

On the Formal Foundation of a Verification Approach for System-Level Concurrent Programs



Dissertation

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

Matthias Daum

md11@wjpserver.cs.uni-saarland.de

Saarbrücken, 10. August 2010

Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit selbständig 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, den 26. Februar 2010

Tag des Kolloquiums: 28. Juni 2010
Dekan: Prof. Dr. Holger Hermanns
Vorsitzender des Prüfungsausschusses: Prof. Dr. Raimund Seidel
1. Berichterstatter: Prof. Dr. Wolfgang J. Paul
2. Berichterstatter: Prof. Dr. Burkhard Wolff
3. Berichterstatter: Prof. Dr. Reinhard Wilhelm
akademischer Mitarbeiter: Dr. Eyad Alkassar

In memoriam
RICHARD PRASSE
1915 – 2008

I wake up every morning, and I still haven't finished the book.

Donald Knuth about "The Art of Computer Programming"

Acknowledgments

During graduation, many people have supported me, and though I cannot name everyone, I am grateful to all of them. In particular, I am indebted to Prof. Wolfgang Paul for the opportunity to work at his chair. Furthermore, I thank my colleagues, who devoted much time to many lively and fruitful discussions, among them Sebastian Bogan, Jan Dörrenbächer, Mark Hillebrand, Dirk Leinenbach, Norbert Schirmer, Mareike Schmidt, and Burkhard Wolff. Last but not least, I am much obliged to my family and friends who never grew tired in encouraging me.

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

Though program verification is known and used since decades, the verification of a complete computer system still remains a grand challenge. In essence, this challenge stems from the interaction of various programs. Different techniques have been proposed for the verification of communicating programs. Common to all, however, is that they rely on several (usually implicit) assumptions about the underlying system. Typically, such assumptions include compiler correctness, scheduler fairness, and a certain non-interference between the local program behavior and its environment. This thesis aims at discharging these assumptions for the processes of the microkernel VAMOS. More specifically, this work formally justifies the abstraction from a kernel model with explicit, deterministic scheduling to a concurrent process system with non-deterministic but temporally fair scheduling. Our formal results form the foundation of a verification approach for system-level concurrent programs. We outline this approach on example properties of a user-mode operating system.

Kurze Zusammenfassung

Obwohl es schon jahrzehntelang Programmverifikation gibt, wird die Verifikation eines kompletten Computersystems auch heute noch als eine große Herausforderung angesehen. Im Wesentlichen ergibt sich diese Herausforderung aus der vielfältigen Interaktion von Programmen. Verschiedene Techniken wurden für die Verifikation kommunizierender Programme vorgeschlagen. Alle haben jedoch gemein, dass sie sich auf mehrere (meist implizite) Annahmen über das zugrunde liegende System stützen. In der Regel sind solche Annahmen Compiler-Korrektheit, Scheduler-Fairness und eine gewisse Störfreiheit des lokalen Programmverhaltens vom Verhalten seiner Umgebung. Die vorliegende Dissertation beschäftigt sich mit der Entlastung dieser Annahmen für die Prozesse des Mikrokernelns VAMOS. Genauer gesagt, rechtfertigt diese Arbeit formal die Abstraktion von einem Kernmodell mit explizitem, deterministischem Scheduling zu einem nebenläufigen Prozesssystem mit nicht-deterministischem, aber temporal fairem Scheduling. Die formalen Ergebnisse bilden die Grundlage eines Verifikationsansatzes für nebenläufige, systemnahe Programme. Dieser Ansatz wird am Beispiel von Eigenschaften eines User-Mode-Betriebssystems erläutert.

Ausführliche Zusammenfassung

Werden Computer in sicherheitskritischen Bereichen eingesetzt, spielt Robustheit – und insbesondere die Vermeidung menschlicher Irrtümer – eine entscheidende Rolle. *Verifikation* ist die sicherste Technik zum Ausschluss systematischer Fehler beim Systementwurf. Dazu wird das korrekte Verhalten des untersuchten Systems durch ein mathematisches Modell spezifiziert und ein Beweis dafür entwickelt, dass der Systementwurf dieser Spezifikation genügt. Von *formaler Verifikation* sprechen wir, wenn der Beweis durch einen Computer überprüft wird.

In der Regel werden Computersysteme in mehreren Ebenen realisiert: Schon die Hardware ist auf den verschiedenen Abstraktionsebenen modelliert; das Betriebssystem stützt sich auf die abstrakteste Hardware-Schicht, und dieses bietet wiederum eine Abstraktion an, auf deren Basis schließlich die Benutzerprogramme implementiert werden. Will man alle systematischen Fehler im kompletten Systementwurf sicher ausschließen, muss man alle Ebenen in die Verifikation mit einbeziehen. Die Verifikation eines Systems über mehrere Abstraktionsebenen hinweg nennen wir *durchgängig*.

Obwohl Programmverifikation seit Jahrzehnten bekannt ist und vielfach eingesetzt wird, bleibt die durchgängige formale Verifikation eines vollständigen Computersystems – angefangen vom Hardware-Design bis hinauf zu den Benutzerprogrammen – nach wie vor eine große Herausforderung [Moo02]. Ein ganz wesentlicher Aspekt dieser Herausforderung besteht darin, dass reale Programme vielfältig miteinander kommunizieren. Selbst einfachste Beispiele wie ein “Hallo Welt”-Programm erzeugt Ausgabedaten; in der Regel wird das Betriebssystem angewiesen, diese Daten an ein Ausgabegerät zu übertragen.

Betriebssystemkerne stellen Benutzerprogrammen eine Prozessorabstraktion mit exklusivem Zugang zu Ressourcen wie Registern und Speicher zur Verfügung. Diese Abstraktion wird üblicherweise *Prozess* genannt [Kno74]. Ein wichtiges Merkmal von Prozessen ist ein Mittel zur Kommunikation mit dem Kern zur Anfrage weiterer Dienste. Da die meisten Programme kompiliert werden, um schließlich als Prozess unter einem Betriebssystemkern zu laufen, verallgemeinern wir den üblichen Prozessbegriff etwas auf eine Programmsemantik mit Primitiven zur Kommunikation mit einem Kern.

Zur Verifikation kommunizierender Programme wurden verschiedene Techniken vorgeschlagen. Allen gemeinsam ist jedoch, dass sie auf mehreren (meist impliziten) Annahmen über das zugrunde liegende System beruhen. Häufig schließen diese Annahmen Compiler-Korrektheit, Scheduler-Fairness und eine gewisse Störfreiheit des lokalen Programmverhaltens vom Verhalten seiner Umgebung ein. Die vorliegende Dissertation beschäftigt sich mit der Entlastung dieser Annahmen für die Prozesse des Mikrokerns VAMOS. Genauer gesagt, rechtfertigt diese Arbeit formal die Abstraktion von einem Kernmodell mit explizitem, deterministischem Scheduling zu einem nebenläufigen Prozesssystem mit nicht-deterministischem, aber temporal fairem Scheduling.

Im Rahmen der vorliegenden Arbeit wurde der VAMOS-Kern verbessert, indem sowohl Beschränkungen der maximalen Laufzeit des Kerns aufgehoben wurden, als auch die Fairness des Scheduling-Algorithmus hergestellt wurde.

Darüber hinaus wurde ein Compiler-Satz für sequentielle Sprachen auf die Semantik von Prozessen erweitert. Dieses Ergebnis basiert auf dem von Leinenbach & Petrova [LP08] formal verifizierten Compiler. Dieser Compiler übersetzt aus dem C-Dialekt C0 in Assembler. Darauf aufbauend wurden (a) die Kommunikationsprimitive des VAMOS-Kerns formalisiert, (b) C0- und Assembler-Prozesse spezifiziert, indem die sequentiellen Sprachen um die Kommunikationsprimitive erweitert wurden und (c) ein Simulationstheorem entwickelt, das die Prozessmodelle verbindet und den ursprünglichen, sequentiellen Compiler-Satz erweitert.

Die Prozessmodelle werden in zwei Kernmodellen verwendet: ein deterministisches Modell, welches das genaue Verhalten von VAMOS einschließlich seines Schedulers spezifiziert, und ein nicht-deterministisches, nebenläufiges Modell kommunizierender Prozesse, das vom Scheduler abstrahiert. Zur Entwicklung des ersteren haben wir wesentlich beigetragen, das letztere haben wir allein entwickelt. In einem noch laufenden Verifikationsversuch wird das deterministische Modell benutzt, um die korrekte Implementierung von VAMOS zu zeigen [DDW09, Sect. 6]. Teil der vorliegenden Arbeit war es dagegen, die Simulation zwischen dem deterministischen und dem nebenläufigen System zu zeigen.

Schließlich wurde ein Begriff temporaler Fairness definiert. Basierend auf der Kernspezifikation wurde dann formal gezeigt, dass der VAMOS-Scheduler dieser Definition gerecht wird.

Zusammengefasst umfasst die Arbeit drei formale Hauptergebnisse: (a) einen Ansatz zum Erweitern von Compiler-Korrektheit von rein sequentiellen Sprachen auf Prozesse, (b) einen Simulationssatz zwischen einem VAMOS-Modell mit explizitem Scheduler und einem nebenläufigen Kernmodell und (c) einen formalen Beweis für das temporal faire Scheduling von Prozessen in VAMOS. Gemeinsam bilden diese Ergebnisse die Grundlage eines Verifikationsansatzes für nebenläufige, systemnahe Programme. Dieser Ansatz wird am Beispiel von Eigenschaften eines User-Mode-Betriebssystems erläutert.

