp\ (s_{\mathsf{V}}, s_{\mathsf{D}}))$.

Proof Sketch The correctness of the simulation theorem is based on the correctness of all functions applied in the scope of system_step. Thus, for all functions we have to define and prove lemmata stating their correctness. Eventually, we connect these lemmata in order to prove the correctness of the overall system. The proof of the theorem mainly relies on the correctness of the function dispatcher_kernel representing the main function of the VAMOS implementation. The correctness of dispatcher_kernel, in turn, depends on the correctness of its four different parts: the initialization, the handling of process exceptions, the timer-interrupt handler and the delivery of device interrupts.

We have not completed the proof of the whole theorem but mayor parts of it. The next chapter presents the already proven ones.

Chapter 6

Implementation Correctness

Contents

The C0 implementation of the VAMOS kernel is based on the CVM primitives provided by the CVM framework. Apart from that, the implementation relies on a library of functions on doubly-linked lists which was specified and verified in the Hoare logic against its C0 implementation [59, 71].

Except the functions of the list library and the CVM primitives, Figure 6.1 depicts the call graph of the VAMOS implementation.

The functions highlighted in green are completely verified, the correctness of the ones in yellow is shown but the proof relies on specifications of sub-functions which are not yet proven.

Often recurring operations are encapsulated in auxiliary functions. Subsequently, we briefly introduce their functionalities and specifications. Based on these functions, we proceed with the correctness statements for the timer-interrupt handler handle_timer and the trap handler which is implemented in function dispatcher_kernel_call. Under the assumption that the remaining parts, i. e., the initialization, the interrupt delivery and the termination of processes are correctly implemented, we formulate and prove the correctness of the VAMOS top-level function dispatcher_kernel. Finally, the VAMOS correctness statement is integrated in function system_step and we prove the overall correctness statement as formulated in Theorem 5.3.1.

The correctness statement of the VAMOS kernel relies on a slightly adjusted abstraction relation which takes the invocation context of function

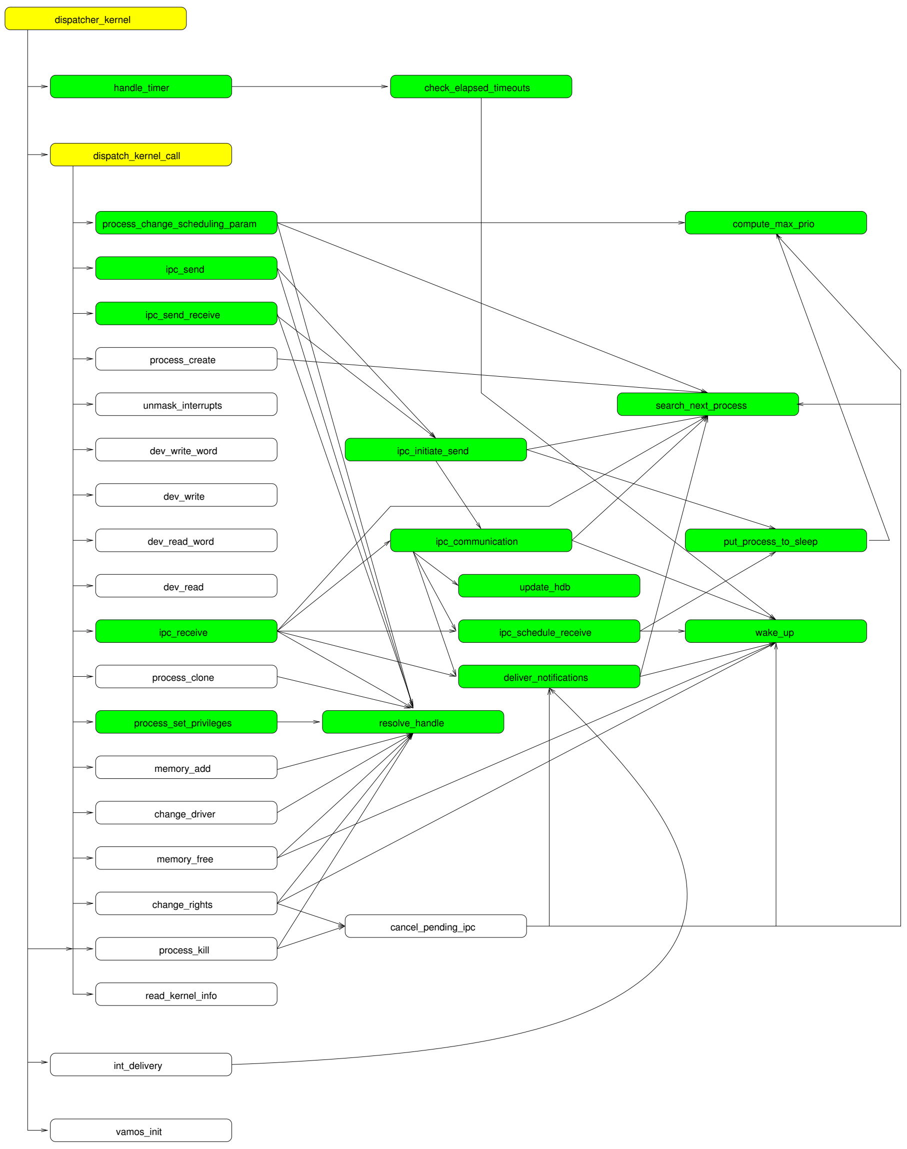

Figure 6.1: Call Graph of the VAMOS Implementation

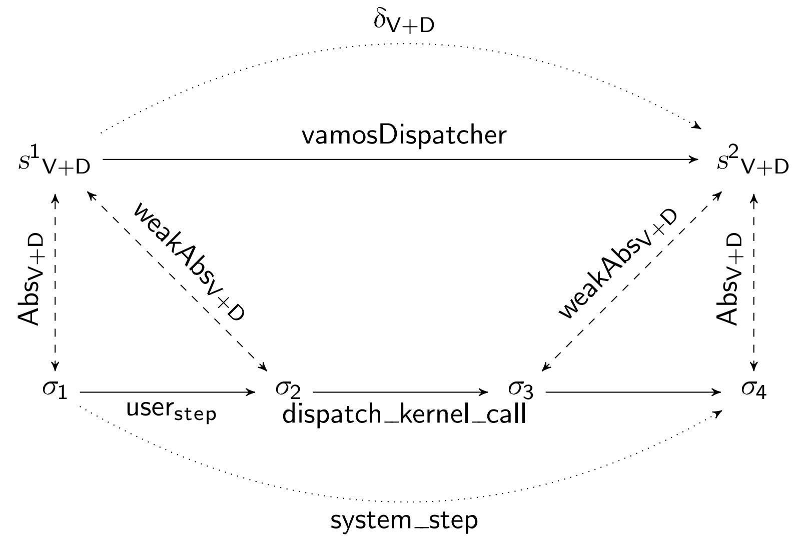
$$\begin{tikzcd}
s^1{}_{\mathsf{V+D}} \arrow[rrr, "\mathsf{vamosDispatcher}"] \arrow[rrr, dotted, bend left, "\delta_{\mathsf{V+D}}"] \arrow[d, dashed, leftrightarrow, "\mathsf{Abs}_{\mathsf{V+D}}"'] \arrow[dr, dashed, leftrightarrow, "\mathsf{weakAbs}_{\mathsf{V+D}}"] & & & s^2{}_{\mathsf{V+D}} \arrow[d, dashed, leftrightarrow, "\mathsf{Abs}_{\mathsf{V+D}}"] \arrow[dl, dashed, leftrightarrow, "\mathsf{weakAbs}_{\mathsf{V+D}}"'] \\
\sigma_1 \arrow[r, "\mathsf{user}_{\mathsf{step}}"'] \arrow[rrr, dotted, bend right, "\mathsf{system\_step}"'] & \sigma_2 \arrow[r, "\mathsf{dispatch\_kernel\_call}"'] & \sigma_3 \arrow[r] & \sigma_4
\end{tikzcd}$$

Figure 6.2: Data abstraction in the case of trap handling

dispatcher_kernel into account. Thus, we first introduce this weakening of the abstraction relation in the next section and proceed than as described above.

6.1 Weakening the Abstraction Relation

The weakening of the abstraction relation $\mathsf{Abs}_{\mathsf{V+D}}$ only affects the abstraction of the virtual machines. As mentioned in Section 5.1.1, the reason for this is the different handling of the user-generated exceptions in the implementation and the model.

In general, the user-generated exception handling in the concrete system happens as follows: The current process computes on the VAMP processor and executes an instruction. Executing this instruction triggers an exception and the VAMP calls the interrupt service routine (ISR). The ISR passes the exception on to the VAMOS kernel which – on the basis of the exception cause and data – handles this exception. Thus, the current process runs into the exception which is then handled in a sequential way.

The VAMOS model, in contrast, abstracts from the sequential exception handling and combines it into one step. Thereby, the main difference is that the instruction causing the exception is not executed right at the beginning in order to determine any exceptions. The model rather considers the corresponding process output by means of the function ω_{proc} which inspects the next instruction of the current process before the actual execution. If the execution of the instruction would lead to an exception, the model handles the exception in advance and delivers the result in the same moment as it executes the instruction. Thus, the virtual machine is kept in the original state as long as the exception handling is not completed.

We exemplify the different approaches by the trap handling and explain by this example, how we adjust the abstraction of the virutal machines to cope with this situation in the proofs.

Figure 6.2 illustrates the situation on the concrete and abstract levels. In our setting, the kernel-visible effects of the ISR are described by function system_step. It executes the instruction of the current process by means of $\mathsf{user}_{\mathsf{step}}$ and calls function dispatcher_kernel which implements the VAMOS kernel. As we only consider the trap handling in this case, the invocation of the VAMOS kernel mainly boils down to the function dispatch_kernel_call. The model represents a kernel step by means of the transition function $\delta_{\mathsf{V+D}}$. As in the implementation, this function results in calling vamosDispatcher.

In our example, the states σ_1 and $s^1{}_{\mathsf{V+D}}$ are correlated with each other via the abstraction relation $\mathsf{Abs}_{\mathsf{V+D}}$ and represent the situation before the actual kernel step. In the concrete system, the TRAP instruction is executed by means of function $\mathsf{user}_{\mathsf{step}}$ and function dispatch_kernel_call is invoked in state σ_2. Executing the TRAP instruction in the concrete system does not yet have a counterpart in the model, i. e., the model still resides in state $s^1{}_{\mathsf{V+D}}$.

As we want to use function vamosDispatcher for describing the semantic effect of the function dispatch_kernel_call, we establish a weak abstraction relation $\mathsf{weakAbs}_{\mathsf{V+D}}$ between σ_2 and $s^1{}_{\mathsf{V+D}}$. It is pretty similar to the overall abstraction relation $\mathsf{weakAbs}_{\mathsf{V+D}}$ but certainly takes the differing virtual machine states of the current process into account.

The latter is reflected in the weak abstraction relation $\mathsf{weak\_rel\_procs}_{\mathsf{v}}$ for the virtual user machines:

```
$\mathsf{weak\_rel\_procs}_{\mathsf{v}}\ \sigma\ ptr_{\mathsf{cp}}\ diff\ inactQ\ waitQ\ uprocs \equiv$
  $\forall x.$ if $x \in \mathsf{set}\ inactQ$ then $uprocs\ x = \bot$
      else if $x \in \mathsf{set}\ waitQ$
          then $\mathsf{incPcs}\ \lceil uprocs\ x \rceil = (\mathsf{cvm\_ups}\ {}^{\sigma}\mathsf{cvmX}).\mathsf{userprocesses}\ x\ \wedge$
              $uprocs\ x \neq \bot$
          else if $diff \wedge ptr_{\mathsf{cp}} \neq \mathsf{Null} \wedge x = {}^{\sigma}\mathsf{pid}\ ptr_{\mathsf{cp}}$
              then $\mathsf{user}_{\mathsf{step}}\ \lceil uprocs\ x \rceil = (\mathsf{cvm\_ups}\ {}^{\sigma}\mathsf{cvmX}).\mathsf{userprocesses}\ x\ \wedge$
                  $uprocs\ x \neq \bot$
              else $uprocs\ x = \lfloor (\mathsf{cvm\_ups}\ {}^{\sigma}\mathsf{cvmX}).\mathsf{userprocesses}\ x \rfloor$
```

Virtual machines of inactive and waiting processes are abstracted in the same way as with $\mathsf{rel\_procs}_{\mathsf{v}}$. Expressing the differing virtual machine states of the current process is made possible by two additional arguments: a pointer ptr_{cp} to the current process at kernel entry and a flag *diff* denoting whether its virtual machine states differ. If a current process existed, i. e., $ptr_{\mathsf{cp}} \neq \mathsf{Null}$ and, in addition, flag *diff* is active, the virtual machine of the current process in the model is the one that after the application of $\mathsf{user}_{\mathsf{step}}$ corresponds with its machine in the implementation. In all other cases, the virtual machines are identical, again.

The relation weak_rel_procs$_\mathsf{v}$ is also applicable in the postcondition of dispatch_kernel_call in order to relate the virtual machines in the implementation state σ_3 and the ones in $s^2{}_{\mathsf{V+D}}$. If the trap handling succeeds, function dispatch_kernel_call writes the corresponding result into the result register of the current process. In the model, function vamosDispatcher also delivers the result and executes the TRAP instruction. As a consequence, the states of the virtual machines of the current process are in-sync again. The same applies, if the trap handling failed and the current process received an error message.

In the remaining cases, the trap handling is not yet finished. For instance, if the current process triggered an IPC operation but no suitable communication partner was available. As it has not received an error message, it is apparently willing to wait. Accordingly, it is put into the wait queue. For members of the wait queue, predicate weak_rel_procs$_\mathsf{v}$ establishes the corresponding relation between the virtual machine states.

In a similar way, we can use weak_rel_procs$_\mathsf{v}$ in the other application areas, i. e., in the pre- and postconditions of the VAMOS scheduler or the interrupt delivery. These entities, however, also involve the remaining datastructures. Thus, we define a weak abstraction relation weakAbs$_\mathsf{V+D}$:

weakAbs$_\mathsf{V+D}$ σ *cup pendingStep* (s_V, s_D) $\equiv$
  cvm_devs ${}^\sigma$cvmX $=$ s_D $\wedge$
  weak_rel_procs$_\mathsf{v}$ σ *cup pendingStep* s_V.schedds.inactive s_V.schedds.wait
   s_V.procs $\wedge$
  rel_schedds$_\mathsf{v}$ σ s_V.schedds $\wedge$
  rel_priodb$_\mathsf{v}$ σ s_V.priodb s_V.schedds.inactive $\wedge$
  rel_sndstatdb$_\mathsf{v}$ σ s_V.procs s_V.sndstatdb $\wedge$
  rel_rightsdb$_\mathsf{v}$ σ s_V.schedds.inactive s_V.rightsdb $\wedge$ rel_devds$_\mathsf{v}$ σ s_V.devds

It is similar to Abs$_\mathsf{V+D}$ but uses weak_rel_procs$_\mathsf{v}$ instead of rel_procs$_\mathsf{v}$ to abstract the virtual machines. In addition, it does not enforce the equivalence between the current process identifiers of CVM and VAMOS.

6.2 Auxiliary Functions

While specifying the auxiliary functions, σ denotes the implementation state before the invocation and τ the one after the execution. We do not describe the complete Hoare triples but only the semantic effect of the particular functions. The complete formal definitions of the pre- and postconditions comprise many technical details. For instance, in most cases, we cannot rely on the complete implementation invariant but only on (adjusted) parts of it. Furthermore, the postconditions contain information on how certain variables have changed. Thus, a more detailed inspection would be tedious and result rather in confusion than clarification. For this reason, we only focus on the essential aspects. However, the formal definitions of the pre- and postconditions together with the corresponding Hoare triples and the functional correctness proofs can be found in the provided theory files.

Resolving Handles. The function resolve_handle implements the mapping from process-local handles to process-identifiers. Invoked by a process p, it resolves the given handle hn in p's context and returns the corresponding process-identifier pid. The specification basically relies on the assumption that $\mathsf{rel\_rightsdb_v}$ establishes a relation between the concrete rights datastructures in σ and the abstract ones $rightsdb$. Return values $pid \in \{\mathsf{0}, \mathsf{PID\_MAX}\}$ indicate that handle hn is invalid in the context of p. On the abstract level, this situation is expressed by $\lceil rightsdb\ ({}^{\sigma}\mathsf{pid}\ p)\rceil.\mathsf{hdb}\ hn = \bot$ and $\neg$ valid_handle $rightsdb\ ({}^{\sigma}\mathsf{pid}\ p)\ hn$. The handle hn is valid, if pid denotes a process-identifier. Accordingly, the entry in the abstract handle database of p stores the corresponding process number, i. e., $\lceil rightsdb\ ({}^{\sigma}\mathsf{pid}\ p)\rceil.\mathsf{hdb}\ hn = \lfloor pid\rfloor$.

The theory file `vamosResolveHandle.thy` contains the corresponding Hoare triple together with the functional correctness proof. Proving the functional correctness of function resolve_handle is rather simple which is reflected by only around 60 proof steps.

Putting processes to sleep and waking them up. The task of putting a process to sleep is encapsulated in function put_process_to_sleep and function wake_up is used to wake up processes. At the very beginning, the specifications rely on the assumption that the concrete scheduling lists in σ can be related to their abstract counterparts via Queue. In both cases, the ready queues $RdyQs$ and the wait queue $waitQ$ are used to describe the semantic effect. Furthermore, there is the abstract priority database $priodb$ which is obtained by $\mathsf{rel\_priodb_v}$.

Putting a process p to sleep involves its removal from the ready queue and its appending to the wait queue. The former is described by $RdyQs(prio_{\mathsf{p}} := [x{\in}RdyQs\ prio_{\mathsf{p}}\ .\ x \neq {}^{\sigma}\mathsf{pid}\ p])$, where $prio_{\mathsf{p}} = \lceil priodb\ ({}^{\sigma}\mathsf{pid}\ p)\rceil$. Updating the wait queue is also straightforward: $WkpQ$ @ $[{}^{\sigma}\mathsf{pid}\ p]$. The function put_process_to_sleep does not set the timeout value of p. Apart from that, its semantic complies with function v_ipc_wait_updt_schedds.