Contents

1	Introduction	1
1.1	Notation	3
1.2	Basic Concepts	5
1.2.1	Kleene Algebras and the Kleene Star	5
1.2.2	Future-Time Linear Temporal Logic	6
1.3	Related Work	7
1.3.1	Verification Methods for Concurrent Programs	8
1.3.2	Verification Frameworks for Concurrent Programs	9
1.3.3	Kernel Verification	10
1.3.4	Operating-System Verification	11
2	Fundamentals	13
2.1	Implementing the Academic System: The Software Stack	14
2.2	Verifying Sequential Imperative Programs in Verisoft	15
2.3	The Language C0	16
2.4	The VAMP Assembly Language	18
2.5	On a Correct Compiler	20
2.6	Isabelle/Simpl – a Verification Environment for C0	25
2.7	Extending the Language Stack: a Roadmap to Concurrency	25
3	The VAMOS Microkernel: Design and Implementation	27
3.1	Requirements on a Correct Microkernel’s Design	28
3.2	Functionality	28
3.2.1	Inter-Process Communication	30
3.2.2	Process Scheduling	31
3.3	Overflow-Safe Implementation of the Timeout Management	31
3.4	Conclusion	34
4	Process Abstraction	35
4.1	Motivation	36
4.2	Process Models	37
4.3	The Assembly-Process Model	43
4.4	The C0-Process Model	47
4.5	Extended Compiler Correctness	55
4.6	Conclusion	62

5	Adding Concurrency: The Kernel Models	63
5.1	Overview of the Model Components	64
5.2	A Simple Kernel Call: <code>SET_PRIVILEGED</code> Revisited	67
5.3	Process Termination	68
5.4	Inter-Process Communication	71
5.4.1	IPC Rights Management	71
5.4.2	Specifying IPC in the VAMOS Model	72
5.4.3	IPC in CoUP	76
5.5	The Scheduler Specification	78
5.6	Invariants over Execution Traces	81
5.7	Conclusion	85
6	Simulation: Formally Relating the Kernel Models	87
6.1	Kernel-Model Simulation	87
6.2	Abstracting Concurrent Assembly Processes to C0 Processes	90
6.2.1	On the Commutability of Transitions: From Trace Theory to Kleene Algebra	91
6.2.2	Process Simulation Meets True Concurrency	92
6.2.3	Commuting Transitions in CoUP	93
6.3	Conclusion	95
7	A Temporal Property: Scheduler Fairness	97
7.1	An Introductory Scheduling Example	97
7.2	Common Notions of Fairness	98
7.3	Developing and Proving Prioritized Fairness	99
7.4	Rephrasing the Temporal Property in LTL	103
7.5	Conclusion	104
8	Towards Verifying a User-Mode Operating System	107
8.1	Non-Termination	108
8.2	Fair Scheduling of Applications	110
8.2.1	Priorities in the SOS	110
8.2.2	Applying Scheduler Fairness to Applications	110
8.3	Conclusion	112
9	Conclusion and Future Work	115
9.1	Formal Results	115
9.2	Measuring the Formal Verification Effort	117
9.3	Outlook	118

1 Introduction

What is truth?

Pilate, according to John, 18:38

Contents

1.1	Notation	3
1.2	Basic Concepts	5
1.2.1	Kleene Algebras and the Kleene Star	5
1.2.2	Future-Time Linear Temporal Logic	6
1.3	Related Work	7
1.3.1	Verification Methods for Concurrent Programs	8
1.3.2	Verification Frameworks for Concurrent Programs	9
1.3.3	Kernel Verification	10
1.3.4	Operating-System Verification	11

If computers are used in a highly secure or even safety-critical context, robustness – and in particular the avoidance of human errors – plays an essential role. *Verification* is the most rigorous technique for the exclusion of systematic design errors. This approach specifies the correct behavior of the examined system by a mathematical model and develops a proof that the system design meets this specification. We speak of *formal* verification if the proof is checked by a computer.

Usually, computer systems are implemented in several layers: Even the hardware is modeled at different levels of abstraction, the operating system is based on the hardware layer, and on top of that, the user programs are implemented. In order to safely exclude all systematic errors in the design of the complete system, all layers must be regarded. If a verification attempt spans over multiple layers of a system, we call it *pervasive*.

While program verification has been known and used over decades, the pervasive, formal verification of a complete computer system from the hardware design up to application programs still remains a grand challenge [Moo02]. In essence, the challenge stems from the interaction of various programs in a real system – even a simplistic “Hello World”-program produces output data, usually instructing the operating system to transfer the data to the output peripheral.

Operating-system kernels provide a processor abstraction to user programs with the exclusive access to resources like registers and memory. This abstraction is commonly referred to as a *process* [Kno74]. An important feature of processes is a means to communicate with the kernel to request further services. As most programs are eventually

compiled to run as a process under a kernel, we slightly generalize the common process notion to a program semantics with primitives for the communication with a kernel.

Different techniques have been proposed for the verification of communicating programs. Common to all, however, is that they rely on several (usually implicit) assumptions about the underlying system. Typically, such assumptions include compiler correctness, fairness, and a certain non-interference between the local program behavior and its environment. This thesis aims at discharging these assumptions for the processes of the microkernel VAMOS. More specifically, this work formally justifies the abstraction from a kernel model with explicit, deterministic scheduling to a concurrent process system with non-deterministic but temporally fair scheduling.

In the course of our work, we have improved the VAMOS kernel by overcoming limitations of the maximal runtime, on the one hand, and by enabling fair scheduling, on the other hand.

Furthermore, we have extended a compiler theorem for sequential languages to processes. In particular, our work is based on Leinenbach & Petrova’s [LP08] formally verified compiler. Their compiler translates from the C dialect *C0* into assembly language. We have (a) formalized the communication primitives of the VAMOS kernel, (b) specified *C0* and assembly processes by enriching the sequential languages with communication primitives, and (c) formally stated a simulation theorem relating the process models, which extends the original compiler-correctness theorem that relates the sequential semantics.

The process models are used in two kernel models: a deterministic model specifying the exact VAMOS behavior, and a non-deterministic, concurrent model of communicating processes, which abstracts from the scheduler. We have contributed to the former and have developed the latter. While an ongoing verification attempt uses the deterministic specification to show that VAMOS is correctly implemented [DDW09, Sect. 6], we have shown that the concurrent model indeed simulates the kernel specification.

Finally, we have defined a temporal fairness notion. Based on the kernel specification, we have formally shown that the VAMOS scheduler is fair according to this definition.

Summarizing, this thesis presents three main results: (a) an approach to extend compiler correctness from purely sequential languages to processes, (b) a simulation theorem from a VAMOS model with explicit scheduling to a concurrent kernel model, and (c) a formal proof for the fair scheduling of processes in VAMOS. These results form the foundation of a verification approach for system-level concurrent programs. We sketch this approach on example properties of a user-mode operating system.

Outline. The remainder of this chapter introduces notation, recapitulates general mathematical and logical concepts, and presents related work. The next chapter, in contrast, describes specific background information about the context and prerequisites of this thesis. It concludes with a detailed overview of the actual thesis work.

The main part of this thesis starts in [Chapter 3](#), which provides an insight into the design principles of the VAMOS kernel and exemplifies an implementation challenge. [Chapter 4](#) details the kernel communication primitives and the semantics of assembly and *C0*

processes. Furthermore, compiler correctness is extended to processes with communication primitives. [Chapter 5](#) then introduces two kernel models describing the interleaved execution of processes in the kernel. [Chapter 6](#) sketches the proof of independence of process transitions in an interleaved execution between adjacent communication points as well as the embedding of C0 processes into a concurrent kernel model. [Chapter 7](#) reports on the verification that the scheduler of our microkernel is indeed fair. Finally, [Chapter 8](#) exemplifies how the provided models can be used for the verification of a user-mode operating system. We conclude in [Chapter 9](#) and present future work.

1.1 Notation¹

The formalizations presented in this thesis are mechanized and checked within the generic interactive theorem prover *Isabelle* [Pau94]. 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* [NPW02], which is based on the typed λ -calculus.

This article 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`. Keywords are set 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 document when we present theorems and lemmas. The *Pure* logic itself is intuitionistic higher-order logic, where universal quantification is \bigwedge , implication is \implies , and equality is \equiv . Nested implications like $P_1 \implies P_2 \implies P_3 \implies C$ are abbreviated with $\llbracket P_1; P_2; P_3 \rrbracket \implies C$, where we refer to P_1 , P_2 , and P_3 as the premises and to C as the conclusion. We may also write:

$$\frac{P_1 \quad P_2 \quad P_3}{C}$$

In the object logic HOL, universal quantification is \forall , implication is \longrightarrow , and equality is $=$. We sometimes use the abbreviation $P \longleftrightarrow Q$ for $(P \longrightarrow Q) \wedge (Q \longrightarrow P)$. The other logical and mathematical notions follow the standard notational conventions with a bias towards functional programming. *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.

¹For taming Isabelle’s powerful document-generation mechanism in general, and in particular for the major part of this section, I am indebted to Schirmer *et al.* [Sch06, AHL⁺09]

Functions. In Isabelle/HOL, all functions are total. An unnamed function can be specified using the λ -operator, e. g., $\lambda x. x$ is the identity function. In general, we prefer curried functions over functions taking an n-tuple as argument, e. g., $f(a)(b)$ instead of $f(a, b)$. As the parentheses for function application soon become distracting, we omit them and write $f a b$, instead. Note that function application binds tighter than any other operator, i. e., $f i + g j$ means $(f i) + (g j)$. We write $f \circ g$ for functional composition and recursively define the n-fold function application by $f^0 = (\lambda x. x)$ and $f^{n+1} = f \circ f^n$. Function update is $f(y := v) \equiv \lambda x. \mathbf{if} x = y \mathbf{then} v \mathbf{else} f x$.