The theory file `vamosPutProcessToSleep.thy` contains the Hoare triple for put_process_to_sleep together with the functional correctness proof. The most effort is necessary to establish the invariants regarding the wait and the ready lists. In total, the proof comprises around 570 steps.

The awakening of a process p involves its removal from $waitQ$ and its appending to $RdyQs\ prio_{\mathsf{p}}$. Function wake_up, however, does not reset the consumed time to 0. Nevertheless, apart from the latter, the semantic effect on the ready and wait queues complies with the one described by v_wkp_updt_schedds.

The theory file `vamosWakeUp.thy` contains the Hoare triple for wake_up together with the functional correctness proof. Again, the most effort is necessary to establish the invariants regarding the wait and the ready lists. The proof comprises ca. 640 steps.

Computing the current maximum priority. As the call graph illustrates, putting a process to sleep entails the re-computation of the maximal priority current_max_prio by the function compute_max_prio. Based on abstract ready queues *RdyQs*, the result of this computation can be expressed as follows: $^{\tau}\mathsf{current\_max\_prio} = \mathsf{Max}\ (\{i.\ i < \mathsf{PRIOCNT} \wedge s_{\mathsf{V}}.\mathsf{schedds.ready}\ i \neq []\} \cup \{0\})$.

The theory file `vamosComputeMaxPrio.thy` contains the corresponding Hoare triple together with the functional correctness proof. Proving the functional correctness is rather straightforward and the proof only requires 60 steps.

Searching the next process. Usually, the awakening of a process or putting it to sleep requires the re-computation of the current process current_process. The function search_next_process encapsulates this task. Its specification relies on the assumption that, at the beginning, ready queues *RdyQs* can be abstracted from the implementation state σ via Queue. Setting current_process to Null is equivalent with the statement that the ready queue of the highest priority is empty: $(^{\tau}\mathsf{current\_process} = \mathsf{Null}) = (RdyQs\ ^{\sigma}\mathsf{current\_max\_prio} = [])$. If $^{\tau}\mathsf{current\_process} \neq \mathsf{Null}$, the corresponding process number denotes the head of the ready queue with the highest priority, i. e., $^{\tau}\mathsf{pid}\ ^{\tau}\mathsf{current\_process} = \mathsf{hd}\ (RdyQs\ ^{\sigma}\mathsf{current\_max\_prio})$.

The theory file `searchNextProcess.thy` contains the Hoare triple together with the functional correctness proof which comprises around 30 steps.

6.3 Correctness of the Timer-Interrupt Handler

The implementation correctness of the timer-interrupt handler was already briefly covered in [27][1] and lays the foundation for the fairness proof of the Vamos scheduler [27, 25].

The Vamos scheduler is implemented in function handle_timer (see Figure 6.3), which is called by dispatcher_kernel, whenever a timer interrupt

[1] A very early attempt of showing the implementation correctness of the timer-interrupt handler presents [19]. However, we only mention this work for the sake of completeness. On the one hand, the specification of the Vamos scheduler has significantly changed in the course of time and, on the other hand, the work only presents rudimentary approaches regarding the actual functional verification.

occurs. Recall that the scheduler might be invoked after the trap handling which might change the variable current_process. In order to charge the correct process, we store the pointer current_process at the very beginning in variable old_cup and provide this variable as input to function handle_timer which performs the following tasks:

- acknowledge the timer interrupt by reading a word from the timer device by calling cvm_in_word, such that the latter may lower its interrupt line.
- if there exist finite timeouts, i. e., next_timeout $\neq$ INFINITE_TIMEOUT, increase the current time and check for elapsed timeouts. This check is encapsulated in function check_elapsed_timeouts, which traverses the wait queue, wakes up all processes with elapsed timeouts, and subtracts the current time from all other timeouts. Finally, the current time is set to zero which avoids the overflow of the current_time variable.
- charge the process old_cup that computed in the last step, if it is still ready. If the so far consumed time of old_cup is greater than or equal to its timeslice, the process is moved to the end of its ready list. Therefore, it is first deleted from the list and then appended again. In addition, the consumed time is set to zero. Otherwise, the consumed time is increased by one.
- elect the process that runs in the next kernel step by means of the function search_next_process, which returns the first element in the ready list with the current maximum priority.

The functional correctness of the timer-interrupt handler mainly relies on the functional correctness of check_elapsed_timeouts. Thus, we first briefly describe the proof of the latter and continue afterwards with the correctness theorem of the timer-interrupt handler.

6.3.1 Implementation Correctness of check_elapsed_timeouts

As with the auxiliary functions, we do not present the complete formal specification of function check_elapsed_timeouts but only describe its semantic effect. For more details, we refer to the corresponding theory file.

The semantic effect of function check_elapsed_timeouts only affects the virtual machines and the scheduling datastructures. Accordingly, the proof is based on the assumption that the relations $\mathsf{rel\_procs}_\mathsf{v}$ and $\mathsf{rel\_schedds}_\mathsf{v}$ can be established between the concrete and abstract datastructures. Note that the precondition cannot assume the complete implementation invariant implInv but only a part of it together with several adjusted properties. Recall that check_elapsed_timeouts is applied after increasing the time

```
procedures handle_timer(old_cup | res_int) =
  (* acknowledge the timer interrupt *)
  'dummy :== CALL_g cvm_in_word(DEVICE_TIMER, 0);

  (* detect and handle elapsed IPC timeouts *)
  IF_g 'next_timeout $\neq$ INFINITE_TIMEOUT THEN
    'current_time :==_g 'current_time + 1
  FI;
  IF_g 'current_time $\geq$ 'next_timeout THEN
    'dummy_i :== CALL_g check_elapsed_timeouts()
  FI;

  (* charge the process that computed in the last step *)
  IF_g 'old_cup $\neq$ Null $\wedge$ 'old_cup $\rightarrow$ 'state = READY_STATE  THEN
    IF_g 'old_cup$\rightarrow$'consumed_time $\geq$ 'old_cup$\rightarrow$'timeslice THEN
      'ready_lists ! ('old_cup$\rightarrow$'priority) :==
              CALL_g queueDelete('ready_lists ! ('old_cup$\rightarrow$'priority), 'old_cup);
      'ready_lists ! ('old_cup$\rightarrow$'priority) :==
              CALL_g queueAppend('ready_lists ! ('old_cup$\rightarrow$'priority), 'old_cup);
      'old_cup$\rightarrow$'consumed_time :==_g 0
    ELSE
      'old_cup$\rightarrow$'consumed_time :==_g 'old_cup$\rightarrow$'consumed_time + 1
    FI
  FI;

  (* select the process that runs in the next step *)
  'dummy_i :== CALL_g search_next_process();
  'res_int :==_g 0
```

Figure 6.3: Function handle_timer

in function handle_timer, which may lead to a situation where, e. g., the time is no longer smaller than MAX_TIME. Thus, in the precondition for check_elapsed_timeouts, we have to adjust, among other things, the time-related properties of implInv, accordingly. The same applies for the post-condition. For instance, we cannot establish the complete validity requirements for ready lists, where the head of the ready list of the highest priority determines the current process. Waking up processes, as it happens in check_elapsed_timeouts could violate this property, because the highest priority might change. This violation is not resolved until the point, where function search_next_process is called in handle_timer. Apart from that the post-condition also states the semantic effect of calling check_elapsed_timeouts. Both, the virtual machines and the scheduling datastructures, are updated in the way as described by the model function checkTimeouts. The Hoare triple of checkTimeouts and the formal functional correctness proof can be found in the theory file `vamosCheckElapsedTimeouts_proof.thy`. Due to the traversing of the wait queue and the awakening of processes, showing the functional correctness of function checkTimeouts is pretty elaborate which is reflected in almost 4700 proof steps.

6.3.2 Implementation Correctness of handle_timer

For a better reusability in the correctness proof of function dispatcher_kernel, we encapsulate the pre- and postconditions of function handle_timer in predicates.

The predicate $\mathsf{handleTimer}_{\mathsf{pre}}$ formulates the precondition:

$$\begin{aligned}&\mathsf{handleTimer}_{\mathsf{pre}}\ \sigma\ \mathit{old\_cup}\ \mathit{diff}\ s_{\mathsf{V}}\ s_{\mathsf{D}} \equiv \mathsf{weakAbs}_{\mathsf{V+D}}\ \sigma\ \mathit{old\_cup}\ \mathit{diff}\ (s_{\mathsf{V}}, s_{\mathsf{D}}) \wedge\\ &\quad \mathsf{implInv}\ \sigma\ s_{\mathsf{V}} \wedge \mathit{old\_cup} = \mathsf{Null} \vee \mathit{old\_cup} \in \mathsf{set}\ {}^{\sigma}\mathsf{pib} \wedge\\ &\quad (\mathit{old\_cup} \neq \mathsf{Null} \wedge \mathit{diff} \longrightarrow {}^{\sigma}\mathsf{pid}\ \mathit{old\_cup} \notin \mathsf{set}\ s_{\mathsf{V}}.\mathsf{schedds.wait})\end{aligned}$$

Relation $\mathsf{weakAbs}_{\mathsf{V+D}}$ depends on *old_cup* and *diff*, whereas implInv is stated as usual. The *old_cup* is either Null or denotes a pointer to a process information block. Furthermore, if a current process *old_cup* has been apparent at kernel entry and flag *diff* is active, the process is not waiting. This statement actually reflects the situation that *old_cup* has neither triggered a runtime-error nor a trap exception but performed a process-local transition. Accordingly, the timer-interrupt handler is the first function which is called in the VAMOS kernel and the virtual machines of *old_cup* still differ. This property allows to exclude process *old_cup* from the check for elapsed timeouts and is thus passed on as a precondition of check_elapsed_timeouts.

The postcondition is stated by the predicate $\mathsf{handleTimer}_{\mathsf{post}}$:

$$\begin{aligned}&\mathsf{handleTimer}_{\mathsf{post}}\ \tau\ \sigma\ \mathit{old\_cup}\ \mathit{diff}_{\mathsf{cp}}\ s_{\mathsf{V}}\ s_{\mathsf{D}} \Longrightarrow\\ &\mathbf{let}\ s_{\mathsf{D}}{}' = \mathsf{fst}\ (\delta_{\mathsf{Dint}}\ (\mathsf{read}_{\Omega\mathsf{V}}\ \mathsf{DEV\_TIMER}\ 0\ 1)\ s_{\mathsf{D}});\\ &\qquad s_{\mathsf{V}}{}' = \mathsf{handleTimer}\ s_{\mathsf{V}}\ (\mathbf{if}\ \mathit{old\_cup} = \mathsf{Null}\ \mathbf{then}\ \bot\ \mathbf{else}\ \lfloor {}^{\sigma}\mathsf{pid}\ \mathit{old\_cup} \rfloor)\\ &\mathbf{in}\ \mathsf{weakAbs}_{\mathsf{V+D}}\ \tau\ \mathit{old\_cup}\ \mathit{diff}_{\mathsf{cp}}\ (s_{\mathsf{V}}{}', s_{\mathsf{D}}{}') \wedge \mathsf{implInv}\ \tau\ s_{\mathsf{V}}{}'\end{aligned}$$

The abstract device state $s_D{}'$ reflects the acknowledgement of the timer interrupt by reading one word from port 0 and function handleTimer computes the adjacent VAMOS state $s_V{}'$. Similar to handle_timer, it gets the identifier of the current process at kernel entry. This identifier denotes $\perp$, if no current process was apparent at kernel entry, i. e., *old_cup* = Null. Otherwise, it delivers the corresponding abstract process number. Again, $\mathsf{weakAbs}_{V+D}$ and implInv are preserved.

Note that the postcondition comprises also statements that some datastructures are not changed during execution. This information is rather technical and thus omitted in this presentation.

With the assumption that the CVM primitive cvm_in_word, the list library, and the functions check_elapsed_timeouts and search_next_process are implemented correctly, we finally can prove the correctness of the function handle_timer:

Theorem 6.3.1 (Correctness of the Timer-Interrupt Handler) *The semantic effect of the function* handle_timer *can be described by* handleTimer *and the execution preserves the abstraction relation* $\mathsf{weakAbs}_{V+D}$ *and the implementation invariant* implInv.

$$
\begin{array}{l}
\forall \sigma\ \mathit{diff}\ s_V\ s_D.\\
\quad \Gamma \vdash_t \{\!|\sigma.\ \mathsf{handleTimer_{pre}}\ \sigma\ {}^{\sigma}\mathsf{old\_cup}\ \mathit{diff}\ s_V\ s_D|\!\}\\
\qquad\quad {}'\mathsf{res\_int} :== \mathbf{PROC}\ \mathsf{handle\_timer}({}'\mathsf{old\_cup})\\
\qquad\quad \{\!|\tau.\ \mathsf{handleTimer_{post}}\ \tau\ \sigma\ {}^{\sigma}\mathsf{old\_cup}\ \mathit{diff}\ s_V\ s_D|\!\}
\end{array}
$$

Proof The proof of this function involves at the one hand, a case split over the three cases distinguished in the implementation. Hence, we show that the charging policy is correctly implemented with regard to the specification function charge. At the other hand, we have to establish the precondition for check_elapsed_timeouts. With this prerequisite in place, we use its specification checkTimeouts to establish the claim of our theorem. Additionally, the state invariant implInv is again recovered. The Hoare triple of function handle_timer and the functional correctness proof can be found in theory file `vamosHandleTimer_proof.thy`. Almost 4900 proof steps are necessary to show the latter. □

6.4 Correctness of the Trap Handler

The function dispatch_kernel_call represents the VAMOS trap handler in the implementation. Whenever the current process has triggered a trap exception, it is called by the VAMOS kernel and provided with the trap number *trapnr* as input. Based on this trap number, dispatch_kernel_call implements a case distinction, reads the arguments out of the registers of the current process and executes the desired kernel call. For example, the trap number 6 is associated with the VAMOS call SET_PRIVILEGES, which is realized by the function set_privileges. Except for the IPC calls, the dedicated functions return with a result value, which is written into the result register of the current process. The result delivery in case of an IPC call is more elaborate and accomplished within the according functions. Nevertheless, writing the result value completes the trap handling and dispatch_kernel_call returns to dispatcher_kernel.

The implementation correctness of dispatch_kernel_call involves the implementation correctness of the particular VAMOS calls. Subsequently, we sketch the specifications and proofs of the already proven ones and combine them to the correctness statement of the VAMOS trap handler, later on.

6.4.1 Implementation Correctness of Function set_privileges

The VAMOS trap handler calls function set_privileges, if the trap number denotes 6. As input parameter, it provides a handle hn, which is taken out of the register 11 of the current process.

Functionality. In a first step, function set_privileges resolves the handle hn by means of function resolve_handle. If the return value pid denotes a process-identifier and the current process is privileged, it activates the privileged status of process pid. In addition, it signals the successful operation by setting the return value to 0. The return value is set to an error code, if either the current process is not privileged or pid does not denote a valid process identifier, i. e., hn is invalid.

Precondition. Predicate $\mathsf{setPrivileges_{pre}}$ describes the precondition and must be discharged in the context of dispatcher_kernel_call, whenever it calls the function set_privileges:

$\mathsf{setPrivileges_{pre}}\ \sigma\ hn\ s_\mathsf{V}\ s_\mathsf{D} \equiv$
$\mathsf{weakAbs_{V+D}}\ \sigma\ {}^\sigma\mathsf{current\_process}\ \mathsf{True}\ (s_\mathsf{V},\ s_\mathsf{D}) \wedge \mathsf{implInv}\ \sigma\ s_\mathsf{V} \wedge$
${}^\sigma\mathsf{current\_process} \neq \mathsf{Null} \wedge 0 \leq hn < 2^{32} \wedge$
$(\exists hn.\ \omega_\mathsf{proc}\ \lceil s_\mathsf{V}.\mathsf{procs}\ \lceil \mathsf{v\_cup}\ s_\mathsf{V}.\mathsf{schedds}\rceil\rceil = \textsc{set\_privileged}\ hn)$

The implementation state σ together with the abstract states s_V and s_D fulfill the abstraction relation $\mathsf{weakAbs_{V+D}}$ and the state invariant implInv. As a trap instruction was triggered, it is further assumed that the current process exists, i. e., current_process $\neq$ Null. This also involves that the flag *diff* in $\mathsf{weakAbs_{V+D}}$ is set to True. In addition, hn denotes a 32-bit natural number and the process output of the current process denotes SET_PRIVILEGED.

Postcondition. The predicate $\mathsf{setPrivileges_{post}}$ encapsulates the postconditions of function set_privileges:

$\mathsf{setPrivileges_{post}}\ \tau\ \sigma\ res\ hn\ s_\mathsf{V}\ s_\mathsf{D} \Longrightarrow$
let $s_\mathsf{D}{}' = s_\mathsf{D};\ s_\mathsf{V}{}' = \mathsf{vamos\_set\_privileges\_inter}\ s_\mathsf{V}\ (\mathsf{to\_int32}\ hn)$

$$\begin{array}{l}
\textbf{in } res = \\
\quad \mathsf{result\_value\_int} \\
\quad\; (\mathsf{vamos\_result\_set\_privileges}\ s_{\mathsf{V}}.\mathsf{rightsdb}\ \lceil \mathsf{v\_cup}\ s_{\mathsf{V}}.\mathsf{schedds}\rceil\ (\mathsf{to\_int32}\ hn)) \wedge \\
\quad \mathsf{weakAbs}_{\mathsf{V+D}}\ \tau\ {}^{\sigma}\mathsf{current\_process}\ \mathsf{True}\ (s_{\mathsf{V}}{}', s_{\mathsf{D}}{}') \wedge \mathsf{implInv}\ \tau\ s_{\mathsf{V}}{}'
\end{array}$$

It denotes the implementation state after the execution with τ, the result of set_privileges with *res* and the handle to the process whose privileged status should be activated with *hn*. Furthermore, s_{V} and s_{D} represent the original abstract states.

While the devices remain unchanged, the new abstract VAMOS state $s_{\mathsf{V}}{}'$ is determined by function vamos_set_privileges_inter. This function is similar to the specification function vamos_set_privileges but does not yet specify the result delivery to the current process. The reason for this is, that not set_privileges writes the result into the corresponding register of the current process but dispatch_kernel_call. Thus, set_privileges is only concerned with setting the privileged status and with the computation of the result value. Hence, we define:

$$\begin{array}{l}
\mathsf{vamos\_set\_privileges\_inter}\ s_{\mathsf{V}}\ hn \equiv \\
\quad \textbf{let } p_{\mathsf{cp}} = \lceil \mathsf{v\_cup}\ s_{\mathsf{V}}.\mathsf{schedds}\rceil;\ p_{\mathsf{hn}} = \lceil s_{\mathsf{V}}.\mathsf{rightsdb}\ p_{\mathsf{cp}}\rceil.\mathsf{hdb}\ hn; \\
\quad\quad res = \mathsf{vamos\_result\_set\_privileges}\ s_{\mathsf{V}}.\mathsf{rightsdb}\ p_{\mathsf{cp}}\ hn \\
\quad \textbf{in if } \mathsf{is\_error}\ res\ \textbf{then } s_{\mathsf{V}} \\
\quad\quad \textbf{else } s_{\mathsf{V}}(\!|\mathsf{rightsdb} := s_{\mathsf{V}}.\mathsf{rightsdb}(\lceil p_{\mathsf{hn}}\rceil \mapsto \lceil s_{\mathsf{V}}.\mathsf{rightsdb}\ \lceil p_{\mathsf{hn}}\rceil\rceil(\!|\mathsf{priv} := \mathsf{True}|\!))|\!)
\end{array}$$

Apart from that, $\mathsf{setPrivileges}_{\mathsf{post}}$ claims that the result value *res* is equal to the one returned by vamos_result_set_privileges. Finally, the abstraction relation and the state invariant are preserved.

Note that the trap handling is only completed with writing the result register of the current process. As this does not yet happen in set_privileges, the virtual machines of the current process still differ after the execution. Thus, the flag *diff* in $\mathsf{weakAbs}_{\mathsf{V+D}}$ is still True.

Correctness. Based on the pre- and postconditions, we formulate the following correctness theorem.

Theorem 6.4.1 (Correctness of set_privileges) *The semantic effect of the function* set_privileges *is described by* vamos_set_privileges_inter. *Furthermore, the execution of the function* set_privileges *preserves the abstraction relation* $\mathsf{weakAbs}_{\mathsf{V+D}}$ *and the implementation invariant* implInv.

$$\begin{array}{l}
\forall \sigma\ s_{\mathsf{V}}\ s_{\mathsf{D}}. \\
\quad \Gamma \vdash_{\mathsf{t}} \{\!|\sigma.\ \mathsf{setPrivileges}_{\mathsf{pre}}\ \sigma\ {}'\mathsf{hn}\ s_{\mathsf{V}}\ s_{\mathsf{D}}|\!\} \\
\quad\quad {}'\mathsf{res\_int} :== \textbf{PROC}\ \mathsf{process\_set\_privileges}({}'\mathsf{hn}) \\
\quad\quad \{\!|\tau.\ \mathsf{setPrivileges}_{\mathsf{post}}\ \tau\ \sigma\ {}'\mathsf{res\_int}\ {}^{\sigma}\mathsf{hn}\ s_{\mathsf{V}}\ s_{\mathsf{D}}|\!\}
\end{array}$$

Proof The precondition for function resolve_handle directly follows from the one of set_privileges. With it, we get the relation between *pid* (the result from resolving the handle) and the abstract process number p_{hn} used in vamos_set_privileges_inter. Furthermore, we have to establish the relations between the errors in the implementation and their counterparts in the specification. Remember that the VAMOS call SET_PRIVILEGES is reserved for privileged processes. Thus, we have to show, for instance, that privileged s_V.rightsdb ⌈v_cup s_V.schedds⌉ only applies iff privileged current_process.

Based on such relations, we can show that the result value res_int corresponds with the value returned by vamos_result_set_privileges. In case of success, the actual setting of the privileged status is straightforward.

The theory file `vamosSetPrivileges_proof.thy` contains the corresponding Hoare triple and the functional proof which comprises 100 steps. □

6.4.2 Implementation Correctness of Function process_change_scheduling_param

Changing the scheduling parameters of a process is encapsulated in function process_change_scheduling_param. It is called by the VAMOS trap handler, if the current process executed a TRAP instruction with trap number 4. As input parameters, it takes a handle *hn*, a timeslice *tsl* and a priority *prio*, which are taken out of the registers of the current process.

Functionality. In a first step, function process_change_scheduling_param resolves the handle *hn* by means of function resolve_handle. If the return value *pid* does not represent a process-identifier or one of the remaining arguments is invalid, it only sets the return value to the corresponding error code. Otherwise, function process_change_scheduling_param performs the following tasks:

- If the priority of an ready process *pid* should be changed, the process is taken out of its former ready list and appended to the one of priority *prio*. In addition to that, its consumed time is reset to 0. Changing the priority might change the current maximum priority as well as the current process. The former is computed by compute_max_prio and function search_next_process is used to elect the process that runs next.
- Otherwise, only the values of the timeslice and the priority are updated to *tsl* and *prio*, respectively.

Precondition. The precondition $\mathsf{chngSchedParams_{pre}}$ is similar to the one for set_privileges and defined as follows:

$\mathsf{chngSchedParams_{pre}}\ \sigma\ hn\ tsl\ prio\ s_V\ s_D \equiv$
$\quad \mathsf{weakAbs_{V+D}}\ \sigma\ {}^{\sigma}\mathsf{current\_process}\ \mathsf{True}\ (s_V,\ s_D) \wedge \mathsf{implInv}\ \sigma\ s_V \wedge$
$\quad {}^{\sigma}\mathsf{current\_process} \neq \mathsf{Null} \wedge 0 \leq hn < 2^{32} \wedge 0 \leq tsl < 2^{32} \wedge$
$\quad 0 \leq prio < 2^{32} \wedge$

$(\exists hn\ tsl\ prio.$
$\quad \omega_{\mathsf{proc}}\ \lceil s_V.\mathsf{procs}\ \lceil \mathsf{v\_cup}\ s_V.\mathsf{schedds}\rceil\rceil = \text{CHG\_SCHED\_PARAMS}\ hn\ tsl\ prio)$

Postcondition. The postcondition $\mathsf{chgSchedParams}_{\mathsf{post}}$ is also similar to the one of set_privileges. Again, a function vamos_change_sched_param_inter is defined which, except for the result delivery, corresponds with the specification function vamos_change_sched_param:

```
vamos_change_sched_param_inter $s_V$ $hn_{victim}$ $tsl_{new}$ $prio_{new}$ $\equiv$
  let $p_{cp} = \lceil$v_cup $s_V$.schedds$\rceil$; $p_{victim} = \lceil\lceil s_V$.rightsdb $p_{cp}\rceil$.hdb $hn_{victim}\rceil$;
      $res$ = vamos_result_change_sched_param $s_V$.rightsdb $p_{cp}$ $hn_{victim}$ $prio_{new}$
  in if is_error $res$ then $s_V$
     else $s_V$(|schedds := v_chng_sched_param_updt_schedds $s_V$.schedds $s_V$.priodb
                           $s_V$.rightsdb $p_{cp}$ $hn_{victim}$ $tsl_{new}$ $\lceil prio_{new}\rceil$,
              priodb := $s_V$.priodb($p_{victim}$ := $prio_{new}$)|)
```

Accordingly, the postcondition claims that the new VAMOS state $s_V{}'$ is determined by function vamos_change_sched_param_inter. The result *res* has to correspond to the value returned by vamos_result_change_sched_param and the abstraction relation as well as the implementation invariant have to be preserved:

```
chgSchedParams$_{post}$ $\tau$ $\sigma$ $res$ $hn$ $tsl$ $prio$ $s_V$ $s_D$ $\Longrightarrow$
let $s_V{}'$ = vamos_change_sched_param_inter $s_V$ (to_int32 $hn$) $tsl$
            (reg2prio (to_int32 $prio$));
    $s_D{}' = s_D$
in $res$ =
   result_value_int
    (vamos_result_change_sched_param $s_V$.rightsdb $\lceil$v_cup $s_V$.schedds$\rceil$
      (to_int32 $hn$) (reg2prio (to_int32 $prio$))) $\wedge$
   weakAbs$_{V+D}$ $\tau$ $^\sigma$current_process True ($s_V{}'$, $s_D{}'$) $\wedge$ implInv $\tau$ $s_V{}'$
```

Correctness. Based on the assertions above, we formulate the following correctness theorem.

Theorem 6.4.2 (Correctness of process_change_scheduling_param**)** *The semantic effect of function* process_change_scheduling_param *is described by the abstract function* vamos_change_sched_param_inter. *Furthermore, the execution of* process_change_scheduling_param *preserves the abstraction relation* $\mathsf{weakAbs}_{V+D}$ *and the implementation invariant* implInv.

```
$\forall \sigma\ s_V\ s_D$.
  $\Gamma \vdash_t$ {|$\sigma$. chngSchedParams$_{pre}$ $\sigma$ 'hn 'tsl 'prio $s_V$ $s_D$|}
        'res_int :== PROC process_change_scheduling_param('hn,'tsl,
        'prio)
        {|$\tau$. chgSchedParams$_{post}$ $\tau$ $\sigma$ 'res_int $^\sigma$hn $^\sigma$tsl $^\sigma$prio $s_V$ $s_D$|}
```

Proof Similar as above, the precondition for function resolve_handle directly follows from the one of process_change_scheduling_param. With it, we get the relation between *pid* (the result from resolving the handle) and the abstract process number p_{victim} used in vamos_change_sched_param_inter to determine the process whose scheduling parameters should be changed. We also establish the relations between the different error representations in the concrete and abstract levels. More effort is involved, if a rearrangement of the ready queues is necessary. In this case, we have to show that the semantic effect can be described by the function v_chng_sched_param_updt_schedds and that the list properties, as required by implInv, are preserved. The latter is ensured by the application of the functions compute_max_prio and search_next_process. Showing the correspondance of the updates of the timeslice and the priority, in contrast, is straightforward.

The theory file `vamosChngSchedParams_proof.thy` contains the corresponding Hoare triple together with the functional correctness proof comprising 1700 steps. □

6.4.3 Implementation Correctness of IPC

As in the specification, the IPC implementation breaks up into several stages. The first-level functions are those directly called by function dispatch_kernel_call with corresponding arguments. Accordingly, the VAMOS call IPC_SEND is realized by the function ipc_send and IPC_RECEIVE is associated with ipc_receive. Function ipc_send_receive realizes the combined call IPC_REQUEST. Within these functions, the implementation checks for invocation errors, stores the call arguments in the process information block of the calling process and, as shown in the call graph in Figure 6.1, invokes the second-level functions.

The linchpin of these second-level functions is ipc_communication which implements the actual transmission. As the call graph depicts, it invokes various auxiliary functions. One of it is ipc_schedule_receive which is also used in the context of ipc_receive. It is invoked by the latter, if neither a suitable sender exists nor the delivery of kernel notifications is possible. In this case, the receiver either gets an timeout error (if an immediate timeout was specified) or is prepared for waiting. Both actions are implemented by ipc_schedule_receive. In the context of ipc_communication, only preparing a process for waiting is necessary. If the transmission was part of a combined send and receive operation, the sender is forced to wait.

Before the actual transmission, ipc_send and ipc_send_receive first call function ipc_initiate_send. It tries to establish a rendez-vous situation with a receiver and, in case of success, calls ipc_communication. Otherwise, it puts the sender to sleep or writes an timeout error into its result register and aborts the call, after a new current process was elected.

Subsequently, we briefly introduce the functionality and specification of function ipc_communication. Later on, we use the specification and sketch the implementation correctness proofs of the particular IPC calls.

Implementation Correctness of Function ipc_communication

The call graph in Figure 6.1 illustrates that the implementation of the function ipc_communication is far too complex for being presented, here. Thus, we confine ourselves to the intended functionality and relate the different parts to their rough abstract counterparts.

The function ipc_communication realizes the communication between two processes, as seen from the receiver's point of view. For this reason, it is occupied with the process-identifier rcv_pid of the receiving process as input. The corresponding pointer receiver can be obtained from the list of process pointers, i. e., pib ! rcv_pid. The execution of this function relies on the assumption, that the arguments are stored in the PIBs of the processes involved and that these arguments already passed through the initial checks, i. e., are valid.

Determining the Sender. In a first step, ipc_communication determines the sender. The desired IPC partner of the receiver is stored in ipc_pid receiver. If this entry delivers 0, the receiver performs an open-receive and the potential senders can be found in the send queue accessed by send_queue receiver. Otherwise, the entry denotes the PID of the desired sender.

The implementation actually performs the same checks as in the specification. If the message length of a potential sender is too large, the corresponding process is provided with an error message and waken up. The awakening of a process is taken over by the function wake_up. Setting the result register is done by means of the Cvm primitive cvm_set_vm_gpr.

The description of the semantic effect of the error message delivery is included in the specification function v_ipc_rcv_updt_procs. Waking up the processes is covered by function v_ipc_rcv_updt_schedds.

The theory file `vamosIpcComm_determineSender.thy` contains the Hoare triple of the part which determines the sender and the functional correctness proof. Due to the various case distinctions, this proof is complex and requires around 4600 steps.

Transmission. If a suitable sender remains after the checks above, the actual transmission starts. Thus, the send message is copied into the virtual memory of the receiver and, according to the call arguments of the sender, the receiver's handle database is updated. The former is done by means of the Cvm primitive cvm_copy, whereas the update of an entry in the handle database is encapsulated into the auxiliary function update_hdb. For details regarding the Hoare triple of update_hdb and the corresponding proof of the implementation correctness (around 800 steps), see `vamosUpdateHdb.thy`.

Apart from the updates of the internal kernel datastructures, the sender as well as the receiver are informed about the successful operation. For this purpose, the kernel uses the Cvm primitive cvm_set_vm_gpr to write the according result registers. In addition, the receiver is also provided with kernel notifications which are written into certain registers, as well. The semantic effect of the message copying and the result delivery is part of the specification function v_ipc_trans_updt_procs. Updating the handle database is covered by function v_ipc_trans_updt_rightsdb.

In the remaining steps, the kernel cleans up and sets the implementation into a consistent state again. Thus, as long as the sender is not allowed to perform multiple send operations to the receiver, the Vamos kernel revokes all its rights to the receiver. The effect of this revoking describes the update of the sender's rights database in v_ipc_trans_updt_rightsdb.

A transmission also affects the process states. Thereby, the kernel distinguishes whether the receiver or the sender acted as current process. If the receiver represents the current process, it suffices to set its state to ready again. Otherwise, it has been waiting before and function wake_up is applied in order to wake it up again. The effects of the latter operation describes v_wkp_updt_schedds. Regarding the sender, the kernel distinguishes whether the transmission was part of an ordinary send or a combined send and receive operation. In the former case, the sender has to be waken up as long as it is not the current process. Otherwise, only its state is set to ready again. In the latter case, the sender is prepared for the succeeding receive phase. The auxiliary function ipc_schedule_receive encapsulates the required steps. In this situation, the specification function v_ipc_wait_updt_schedds describes its semantic effect. The theory file `vamosIpcScheduleReceive.thy` contains the Hoare triple of the function ipc_schedule_receive and the corresponding formal correctness proof comprising almost 400 steps.

No Transmission. In case that no suitable sender can be found, the Vamos kernel examines the possibility of delivering kernel notifications. If possible, the auxiliary function deliver_notifications writes the relevant information into dedicated registers of the receiver. Otherwise, the receiver either receives an error message due to a timeout error or is prepared for waiting. For this purpose, the kernel again uses function ipc_schedule_receive. The semantic effect is part of the specification functions v_ipc_rcv_updt_procs and v_ipc_rcv_updt_schedds, if no communication takes place.

Summary. Specifying and proving the implementation correctness of function ipc_communication is pretty elaborate, because many invocation scenarios have to be taken into account. Although not explicitly specified in this way, in the overall context of an IPC operation, the semantic effect of ipc_communication can be described, more or less accurately, by the dedicated update functions defined in the context of vamos_ipc_receive, like v_ipc_rcv_updt_procs for the virtual machine update. If ipc_communication is called in the context of an IPC send operation, these functions boil down to the transition functions, i. e., v_ipc_trans_updt_procs, etc.. The complete formal specification of the function ipc_communication together with the associated implementation correctness proof can be found in the theory file `vamosIpcCommunication.thy`. The complexity of ipc_communication is also reflected in the correctness proof which altogether comprises around 20.000 steps. Thereby, the parts which determine the sender (4600 steps), update the handle databases (1500 steps), and update the process states (1600 steps) are the most complex ones. Furthermore, the combination of all these particular parts entails a lot of effort.

Implementation Correctness of Function ipc_send

The IPC send operation is encapsulated in function ipc_send. It is called by the VAMOS trap handler, if the current process executed a TRAP instruction with trap number 12. As input parameters, it takes the handle *rcv_handle* to the desired receiver and the rights *snd_rights* that should be granted, the handle *add_handle* to the additional process together with the rights *add_rights*, the start address *snd_msg_ptr* and the length *snd_msglen* of the memory message, and the timeout *snd_timeout*.

Functionality. By means of function resolve_handles, function ipc_send first resolves the given handles. If this results in invalid process-identifiers or if the other arguments are invalid, an error code is written into the result register of the current process and the call is aborted. Otherwise, the arguments are stored in the corresponding components of the PIB of the current process and function ipc_initiate_send is invoked. This function tries to establish a rendez-vous situation with a receiver and invokes ipc_communication, in case of success. Otherwise, the sender either gets a timeout message or is scheduled for a later sending. The latter involves function put_process_to_sleep in order to put the sender into the wait list. This, in turn, may change the current process and function search_next_process is invoked to elect the next one.