Partial functions are usually formalized in HOL with the option type. This type is a data type with two constructors, one to inject values of the base type, e. g., $[x]$, and the other one is simply the additional element \perp . With $[y]$, the original base value x can be obtained such that $y = [x]$. Note that there is no base value x iff $y = \perp$. As HOL is a total logic, the term $[\perp]$ is nevertheless a well-defined yet unspecified value. For an update of a partial function, we write $f(y \mapsto v)$.

For data types, we write the structural case distinction over some value v as **case** v **of** $\perp \Rightarrow g \mid [x] \Rightarrow f x$. 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** $\text{FOO} \Rightarrow x \mid _ \Rightarrow y$, which means value x if v has the value FOO , and otherwise value y .

Sets, Intervals, and Relations. Sets come along with the standard operations for union, i. e., $A \cup B$, intersection, i. e., $A \cap B$ and membership, i. e., $x \in A$. We denote the interval of the numbers a and b excluding the endpoints by $\{a <..<b\}$, and including the endpoints by $\{a..b\}$. Furthermore, we write $a < c \leq b$ to denote that a number c is in the interval $\{a <..b\}$. Relational composition is written as $R_1 \circ R_2$.

Lists. The syntax and the operations for lists are similar to functional programming languages like ML or Haskell. The empty list is $[]$ and by $x \odot xs$ the list xs is preceded with the element x . We write $[a, b, c]$ instead of $a \odot b \odot c \odot []$. With $xs \odot ys$, list ys is appended to list xs . The function `concat` takes a list of lists as argument and returns the concatenation of these lists. The length of a list xs is written $|xs|$, 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 $[x \in xs . P x]$. With `replicate n e` we denote a list that consists of n elements e .

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

1.2 Basic Concepts

1.2.1 Kleene Algebras and the Kleene Star

In automata theory as well as in the theory of formal languages, *Kleene algebras* are an important class of algebraic structures. Various inequivalent definitions can be found in the literature; we adopt the one by Kozen [Koz90].

Definition 1.1 (Semiring). An algebraic structure $(U, +, \cdot)$ with the universe U and the binary operators $+$ (addition) and \cdot (multiplication) is a *semiring* if it satisfies the following constraints:

1. $(U, +)$ is a commutative monoid with identity element 0, i. e.,
 - a) $(a + b) + c = a + (b + c)$
 - b) $0 + a = a$
 - c) $a + b = b + a$
2. (U, \cdot) is a monoid with identity element 1, i. e.,
 - a) $(a \cdot b) \cdot c = a \cdot (b \cdot c)$
 - b) $1 \cdot a = a$
 - c) $a \cdot 1 = a$
3. Multiplication distributes over addition:
 - a) $(a + b) \cdot c = a \cdot c + b \cdot c$
 - b) $a \cdot (b + c) = a \cdot b + a \cdot c$
4. 0 annihilates U with respect to multiplication:
 - a) $0 \cdot a = 0$
 - b) $a \cdot 0 = 0$

Definition 1.2 (Idempotent Semiring). We call a semiring $(U, +, \cdot)$ *idempotent* if its addition is idempotent, i. e., $a + a = a$.

Definition 1.3 (Kleene Algebra). An idempotent semiring $(U, +, \cdot)$ is called a *Kleene algebra* iff

1. the universe U is partially ordered such that $a \leq b \equiv a + b = b$, and
2. a unary operation $*$ is defined satisfying the following axioms:
 - a) $1 + a \cdot a^* \leq a^*$
 - b) $1 + a^* \cdot a \leq a^*$
 - c) $a \cdot x \leq x \implies a^* \cdot x \leq x$
 - d) $x \cdot a \leq x \implies x \cdot a^* \leq x$

The operation $*$ is also called *Kleene star*. Often, it is defined as the sum over all powers of the operand, i. e., $a^* \equiv \sum_{n \geq 0} a^n$ where $a^0 \equiv 1$ and $a^{n+1} \equiv a \cdot a^n$. Depending on the application field, the Kleene star has different interpretations: In programming-language theory, it might be used as “iteration” (where multiplication means “sequencing”). In the context of concurrency, in contrast, the star often symbolizes a varying number of concurrent threads of control (with multiplication meaning “concurrency”).

1.2.2 Future-Time Linear Temporal Logic

Linear temporal logic (LTL) [Lam80b, EH86] is a modal logic [Pri57], which features modalities of time like in the statements “property P has been true” and “ P will be true”. In this document, however, we are only interested in the future modality. Furthermore, we are only concerned with the LTL language but do not define any proof rules. Below, we introduce the syntax, specify its semantics, and shortly reflect on practical aspects.

Syntax. We define the syntax of LTL recursively. An LTL formula is one of:

1. a propositional variable P ,
2. a negated LTL formula $\neg\varphi$,
3. a conjunct of two LTL formulae $\varphi \wedge \psi$,
4. the unary *next* operation applied to an LTL formula $\mathcal{N}\varphi$, or
5. the binary *until* operation of two LTL formulae $\varphi \mathcal{U} \psi$.

Semantics. Temporal logics are traditionally interpreted in terms of *Kripke structures*, a type of non-deterministic finite-state machine proposed by Saul Kripke [Kri63] in 1963. A Kripke structure is a triple (S, Δ, L) with a set S of states, a binary relation $\Delta \in S \times S$, and a set L of propositions over S . Temporal logics formulate properties on state sequences of such finite-state machines.

In Isabelle/HOL, we define the LTL semantics over sets of infinite state sequences, or *traces*, and represent a trace as a function t from natural numbers to states. In particular, the validity of a formula at time j of a trace t is defined by structural induction:

$$\begin{aligned}
(t, j) \models P &\equiv P(t, j) \\
(t, j) \models \neg\varphi &\equiv \neg((t, j) \models \varphi) \\
(t, j) \models \varphi \wedge \psi &\equiv ((t, j) \models \varphi) \wedge ((t, j) \models \psi) \\
(t, j) \models \mathcal{N}\varphi &\equiv (t, j + 1) \models \varphi \\
(t, j) \models \varphi \mathcal{U} \psi &\equiv \exists k \geq j. ((t, k) \models \psi) \wedge (\forall i \in \{j..<k\}. (t, i) \models \varphi)
\end{aligned}$$

with propositions P over states and LTL formulae φ and ψ . Furthermore, we define the validity of formula φ for a set K of traces (usually specified by a Kripke structure) as follows:

$$K \models \varphi \equiv \forall t \in K. (t, 0) \models \varphi$$

Ultimately, we specify the term $\langle \mathcal{S}^0, \Sigma, \delta \rangle_{\mathbf{A}}$ describing an *action Kripke structure with input*. This structure specifies the set of all traces that can be produced by the set of initial states \mathcal{S}^0 , the input alphabet Σ , and the transition function δ . Each trace in this set is a function from natural numbers to triples (i, s, n) , where i is the input for the transition, s is the current state, and n is the next state. Formally, we define:

$$\begin{aligned}
\langle \mathcal{S}^0, \Sigma, \delta \rangle_{\mathbf{A}} &\equiv \\
&\{t. (\exists i \ s \ n. t \ 0 = (i, s, n) \wedge s \in \mathcal{S}^0) \wedge \\
&\quad (\forall j. \exists i \ s \ n. \\
&\quad \quad t \ j = (i, s, n) \wedge \\
&\quad \quad i \in \Sigma \wedge \delta \ i \ s = n \wedge (\exists i' \ s' \ n'. t \ (j + 1) = (i', s', n') \wedge s' = n))\}
\end{aligned}$$

Abbreviations. For succinctness, we define the usual logical abbreviations truth \top_L , falsehood \perp_L , disjunct $\varphi \vee \psi$, implication $\varphi \longrightarrow \psi$, as well as the following specifically temporal-logical abbreviations:

- the unary *eventually* operation: $\diamond\varphi \equiv \top_L \mathcal{U} \varphi$,
- the unary *always* operation: $\square\varphi \equiv \neg\diamond\neg\varphi$,
- the binary *release* operation: $\varphi \mathcal{R} \psi \equiv \neg(\neg\varphi \mathcal{U} \neg\psi)$, and
- the binary *weak until* operation: $\varphi \mathcal{W} \psi \equiv \varphi \mathcal{U} \psi \vee \square\varphi$.

The first two operations can be perceived as existential and universal quantifier over traces. Release and weak until are both variations of the until operation.

Common to all formal logics is that they precisely define certain terms of everyday language. In temporal logic, this is particularly true for the temporal conjunction until: In everyday language, it is unclear whether the terminating event must occur. A statement $\varphi \mathcal{U} \psi$, however, is definitely false if ψ never becomes true. Some authors therefore call this operator the *strong until*. For the weaker statement that the formula φ remains valid as long as the formula ψ is not satisfied, we denote: $\varphi \mathcal{W} \psi$ (weak until). A similar statement is expressed by the release operator, where, in contrast to weak until, both substatements must simultaneously hold in the terminating state, i. e.,

$$(K \models \varphi \mathcal{R} \psi) \longleftrightarrow (K \models \psi \mathcal{W} \psi \wedge \varphi)$$

which easily follows from the definitions of the involved operations.

Terminology. Certainly, the operations might be combined, e. g., $\square\diamond\varphi$. In the context of temporal logics, it is custom to state that property φ holds *always eventually*. In natural language, such a statement might confuse. Note that temporal logics implicitly refers to a trace, such that *always* can be understood as *on any suffix* of a trace. You may equally refer to $\square\diamond\varphi$ as *infinitely often* φ .