Precondition. The precondition $\mathsf{ipcSend}_{\mathsf{pre}}$ of function ipc_send is defined as follows:

$\mathsf{ipcSend}_{\mathsf{pre}}\ \sigma\ \mathit{rcv\_handle}\ \mathit{snd\_rights}\ \mathit{add\_handle}\ \mathit{add\_rights}\ \mathit{snd\_timeout}$
$\ \mathit{snd\_msg\_ptr}\ \mathit{snd\_msglen}\ s_{\mathsf{V}}\ s_{\mathsf{D}} \equiv$
$\ \ \mathsf{weakAbs}_{\mathsf{V+D}}\ \sigma\ {}^{\sigma}\mathsf{current\_process}\ \mathsf{True}\ (s_{\mathsf{V}},\ s_{\mathsf{D}}) \wedge \mathsf{implInv}\ \sigma\ s_{\mathsf{V}} \wedge$
$\ \ {}^{\sigma}\mathsf{current\_process} \neq \mathsf{Null} \wedge$
$\ \ (\exists \mathit{hn}_{\mathsf{rcv}}\ \mathit{rights}_{\mathsf{snd}}\ \mathit{msg}\ \mathit{hn}_{\mathsf{add}}\ \mathit{rights}_{\mathsf{add}}\ \mathit{to}_{\mathsf{snd}}.$
$\ \ \ \ \omega_{\mathsf{proc}}\ \lceil s_{\mathsf{V}}.\mathsf{procs}\ \lceil \mathsf{v\_cup}\ s_{\mathsf{V}}.\mathsf{schedds}\rceil\rceil =$
$\ \ \ \ \textsc{ipc\_send}\ \mathit{hn}_{\mathsf{rcv}}\ \mathit{rights}_{\mathsf{snd}}\ \mathit{msg}\ \mathit{hn}_{\mathsf{add}}\ \mathit{rights}_{\mathsf{add}}\ \mathit{to}_{\mathsf{snd}}) \wedge$
$\ \ \mathsf{to\_nat32}\ (\lceil s_{\mathsf{V}}.\mathsf{procs}\ ({}^{\sigma}\mathsf{pid}\ {}^{\sigma}\mathsf{current\_process})\rceil.\mathsf{gprs}\ !\ 11) =$

$rcv\_handle\ \wedge$
to_nat32 ($\lceil s_V$.procs ($^\sigma$pid $^\sigma$current_process)$\rceil$.gprs ! 12) =
$snd\_rights\ \wedge$
to_nat32 ($\lceil s_V$.procs ($^\sigma$pid $^\sigma$current_process)$\rceil$.gprs ! 13) =
$snd\_msg\_ptr\ \wedge$
to_nat32 ($\lceil s_V$.procs ($^\sigma$pid $^\sigma$current_process)$\rceil$.gprs ! 14) =
$snd\_msglen\ \wedge$
to_nat32 ($\lceil s_V$.procs ($^\sigma$pid $^\sigma$current_process)$\rceil$.gprs ! 18) =
$add\_rights\ \wedge$
$\lceil s_V$.procs ($^\sigma$pid $^\sigma$current_process)$\rceil$.gprs ! 19 = $snd\_timeout\ \wedge$
to_nat32 ($\lceil s_V$.procs ($^\sigma$pid $^\sigma$current_process)$\rceil$.gprs ! 17) =
add_handle

As usual, the precondition requires the abstraction relation and the implementation invariant. Furthermore, the current process (the sender) must exist, i. e., current_process $\neq$ Null. Apart from that, predicate $\text{ipcSend}_{\text{pre}}$ also fixes the sources registers of the arguments. This is necessary in order to preserve the properties of IPC operations as requested by ipcProps. For the same reason, it is also required that the process output of the current process denotes IPC_SEND.

Postcondition. The predicate $\text{ipcSend}_{\text{post}}$ describes the postcondition of function ipc_send:

$\text{ipcSend}_{\text{post}}$ τ σ rcv_handle $rights$ add_handle add_rights $snd\_timeout$
 snd_msg_ptr snd_msglen s_V s_D $\Longrightarrow$
let $s_V{}' =$
    vamos_ipc_send s_V (to_int32 rcv_handle)
     (num2rights (to_int32 $rights$))
     (build_memobj $\lceil s_V$.procs $\lceil$v_cup s_V.schedds$\rceil\rceil$ snd_msg_ptr
      snd_msglen)
     (to_int32 add_handle) (num2rights (to_int32 add_rights))
     (int2timeout $snd\_timeout$);
   $s_D{}' = s_D$
in $\text{weakAbs}_{\text{V+D}}$ τ $^\sigma$current_process False ($s_V{}'$, $s_D{}'$) $\wedge$ implInv τ $s_V{}'$

The devices remain unchanged and the new VAMOS state $s_V{}'$ is computed by the specification function vamos_ipc_send. Together with the new implementation state τ, it fulfills the abstraction relation as well as the implementation invariant.

Correctness. The implementation correctness of ipc_send is stated with the following theorem.

Theorem 6.4.3 (Correctness of ipc_send) *The semantic effect of function* ipc_send *is described by the model function* vamos_ipc_send*. Furthermore, the execution of the function* ipc_send *preserves the abstraction relation* $\text{weakAbs}_{\text{V+D}}$ *and the implementation invariant* implInv*.*

$$\begin{array}{l}
\forall \sigma\ s_V\ s_D.\\
\quad \Gamma \vdash_t \{\!|\sigma.\ \mathsf{ipcSend_{pre}}\ \sigma\ {}'\mathsf{rcv\_handle}\ {}'\mathsf{rights}\ {}'\mathsf{add\_handle}\ {}'\mathsf{add\_rights}\\
\qquad\qquad {}'\mathsf{snd\_timeout}\ {}'\mathsf{snd\_msg\_ptr}\ {}'\mathsf{snd\_msglen}\ s_V\ s_D|\!\}\\
\qquad {}'\mathsf{res\_int} :== \textbf{PROC}\ \mathsf{ipc\_send}({}'\mathsf{rcv\_handle}, {}'\mathsf{rights}, {}'\mathsf{snd\_msg\_ptr},\\
\qquad {}'\mathsf{snd\_msglen}, {}'\mathsf{add\_handle}, {}'\mathsf{add\_rights}, {}'\mathsf{snd\_timeout})\\
\qquad \{\!|\tau.\ \mathsf{ipcSend_{post}}\ \tau\ \sigma\ {}^{\sigma}\mathsf{rcv\_handle}\ {}^{\sigma}\mathsf{rights}\ {}^{\sigma}\mathsf{add\_handle}\ {}^{\sigma}\mathsf{add\_rights}\\
\qquad\qquad {}^{\sigma}\mathsf{snd\_timeout}\ {}^{\sigma}\mathsf{snd\_msg\_ptr}\ {}^{\sigma}\mathsf{snd\_msglen}\ s_V\ s_D|\!\}
\end{array}$$

Proof The initial checks regarding the validity of the call arguments basically realize the abstract function ipc_send_invoc_err. Calling initiate_send has the following impacts: If a transmission is possible, it calls the function ipc_communication. Thus, we have to discharge the precondition of ipc_communication and may then use its specification to establish the claim of the theorem. If no transmission is possible, initiate_send either calls put_process_to_sleep to prepare the sender for waiting or sends an error message due to a timeout error. Both actions can directly be related to the effects described in vamos_ipc_send.

The theory file `vamosInitiateSend.thy` contains the Hoare triple and the functional correctness proof (1300 steps) of function initiate_send. Relying on this, `vamosIpcSend_proof.thy` presents the same one for the function vamos_ipc_send. Showing the functional correctness of vamos_ipc_send requires approximately 2000 steps. □

In a similar way, we can specify and show the implementation correctness of the function ipc_send_receive.

The theory file `vamosIpcSendReceive_proof.thy` contains the corresponding Hoare triple and the functional correctness proof. The latter comprises 2100 steps.

Implementation Correctness of Function ipc_receive

Function ipc_receive implements the IPC receive operation. It is called by the VAMOS trap handler, if the current process executed a TRAP instruction with trap number 11. As input parameters, it takes the handle *snd_handle* to the desired sender, the start address *rcv_msg_ptr* and the length *rcv_msglen* of the buffer, and the timeout *rcv_timeout*.

Functionality. By means of function resolve_handles, ipc_receive first resolves the handle to the sender. If this results in an invalid process-identifier and neither an open-receive nor a closed-receive from the kernel is desired, the according error code is written into the result register and the call is aborted. The same happens, if the remaining arguments are invalid. Otherwise, the arguments are stored in the corresponding components of the PIB of the current process.

If a rendez-vous situation can be established, ipc_receive initiates the transmission by calling function ipc_communication. If no rendez-vous situation can be established and the receiver is ready to receive kernel notifications, function deliver_notifications is called. Otherwise, the receiver either gets an error message due a timeout or is prepared for waiting. For both actions function ipc_schedule_receive is used. In addition, function search_next_process elects the next current process.

Precondition. The precondition $\mathsf{ipcReceive}_{\mathsf{pre}}$ of function ipc_receive is similar to the one of ipc_send. Only the source registers and the process output are adjusted:

$\mathsf{ipcReceive}_{\mathsf{pre}}\ \sigma\ \mathit{snd\_handle}\ \mathit{rcv\_msg\_ptr}\ \mathit{rcv\_msglen}\ \mathit{rcv\_timeout}\ s_{\mathsf{V}}\ s_{\mathsf{D}} \equiv$
$\quad \mathsf{weakAbs}_{\mathsf{V+D}}\ \sigma\ {}^{\sigma}\mathsf{current\_process}\ \mathsf{True}\ (s_{\mathsf{V}},\ s_{\mathsf{D}}) \wedge \mathsf{implInv}\ \sigma\ s_{\mathsf{V}} \wedge$
$\quad {}^{\sigma}\mathsf{current\_process} \neq \mathsf{Null} \wedge$
$\quad (\exists hn_{\mathsf{snd}}\ \mathit{buffer}\ to_{\mathsf{rcv}}.$
$\qquad \omega_{\mathsf{proc}}\ \lceil s_{\mathsf{V}}.\mathsf{procs}\ \lceil \mathsf{v\_cup}\ s_{\mathsf{V}}.\mathsf{schedds}\rceil\rceil =$
$\qquad \textsc{ipc\_receive}\ hn_{\mathsf{snd}}\ \mathit{buffer}\ to_{\mathsf{rcv}}) \wedge$
$\quad \mathsf{to\_nat32}\ (\lceil s_{\mathsf{V}}.\mathsf{procs}\ ({}^{\sigma}\mathsf{pid}\ {}^{\sigma}\mathsf{current\_process})\rceil.\mathsf{gprs}\ !\ 11) =$
$\quad \mathit{snd\_handle} \wedge$
$\quad \mathsf{to\_nat32}\ (\lceil s_{\mathsf{V}}.\mathsf{procs}\ ({}^{\sigma}\mathsf{pid}\ {}^{\sigma}\mathsf{current\_process})\rceil.\mathsf{gprs}\ !\ 15) =$
$\quad \mathit{rcv\_msg\_ptr} \wedge$
$\quad \mathsf{to\_nat32}\ (\lceil s_{\mathsf{V}}.\mathsf{procs}\ ({}^{\sigma}\mathsf{pid}\ {}^{\sigma}\mathsf{current\_process})\rceil.\mathsf{gprs}\ !\ 16) =$
$\quad \mathit{rcv\_msglen} \wedge$
$\quad \lceil s_{\mathsf{V}}.\mathsf{procs}\ ({}^{\sigma}\mathsf{pid}\ {}^{\sigma}\mathsf{current\_process})\rceil.\mathsf{gprs}\ !\ 20 = \mathit{rcv\_timeout}$

Postcondition. The predicate $\mathsf{ipcReceive}_{\mathsf{post}}$ describes the postcondition of ipc_receive:

$\mathsf{ipcReceive}_{\mathsf{post}}\ \tau\ \sigma\ \mathit{snd\_handle}\ \mathit{rcv\_msg\_ptr}\ \mathit{rcv\_msglen}\ \mathit{rcv\_timeout}\ s_{\mathsf{V}}$
$s_{\mathsf{D}} \Longrightarrow$
$\mathbf{let}\ s_{\mathsf{V}}{}' =$
$\qquad \mathsf{vamos\_ipc\_receive}\ s_{\mathsf{V}}\ (\mathsf{to\_int32}\ \mathit{snd\_handle})$
$\qquad (\mathsf{build\_buffer}\ \lceil s_{\mathsf{V}}.\mathsf{procs}\ \lceil \mathsf{v\_cup}\ s_{\mathsf{V}}.\mathsf{schedds}\rceil\rceil\ \mathit{rcv\_msg\_ptr}$
$\qquad\quad \mathit{rcv\_msglen})$
$\qquad (\mathsf{int2timeout}\ \mathit{rcv\_timeout});$
$\quad s_{\mathsf{D}}{}' = s_{\mathsf{D}}$
$\mathbf{in}\ \mathsf{weakAbs}_{\mathsf{V+D}}\ \tau\ {}^{\sigma}\mathsf{current\_process}\ \mathsf{False}\ (s_{\mathsf{V}}{}',\ s_{\mathsf{D}}{}') \wedge \mathsf{implInv}\ \tau\ s_{\mathsf{V}}{}'$

The devices remain unchanged and the new Vamos state $s_{\mathsf{V}}{}'$ is computed by the specification function vamos_ipc_receive. Together with the new implementation state τ, it fulfills the abstraction relation as well as the implementation invariant.

Correctness. The implementation correctness of ipc_receive is stated with the following theorem.

Theorem 6.4.4 (Correctness of ipc_receive) *The semantic effect of function* ipc_receive *is described by the model function* vamos_ipc_receive*. Furthermore, the execution of* ipc_receive *preserves the abstraction relation* $\mathsf{weakAbs}_{\mathsf{V+D}}$ *and the implementation invariant* implInv.

$\forall \sigma\ s_{\mathsf{V}}\ s_{\mathsf{D}}.$
$\quad \Gamma \vdash_{\mathsf{t}} \{\!|\sigma.\ \mathsf{ipcReceive}_{\mathsf{pre}}\ \sigma\ '\mathsf{snd\_handle}\ '\mathsf{rcv\_msg\_ptr}\ '\mathsf{rcv\_msglen}$
$\qquad\qquad '\mathsf{rcv\_timeout}\ s_{\mathsf{V}}\ s_{\mathsf{D}}|\!\}$
$\qquad '\mathsf{res\_int} :== \mathbf{PROC}\ \mathsf{ipc\_receive}('\mathsf{snd\_handle}, '\mathsf{rcv\_msg\_ptr},$
$\qquad '\mathsf{rcv\_msglen}, '\mathsf{rcv\_timeout})$
$\qquad \{\!|\tau.\ \mathsf{ipcReceive}_{\mathsf{post}}\ \tau\ \sigma\ {}^{\sigma}\mathsf{snd\_handle}\ {}^{\sigma}\mathsf{rcv\_msg\_ptr}\ {}^{\sigma}\mathsf{rcv\_msglen}$
$\qquad\qquad {}^{\sigma}\mathsf{rcv\_timeout}\ s_{\mathsf{V}}\ s_{\mathsf{D}}|\!\}$

Proof The initial checks can basically be described by the abstract function ipc_rcv_invoc_err. We have to discharge the precondition of the function ipc_communication, if a transmission is possible. With its specification, we are able to establish the claim of the theorem. If no transmission is possible but the delivery of kernel notifications, we have to discharge the precondition of deliver_notifications. In this case, the abstract function v_ipc_rcv_updt_procs describes the updates of the virtual machines. The remaining datastructures remain untouched and we can establish the claim of the theorem. Otherwise, we have to discharge the preconditions of the functions ipc_schedule_receive and search_next_process and use their postconditions to proof the theorem.

The theory file `vamosIpcReceive_proof.thy` contains the Hoare triple and the functional correctness proof with almost 2900 steps. □

6.4.4 Implementation Correctness of Function dispatcher_kernel_call

The previous sections introduced the correctness statements and proofs of the IPC functions, process_change_sched_param and set_privileges. In a similar way, we can formulate such statements for the remaining VAMOS calls. Even if not proven, the particular specifications can be used for the correctness proof of the VAMOS trap handler implemented in function dispatcher_kernel_call.

As usual, the correctness statement relies on a pre- and postcondition.

Precondition. The predicate $\mathsf{handleTrap}_{\mathsf{pre}}$ encapsulates the precondition:

$\mathsf{handleTrap}_{\mathsf{pre}}\ \sigma\ \mathit{trapnr}\ s_{\mathsf{V}}\ s_{\mathsf{D}} \equiv$
$\quad \mathsf{weakAbs}_{\mathsf{V+D}}\ \sigma\ {}^{\sigma}\mathsf{current\_process}\ \mathsf{True}\ (s_{\mathsf{V}}, s_{\mathsf{D}}) \wedge$
$\quad \mathsf{implInv}\ \sigma\ s_{\mathsf{V}} \wedge$
$\quad {}^{\sigma}\mathsf{current\_process} \neq \mathsf{Null} \wedge$
$\quad (\mathsf{mca\_nat}\ (s_{\mathsf{V}}.\mathsf{procs}\ ({}^{\sigma}\mathsf{pid}\ {}^{\sigma}\mathsf{current\_process}))\ s_{\mathsf{D}}\ (\mathsf{cvm\_ups}\ {}^{\sigma}\mathsf{cvmX}).\mathsf{statusreg} \wedge_{\mathsf{u}}$

$\mathsf{UEXCEPT\_MASK}) =$
$0 \wedge$
$(\mathsf{mca\_nat}\ (s_{\mathsf{V}}.\mathsf{procs}\ ({}^{\sigma}\mathsf{pid}\ {}^{\sigma}\mathsf{current\_process}))\ s_{\mathsf{D}}\ (\mathsf{cvm\_ups}\ {}^{\sigma}\mathsf{cvmX}).\mathsf{statusreg} \wedge_{\mathsf{u}}$
$\mathsf{EXCEPT\_TRAP}) \neq$
$0 \wedge$
$\mathsf{edata\_nat}\ \lceil s_{\mathsf{V}}.\mathsf{procs}\ ({}^{\sigma}\mathsf{pid}\ {}^{\sigma}\mathsf{current\_process})\rceil = trapnr$

Apart from claiming the abstraction relation, the implementation invariant and an existing current process, $\mathsf{handleTrap_{pre}}$ also makes some demands on the invocation context of dispatch_kernel_call. In function system_step, we have already seen, that the exception cause and data are computed by means of the functions mca_nat and edata_nat. The precondition reproduces this computations and thus establishes a relation to the invocation context. In particular, it demands that the exception cause signals a trap instruction without runtime errors. The former corresponds to an active bit EXCEPT_TRAP, the latter is ensured by excluding any active bits in the mask UEXCEPT_MASK. In addition, the trap number *trapnr* provided as input has to correspond to the value returned from edata_nat.