Application Domain. LTL is usually employed to show certain kinds of properties:

Safety properties claim that a certain property holds for all states in a trace: $\square P$

Liveness properties claim that some property is eventually satisfied: $\diamond P$

Fairness properties claim that a property holds infinitely often in a trace: $\square\diamond P$

Persistence properties claim that a property is established once and for all: $\diamond\square P$.

Certainly, hybrid forms are possible like a fairness property relying on some persistence: $\diamond\square P \longrightarrow \square\diamond Q$. We may view partial correctness of a program as a safety property: Given that a terminating state is reached, the computational result should have some form or value: $\square(\lambda s. \text{terminated } s \longrightarrow \text{corr_res } s)$. Program termination, in contrast, is a liveness property: $\diamond\text{terminated}$.

1.3 Related Work

The work in this thesis is mainly related to two research areas: the verification of concurrent programs and the pervasive verification of operating systems. Both research

areas can further be subdivided. In the former, we distinguish abstract verification methods and formal verification frameworks implementing such methods. In the latter, we regard the verified layer: Most operating systems cope with the vast complexity of their task by separating their core functionality, which runs in the privileged *system mode* of the processor, from additional services, which run in the unprivileged *user mode*. Consequently, we divide verification efforts into those targeting the core or *kernel* and those aiming for correct user-mode components. Note that this thesis draws upon a verified operating-system kernel and provides the necessary basis to pervasively verify the user-mode parts of an operating system. We summarize important advances in all four research areas.

1.3.1 Verification Methods for Concurrent Programs

Numerous methods have been proposed for reasoning about concurrent programs. Most notably, the methods vary in the means of communication between different threads of execution. In practice, we distinguish communication via synchronous message passing and shared memory with atomic operations. Note, however, that in principle, both concepts are equivalent, i. e., a message passing system can be transformed into one with shared memory and vice versa [LS84].

Logics for message-passing systems are Hoare’s *Communicating Sequential Processes* [Hoa85] and Milner’s *Calculus of Communicating Systems* [Mil82]. These logics are based on the fact that the observable behavior of a communicating thread of execution is solely determined by the messages it sends and receives. Thus, they describe a thread by its communication pattern and without an internal state. Certainly, the internal state is theoretically irrelevant unless reflected by the communication pattern. Practically, however, programs have state and justifying the transformation of a program into its stateless communication pattern is a most complex task. Hence, we rather prefer a process description with an internal state.

There are several verification methods for reasoning about concurrent programs with shared memory and atomic operations. Owicki & Gries [OG76] have proposed a method, where the program is annotated with assumptions on the shared variables between each atomic operation. These assumptions are required to remain stable on every operation of the other threads. This dependency on the other threads, however, makes the Owicki-Gries method effectively non-modular because the inference step, i. e., the combination of proof results for two threads, becomes extremely laborious.

Ashcroft [Ash75] has suggested to formulate a huge, single state invariant to verify concurrent programs. This method disposes of the inference step, making verification completely thread-modular. Jones [Jon83], in contrast, has recommended to abstract the intermediate assumptions of the Owicki-Gries method into global rely-guarantee conditions, which substantially simplifies the inference step and hence supports thread-modular reasoning to some extent, as well. Our process model specifies sequences of atomic transitions and features a quite general global invariant, which constrains the process inputs and outputs (see the process-validity constraints in Section 4.2 as well as the validity of process inputs in Section 4.5).

Note that though thread-modular, the methods of Ashcroft and Jones are not data-modular. Recently, Vafeiadis & Parkinson [VP07] have proposed a “marriage” of the rely-guarantee method with separation logic. Originally, Reynolds & O’Hearn [Rey02, O’H04] have developed separation logic to improve data modularity when reasoning about sequential programs with pointers. The key concept are separable assumptions about disjoint chunks of the heap memory, which can be combined with a *separating-conjunction* operator according to a so-called *frame rule*. The combination of rely-guarantee and separation logic yields a thread- and data-modular verification method.

Though all these verification methods differ in the basic means of communication from our process model, they share the notion of an internal state with our approach. It might be worthwhile to reuse these models with an adapted communication mechanism when reasoning about complex protocols between processes.

1.3.2 Verification Frameworks for Concurrent Programs

Langenstein *et al.* [LNRS07a, LNRS07b] report on a verification framework for reactive systems, which is based on an interaction-centered rely-guarantee reasoning and refinement. The basic building block of their method is the concept of communicating transition systems, whose behavior is specified by finite interaction sequences. An interaction sequence is called *history*, and collects assumptions on the inputs from the environment, on the one hand, as well as guarantees on the output of a transition system, on the other hand. In order to distinguish assumptions from guarantees in the history, the transition system as well as each interaction event in the history bear an identifier. If both coincide, the event is a guarantee, otherwise an assumption. The verification framework supports the extension of a history along with an inductive reasoning—similar to our LTL-based approach (see Chapter 8). A transition system can be refined by decomposition into communicating subsystems. These subsystems then share the same history specification—each subsystem, however, regards different events in the history as its guarantees. The verification of these subsystems supports the same kind of modularity as our approach. Langenstein’s verification framework, however, does not support propositions about states, which is an elementary prerequisite for the integration with a classic Hoare logic for program verification. Furthermore, the verification method relies on “a kind of simulation theorem . . . established by looking at the C0 execution mechanism” [LNRS07a, Section 2.2]. Our approach, in contrast, fits into the existing verification infrastructure for sequential reasoning and facilitates the formal verification of a simulation theorem as opposed to Langenstein’s educated guess.

A currently active research field is the use of separation logic for concurrent programs. Hobor *et al.* [HAN08] have formalized such a system in the theorem prover Coq. They have extended a sequential operational semantics for a C dialect with shared memory, spawnable threads and locks. Their approach allows for a so-called *modular* reasoning, i. e., arguments about sequential control and data-flow can be separated from arguments about concurrency. Their research, however, focusses around the specific problems of optimizing compilation of concurrent programs with shared memory. When they claim that properties proven in their concurrent separation logic are true for the program that

actually executes on the machine, they most likely assume a correct operating system.²

Prensa Nieto [Pre02] has formalized the Owicki/Gries method for correctness proofs of concurrent imperative programs with shared variables in the theorem prover Isabelle/HOL. Syntax, semantics and proof rules are defined in higher-order logic. Furthermore, the correctness of the proof rules is proven with respect to the semantics. Though this proof method currently lacks the support for procedures, the general approach can be applied to the Hoare logic of Schirmer [Sch05], which we use for the verification of C0 programs. Thus, Prensa Nieto’s approach provides a promising technique to construct a concurrent Hoare logic, whose verification results can be formally transferred down to our low-level kernel model with the help of our foundational work.

1.3.3 Kernel Verification

We are indebted to Klein for a comprehensive article [Kle09] on past and present approaches to the verification of operating-system kernels. Summarizing, Klein only presents a single, fully verified kernel: KIT, a small assembly program, that provides task isolation, device I/O, and single-word message passing. This verification project can only be referred to as groundbreaking in the area of pervasive verification. KIT is very far from any real system and the verification is based on a fairly abstract LISP execution model. Moreover, the corner-stone theorem of this work is limited to memory separation of processes.

Several past verification projects concentrated on the specification but fell short on the actual verification, e.g., UCLA Secure Unix [WKP80] or VFiasco/Robin [Tew07]. Other projects, like Embedded Device [HALM06], were successful in the verification but did not reach down to the code level. Furthermore, the Flint project [FSDG08] verified the correctness of certain low-level assembly fragments but did not yet reach full code coverage.

Since the publication of Klein’s article, two verification projects announced their success: First, functional correctness has been shown [DSS09] for the tiny real-time operating system OLOS, which fully runs in system mode. The correctness theorem extends over unbounded execution traces, and the verification comprises boot-up and assembly code. Second, the L4.verified project [HEK⁺07, KEH⁺09] completed the world’s first refinement proof for a general-purpose microkernel. Their verification target, *seL4*, is a third-generation microkernel of L4 provenance comparable to other high-performance L4 kernels. It comprises 8,700 lines of C and 600 lines of assembly code. Note that neither boot-up nor any assembly code have been verified; furthermore, the proof relies on a certain standard behavior of memory.³

Among ongoing verification projects is Verisoft [Pau08, HP07] and its successor, Verisoft XT.⁴ The former has not only proven functional correctness of OLOS but applies the same techniques to the larger VAMOS microkernel [DDW09, Sect.6]. Though the

²Certainly, it is possible to build a distributed system, where each thread exclusively runs on its own processor. If, however, several threads share a processor, a thread switch is required.

³For proof assumptions, see also <http://ertos.nicta.com.au/research/l4.verified/proof.pml>.

⁴More information about this project is available at <http://verisoftxt.de/>.

verification is not yet completed, it has reached a mature state. Note that the thesis at hand hinges on the correctness of VAMOS.

The latter project aims, amongst others, at the verification of the PikeOS kernel [Kai07, BBBB09] and the virtualization environment HYPER-V [Sam08, LS09]. Both verification targets are part of commercial products. PikeOS is available for Intel’s x86, PowerPC, and ARM, and has an L4-like kernel with about 6,000 lines of code. The HYPER-V, in contrast, comprises over 50,000 lines of kernel code and is based on Intel’s x86. The proofs in Verisoft XT are carried out by the VCC verification environment [CMST09, CDH⁺09, DMS⁺09], which uses a trusted tool chain comprising the C translator *VCC*, the verification condition generator *Boogie*, and the automated theorem prover *Z3*.