Postcondition. The effects of the trap handling are described by the postcondition $\mathsf{handleTrap_{post}}$:

$\mathsf{handleTrap_{post}}\ \tau\ \sigma\ s_{\mathsf{V}}\ s_{\mathsf{D}} \Longrightarrow$
$\textbf{let}\ \mathit{data}_{\mathsf{dev}} = \textbf{if}\ \omega_{\mathsf{V}}\ s_{\mathsf{V}} = \mathsf{idle}_{\Omega\mathsf{V}}\ \textbf{then}\ \mathsf{idle}_{\Sigma\mathsf{V}}\ \textbf{else}\ \mathsf{fst}\ (\mathsf{snd}\ (\delta_{\mathsf{Dint}}\ (\omega_{\mathsf{V}}\ s_{\mathsf{V}})\ s_{\mathsf{D}}));$
$\quad s_{\mathsf{D}}' = \textbf{if}\ \omega_{\mathsf{V}}\ s_{\mathsf{V}} = \mathsf{idle}_{\Omega\mathsf{V}}\ \textbf{then}\ s_{\mathsf{D}}\ \textbf{else}\ \mathsf{fst}\ (\delta_{\mathsf{Dint}}\ (\omega_{\mathsf{V}}\ s_{\mathsf{V}})\ s_{\mathsf{D}});$
$\quad s_{\mathsf{V}}' = \mathsf{vamosDispatcher}\ s_{\mathsf{V}}\ \mathit{data}_{\mathsf{dev}}$
$\textbf{in}\ \mathsf{weakAbs_{V+D}}\ \tau\ {}^{\sigma}\mathsf{current\_process}\ \mathsf{False}\ (s_{\mathsf{V}}', s_{\mathsf{D}}') \wedge \mathsf{implInv}\ \tau\ s_{\mathsf{V}}'$

We distinguish two kinds of traps resp. VAMOS calls: those with and those without device interaction. In the latter case, the VAMOS output ω_{V} to the device subsystem is $\mathsf{idle}_{\Omega\mathsf{V}}$. Accordingly, the devices remain untouched and no device data is provided as input to vamosDispatcher which computes the new abstract VAMOS state s_{V}'. If device interaction is involved, we first perform the request to the device system. This is done by applying δ_{Dint} with the according VAMOS output as input. As result, we get the new state s_{D}' of the device system, on the one hand, and the response in form of device data $\mathit{data}_{\mathsf{dev}}$, on the other one. The latter is provided as input to vamosDispatcher. Finally, the concrete as well as the abstract states are related via $\mathsf{weakAbs_{V+D}}$ and implInv is preserved.

Note that the virtual machines of the current process do no longer need a separate handling. The trap is either completely handled and the machines are in-sync again or the process is waiting in a pending IPC operation.

Correctness. The implementation correctness of dispatch_kernel_call is stated with the following theorem.

Theorem 6.4.5 (Correctness of the Trap Handler) *The semantic effect of function* dispatch_kernel_call *is described by the model function* vamosDispatcher*. Furthermore, the execution of* dispatch_kernel_call *preserves the abstraction relation* $\mathsf{weakAbs}_{\mathsf{V+D}}$ *and the implementation invariant* implInv*.*

$$\begin{array}{l}\forall \sigma\ s_{\mathsf{V}}\ s_{\mathsf{D}}.\\ \quad \Gamma \vdash_{\mathsf{t}} \{\!|\sigma.\ \mathsf{handleTrap}_{\mathsf{pre}}\ \sigma\ {}^{\sigma}\mathsf{i}\ s_{\mathsf{V}}\ s_{\mathsf{D}}|\!\}\ '\mathsf{res\_int} :== \mathbf{PROC}\ \mathsf{dispatch\_kernel\_call}('\mathsf{i})\\ \qquad\quad \{\!|\tau.\ \mathsf{handleTrap}_{\mathsf{post}}\ \tau\ \sigma\ s_{\mathsf{V}}\ s_{\mathsf{D}}|\!\}\end{array}$$

Proof The proof of this function mainly involves a distinction over the various possible trap numbers. Establishing the preconditions of the particular functions is the main effort. With these prerequisites in place, we use the specifications to establish the claim of our theorem.

As an example, we consider the case that the trap number equals 6, i. e., the current process triggered the VAMOS call SET_PRIVILEGES.

The precondition $\mathsf{setPrivileges}_{\mathsf{pre}}$ claims the abstraction relation and the implementation invariant as well as the existence of the current process. All of these properties can directly be derived from the precondition of the function dispatch_kernel_call. Furthermore, it claims that the handle *hn* to the process (whose privileges should be activated) represents a 32-bit natural number. As the handle *hn* is taken out of a virtual machine register of the current process, this property directly follows from the requirements for the virtual machines which are part of the implementation invariant and encapsulated in predicate $\mathsf{validVMs}_{\mathsf{v}}$. Finally, we have to show that the invocation context causes the process output SET_PRIVILEGED on the abstract level. This property states the following lemma:

$$\begin{array}{l}[\![\mathsf{weakAbs}_{\mathsf{V+D}}\ \sigma\ {}^{\sigma}\mathsf{current\_process}\ \mathsf{True}\ (s_{\mathsf{V}}, s_{\mathsf{D}});\ \mathsf{implInv}\ \sigma\ s_{\mathsf{V}};\\ \ {}^{\sigma}\mathsf{current\_process} \neq \mathsf{Null};\\ \ (\mathsf{mca\_nat}\ (s_{\mathsf{V}}.\mathsf{procs}\ ({}^{\sigma}\mathsf{pid}\ {}^{\sigma}\mathsf{current\_process}))\ s_{\mathsf{D}}\\ \quad (\mathsf{cvm\_ups}\ {}^{\sigma}\mathsf{cvmX}).\mathsf{statusreg}\ \wedge_{\mathsf{u}}\\ \ \ \mathsf{UEXCEPT\_MASK}) =\\ \ 0;\\ \ (\mathsf{mca\_nat}\ (s_{\mathsf{V}}.\mathsf{procs}\ ({}^{\sigma}\mathsf{pid}\ {}^{\sigma}\mathsf{current\_process}))\ s_{\mathsf{D}}\\ \quad (\mathsf{cvm\_ups}\ {}^{\sigma}\mathsf{cvmX}).\mathsf{statusreg}\ \wedge_{\mathsf{u}}\\ \ \ \mathsf{EXCEPT\_TRAP}) \neq\\ \ 0;\\ \ \mathsf{edata\_nat}\ \lceil s_{\mathsf{V}}.\mathsf{procs}\ ({}^{\sigma}\mathsf{pid}\ {}^{\sigma}\mathsf{current\_process})\rceil = 6]\!]\\ \Longrightarrow \exists hn.\ \omega_{\mathsf{proc}}\ \lceil s_{\mathsf{V}}.\mathsf{procs}\ \lceil \mathsf{v\_cup}\ s_{\mathsf{V}}.\mathsf{schedds}\rceil\rceil = \textsc{set\_privileged}\ hn\end{array}$$

With discharging the precondition $\mathsf{setPrivileges}_{\mathsf{pre}}$, we can use the post-condition $\mathsf{setPrivileges}_{\mathsf{post}}$ to establish the claim of the theorem above. The remaining steps are straightforward and concern the result delivery to the virtual machine of the current process. Finally, the virtual machines of the current process are in-sync, again.

In a similar way, we proceed with the remaining VAMOS calls. The theory file `vamosDispatchKernelCall_proof.thy` contains the Hoare triple

of dispatch_kernel_call and the functional correctness proof with 2000 steps.
□

6.5 Correctness of the Vamos Top-level Function

Function dispatcher_kernel describes the top-level function of the VAMOS kernel and is invoked by the CVM framework. This framework also computes the two input parameters of dispatcher_kernel: the exception cause eca and the corresponding data edata. As the call graph suggests, the VAMOS kernel combines the function representing the initialization, the handling of user-generated exceptions, the scheduler, and the interrupt delivery. The previous sections introduced the functions regarding the trap handling and the VAMOS scheduler and their specifications. In a similar way, we can specify the remaining functions and use them in the context of the implementation correctness proof of function dispatcher_kernel.

Subsequently, we first define the pre- and postcondition of the VAMOS kernel and state the correctness statement, afterwards.

Precondition. The precondition $\mathsf{kdispatch}_{\mathsf{pre}}$ encapsulates the requirements for the invocation context:

$\mathsf{kdispatch}_{\mathsf{pre}}\ \sigma\ eca\ edata\ s_{\mathsf{V}}\ s_{\mathsf{D}} \equiv$
$\quad ((eca \wedge_{\mathsf{u}} 1) \neq 0 \longrightarrow {}^{\sigma}\mathsf{cvmX} = (\mathsf{init\_cvmup},\ s_{\mathsf{D}})) \wedge$
$\quad ((eca \wedge_{\mathsf{u}} 1) = 0 \longrightarrow$
$\quad\ \mathsf{weakAbs}_{\mathsf{V+D}}\ \sigma\ {}^{\sigma}\mathsf{current\_process}\ (\mathsf{v\_cup}\ s_{\mathsf{V}}.\mathsf{schedds} \neq \bot)\ (s_{\mathsf{V}},\ s_{\mathsf{D}}) \wedge$
$\quad\ \mathsf{implInv}\ \sigma\ s_{\mathsf{V}} \wedge$
$\quad\ eca =$
$\quad\ \mathsf{mca\_nat}$
$\quad\ \ (\mathbf{if}\ \mathsf{v\_cup}\ s_{\mathsf{V}}.\mathsf{schedds} \neq \bot\ \mathbf{then}\ s_{\mathsf{V}}.\mathsf{procs}\ \lceil \mathsf{v\_cup}\ s_{\mathsf{V}}.\mathsf{schedds} \rceil\ \mathbf{else}\ \bot)\ s_{\mathsf{D}}$
$\quad\ \ (\mathsf{cvm\_ups}\ {}^{\sigma}\mathsf{cvmX}).\mathsf{statusreg} \wedge$
$\quad\ edata =$
$\quad\ (\mathbf{if}\ \mathsf{v\_cup}\ s_{\mathsf{V}}.\mathsf{schedds} \neq \bot\ \mathbf{then}\ \mathsf{edata\_nat}\ \lceil s_{\mathsf{V}}.\mathsf{procs}\ \lceil \mathsf{v\_cup}\ s_{\mathsf{V}}.\mathsf{schedds} \rceil \rceil$
$\quad\ \ \mathbf{else}\ 0))$

Based on *eca*, it distinguishes two cases. The first case consists of a reset which is denoted by $(eca \wedge_{\mathsf{u}} 1) \neq 0$. In this situation, the precondition solely claims that the external state component cvmX is initialized as described in system_step.

Apart from the reset case, the precondition requires the abstraction relation $\mathsf{weakAbs}_{\mathsf{V+D}}$ followed by the implementation invariant implInv. The former has to cope with differing virtual machines, if a current process exists, i. e., $\mathsf{v\_cup}\ s_{\mathsf{V}}.\mathsf{schedds} \neq \bot$. In addition, the input arguments *eca* and *edata* have to fulfill certain requirements which reflect the computations in the CVM framework. The exception cause has to correspond to the value returned by function mca_nat whereas data is either 0 (if no process is running) or the return value of edata_nat.

Postcondition. As the precondition, the postcondition $\mathsf{kdispatch}_{\mathsf{post}}$ also distinguishes two cases:

$\mathsf{kdispatch}_{\mathsf{post}}\ \tau\ \sigma\ res\ eca\ edata\ s_{\mathsf{V}}\ s_{\mathsf{D}} \Longrightarrow$
let $(s_{\mathsf{V}}', s_{\mathsf{D}}') =$
    if $(eca \wedge_{\mathsf{u}} 1) \neq 0$ **then** $(\mathsf{initialConf}\ (\mathsf{getOSimg}\ s_{\mathsf{D}})\ \mathsf{OS\_PAGES}, s_{\mathsf{D}})$
    else $\mathsf{fst}\ (\delta_{\mathsf{V+D}}\ \perp\ (s_{\mathsf{V}}, s_{\mathsf{D}}))$
in $\mathsf{weakAbs}_{\mathsf{V+D}}\ \tau\ {}^{\tau}\mathsf{current\_process}\ \mathsf{False}\ (s_{\mathsf{V}}', s_{\mathsf{D}}') \wedge$
  $\mathsf{implInv}\ \tau\ s_{\mathsf{V}}' \wedge$
  $\mathsf{v\_cup}\ s_{\mathsf{V}}'.\mathsf{schedds} = (\textbf{if}\ 1 \leq res < \mathsf{PID\_MAX}\ \textbf{then}\ \lfloor res \rfloor\ \textbf{else}\ \perp)$

The semantic effect of the initialization is described by the abstract state s_{V}, which is obtained by function initialConf. Function getOSimage delivers the operating system image by reading it from the harddisk. Parameter OS_PAGES determines the number of memory pages reserved for the OS.

For all other cases, we use function $\delta_{\mathsf{V+D}}$ to specify the resulting abstract states. The abstract state together with the concrete one satisfy $\mathsf{weakAbs}_{\mathsf{V+D}}$ as well as implInv. Finally, $\mathsf{kdispatch}_{\mathsf{post}}$ establishes a connection between the result value *res* and the current process of s_{V}'. Later on, this statement helps to reproduce the correspondence between the current process in the Cvm framework and the Vamos model.

Recall that the transition function $\delta_{\mathsf{V+D}}$ uses its input parameter to determine whether the kernel or solely the device subsystem (e. g., by receiving a network package) advance during a transition of the overall system. If this input is not $\perp$, the kernel remains constant. Knowing that the abstraction relation of the kernel data structures and the one of the device subsystem are independent, we may consequently disregard changes that are limited to the device subsystem in the consideration of the kernel refinement. While we are interested in the kernel-device interaction, we disregard any external device input from the environment and the according output to it. As a consequence, this input is fixed at $\perp$ in the theorem above. Furthermore, the transition function $\delta_{\mathsf{V+D}}$ returns a pair of the updated state and the external output. We are, however, only interested in the updated state and thus take the first component of this pair, i. e., $(s_{\mathsf{V}}', s_{\mathsf{D}}') = \mathsf{fst}\ (\delta_{\mathsf{V+D}}\ \perp\ (s_{\mathsf{V}}, s_{\mathsf{D}}))$.

Correctness. The implementation correctness of dispatcher_kernel is stated with the following theorem.

Theorem 6.5.1 (Correctness of the Vamos Top-level Function) *The semantic effect of the function* dispatcher_kernel *is described by the transition function* $\delta_{\mathsf{V+D}}$ *of the* Vamos *model. Furthermore, the execution of* dispatcher_kernel *preservers the abstraction relation* $\mathsf{weakAbs}_{\mathsf{V+D}}$ *and the implementation invariant* implInv.

$$\begin{array}{l}\forall \sigma\ s_V\ s_D.\\ \quad \Gamma \vdash_t \{\!|\sigma.\ \mathsf{kdispatch_{pre}}\ \sigma\ {}^{\sigma}\mathsf{eca}\ {}^{\sigma}\mathsf{edata}\ s_V\ s_D|\!\}\\ \qquad {}'\mathsf{res\_nat} :== \mathbf{PROC}\ \mathsf{dispatcher\_kernel}({}'\mathsf{eca}, {}'\mathsf{edata})\\ \qquad \{\!|\tau.\ \mathsf{kdispatch_{post}}\ \tau\ \sigma\ {}'\mathsf{res\_nat}\ {}^{\sigma}\mathsf{eca}\ {}^{\sigma}\mathsf{edata}\ s_V\ s_D|\!\}\end{array}$$

Proof As illustrated in the sections Section 6.3 and Section 6.4, we use the abstract model functions to describe the semantic effect of the particular parts of the VAMOS kernel.

Aim of this proof is the combination of these single parts to the overall transition function δ_{V+D}. In doing this, the crux of the matter is to follow the different case distinctions of the implementation in the model. For this reason, we have to establish a series of relations. The abstract states are given, as usual, by s_V and s_D. Furthermore, we use the following abbreviations: vm_{cp} denotes the virtual machine of the current process and *stat* the value of the status register, such that $eca = \mathsf{mca\_nat}\ vm_{cp}\ s_D\ stat$ denotes the exception cause. Subsequently, we only give the claims of the lemmata. The complete proofs can be found in `mcaLemmata.thy`.

1. The implementation uses the bits of mask UEXCEPT_MASK in the status register to determine runtime errors. In the model, the process output RUNTIME_ERROR denotes runtime errors. Thus, if the implementation recognizes runtime errors, the model as well has to recognize them:

   $$(eca \wedge_u \mathsf{UEXCEPT\_MASK}) \neq 0 \Longrightarrow \omega_{asm}\ \lceil vm_{cp} \rceil = \text{RUNTIME\_ERROR}$$

2. An active bit EXCEPT_TRAP in *eca* signals a trap exception and causes the invocation of dispatch_kernel_call. The model, in turn, handles traps with function vamosDispatcher which is called as long as a current process exists:

   $$(eca \wedge_u \mathsf{EXCEPT\_TRAP}) \neq 0 \Longrightarrow \mathsf{v\_cup}\ s_V.\mathsf{schedds} \neq \bot$$

3. Timer interrupts are identified by an active bit DEVICE_TIMER_BIT in *eca*. In contrast, the model utilizes predicate isTimer for this purpose. The following statement establishes a relation between both:

   $$((eca \wedge_u \mathsf{DEVICE\_TIMER\_BIT}) \neq 0) = \mathsf{isTimer}\ \{x.\ \mathsf{is\_int\_devs\_single}\ (s_D\ x)\}$$

4. The implementation uses the bitmask UEXT_INT_MASK to identify external device interrupts. The model subsumes the external interrupts in the set $\{\mathsf{x.\ is\_int\_devs\_single\ s_D}\} \cap \mathsf{v\_mask}\ (\mathsf{v\_devds}\ \mathsf{s_V}) - \{\mathsf{DEV\_TIMER}\}$. The relation is given by:

   $$\begin{array}{l}((eca \wedge_u \mathsf{UEXT\_INT\_MASK}) = 0) =\\ (\{x.\ \mathsf{is\_int\_devs\_single}\ (s_D\ x)\} \cap s_V.\mathsf{devds.enabled} - \{\mathsf{DEV\_TIMER}\} = \{\})\end{array}$$

In order to ensure that the implementation and the model serve the same interrupts, the following relation is estabished:

$\lceil \mathsf{mca2ints}\ eca \rceil =$
$\{x.\ \mathsf{is\_int\_devs\_single}\ (s_{\mathsf{D}}\ x)\} \cap s_{\mathsf{V}}.\mathsf{devds.enabled} - \{\mathsf{DEV\_TIMER}\}$

These relations enable the model to reproduce the function calls in the implementation. Combining the particular parts is tedious due to the various case distinctions but straightforward in the end.

The remaining steps in dispatcher_kernel apply to the return value of the VAMOS invocation. It contains the process identifier of the process to be scheduled next, if any such process exists. Otherwise, the VAMOS kernel returns 0. Establishing the connection between this return value and the current process identifier v_cup in the model, as required in the postcondition, is straightforward Later on, this connection is used to establish the equivalence between currentp in CVM and v_cup in the VAMOS model.

The theory file `vamosDispatcherKernel_proof.thy` contains the Hoare triple of dispatcher_kernel and the functional correctness proof with around 1300 steps. □

6.6 Correctness of a System Step

Theorem 5.3.1 already introduced the overall correctness statement of a system step.

Proof The proof of the theorem mainly depends on the correctness statement for the VAMOS kernel in Theorem 6.5.1. Comparing pre- and post-conditions of the functions system_step and dispatcher_kernel reveals great analogies. Differences are the abstraction relations and the representation of the reset case.

As shown in the definition, $\mathsf{Abs}_{\mathsf{V+D}}$ requires, compared to $\mathsf{weakAbs}_{\mathsf{V+D}}$, the equivalence of the current processes in CVM and VAMOS. This relation is not needed in the context of the VAMOS kernel and simply ignored in its precondition $\mathsf{kdispatch}_{\mathsf{pre}}$. Thus, we just have to deal with the different representations of the reset case, i. e., on the one hand, the external signal reset and, on the other hand, the first bit of the exception cause eca.

Looking into the implementation of system_step displays that an active reset entails the setting of values eca to 1 and edata to 0. Thus, on the level of dispatcher_kernel, an active reset is equivalent to $(\mathsf{eca}\ \wedge_{\mathsf{u}}\ 1) \neq 0$. With the external component cvmX initialized as requested, the precondition $\mathsf{kdispatch}_{\mathsf{pre}}$ follows immediately.

Without an active reset signal, the functions mca_nat and edata_nat determine the values of eca and edata. Additionally, the running process (if it exists) is advanced through $\mathsf{user}_{\mathsf{step}}$ before the VAMOS kernel comes into play. Again, deriving the precondition of dispatcher_kernel from this context is rather straightforward. Instead of currentp, the precondition $\mathsf{kdispatch}_{\mathsf{pre}}$ determines the current process by means of v_cup. The equivalence of both is given by $\mathsf{Abs}_{\mathsf{V+D}}$. Furthermore, the values for eca and edata rely on the concrete virtual machine of the current process, whereas $\mathsf{kdispatch}_{\mathsf{pre}}$ uses the abstract one to recompute the values. Both computations, however, are consistent because both machines are in-sync before the application of $\mathsf{user}_{\mathsf{step}}$.

With discharging the precondition of the VAMOS kernel, its postcondition $\mathsf{kdispatch}_{\mathsf{post}}$ becomes available. With it, the remaining proof steps apply to the re-establishing of the equivalence of currentp and v_cup. Postcondition $\mathsf{kdispatch}_{\mathsf{post}}$ does not preserve this equivalence but lays the foundation by relating the current process identifier v_cup of the VAMOS model with the result value of function dispatcher_kernel. Setting the current process currentp of CVM in accordance to this result value establishes the relation between currentp and v_cup as required by $\mathsf{Abs}_{\mathsf{V+D}}$.

The theory file `kernelStep.thy` contains the Hoare triple of system_step and the functional correctness proof with around 300 steps. □

Chapter 7

Conclusion

This work deals with the VAMOS kernel as part of Verisoft's Academic System. Both the kernel implementation as well as the abstract model are fully embedded into the complete system. The VAMOS kernel runs on the verified VAMP processor and provides enough functionality to serve as basis for a simple operating system [15] which, in turn, enables applications for exchanging and managing signed and encrypted e-mails.

On the abstract level, the VAMOS model is also completely integrated in the model stack. It is directly based on the underlying layers, like CVM, and uses definitions from there literally. In a similar way, subsequent models, such as COUP [25] or SOS [15], literally rely on definitions of VAMOS. However, the seamless integration of both, the implementation and the abstract model of the VAMOS kernel, is not the only achievement. In addition, the VAMOS model is used to show the functional correctness of the VAMOS implementation.

It can therefore be stated that the VAMOS model represents a general-purpose kernel model that is not tailor-made for one specific purpose. This fact contributes significantly to the relevance of the VAMOS model and constitutes the fundamental difference between our *pervasive* approach compared to *isolated* approaches that only focus on a single layer. In the latter approaches, the models are usually adapted to the actual verification goal. If it is the only goal to show that the formal specification precisely describes the implementation, the specification tends to move closely towards the implementation. If a model is used as fundament of an underlying component without a proof, there is a likelihood that the model over-abstracts the actual implementation. Furthermore, regarding one single layer at a time necessarily leads to self-contained, independent models. When combining those different layers later on, there is a tremendous proof effort necessary to establish simulation theorems between the adjacent layers. Our approach prevents this effort by the specification design.

Establishing a simulation proof between the abstract VAMOS model and the concrete, fairly realistic implementation of the VAMOS kernel is a further achievement and ensures that the model can safely be used in the upper layers, as it happened with the operating system model of Bogan [15] and Daum's proof of the VAMOS scheduler fairness [25].

To our knowledge, such a proof based on a model stack of this concrete level of detail and with such a clean, seamless logical foundation has been undertaken for the first time.

We believe that our work represents a significant step towards the grand challenge of "real code verification", although we compromised in a number of ways in order to achieve our goal:

- the VAMP is not a "real", i. e., industrial-strength processor,
- C0 is a typed, simplified fragment of C; this forces to shift more low-level computations into assembly programs as necessary in a more powerful C execution model, and
- the VAMOS code has been written by the verification engineers themselves, and often with an eye on the tool-chain and the verification task.

Despite these simplifications, which were partly applied for project-pragmatic reasons, we maintain that our models are still not too far away from industrial-strength processors, C code and microkernel implementations such as the PikeOS kernel. Whether it is ever possible to verify system-level programming code of a substantial size written *without* any regard to verification is a fully open question at present.

We would like to argue that the possibility of adapting specifications, code, and tool-chain to each other simplified the task of achieving our goal. Besides the obvious foresight that all code was written in C0 and had to live with the restrictions imposed by our tool-chain, we see the following (incomplete) list of mutual influences:

- The abstract model uses infinite datatypes to model key entities like time, processes, etc., while the concrete implementation of these entities is bound to bit-vector representations of numbers. We ensured the abstraction relation by using a solely relative notion of time within the kernel. Furthermore, a capability-like management of process identifiers allows for a conceptually infinite name space of identifiers.
- The fairly simple abstraction scheme between $\mathsf{system\_step}$ and $\delta_{\mathsf{V+D}}$ required a lot of experimenting within the definition of the abstraction relation, which has to cope with the fact that parts of the concrete computations "overtake" their abstract counterparts and vice versa.
- The precise form of the contracts and the invariants in the implementation certainly needed the consideration on what was actually required in the proofs at higher levels.

It turned out that an obstacle to our work was the lack of early, systematic validation of the specification; at the end, we found substantially more errors there than in the (fairly well-tested) implementation, although some intricate bugs could only be revealed by the verification. One of the bugs revealed during the verification was a race condition involving the coincidence of four special cases at the same time: If (a) a process issued an IPC call, (b) the IPC partner was not yet waiting, and in the same processor cycle, (c) the timer interrupt was raised, and (d) the timeslice of the process was used up, then, the process was erroneously re-added to the ready queue.

That bugs are not necessarily revealed during the verification of a single layer examplifies the following: An earlier version of function system_step did not assure that at least one assembly step of the current process is executed. That was not a problem regarding the code correctness proof but breaks the fairness theorem of the VAMOS scheduler.

Thus, in conclusion, we point out that there is a fundamental difference between our pervasive approach and the idea of an isolated verification focusing at one layer at a time.

Furthermore, we experienced that pervasiveness entails more than just cumulative verification efforts on several (isolated) layers. In fact, it was a challenging task to integrate models and proofs into a uniform, coherent theory.

Establishing the abstract VAMOS model took nearly two person years and comprises about 2.000 lines of code in Isabelle/HOL.

Our kernel implementation (without CVM) consists of roughly three thousand lines of C0 code. The functional correctness proof covers around two thirds of the VAMOS implementation and comprises approximately 60.000 proof steps. With about 30.000 proof steps, a large portion of these steps applies to the IPC-relevant parts. Showing the functional correctness of the VAMOS scheduler involves roughly 9.000 proof steps. The remaining steps were mainly deployed to combine the particular parts of the VAMOS kernel and thus to show Theorem 5.3.1. The total effort regarding the code correctness proofs was around two person years.

Note that the specifications of the not yet proven parts of the VAMOS kernel are completely integrated in our framework and used to show the overall kernel correctness. The remaining proofs would be time-consuming (approximately 12 to 15 person months) but clearly follow the same scheme as the ones that have already been accomplished.

Due to the usage of while-loops in the initialization, we expect some effort in order to find the corresponding loop-invariants. Once found, however, the proof will probably be straightforward. Proving the correctness of the functions process_kill und change_rights might also be a bit expensive as they affect pending IPC operations. Nevertheless, we do not see any serious difficulties because the influences are similar to those, the timer-interrupt handler has on IPC operations. The main effort regarding the remaining functions is to show that the error cases in the implementation correspond to those in the model. Here again, we see no difficulties.

The formal verification work of this thesis is complete with the exception of the contracts for the Cvm operations, which have been shown by Tsyban *et al.* [40, 73, 75] but on a lower abstraction level than the form used in the thesis. The missing link is a transfer lemma similar to the one shown by Alkassar *et al.* [6].

7.1 Future Work

Additional microkernel features could certainly extend our work. As an example, we could introduce multi-threading generalizing the multi-processing of Vamos. This feature amounts to sharable address spaces (possibly including user-level paging) and is primarily an extension of the Cvm framework.

A similar extension is the mapping of device addresses directly into user memory. This change would abandon the kernel calls DEV_READ and DEV_WRITE for device communication, and thus substantially improve the system performance by reducing both, the size of the kernel and the frequency of kernel calls. Despite these benefits, we complicate the abstract kernel model with respect to device communication because we need a more sophisticated detection of device accesses. For that reason, we did not implement this optimization right from the beginning. From our experience today, however, we do not foresee substantial obstacles in this change.

Following the last two arguments, we could even strive for the direct memory access (DMA) by devices. Admittedly, this feature requires a more elaborate reorganization of the abstract kernel models and prevents the clean isolation of processor and devices.

Another direction of further research is a port of the Cvm framework to a mass-market processor such as an embedded PowerPC core or to an optimizing compiler supporting a larger subset of C. In contrast to additional microkernel features, this change would not necessarily require changes to the implementation of Vamos (apart from Cvm). Provided that the ported hardware maintains the same Cvm specification, we could hence draw on the current proofs of the code correctness.

Chapter 8

Appendix

8.1 Formal Specification of the Vamos Trap Dispatcher

As soon as the current instruction of the current process denotes a TRAP instruction, the VAMOS kernel invokes the trap handler. Each TRAP instruction comes with an immediate constant *imm* and the trap handles realizes a case distinction over the different values of *imm*. Based on that, it specifies the argument passing and converts the register values into the abstract ones in order to determine the corresponding abstract process output.

Function trap_dispatch encapsulates the formal specification of the VAMOS trap handler. As the formal definition of trap_dispatch is quite substantial, we split its definition up into several parts and present them (as already done in Section 3.3.3) in form of implications, subsequently.

Creating a new process. The VAMOS call PROCESS_CREATE is associated with the trap number 1:

```
$\mathsf{current\_instr}\ s_{\mathsf{proc}} = \text{TRAP}\ 1 \Longrightarrow$
$\mathsf{trap\_dispatch}\ s_{\mathsf{proc}} \equiv$
  $\textbf{let}\ \mathit{tsl}_{\mathsf{reg}} = s_{\mathsf{proc}}.\mathsf{gprs}\ !\ 11;\ \mathit{prio}_{\mathsf{reg}} = s_{\mathsf{proc}}.\mathsf{gprs}\ !\ 12;$
      $\mathit{saddr}_{\mathsf{reg}} = s_{\mathsf{proc}}.\mathsf{gprs}\ !\ 13;\ \mathit{len}_{\mathsf{reg}} = s_{\mathsf{proc}}.\mathsf{gprs}\ !\ 14;$
      $\mathit{pgs}_{\mathsf{reg}} = s_{\mathsf{proc}}.\mathsf{gprs}\ !\ 15$
  $\textbf{in}\ \text{PROCESS\_CREATE}\ (\mathsf{to\_nat32}\ \mathit{tsl}_{\mathsf{reg}})\ (\mathsf{reg2prio}\ \mathit{prio}_{\mathsf{reg}})$
      $(\mathsf{build\_memobj}\ s_{\mathsf{proc}}\ (\mathsf{to\_nat32}\ \mathit{saddr}_{\mathsf{reg}})\ (\mathsf{to\_nat32}\ \mathit{len}_{\mathsf{reg}}))$
      $(\mathsf{to\_nat32}\ \mathit{pgs}_{\mathsf{reg}})$
```

Register 11 specifies the timeslice for the new process while register 12 determines its priority. The initial memory image is taken from the virtual memory of the calling process and specified by the start address in register 13 and the length in register 14. The number of pages that should be allocated for the new process is stored in register 15. Based on these register values, function trap_dispatch determines the corresponding abstract process output.

Cloning a process. The VAMOS call PROCESS_CLONE is associated with the trap number 2:

$\mathsf{current\_instr}\ s_{\mathsf{proc}} = \text{TRAP}\ 2 \Longrightarrow$
$\mathsf{trap\_dispatch}\ s_{\mathsf{proc}} \equiv \text{PROCESS\_CLONE}\ (s_{\mathsf{proc}}.\mathsf{gprs}\ !\ 11)$

Register 11 specifies the handle to the process that should be cloned. Based on this handle, function trap_dispatch determines the corresponding abstract process output.

Terminating a process. The VAMOS call PROCESS_KILL is associated with the trap number 3:

$\mathsf{current\_instr}\ s_{\mathsf{proc}} = \text{TRAP}\ 3 \Longrightarrow$
$\mathsf{trap\_dispatch}\ s_{\mathsf{proc}} \equiv \text{PROCESS\_KILL}\ (s_{\mathsf{proc}}.\mathsf{gprs}\ !\ 11)$

Register 11 specifies the handle to the process that should be terminated. Based on this handle, function trap_dispatch determines the corresponding abstract process output.

Changing the scheduling parameters of a process. The VAMOS call CHG_SCHED_PARAMS is associated with the trap number 4:

```
$\mathsf{current\_instr}\ s_{\mathsf{proc}} = \text{TRAP}\ 4 \Longrightarrow$
$\mathsf{trap\_dispatch}\ s_{\mathsf{proc}} \equiv$
  $\textbf{let}\ \mathit{hn}_{\mathsf{reg}} = s_{\mathsf{proc}}.\mathsf{gprs}\ !\ 11;\ \mathit{tsl}_{\mathsf{reg}} = s_{\mathsf{proc}}.\mathsf{gprs}\ !\ 12;$
      $\mathit{prio}_{\mathsf{reg}} = s_{\mathsf{proc}}.\mathsf{gprs}\ !\ 13$
  $\textbf{in}\ \text{CHG\_SCHED\_PARAMS}\ \mathit{hn}_{\mathsf{reg}}\ (\mathsf{to\_nat32}\ \mathit{tsl}_{\mathsf{reg}})\ (\mathsf{reg2prio}\ \mathit{prio}_{\mathsf{reg}})$
```

Register 11 specifies the handle to the process whose parameters should be changed. The registers 12 and 13 specify the new timeslice and priority. Based on these register values, function trap_dispatch determines the corresponding abstract process output.