1.3.4 Operating-System Verification

Bogan [Bog08] provides a comprehensive overview of advances in operating-system verification. Thus, we just highlight a few results. Verification has a long tradition for improving the confidence in very complex operation-systems functionality but seldom attempts full code coverage. A fruitful target for verification have been, for instance, file systems [AZKR04] and network protocols [Smi96]. Furthermore, the level of abstraction and the verified property vary widely. Bevier *et al.* [BCT95], for instance, have specified the interface of the Synergy file system on a very abstract level while Yang *et al.* [YTEM06] used model checking to systematically check implementations for specific file-system errors.

Reliable software is particularly important for security-sensitive applications. Secure system architectures are specifically developed for such applications. These architectures are usually based on a microkernel that facilitates the separation of legacy operating-system functionality from security-sensitive services, thus limiting the trusted computing base for security-sensitive applications while providing the same functionality as traditional operating systems. This approach has been taken by the PERSEUS security framework [PRS⁺01] as well as the Nizza secure-system architecture [HHF⁺05].

In spite of the high relevance of reliance, formal verification for these architectures has to our knowledge yet been limited to the used operating-system kernel. Both architectures are based on the Fiasco kernel, whose verification has been attempted in VFiasco/Robin (see above). Beyond that, the PERSEUS project aims at an evaluation “according to the Common Criteria at a later date”.⁵ The PERSEUS framework is the basis for the TURAYA security platform [LP06]. In the scope of the Verisoft XT project, some security-critical functionality of this platform has been verified but unfortunately, the results have not yet been published.

⁵The announcement has been found on the project’s homepage, <http://www.perseus-os.org/content/pages/Evaluation.htm>.

2 Fundamentals

Jedem Anfang wohnt ein Zauber inne.

Hermann Hesse in: "Stufen"

Contents

2.1	Implementing the Academic System: The Software Stack	14
2.2	Verifying Sequential Imperative Programs in Verisoft	15
2.3	The Language C0	16
2.4	The VAMP Assembly Language	18
2.5	On a Correct Compiler	20
2.6	Isabelle/Simpl – a Verification Environment for C0	25
2.7	Extending the Language Stack: a Roadmap to Concurrency	25

Our work belongs to the Verisoft project, a large-scale research project with partners from industry and academia, funded by the German government. The project adheres to the very idea of pervasive formal verification [BHMY89] of computer systems: Each abstraction layer is justified by simulation theorems that permit transferring the results to the low-level models. All theory development is mechanized in the uniform logical framework of the interactive theorem prover Isabelle/HOL, and hence, it is rigorously checked that all verification results fit together.

The general focus of the Verisoft project is threefold: First, methods and tools should be created that permit the pervasive formal verification of computer-system designs. Second, the project pursues an increase of industrial productivity and quality. Third, Verisoft aims at a prototypical implementation and verification of four specific computer systems, three of which are from the industrial sector.

More specifically, this thesis belongs to the so-called *Academic System* (as opposed to the industrial systems), which is a computer system for the secure management and exchange 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, an e-mail client [BHW06, BB04], an e-mail server [LNRS07a, LNRS07b], and a cryptography server run on top of this operating system. These applications might be arbitrarily distributed over the trusted computers in the network.

This chapter continues with a short introduction of the implementation layers in the computer system. Furthermore, we summarize the general verification approach that has

been developed for sequential programs within Verisoft. A key feature of this approach is that it can cope with mixed-language implementations as they frequently occur in system-level software. More specifically, these languages are C0 [Lei08] and VAMP assembly [MP00]. We dedicate a subsection to each language. Afterwards, we sketch the compiler-correctness theorem, and introduce the verification environment *Isabelle/Simpl*. Finally, we present an overview of the actual task of the work at hand: the extension of the existing verification technology to concurrently executed processes.

2.1 Implementing the Academic System: The Software Stack

Figure 2.1 depicts the software layers of the academic system. The lowest software layer is called *communicating virtual machines* (CVM) [GHLP05, IT08], a generic programming framework for the implementation of operating-system kernels on the VAMP processor [MP00]. This layer encapsulates the hardware-specific low-level functionality, which employs inlined assembly. Using this framework, the microkernel VAMOS [DDB08] is implemented in C0 without extra portions of inlined assembly. Both layers interact via C0 function calls: The CVM framework calls the `kdispatch` function of VAMOS, and VAMOS calls CVM *primitives* to access low-level functionality. While these two layers run in the privileged system mode of the processor, processes run in the unprivileged user mode. In the figure, we labeled one process “OS” for the operating system and the others “App” abbreviating application (e.g., e-mail client or server). All processes communicate with VAMOS via so-called *kernel calls*.

CVM’s major task is process separation and memory virtualization. Hence, CVM includes a page-fault handler with a simple memory-swapping facility [ASS08]. All remaining kernel functionality is to be implemented in the hardware-independent part, the so-called *abstract kernel*. Technically, the CVM framework consists of the interrupt-service routine (ISR), on the one hand, and the *primitives*, on the other hand. The ISR is stored at a specific memory address and the processor executes the code at this address whenever an interrupt occurs. CVM’s ISR saves the old processor context, establishes a suitable C0 environment and calls the C0 function `kdispatch` of the abstract kernel. The CVM primitives are C0 functions employing inlined assembly code to provide low-level functionality to the abstract kernel, e.g., for the manipulation of process memory or registers. The return value of `kdispatch` instructs CVM, which process is to resume when the kernel execution finishes.

The functionality of VAMOS is accessible for processes via *kernel calls*. Technically, a kernel call is implemented using the special instruction `trap`: The sole effect of this instruction is an exception similar to, e.g., an arithmetic overflow. *Exceptions* are a

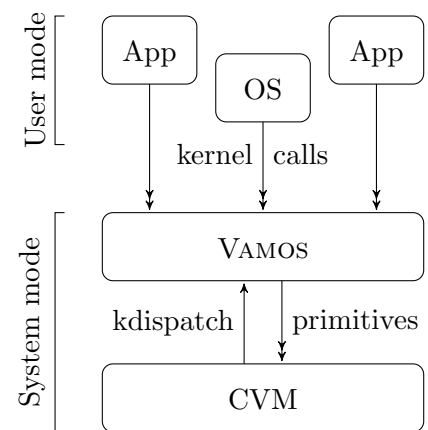


Figure 2.1: Software stack

special class of interrupts that are generated as a direct result of the program execution (as opposed to the so-called *external* or *device interrupts* that are caused by the peripherals). Thus, the exception causes – like any interrupt – a jump to CVM’s ISR, which eventually invokes VAMOS. VAMOS can examine and alter the state of the process using CVM primitives. That means, the calling process can store parameters to a kernel call in registers and memory in order to describe its specific demand from VAMOS. VAMOS, in turn, interprets the inquiry by examining the process state and in response, it alters the process state accordingly. The *application binary interface* (ABI) precisely defines this interaction between kernel and processes by assigning a kernel-call semantics to register values and memory contents.

The OS process constitutes the highest layer of our operating system. This process is initially started by VAMOS, equipped with more permissions than applications, and in charge to set up any additionally required devices and processes. Sebastian Bogan [Bog08] has designed, implemented, and formally specified the *Simple Operating System* (SOS) for this purpose. The SOS 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. Initially, the SOS sets up virtual terminals with login processes that wait for keyboard input.

2.2 Verifying Sequential Imperative Programs in Verisoft

A typical computer system comprises a stack of hardware and software layers. Each layer provides a different level of abstraction and potentially employs several implementation languages. CVM, for example, provides the abstraction of separate processes with virtual memory, and is implemented in a mixture of C0 and VAMP assembly. The reason for two implementation languages is apparent: C0 is a C-like high-level programming language and provides a comfortable abstraction layer for most programming tasks. Specific low-level functionality, however, like the access to special-purpose registers, cannot be expressed in C0. The resort for these tasks is the assembly language.

Verisoft takes this mixed-language architecture [AHL⁺09] into account by using a two-dimensional verification approach: The first axis, called *system stack*, traces the different implementation layers. The behavior of each implementation layer is specified by a corresponding computational model. The second axis follows the language semantics and, hence, we name it *language stack*.

While the full language stack reaches down to the instruction-set architecture (ISA), we are only concerned with the languages depicted in Figure 2.2. At the lower end, we find VAMP assembly (denoted “asm” for short), above it C0, and at the top, the language *Simpl*. The latter is a generic programming model for sequential imperative programming languages. This generic model provides the basis for the versatile verification environment

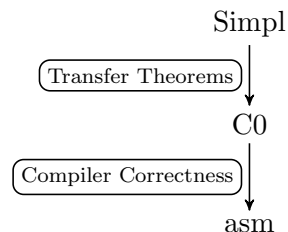


Figure 2.2: Language stack

Isabelle/Simpl.

The major challenge of a pervasive verification effort is the integration of the various verification results into a coherent theory. Thus, an important result of the Verisoft project is a compiler-correctness theorem [LP08] that links C0 and the VAMP assembly language. Furthermore, transfer theorems [Sch06] establish an embedding of C0 into the generic programming model Simpl and facilitate the transfer of properties proven in Simpl down to the C0 semantics.

In the following sections, we present the languages C0 and VAMP assembly in detail, we sketch the compiler-correctness theorem, and introduce Isabelle/Simpl.

2.3 The Language C0

ANSI C [Ame99] has a complex and highly underspecified semantics. Low-level programs such as device drivers, however, explicitly *use* properties of a particular compiler on a target hardware, like register bindings or the internal representation of data types. They can therefore not be verified based only on the vague ANSI C semantics. In Verisoft, we constrained ourselves to the C-like imperative language C0 [Lei08], which has sufficient features to implement low-level software but is interpreted by a more concrete semantics. 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.