Setting the privileges of a process. The VAMOS call SET_PRIVILEGES is associated with the trap number 6:

$\mathsf{current\_instr}\ s_{\mathsf{proc}} = \text{TRAP}\ 6 \Longrightarrow$
$\mathsf{trap\_dispatch}\ s_{\mathsf{proc}} \equiv \text{SET\_PRIVILEGED}\ (s_{\mathsf{proc}}.\mathsf{gprs}\ !\ 11)$

Register 11 specifies the handle to the process whose privileges should be set. Based on this handle, function trap_dispatch determines the corresponding abstract process output.

Increasing the memory amount of a process. The VAMOS call MEMORY_ADD is associated with the trap number 7:

$\mathsf{current\_instr}\ s_{\mathsf{proc}} = \textsc{trap}\ 7 \Longrightarrow$
$\mathsf{trap\_dispatch}\ s_{\mathsf{proc}} \equiv$
$\quad \textbf{let}\ \mathit{hn}_{\mathsf{reg}} = s_{\mathsf{proc}}.\mathsf{gprs}\ !\ 11;\ \mathit{pgs}_{\mathsf{reg}} = s_{\mathsf{proc}}.\mathsf{gprs}\ !\ 12$
$\quad \textbf{in}\ \textsc{memory\_add}\ \mathit{hn}_{\mathsf{reg}}\ (\mathsf{to\_nat32}\ \mathit{pgs}_{\mathsf{reg}})$

Register 11 specifies the handle to the process that should get more memory and register 12 contains the number of the additional memory pages. Based on these register values, function trap_dispatch determines the corresponding abstract process output.

Decreasing the memory amount of a process. The VAMOS call MEMORY_ADD is associated with the trap number 8:

$\mathsf{current\_instr}\ s_{\mathsf{proc}} = \textsc{trap}\ 8 \Longrightarrow$
$\mathsf{trap\_dispatch}\ s_{\mathsf{proc}} \equiv$
$\quad \textbf{let}\ \mathit{hn}_{\mathsf{reg}} = s_{\mathsf{proc}}.\mathsf{gprs}\ !\ 11;\ \mathit{pgs}_{\mathsf{reg}} = s_{\mathsf{proc}}.\mathsf{gprs}\ !\ 12$
$\quad \textbf{in}\ \textsc{memory\_free}\ \mathit{hn}_{\mathsf{reg}}\ (\mathsf{to\_nat32}\ \mathit{pgs}_{\mathsf{reg}})$

Register 11 specifies the handle to the process and register 12 contains the number of the memory pages that should be released. Based on these register values, function trap_dispatch determines the corresponding abstract process output.

Changing a driver. The VAMOS call CHANGE_DRIVER is associated with the trap number 9:

$\mathsf{current\_instr}\ s_{\mathsf{proc}} = \textsc{trap}\ 9 \Longrightarrow$
$\mathsf{trap\_dispatch}\ s_{\mathsf{proc}} \equiv$
$\quad \textbf{let}\ \mathit{hn}_{\mathsf{reg}} = s_{\mathsf{proc}}.\mathsf{gprs}\ !\ 11;\ \mathit{ints}_{\mathsf{reg}} = s_{\mathsf{proc}}.\mathsf{gprs}\ !\ 12$
$\quad \textbf{in}\ \textsc{change\_driver}\ \mathit{hn}_{\mathsf{reg}}$
$\qquad (\mathsf{reg2ints}\ (\textbf{if}\ \mathsf{reg2ints}\ \mathit{ints}_{\mathsf{reg}} \neq \bot\ \textbf{then}\ \mathit{ints}_{\mathsf{reg}}\ \textbf{else}\ \neg_{\mathsf{s/32}}\ \mathit{ints}_{\mathsf{reg}}))$
$\qquad (\mathsf{reg2ints}\ \mathit{ints}_{\mathsf{reg}} \neq \bot)$

Register 11 specifies the handle to the corresponding driver and register 12 the interrupts that should be registered or deregistered. Based on these register values, function trap_dispatch determines the corresponding abstract process output.

Enabling Interrupts. The VAMOS call ENABLE_INTERRUPTS is associated with the trap number 10:

$\mathsf{current\_instr}\ s_{\mathsf{proc}} = \textsc{trap}\ 10 \Longrightarrow$
$\mathsf{trap\_dispatch}\ s_{\mathsf{proc}} \equiv \textsc{enable\_interrupts}\ (\mathsf{reg2ints}\ (s_{\mathsf{proc}}.\mathsf{gprs}\ !\ 11))$

Register 11 specifies the interrupts that should be enabled. Based on this register value, function trap_dispatch determines the corresponding abstract process output.

Receiving from a process. The VAMOS call IPC_RECEIVE is associated with the trap number 11:

```
current_instr $s_{\mathsf{proc}}$ = TRAP 11 $\Longrightarrow$
trap_dispatch $s_{\mathsf{proc}}$ $\equiv$
  let $shn_{\mathsf{reg}} = s_{\mathsf{proc}}.\mathsf{gprs}\ !\ 11$; $saddr_{\mathsf{reg}} = s_{\mathsf{proc}}.\mathsf{gprs}\ !\ 15$;
      $len_{\mathsf{reg}} = s_{\mathsf{proc}}.\mathsf{gprs}\ !\ 16$; $to_{\mathsf{reg}} = s_{\mathsf{proc}}.\mathsf{gprs}\ !\ 20$
  in IPC_RECEIVE $shn_{\mathsf{reg}}$
      (build_buffer $s_{\mathsf{proc}}$ (to_nat32 $saddr_{\mathsf{reg}}$) (to_nat32 $len_{\mathsf{reg}}$))
      (int2timeout $to_{\mathsf{reg}}$)
```

Register 11 specifies the handle to the sender. The buffer is determined by the start address in register 15 and the length in register 16. The timeout for this operation is stored in register 20. Based on these register values, function trap_dispatch determines the corresponding abstract process output.

Sending to a process. The VAMOS call IPC_SEND is associated with the trap number 12:

```
current_instr $s_{\mathsf{proc}}$ = TRAP 12 $\Longrightarrow$
trap_dispatch $s_{\mathsf{proc}}$ $\equiv$
  let $rhn_{\mathsf{reg}} = s_{\mathsf{proc}}.\mathsf{gprs}\ !\ 11$; $srights_{\mathsf{reg}} = s_{\mathsf{proc}}.\mathsf{gprs}\ !\ 12$;
      $saddr_{\mathsf{reg}} = s_{\mathsf{proc}}.\mathsf{gprs}\ !\ 13$; $len_{\mathsf{reg}} = s_{\mathsf{proc}}.\mathsf{gprs}\ !\ 14$;
      $ahn_{\mathsf{reg}} = s_{\mathsf{proc}}.\mathsf{gprs}\ !\ 17$; $arights_{\mathsf{reg}} = s_{\mathsf{proc}}.\mathsf{gprs}\ !\ 18$;
      $to_{\mathsf{reg}} = s_{\mathsf{proc}}.\mathsf{gprs}\ !\ 19$
  in IPC_SEND $rhn_{\mathsf{reg}}$ (num2rights $srights_{\mathsf{reg}}$)
      (build_memobj $s_{\mathsf{proc}}$ (to_nat32 $saddr_{\mathsf{reg}}$) (to_nat32 $len_{\mathsf{reg}}$)) $ahn_{\mathsf{reg}}$
      (num2rights $arights_{\mathsf{reg}}$) (int2timeout $to_{\mathsf{reg}}$)
```

The handle to the receiver is given in register 11 and register 12 specifies the rights, the sender wants to grant to the receiver. By means of the start address in register 13 and the length in register 14, the sender determines the memory message. If an additional process should be introduced, register 17 contains the corresponding handle and register 18 the associated rights. Register 19 determines the timeout of the operation. Based on these register values, function trap_dispatch determines the corresponding abstract process output.

Combined Sending and Receiving. The VAMOS call IPC_REQUEST is associated with the trap number 15:

```
current_instr $s_{\mathsf{proc}}$ = TRAP 15 $\Longrightarrow$
trap_dispatch $s_{\mathsf{proc}}$ $\equiv$
  let $hn_{\mathsf{rcv}} = s_{\mathsf{proc}}.\mathsf{gprs}\ !\ 11$; $rights_{\mathsf{snd}} = s_{\mathsf{proc}}.\mathsf{gprs}\ !\ 12$;
      $msg_{\mathsf{ad}} = s_{\mathsf{proc}}.\mathsf{gprs}\ !\ 13$; $msg_{\mathsf{len}} = s_{\mathsf{proc}}.\mathsf{gprs}\ !\ 14$;
      $hn_{\mathsf{add}} = s_{\mathsf{proc}}.\mathsf{gprs}\ !\ 17$; $rights_{\mathsf{add}} = s_{\mathsf{proc}}.\mathsf{gprs}\ !\ 18$;
      $to_{\mathsf{snd}} = s_{\mathsf{proc}}.\mathsf{gprs}\ !\ 19$; $buf_{\mathsf{ad}} = s_{\mathsf{proc}}.\mathsf{gprs}\ !\ 15$;
```

$$\begin{array}{l}
\qquad \mathit{buf}_{\mathsf{len}} = s_{\mathsf{proc}}.\mathsf{gprs}\ !\ 16;\ \mathit{to}_{\mathsf{rcv}} = s_{\mathsf{proc}}.\mathsf{gprs}\ !\ 20 \\
\quad \textbf{in}\ \textsc{ipc\_request}\ \mathit{hn}_{\mathsf{rcv}}\ (\mathsf{num2rights}\ \mathit{rights}_{\mathsf{snd}}) \\
\qquad (\mathsf{build\_memobj}\ s_{\mathsf{proc}}\ (\mathsf{to\_nat32}\ \mathit{msg}_{\mathsf{ad}})\ (\mathsf{to\_nat32}\ \mathit{msg}_{\mathsf{len}}))\ \mathit{hn}_{\mathsf{add}} \\
\qquad (\mathsf{num2rights}\ \mathit{rights}_{\mathsf{add}})\ (\mathsf{int2timeout}\ \mathit{to}_{\mathsf{snd}}) \\
\qquad (\mathsf{build\_buffer}\ s_{\mathsf{proc}}\ (\mathsf{to\_nat32}\ \mathit{buf}_{\mathsf{ad}})\ (\mathsf{to\_nat32}\ \mathit{buf}_{\mathsf{len}})) \\
\qquad (\mathsf{int2timeout}\ \mathit{to}_{\mathsf{rcv}})
\end{array}$$

The combined sending and receiving was already described in Section 3.3.3.

Reading from a device. The Vamos call DEV_READ is associated with the trap numbers 13 and 19, whereas the former denotes a single read and the latter a burst read:

$$\begin{array}{l}
\mathsf{current\_instr}\ s_{\mathsf{proc}} = \textsc{trap}\ 13 \lor \mathsf{current\_instr}\ s_{\mathsf{proc}} = \textsc{trap}\ 19 \Longrightarrow \\
\mathsf{trap\_dispatch}\ s_{\mathsf{proc}} \equiv \\
\quad \textbf{let}\ \mathit{dev}_{\mathsf{reg}} = s_{\mathsf{proc}}.\mathsf{gprs}\ !\ 11;\ \mathit{port}_{\mathsf{reg}} = s_{\mathsf{proc}}.\mathsf{gprs}\ !\ 12; \\
\qquad \mathit{saddr}_{\mathsf{reg}} = s_{\mathsf{proc}}.\mathsf{gprs}\ !\ 13;\ \mathit{len}_{\mathsf{reg}} = s_{\mathsf{proc}}.\mathsf{gprs}\ !\ 14 \\
\quad \textbf{in}\ \mathsf{Let}\ (\textbf{if}\ \mathsf{current\_instr}\ s_{\mathsf{proc}} = \textsc{trap}\ 19\ \textbf{then}\ \mathsf{BufLength}\ 1 \\
\qquad\qquad \textbf{else}\ \mathsf{build\_buffer}\ s_{\mathsf{proc}}\ (\mathsf{to\_nat32}\ \mathit{saddr}_{\mathsf{reg}})\ (\mathsf{to\_nat32}\ \mathit{len}_{\mathsf{reg}})) \\
\qquad (\textsc{dev\_read}\ (\mathsf{reg2devnum}\ \mathit{dev}_{\mathsf{reg}})\ (\mathsf{reg2port}\ \mathit{port}_{\mathsf{reg}}))
\end{array}$$

Register 11 contains the device number and register 12 the corresponding port number. The buffer which is specified by the start address in register 13 and the length in register 14 is only required in the context of a burst read. Based on these register values, function trap_dispatch determines the corresponding abstract process output.

Writing to a device. The Vamos call DEV_WRITE is associated with the trap numbers 14 and 20, whereas the former denotes a single write and the latter a burst write:

$$\begin{array}{l}
\mathsf{current\_instr}\ s_{\mathsf{proc}} = \textsc{trap}\ 14 \lor \mathsf{current\_instr}\ s_{\mathsf{proc}} = \textsc{trap}\ 20 \Longrightarrow \\
\mathsf{trap\_dispatch}\ s_{\mathsf{proc}} \equiv \\
\quad \textbf{let}\ \mathit{dev}_{\mathsf{reg}} = s_{\mathsf{proc}}.\mathsf{gprs}\ !\ 11;\ \mathit{port}_{\mathsf{reg}} = s_{\mathsf{proc}}.\mathsf{gprs}\ !\ 12; \\
\qquad \mathit{saddr}_{\mathsf{reg}} = s_{\mathsf{proc}}.\mathsf{gprs}\ !\ 13;\ \mathit{len}_{\mathsf{reg}} = s_{\mathsf{proc}}.\mathsf{gprs}\ !\ 14 \\
\quad \textbf{in}\ \mathsf{Let}\ (\textbf{if}\ \mathsf{current\_instr}\ s_{\mathsf{proc}} = \textsc{trap}\ 20\ \textbf{then}\ \mathsf{MObjSeq}\ [s_{\mathsf{proc}}.\mathsf{gprs}\ !\ 13] \\
\qquad\qquad \textbf{else}\ \mathsf{build\_memobj}\ s_{\mathsf{proc}}\ (\mathsf{to\_nat32}\ \mathit{saddr}_{\mathsf{reg}})\ (\mathsf{to\_nat32}\ \mathit{len}_{\mathsf{reg}})) \\
\qquad (\textsc{dev\_write}\ (\mathsf{reg2devnum}\ \mathit{dev}_{\mathsf{reg}})\ (\mathsf{reg2port}\ \mathit{port}_{\mathsf{reg}}))
\end{array}$$

Register 11 contains the device number and register 12 the corresponding port number. The message which is specified by the start address in register 13 and the length in register 14 is only relevant in case of a burst write. Otherwise, only the data at the start address is written to the device. Based on these register values, function trap_dispatch determines the corresponding abstract process output.

Changing IPC rights. The Vamos call CHANGE_RIGHTS is associated with the trap number 16:

$\mathsf{current\_instr}\ s_{\mathsf{proc}} = \textsc{trap}\ 16 \Longrightarrow$
$\mathsf{trap\_dispatch}\ s_{\mathsf{proc}} \equiv$
$\quad \textbf{let}\ \mathit{subj\_hn}_{\mathsf{reg}} = s_{\mathsf{proc}}.\mathsf{gprs}\ !\ 11;\ \mathit{obj\_hn}_{\mathsf{reg}} = s_{\mathsf{proc}}.\mathsf{gprs}\ !\ 12;$
$\qquad \mathit{rights}_{\mathsf{reg}} = s_{\mathsf{proc}}.\mathsf{gprs}\ !\ 13$
$\quad \textbf{in}\ \textsc{change\_rights}\ \mathit{subj\_hn}_{\mathsf{reg}}\ \mathit{obj\_hn}_{\mathsf{reg}}\ (\mathit{rights}_{\mathsf{reg}} \leq 15)$
$\qquad (\mathsf{num2rights}\ (\textbf{if}\ \mathit{rights}_{\mathsf{reg}} \leq 15\ \textbf{then}\ \mathit{rights}_{\mathsf{reg}}\ \textbf{else}\ \neg_{\mathsf{s/32}}\ \mathit{rights}_{\mathsf{reg}}))$

Register 11 contains the handle to the subject, whereas register 12 contains the handle to the object. The rights are specified in register 13. Based on these register values, function trap_dispatch determines the corresponding abstract process output.

Reading Kernel Information. The VAMOS call CHANGE_RIGHTS is associated with the trap number 17:

$\mathsf{current\_instr}\ s_{\mathsf{proc}} = \textsc{trap}\ 17 \Longrightarrow$
$\mathsf{trap\_dispatch}\ s_{\mathsf{proc}} \equiv \textsc{read\_kernel\_info}\ (\mathsf{int2kinfo}\ (s_{\mathsf{proc}}.\mathsf{gprs}\ !\ 11))$

Register 11 specifies the type of kernel notifications. Based on this type, function trap_dispatch determines the corresponding abstract process output.

Undefined Traps. A trap number *imm* is considered as undefined, if it fulfills the predicate undefined_trapnr:

$\mathsf{undefined\_trapnr}\ \mathit{imm} \equiv$
$\quad \mathit{imm} \notin \{1, 2, 3, 4, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 19, 20\}$

The specification of an undefined trap number leads to the process output UNDEFINED_TRAP which is reflected in the following implication:

$[\![\mathsf{current\_instr}\ s_{\mathsf{proc}} = \textsc{trap}\ i;\ \mathsf{undefined\_trapnr}\ i]\!]$
$\Longrightarrow \mathsf{trap\_dispatch}\ s_{\mathsf{proc}} = \textsc{undefined\_trap}$

Bibliography

[1] Alkassar, E., Hillebrand, M., Knapp, S., Rusev, R., Tverdyshev, S.: Formal device and programming model for a serial interface. In: B. Beckert (ed.) Proceedings, 4th International Verification Workshop (VERIFY), Bremen, Germany, pp. 4–20. CEUR-WS Workshop Proceedings (2007). URL http://www-wjp.cs.uni-saarland.de/publikationen/Alkassar_AHK-.pdf

[2] Alkassar, E., Hillebrand, M.A.: Formal functional verification of device drivers. In: J. Woodcock, N. Shankar (eds.) Verified Software: Theories, Tools, and Experiments, *LNCS*, vol. 5295, pp. 225–239. Springer (2008)

[3] Alkassar, E., Hillebrand, M.A., Leinenbach, D., Schirmer, N.W., Starostin, A.: The Verisoft approach to systems verification. In: N. Shankar, J. Woodcock (eds.) Verified Software: Theories, Tools, and Experiments, *LNCS*, vol. 5295, pp. 209–224. Springer (2008)

[4] Alkassar, E., Hillebrand, M.A., Leinenbach, D.C., Schirmer, N.W., Starostin, A., Tsyban, A.: Balancing the load – leveraging a semantics stack for systems verification. J. Autom. Reasoning, Special Issue on Operating System Verification (2009). To appear in this volume.