Primitive expressions are variable names and literals. Other expressions can be composed using operators: If e and i are expressions and n is a component name, the following operations are expressions as well: array access $e[i]$, access to structure components $e.n$, dereferencing $*e$, and the “address-of” operation $\&e$. Moreover, C0 supports the usual unary and binary operations. Namely, unary operations are unary minus $-$, bitwise negation \sim , logical negation $!$, and conversion operations between integral values. Binary operations are arithmetic operations ($+$, $-$, $*$, $/$, $\%$), bitwise operations ($|$, $\&$, \wedge , \ll , \gg), and comparisons ($>$, $<$, $==$, $!=$, $>=$, $<=$) as well as the lazy binary operations (Boolean conjunction $\&\&$ and disjunction $||$). Left expressions are expressions that refer to memory objects, namely variable, array and structure accesses as well as pointer dereferencing.

LIT v	literal values v
VARACC vn	access of variable vn
ARRACC $e_a e$	indexing array e_a with index e
STRUCTACC $e cn$	selecting component cn of structure e
BINOP $bop e_1 e_2$	binary operation
LAZYPINOP $lbop e_1 e_2$	lazy binary operation
UNOP $uop e$	unary operation
ADDR OF e_1	address of left-expression e_1
DEREF e	dereferencing e

Table 2.1: C0 expressions e

The statements in C0 permit assignments, dynamic memory allocation, sequential composition, conditional and repeated execution, inlined assembly, function calls, and returns from functions.

In Isabelle/HOL, C0 expressions e and statements s are represented as data types. Their constructors are listed in the Tables 2.1 and 2.2. Only a few of them are of special interest in this thesis:

- Variable accesses VARACC vn are used to read or write the value of variables vn .
- The dereferencing operator Deref e transforms the pointer expression e into a left expression.
- Furthermore, the function-call statement SCALL $vn fn es sid$ calls a C0 function named fn with the argument list es . The returned result is stored to a (global or stack) variable called vn . The statement identifier sid is used in the compiler correctness theorem.
- Finally, the empty statement SKIP does nothing.

Small-Step Semantics. A C0 program is formally defined by a symbol table gst of global variables, a type-name table tt , and a function table ft . A *symbol table* is a list of pairs of variable names and the corresponding data types. The *type-name table* lists pairs of type names and the corresponding data types. And finally, the *function table*

SKIP	the empty statement
COMP $s_1 s_2$	sequential composition
ASS $e_1 e sid$	assignment of expression e to left-expression e_1
PALLOC $e_1 tn sid$	allocation of an object of type name tn and assignment to e_1
SCALL $vn fn es sid$	call of function fn with arguments es and result variable vn
RETURN $e sid$	return from a function
IFTE $e s_1 s_2 sid$	if-then-else with condition e
LOOP $e s sid$	while loop with condition e and body s
ASM $ls sid$	inlined assembly with instruction list ls

Table 2.2: C0 statements s

lists pairs of function names and the corresponding functions definitions.

A *function definition* is represented by a record *fd* consisting of (a) a statement *fd.body* that represents the function body, (b) a symbol table *fd.params* of the function's parameters, (c) the function's return type *fd.rettype*, and (d) a symbol table *fd.stack_vars* of the stack variables.

In contrast to the static program definition, the program state evolves during the execution of a C0 program. A program state s_{C0} comprises:

- the statement $s_{C0}.prog$ of the program that remains to be executed, and
- the current state $s_{C0}.mem$ of the program variables and the heap objects.

The memory model of the C0 small-step semantics is quite complex. As C0 is perfectly type-safe, the memory is typed. Moreover, the model separates the global variables, the variables in the various stack frames, and the heap objects. Each of these memory parts has its own symbol table. The symbol table of the global variables can be extracted from a memory state with the function `gm_st`.

For the work at hand, the memory internals are not relevant. We just use an evaluation function `mem_read` for the look-up, and an update function `mem_write` for the manipulation of memory objects. The function `mem_read tt m e` evaluates the expression *e* in the memory state *m* based on the type-name table *tt*. Similarly, the function `mem_write tt m el v` updates the object denoted by the left-expression *e_l* in memory *m* by the new value *v*.

Note, however, that the expression evaluation and consequently the memory update may fail, e. g., because of an uninitialized variable or a dereferenced null pointer. Thus, the functions are partial. Furthermore, we want to update certain expressions at the same time, i. e., all left expressions are evaluated before the memory is updated. The corresponding update function is called `mem_writes tt m evs` and parametrized over a type-name table *tt*, a memory state *m*, and a list *evs* of pairs of an expression and the corresponding new value.

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

2.4 The VAMP Assembly Language

We regard the assembly language developed for the VAMP architecture. The *VAMP architecture* is based on the DLX architecture [HP96] and was initially presented by Mueller & Paul [MP00]. An implementation of the VAMP has been formally verified in 2003 [BJK⁺03, BJK⁺06]. Since then, the VAMP has been extended with address translation and support for I/O devices [DHP05, AHK⁺07, TS08].

Three models [AHL⁺09, TS08] are related to this processor. From the most concrete to the most abstract one, these are: the gate-level implementation, the instruction set architecture (ISA) specification, and the assembly-language specification. Simulation proofs relate the adjacent models such that properties shown at the assembly level can finally be transferred down to the gate-level.

The VAMP assembly language is the target language of Leinenbach & Petrova's verified C0 compiler. At the same time, its specification is intended to be a convenient layer

for the implementation and the verification of hardware-dependent programs. Thus, the language specification abstracts from certain aspects of the lower layers, which are irrelevant for these purposes.

Most notably, 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, interrupts are not modeled in the VAMP assembly machine. This abstraction extends the very idea of the previous one: We implicitly assume that interrupts can either be handled transparently to the running program (like device interrupts and page faults) or are programming errors (like misaligned memory accesses and undecodable instructions), which should not occur at all. The simulation theorem between the ISA and the assembly-language specification simply holds unless exceptions are generated during the execution of instructions. As part of compiler correctness, it has been shown that a compiled C0 program does not generate exceptions if there are sufficient resources. Note that this separation of matters is equally welcome when verifying inlined assembly code. Besides, the abstraction from interrupts is a reversible convenience – we re-introduce an exception semantics tailored for assembly processes in [Section 4.3](#).

Finally, the bit vectors from the ISA specification are superseded 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. This representation is optimized for assembly programs working with integers; arguments regarding naturals and bit-vector operations require conversions. The two functions `to_nat32` and `to_int32`, for example, convert between 32-bit integers and naturals.

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

- two program counters $s_{asm}.dpc$ and $s_{asm}.pcp$ for implementing the delayed-branch mechanism, which hold the addresses of the current and next instruction,
- the general-purpose and special-purpose register files $s_{asm}.gprs$ and $s_{asm}.sprs$, which are both lists of data, and
- the word-addressable main memory $s_{asm}.mm$, mapping addresses to data.

Note that some well-formedness constraints are not enforced by the record type. We call an assembly state *valid* iff the program counters are 32-bit naturals, the register files contain 32 registers, and all registers and memory cells are 32-bit integers. We subsume these well-formedness constraints in predicate `valid_asm`:

$$\begin{aligned} \text{valid_asm } s_{asm} &\equiv \\ &s_{asm}.dpc < 2^{32} \wedge s_{asm}.pcp < 2^{32} \wedge |s_{asm}.gprs| = 32 \wedge |s_{asm}.sprs| = 32 \wedge \\ &(\forall i \in \{0 \dots 32\}. -2^{31} \leq s_{asm}.gprs ! i < 2^{31}) \wedge \\ &(\forall i \in \text{used_sprs}. -2^{31} \leq s_{asm}.sprs ! i < 2^{31}) \wedge (\forall addr. -2^{31} \leq s_{asm}.mm \text{ addr} < 2^{31}) \end{aligned}$$

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

$\text{current_instr } s_{\text{asm}} \equiv \text{to_instr } (s_{\text{asm}}.\text{mm } (s_{\text{asm}}.\text{dpc div } 4))$ denotes the instruction that is executed next in the assembly machine.

The assembly semantics can equally be employed in user- and system mode. The mode is determined by the special-purpose register `MODE`:

$\text{is_system_mode } s_{\text{asm}} \equiv s_{\text{asm}}.\text{sprs ! MODE} = 0$

In the course of this thesis, we do not regard the system mode. In user mode, it is illegal to access most of the special-purpose registers and solely the page-table length register `PTL` is relevant for us: It determines the size of the main memory in pages of 1024 words. For technical reasons, the register value is a signed integer with an offset of -1 , i. e., -1 denotes a size of 0 pages. We encapsulate this fact in the function `mm_size` and define:

$\text{mm_size } s_{\text{asm}} \equiv \text{to_nat32 } (s_{\text{asm}}.\text{sprs ! PTL} + 1)$

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

The VAMP-assembly transition function δ_{asm} computes for a given assembly state s_{asm} the next state s'_{asm} . In illegal cases, the transition function gets stuck.¹ Otherwise, the transition is specified by a simple case distinction on `current_instr` s_{asm} . We use the notation $\delta_{\text{asm}}^n s_{\text{asm}}$ to denote the result of executing n steps of the assembly machine starting in state s_{asm} .

2.5 On a Correct Compiler

Most software in Verisoft has been implemented and verified at the C0 level. The C0 programs are translated to assembly code in order to be executed on the target machine. Leinenbach & Petrova have developed and verified a non-optimizing compiler from C0 to VAMP assembly [LPP05, Pet07, LP08]. Below, we summarize their results.

Compiler correctness is formulated as a simulation theorem. In essence, the compiler-simulation theorem states that every step i of the source program executed on the C0 small-step semantics simulates a certain number s_i of steps of the VAMP assembly machine executing the compiled code. For the property transfer from the C0 to the VAMP assembly layer, the simulation theorem has to meet special requirements. In particular, the simulation theorem is formulated based on the small-step semantics, which permits the reasoning about non-terminating programs and interleaved executions. Additionally, the compiler-correctness proof considers resource restrictions at the assembly layer and allows to discharge them at the C0 layer.

At first, we present the simulation relation:

Definition 2.1 (C0 Simulation Relation). The simulation relation `consistent` states that a VAMP assembly state s_{asm} encodes a C0 state s_{C0} . The relation is parametrized over

¹Technically, we avoid extra case distinctions for the detection of all possible invalid cases but sort out those cases before we apply the transition function. Though the transition function might sometimes not get stuck, it is simply meaningless in illegal cases.

an allocation function *alloc*, which maps all variables and heap objects in the C0 state to their allocated address in the main memory of the assembly state. For a C0 program with the type-name table *tt* and the function table *ft*, the relation *consistent tt ft s_{C0} alloc s_{asm}* correlates

- the currently remaining program *s_{C0}.prog* to the value of the program counters in the assembly state *s_{asm}* (called *control consistency*),
- the code of the C0 program, which is computed from the type-name table *tt*, the function table *ft*, and the global symbol table² (*gm_st s_{C0}.mem*), to the corresponding memory region in *s_{asm}* (*code consistency*), and
- the values of variables and heap objects in the C0 state *s_{C0}* to their corresponding memory values in *s_{asm}* (*data consistency*).

The formal specification of the simulation relation is a conjunction of three predicates for control, code, and data consistency. We present only the code-consistency predicate:

```
code_consistent tt ft sC0 sasm ≡
  let code = codegen_program tt ft (gm_st sC0.mem)
  in ∀i < |code|.
    decodable (sasm.mm (program_base + i)) ∧
    to_instr (sasm.mm (program_base + i)) = code ! i
```

with the compiled program *codegen_program tt ft (gm_st s_{C0}.mem)*, the predicate *decodable* determining whether an integer can be interpreted as an instruction, and the conversion function *to_instr* realizing this interpretation. The constant *program_base* is a compiler parameter permitting a variable displacement of the program code in the memory. This displacement is used in the kernel implementation to provide space for some low-level functionality of CVM. For user programs, the displacement is not needed and hence set to 0.

Certainly, we need an initial assembly state for a C0 program. For the support of various use cases, the compiler-correctness theorem does not construct a particular initial state but rather formulates the necessary requirements on an assembly state to serve as an initial state for a specific C0 program.

Definition 2.2 (Initial Assembly States for a C0 Program). For a given C0 program (*tt*, *ft*, *gst*), each corresponding initial assembly state *s_{asm}* is well-formed, its program counters point to the beginning of the code, the assembly state is code consistent with the corresponding initial C0 state (denoted by *init_C0 ft gst*), and the memory containing the global variables is zero-initialized.

The formal requirements are collected in the predicate *is_initial_asm tt ft gst s_{asm}*:

²Note that—though constant during the program execution—the symbol table of the global memory is extracted from the current C0 state’s memory component *s_{C0}.mem*.

$\text{is_initial_asm } tt \ ft \ gst \ s_{\text{asm}} \equiv$
 $\text{valid_asm } s_{\text{asm}} \wedge s_{\text{asm}}.\text{dpc} = 4 \cdot \text{program_base} \wedge s_{\text{asm}}.\text{pcp} = s_{\text{asm}}.\text{dpc} + 4 \wedge$
 $\text{code_consistent } tt \ ft \ (\text{init_C0 } ft \ gst) \ s_{\text{asm}} \wedge$
 $(\forall i \in \text{gm_range } tt \ ft \ gst. s_{\text{asm}}.\text{mm } i = 0)$

Recall that certain executions in the assembly semantics are not legal, e.g., if an instruction accesses memory beyond the available size. Compiler correctness is even more demanding with respect to the accessible memory: It distinguishes read-only code regions in the memory from writable data regions in order to prevent self-modifying code. The non-optimizing compiler furthermore maintains that the assembly machine is not in a delay slot between two compiled C0 statements³, i.e., we require $s_{\text{asm}}.\text{pcp} = s_{\text{asm}}.\text{dpc} + 4$. Finally, we assume a well-formed assembly state, i.e., $\text{valid_asm } s_{\text{asm}}$ (see page 19).

A few auxiliary predicates help to formally specify whether a compiled C0 program is successfully executed on the VAMP assembly level: The predicate $\text{mem_write_inside_range}$ holds iff the current instruction writes into a specified memory range, the predicate is_exception holds iff an exception is generated during the execution of the current instruction, and finally, $\text{inside_range } (l, h) \ i$ is defined as $l \leq i \wedge i < h$.

Definition 2.3 (Successful Execution of Assembly Code). We call a computation of the VAMP assembly machine from a state s_{asm} in n steps to s'_{asm} a *successful execution* with respect to a code range crange and an address range arange iff

- the memory contents in crange has not been overwritten,
- all instructions are only fetched from crange ,
- all accessed memory addresses are in arange (encapsulated in $\text{mem_access_inside_range}$),
- no exceptions are generated, and
- all executed instructions are legal (denoted by is_legal_instr).

Formally, the successful execution of assembly code is defined as

$(\text{crange}, \text{arange}) \vdash_{\text{asm}} s_{\text{asm}} \rightarrow^n s'_{\text{asm}} \equiv$
 $\delta_{\text{asm}}^n s_{\text{asm}} = s'_{\text{asm}} \wedge \text{valid_asm } s'_{\text{asm}} \wedge s'_{\text{asm}}.\text{pcp} = s'_{\text{asm}}.\text{dpc} + 4 \wedge$
 $(\forall m < n. \neg \text{mem_write_inside_range } (\delta_{\text{asm}}^m s_{\text{asm}}) \ \text{crange} \wedge$
 $\text{inside_range } \text{crange } (\delta_{\text{asm}}^m s_{\text{asm}}).\text{dpc} \wedge$
 $\text{mem_access_inside_range } (\delta_{\text{asm}}^m s_{\text{asm}}) \ \text{arange} \wedge$
 $\neg \text{is_exception } (\delta_{\text{asm}}^m s_{\text{asm}}) \wedge$
 $\text{is_legal_instr } (\delta_{\text{asm}}^m s_{\text{asm}}) (\text{current_instr } (\delta_{\text{asm}}^m s_{\text{asm}})))$

Note that the successful execution of assembly code depends on two implicit assumptions: The initial assembly configuration s_{asm} needs to be well-formed, and the program counters must not start in a delay slot. These two conditions are invariant under a successful execution. For a particular C0 program (tt, ft, gst) , the corresponding code

³Recall that the VAMP features a delayed-branch mechanism, i.e., a branch instruction is executed after the next instruction has been decoded (see [MP00]). When a C0 statement has been completely executed, the assembly machine should certainly not be about to execute a previously seen branch.

range is computed with the function `code_range tt gst ft`. The address range, in contrast, is not statically fixed. In practice, it is determined by a maximal memory address `max_address`, which results in the user mode from the value of the page-table length register (see [Section 2.4](#)). The function `address_range` converts this address into the corresponding range, which starts with the program-base address. The formal definition of all auxiliary predicates can be found in earlier publications [[AHL⁺09](#), [Lei08](#)].

Compiler correctness relies on several preconditions.

First, the considered C0 program has to be compilable, which comprises well-formedness constraints on the type-name table `tt`, the function table `ft`, and the global symbol table `gst` as well as a definition for the function `main` and static resource restrictions. The latter require, for instance, that the generated code fits into the memory of the target machine and that the jump distances for conditionals, loops, and function calls are not too large (such that they fit into the immediate constants of the VAMP assembly instructions). We collect all static requirements on a C0 program (`tt`, `ft`, `gst`) including its well-formedness and the definedness of `main` in the predicate `is_compilable tt ft gst`.

Second, the required memory of C0 states `sC0` change dynamically during the execution of a C0 program, e. g., with the recursion depth or by the allocation of heap variables. Predicate `sufficient_memory max_address tt ft sC0` checks for these dynamic resource restrictions, assuming that `max_address` denotes the maximal memory address and does not exceed the value 2^{32} .

Third, the compiler-correctness theorem does not hold if inlined assembly code is executed. The predicate `is_Asm` determines, whether a given C0 statement is an assembly statement. We denote the first statement of the remaining program in a C0 state `sC0` by `fst_stmt sC0.prog`.

Fourth, compiler correctness requires the absence of runtime errors like the access of an uninitialized variable. In case of a runtime error, the transition function δ_{C0} of the C0 small-step semantics remains undefined, i. e., it evaluates to \perp .