[5] Alkassar, E., Hillebrand, M.A., Paul, W., Petrova, E.: Automated verification of a small hypervisor. In: P. O'Hearn, G.T. Leavens, S. Rajamani (eds.) Verified Software: Theories, Tools, Experiments (VSTTE 2010), Lecture Notes in Computer Science. Springer, Edinburgh, UK (2010). To appear.

[6] Alkassar, E., Schirmer, N., Starostin, A.: Formal pervasive verification of a paging mechanism. In: TACAS, *LNCS*, vol. 4963, pp. 109–123. Springer (2008)

[7] Baumann, C., Beckert, B., Blasum, H., Bormer, T.: Better avionics software reliability by code verification – A glance at code verification methodology in the Verisoft XT project. In: Embedded World 2009 Conference. Franzis Verlag, Nuremberg, Germany (2009). URL http://www.uni-koblenz.de/~beckert/pub/embeddedworld2009.pdf. To appear.

[8] Baumann, C., Beckert, B., Blasum, H., Bormer, T.: Formal verification of a microkernel used in dependable software systems. In: B. Buth, G. Rabe, T. Seyfarth (eds.) Computer Safety, Reliability, and Security (SAFECOMP 2009), *Lecture Notes in Computer Science*, vol. 5775, pp. 187–200. Springer, Hamburg, Germany (2009). URL http://www.uni-koblenz.de/~beckert/pub/safecomp2009.pdf

[9] Beckert, B., Beuster, G.: Formal specification of security-relevant properties of user interfaces. In: Proceedings, 3rd International Workshop on Critical Systems Development with UML, pp. 139–146. Lisbon, Portugal (2004). URL http://wwwbib.informatik.tu-muenchen.de/infberichte/2004/TUM-I0415.pdf. TU Munich Technical Report TUM-I0415

[10] Beuster, G., Henrich, N., Wagner, M.: Real world verification – Experiences from the Verisoft email client. In: G. Sutcliffe, R. Schmidt, S. Schulz (eds.) Proceedings of the FLoC'06 Workshop on Empirically Successful Computerized Reasoning (ESCoR 2006), *CEUR Workshop Proceedings*, vol. 192, pp. 112–125. CEUR-WS.org (2006). URL http://ftp.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-192/paper08.pdf

[11] Bevier, W.R.: Kit and the short stack. J. Autom. Reasoning **5**(4), 519–530 (1989)

[12] Bevier, W.R., Hunt, Jr., W.A., Moore, J.S., Young, W.D.: An approach to systems verification. J. Autom. Reasoning **5**(4), 411–428 (1989)

[13] Beyer, S., Jacobi, C., Kröning, D., Leinenbach, D., Paul, W.: Instantiating uninterpreted functional units and memory system: functional verification of the vamp. In: D. Geist, E. Tronci (eds.) CHARME 2003, *LNCS*, vol. 2860, pp. 51–65. Springer (2003). URL http://www-wjp.cs.uni-saarland.de/publikationen/BJKLP03.pdf

[14] Beyer, S., Jacobi, C., Kröning, D., Leinenbach, D., Paul, W.J.: Putting it all together: Formal verification of the VAMP. STTT **8**(4-5), 411–430 (2006)

[15] Bogan, S.: Formal specification of a simple operating system. Ph.D. thesis, Saarland University, Saarbrücken (2008). URL http://www-wjp.cs.uni-saarland.de/publikationen/Bog08.pdf

[16] Böhme, S., Leino, K.R.M., Wolff, B.: HOL-Boogie—An interactive prover for the Boogie program-verifier

[17] Böhme, S., Moskal, M., Schulte, W., Wolff, B.: HOL-Boogie — An interactive prover-backend for the Verifying C Compiler. Journal of

Automated Reasoning **44**(1–2), 111–144 (2010). DOI http://dx.doi.org/10.1007/s10817-009-9142-9

[18] Boyton, A.: A verified shared capability model. In: G. Klein, R. Huuck, B. Schlich (eds.) Proceedings of the 4th Workshop on Systems Software Verification, *Electronic Notes in Computer Science*, vol. 254, pp. 25–44. Elsevier, Aachen, Germany (2009)

[19] Chebiryak, Y.: Formal specification and verification of functions of vamos scheduler. Master thesis, Saarland University (2005)

[20] Cohen, E., Alkassar, E., Boyarinov, V., Dahlweid, M., Degenbaev, U., Hillebrand, M., Langenstein, B., Leinenbach, D., Moskal, M., Obua, S., Paul, W., Pentchev, H., Petrova, E., Santen, T., Schirmer, N., Schmaltz, S., Schulte, W., Shadrin, A., Tobies, S., Tsyban, A., Tverdyshev, S.: Invariants, modularity, and rights. In: Perspectives of Systems Informatics (PSI 2009), Lecture Notes in Computer Science. Springer (2009). Invited paper, to appear.

[21] Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: A practical system for verifying concurrent C. In: S. Berghofer, T. Nipkow, C. Urban, M. Wenzel (eds.) Theorem Proving in Higher Order Logics (TPHOLs 2009), *Lecture Notes in Computer Science*, vol. 5674, pp. 23–42. Springer, Munich, Germany (2009). Invited paper.

[22] Cohen, E., Moskal, M., Schulte, W., Tobies, S.: A precise yet efficient memory model for C (2008). Availabe via `http://research.microsoft.com/apps/pubs/default.aspx?id=77174`

[23] Cohen, E., Moskal, M., Schulte, W., Tobies, S.: Local verification of global invariants in concurrent programs. In: B. Cook, P. Jackson, T. Touili (eds.) Computer Aided Verification (CAV 2010), Lecture Notes in Computer Science. Springer, Edinburgh, UK (2010). To appear.

[24] Dalinger, I., Hillebrand, M., Paul, W.: On the verification of memory management mechanisms. In: D. Borrione, W. Paul (eds.) CHARME 2005, LNCS. Springer (2005). URL `http://www-wjp.cs.uni-saarland.de/publikationen/DHP05.pdf`

[25] Daum, M.: On the formal foundation of a verification approach for system-level concurrent programs. Ph.D. thesis, Saarland University, Computer Science Department (2010)

[26] Daum, M., Dörrenbächer, J., Bogan, S.: Model stack for the pervasive verification of a microkernel-based operating system. In:

B. Beckert, G. Klein (eds.) 5th International Verification Workshop (VERIFY'08), *CEUR Workshop Proceedings*, vol. 372, pp. 56–70. CEUR-WS.org (2008). URL http://www-wjp.cs.uni-saarland.de/publikationen/DaumDB-VERIFY08-.pdf

[27] Daum, M., Dörrenbächer, J., Wolff, B.: Proving fairness and implementation correctness of a microkernel scheduler. Journal of Automated Reasoning: Special Issue on Operating System Verification **42, Numbers 2-4**, 349–388 (2009). URL http://www-wjp.cs.uni-saarland.de/publikationen/DaumDW-jarosv08.pdf

[28] Daum, M., Dörrenbächer, J., Wolff, B., Schmidt, M.: A verification approach for system-level concurrent programs. In: J. Woodcock, N. Shankar (eds.) Verified Software: Theories, Tools, and Experiments, *LNCS*, vol. 5295, pp. 161–176. Springer (2008)

[29] Daum, M., Schirmer, N.W., Schmidt, M.: Implementation correctness of a real-time operating system. In: 7th IEEE International Conference on Software Engineering and Formal Methods (SEFM 2009), 23–27 November 2009, Hanoi, Vietnam, pp. 23–32. IEEE (2009)

[30] Elkaduwe, D., Klein, G., Elphinstone, K.: Verified protection model of the seL4 microkernel. In: J. Woodcock, N. Shankar (eds.) Proceedings of Verified Software: Theories, Tools and Experiments 2008, *Lecture Notes in Computer Science*, vol. 5295, pp. 99–114. Springer, Toronto, Canada (2008)

[31] Endrawaty: Verification of the Fiasco IPC implementation. Master's thesis, Dresden University of Technology (2005)

[32] Engler, D.R., Kaashoek, M.F., O'Toole, J.: Exokernel: An operating system architecture for application-level resource management. In: SOSP, pp. 251–266. ACM (1995)

[33] Fleisch, B.D., Co, M.A.A.: Workplace microkernel and OS: a case study. Softw. Pract. Exper. **28**(6), 569–591 (1998)

[34] Gargano, M., Hillebrand, M., Leinenbach, D., Paul, W.: On the correctness of operating system kernels. In: J. Hurd, T.F. Melham (eds.) TPHOLs 2005, *LNCS*, vol. 3603, pp. 1–16. Springer (2005). URL http://www-wjp.cs.uni-sb.de/publikationen/GHLP05.pdf

[35] Heiser, G., Elphinstone, K., Kuz, I., Klein, G., Petters, S.M.: Towards trustworthy computing systems: taking microkernels to the next level. Operating Systems Review **41**(4), 3–11 (2007)

[36] Heitmeyer, C., Archer, M., Leonard, E., McLean, J.: Applying formal methods to a certifiably secure software system. IEEE Transactions on Software Engineering **34**, 82–98 (2008). DOI http://doi.ieeecomputersociety.org/10.1109/TSE.2007.70772

[37] Heitmeyer, C.L., Archer, M., Leonard, E.I., Mclean, J.: Formal specification and verification of data separation in a separation kernel for an embedded system. In: In CCS, pp. 346–355. ACM (2006)

[38] Hennessy, J.L., Patterson, D.A.: Computer Architecture: A Quantitative Approach, 2nd Edition. Morgan Kaufmann (1996)

[39] Hohmuth, M., Tews, H., Stephens, S.G.: Applying source-code verification to a microkernel: the VFiasco project. In: ACM SIGOPS European Workshop, pp. 165–169. ACM (2002). DOI http://doi.acm.org/10.1145/1133373.1133405

[40] In der Rieden, T., Tsyban, A.: CVM – a verified framework for microkernel programmers. In: Systems Software Verification, *ENTCS*, vol. 217, pp. 151–168. Elsevier Science B.V. (2008)

[41] Kaiser, R.: Combining partitioning and virtualization for safety-critical systems. White Paper WP_CPV_10_A4_R10, SYSGO AG (2007). Availabe via http://www.sysgo.com/news-events/whitepapers/

[42] Klein, G.: Correct OS kernel? proof? done! USENIX ;login: **34**(6), 28–34 (2009)

[43] Klein, G.: Operating system verification — an overview. Sādhanā **34**(1), 27–69 (2009)

[44] Klein, G., Andronick, J., Elphinstone, K., Heiser, G., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: Formal verification of an OS kernel. Communications of the ACM **53**(6), 107–115 (2010)

[45] Langenstein, B., Nonnengart, A., Rock, G., Stephan, W.: A history-based verification of distributed applications. In: B. Beckert (ed.) Proceedings, 4th International Verification Workshop (VERIFY), Bremen, Germany, *CEUR Workshop Proceedings*, vol. 259, pp. 70–84. CEUR-WS.org (2007). URL http://ftp.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-259/paper08.pdf

[46] Langenstein, B., Nonnengart, A., Rock, G., Stephan, W.: Verification of distributed applications. In: F. Saglietti, N. Oster (eds.) Computer Safety, Reliability, and Security, 26th International Conference, SAFECOMP 2007, Nuremberg, Germany, September 18–21, 2007, *Lecture Notes in Computer Science*, vol. 4680, pp. 315–328. Springer (2007)

[47] Lassmann, G., Rock, G., Schwan, M., Cheikhrouhou, L.: Verisoft secure biometric identification system. URL http://www.informatik.hu-berlin.de/forschung/gebiete/algorithmenII/Publikationen/Papers/Verisoft.pdf. T-Systems International University Conference, Düsseldorf, 10.-11. October 2005

[48] Leinenbach, D.: Compiler verification in the context of pervasive system verification. Ph.D. thesis, Saarland University (2008). URL http://www-wjp.cs.uni-sb.de/publikationen/Lei08.pdf

[49] Leinenbach, D., Paul, W.J., Petrova, E.: Towards the formal verification of a C0 compiler: Code generation and implementation correctness. In: SEFM, pp. 2–12. IEEE Computer Society (2005)

[50] Leinenbach, D., Petrova, E.: Pervasive compiler verification: From verified programs to verified systems. In: Systems Software Verification, *ENTCS*, vol. 217, pp. 23–40. Elsevier Science B.V. (2008)

[51] Liedtke, J.: Improving IPC by kernel design. In: SOSP, pp. 175–188. ACM (1993)

[52] Liedtke, J.: On μ-kernel construction. In: SOSP, pp. 237–250. ACM (1995)

[53] Liedtke, J.: Towards real microkernels. Commun. ACM **39**(9), 70–77 (1996)

[54] Martin, W., White, P., Taylor, F.S., Goldberg, A.: Formal construction of the mathematically analyzed separation kernel. In: ASE '00: Proceedings of the 15th IEEE international conference on Automated software engineering, p. 133. IEEE Computer Society, Washington, DC, USA (2000)

[55] Moore, J.S.: A grand challenge proposal for formal methods: A verified stack. In: 10th Anniversary Colloquium of UNU/IIST, pp. 161–172. Springer (2002)

[56] Moura, L.D., Bjørner, N.: Z3: An efficient smt solver. In: In Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS (2008)

[57] Müller, S., Paul, W.: Computer Architecture, Complexity and Correctness. Springer Verlag (2000)

[58] Neumann, P.G., Feiertag, R.J.: PSOS revisited. In: ACSAC, pp. 208–216. IEEE Computer Society (2003)

[59] Nguiekom, V.: Verifikation von doppelt verketteten listen auf pointerebene. Master's thesis, Saarland University (2005)

[60] Ni, Z., Yu, D., Shao, Z.: Using XCAP to certify realistic systems code: Machine context management. In: TPHOLs, *LNCS*, vol. 4732, pp. 189–206. Springer (2007)

[61] Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic, *LNCS*, vol. 2283. Springer (2002)

[62] Paulson, L.C.: Isabelle: A Generic Theorem Prover. Springer (1994). LNCS 828

[63] Petrova, E.: Verification of the C0 compiler implementation on the source code level. Ph.D. thesis, Saarland University (2007)

[64] Rashid, R., Avadis Tevanin, J., Young, M., Golub, D., Baron, R.: Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. IEEE Trans. Comput. **37**(8), 896–908 (1988)

[65] In der Rieden, T.: Verified linking for modular kernel verification. Ph.D. thesis, Saarland University, Saarbrücken (2009). URL `http://www-wjp.cs.uni-sb.de/publikationen/Idr09.pdf`

[66] Samman, T.: Verifying 50,000 lines of C code. Futures, Microsoft's European Innovation Magazine **21** (2008)

[67] Schierboom, E.: Verification of Fiasco's IPC implementation. Master's thesis, Radboud Universiteit Nijmegen (2007)

[68] Schirmer, N.: A verification environment for sequential imperative programs in Isabelle/HOL. In: LPAR, LNCS, pp. 398–414. Springer (2005). URL `http://isabelle.in.tum.de/~schirmer/pub/hoare-lpar04.html`

[69] Schirmer, N.: Verification of sequential imperative programs in Isabelle/HOL. Ph.D. thesis, TU Munich (2006)

[70] Shapiro, J., Doerrie, M.S., Northup, E., Sridhar, S., Miller, M.: Towards a verified, general-purpose operating system kernel. In: FM Workshop on OS Verification, Tech. Rep. 0401005T-1, pp. 1–19. National ICT Australia (2004). URL `http://www.coyotos.org/docs/osverify-2004/osverify-2004.pdf`

[71] Starostin, A.: Formal verification of a c-library for strings. Master's thesis, Saarland University (2006). URL `http://www-wjp.cs.uni-saarland.de/publikationen/St06.pdf`

[72] Starostin, A.: Formal verification of demand paging. Ph.D. thesis, Saarland University, Saarbrücken (2010). URL `http://www-wjp.cs.uni-saarland.de/publikationen/St10.pdf`

[73] Starostin, A., Tsyban, A.: Correct microkernel primitives. In: Systems Software Verification, *ENTCS*, vol. 217, pp. 169–185. Elsevier Science B.V. (2008)

[74] Starostin, A., Tsyban, A.: Verified process-context switch for c-programmed kernels. In: N. Shankar, J. Woodcock (eds.) 2nd IFIP Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE'08), *LNCS*, vol. 5295, pp. 240–254. Springer (2008)

[75] Starostin, A., Tsyban, A.: Verified process-context switch for C-programmed kernels. In: N. Shankar, J. Woodcock (eds.) Verified Software: Theories, Tools, and Experiments, *LNCS*, vol. 5295, pp. 240–254. Springer (2008)

[76] Tsyban, A.: Cvm - a verified framework for microkernel programmers. Ph.D. thesis, Saarland University, Saarbrücken (2009)

[77] Tverdyshev, S.: Formal verification of gate-level computer systems. Ph.D. thesis, Saarland University, Computer Science Department (2009). URL http://www-wjp.cs.uni-saarland.de/publikationen/Tv09.pdf

[78] Verisoft Project: Verisoft repository (2010). URL http://www.verisoft.de/VerisoftRepository.html

[79] Walker, B.J., Kemmerer, R.A., Popek, G.J.: Specification and verification of the ucla unix security kernel. Commun. ACM **23**(2), 118–131 (1980). DOI http://doi.acm.org/10.1145/358818.358825