Finally, we can state the compiler-correctness theorem:

Theorem 2.1 (Compiler Correctness). *We assume that (tt, ft, gst) describes a compilable C0 program, there are no runtime errors during the execution of n steps, there is sufficient memory in each execution step, the execution does not involve inlined assembly statements, and s_{asm} denotes an initial assembly state for (tt, ft, gst) .*

In this case, there exists a step number t , an allocation function $alloc'$, and a final assembly state s'_{asm} such that

- *the assembly machine successfully advances in t steps from s_{asm} to s'_{asm} ,*
- *the final C0 state s'_{C0} simulates s'_{asm} under the allocation function $alloc'$, and*
- *no special-purpose registers have been changed.*

Formally:

$$\begin{aligned} & \llbracket is_compilable\ tt\ ft\ gst; \\ & \delta_{C0}^n\ tt\ ft\ (init_C0\ ft\ gst) = [s'_{C0}]; \\ & \forall i \leq n. \text{ sufficient_memory } max_address\ tt\ ft\ [\delta_{C0}^i\ tt\ ft\ (init_C0\ ft\ gst)]; \\ & max_address \leq 2^{32}; \end{aligned}$$

$$\begin{aligned}
& \forall i < n. \neg \text{is_Asm} (\text{fst_stmt} [\delta_{\text{C0}}^i \text{ tt ft } (\text{init_C0} \text{ ft } \text{gst})].\text{prog}); \\
& \text{is_initial_asm} \text{ tt ft } \text{gst} \text{ } s_{\text{asm}} \\
\implies & \exists t \text{ alloc}' s'_{\text{asm}}. \\
& (\text{code_range} \text{ tt } \text{gst} \text{ ft}, \text{address_range} \text{ max_address}) \vdash_{\text{asm}} s_{\text{asm}} \rightarrow^t s'_{\text{asm}} \wedge \\
& \text{consistent} \text{ tt ft } s'_{\text{C0}} \text{ alloc}' s'_{\text{asm}} \wedge s'_{\text{asm}}.\text{sprs} = s_{\text{asm}}.\text{sprs}
\end{aligned}$$

Proof. Leinenbach [Lei08] has shown this theorem by induction on the step number n . The induction start establishes the simulation relation between a successor of the initial assembly state s_{asm} and the initial C0 state $\text{init_C0} \text{ ft } \text{gst}$. Note that simulation does not hold between the initial states. On the assembly level, some initialization code must be successfully executed to set up the machine accordingly. Furthermore, the values of the special-purpose registers have to be preserved by the initialization code.

The induction step claims that under a C0 transition, the code for the current C0 statement is successfully executed, the special-purpose registers remain unchanged, and the simulation relation is preserved under a C0 transition. The proof for this claim involves a second induction over C0 statements.

The proof has been formalized in Isabelle/HOL and is available from the public Verisoft Repository [Ver09]. \square

Note that this correctness theorem explicitly requires to start with the initial state. Thus, we cannot use the theorem for an execution of C0 statements after an inlined assembly statement has been executed. For this purpose, we employ the stronger lemma of the induction step for the proof of the above theorem.

Lemma 2.2 (Compiler-Correctness Induction Step). *We assume that s_{C0} is a well-formed C0 state and s_{asm} is a well-formed assembly state, where the program counters do not start in a delay slot. Moreover, the simulation relation holds for s_{C0} and s_{asm} under an allocation function alloc . Additionally, a C0 transition is legal (i. e., there is no runtime error and the remaining program does not start with inlined assembly) and there is sufficient memory before and after the transition.*

In this case, there exists a step number t , an allocation function alloc' , and an assembly state s'_{asm} such that

- *the assembly machine successfully advances in t steps from s_{asm} to s'_{asm} ,*
- *the final C0 state s'_{C0} simulates s'_{asm} under the allocation function alloc' , and*
- *no special-purpose registers have been changed.*

Formally:

$$\begin{aligned}
& \llbracket \text{valid_C0} \text{ tt ft } s_{\text{C0}}; \text{valid_asm} \text{ } s_{\text{asm}}; s_{\text{asm}}.\text{pcp} = s_{\text{asm}}.\text{dpc} + 4; \\
& \text{consistent} \text{ tt ft } s_{\text{C0}} \text{ alloc} \text{ } s_{\text{asm}}; \delta_{\text{C0}} \text{ tt ft } s_{\text{C0}} = \lfloor s'_{\text{C0}} \rfloor; \\
& \neg \text{is_Asm} (\text{fst_stmt} s_{\text{C0}}.\text{prog}); \text{sufficient_memory} \text{ max_address} \text{ tt ft } s_{\text{C0}}; \\
& \text{sufficient_memory} \text{ max_address} \text{ tt ft } s'_{\text{C0}}; \text{max_address} \leq 2^{32} \rrbracket \\
\implies & \exists t \text{ alloc}' s'_{\text{asm}}. \\
& (\text{code_range} \text{ tt } (\text{gm_st} s_{\text{C0}}.\text{mem}) \text{ ft}, \\
& \text{address_range} \text{ max_address}) \vdash_{\text{asm}} s_{\text{asm}} \rightarrow^t s'_{\text{asm}} \wedge \\
& \text{consistent} \text{ tt ft } s'_{\text{C0}} \text{ alloc}' s'_{\text{asm}} \wedge s'_{\text{asm}}.\text{sprs} = s_{\text{asm}}.\text{sprs}
\end{aligned}$$

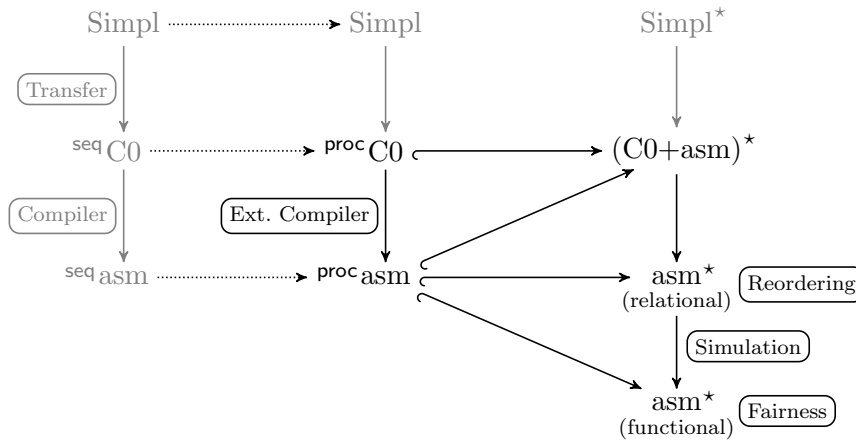


Figure 2.3: A roadmap to concurrency

2.6 Isabelle/Simpl – a Verification Environment for C0

The verification environment Isabelle/Simpl [Sch05, Sch06] is implemented as a conservative extension of the higher-order logic (HOL) instance of the theorem-proving environment Isabelle [NPW02]. Though the verification environment was motivated by C0, it is by no means restricted to C0. In fact, it is a self-contained theory development for a quite generic model of a sequential imperative programming language called *Simpl*. Part of this extensive framework are big- and small-step semantics as well as Hoare logics for both, partial and total correctness. In order to facilitate the usage of the Hoare logics within Isabelle/HOL, the application of the rules is automated as a verification-condition generator. Furthermore, proofs have been developed that the logics are sound and complete with respect to the operational semantics. Soundness is crucial for pervasive verification in order to formally link the results from the Hoare logics to the operational Simpl semantics. Correctness theorems about the embedding of C0 into Simpl then allow us to map these results to the small-step semantics of C0 [Sch06, AHL+08]. Completeness can be viewed as a sophisticated sanity check for the Hoare logics, ensuring that verification cannot get stuck because of missing Hoare rules.

2.7 Extending the Language Stack: a Roadmap to Concurrency

Figure 2.3 illustrates how we extend the existing language stack to support verification about concurrently executing C0 processes (see also [DDWS08]). On the left-hand side, we see the stack of the sequential languages (here marked by ^{seq}) as we know it from Figure 2.2 on page 15. These languages are extended with suitable kernel communication primitives (see Chapter 4) to processes (labeled ^{proc}) as shown in the central column of the figure. Note that Simpl features a very general language model such that we can directly embed the process semantics without an extension. Certainly, the language

extensions require an extended compiler-correctness theorem as developed in [Section 4.5](#). The extension of the transfer theorems remains future work.

The most demanding step, however, is the embedding of these process semantics into the interleaved execution of the VAMOS kernel. The third column in the figure depicts a language-centered view of the involved kernel models⁴ (all introduced in [Chapter 5](#)): At the lower end, there is a functional kernel model featuring assembly processes “asm*”: This model is the VAMOS specification \mathcal{A}_V , which describes the exact behavior of the kernel and is also used for the code verification of VAMOS [[DDW09](#), Sect. 6]. At that level, we show fair scheduling in [Chapter 7](#).

Above, there is a relational kernel model that features only assembly processes. This model abstracts from the scheduler and is employed to show that scheduling decisions may be reordered. A simulation theorem establishes a link between both kernel models. Combining reordering and extended compiler correctness, we can then abstract some⁵ assembly processes to C0 processes and obtain a kernel model with both semantics: “(C0+asm)*”. Formally, we only specify the latter and regard the relational “asm*” as a special case. This relational model \mathcal{A}_{VC} is called *Communicating User Processes* (CoUP). The described abstraction steps are covered by [Chapter 6](#).

The top-most kernel model “Simpl*” suggests a language extension of Simpl towards several threads of execution. This extension as well as a corresponding verification environment with a concurrent Hoare logic and its integration into this framework remain future work. For the particular purpose of the correctness proof for the simple operating system employed in the academic system, however, it is sufficient to regard only one process and its interaction with the kernel. [Chapter 8](#) explains this approach in detail.

⁴We use the Kleene star to express concurrently executing processes in the figure.

⁵Note that not all possible assembly processes can be abstracted to C0 processes.

3 The VAMOS Microkernel: Design and Implementation

Ist es Schatten? ist's Wirklichkeit?
Wie wird mein Pudel lang und breit!

Johann Wolfgang von Goethe, in: "Faust I"

Contents

3.1	Requirements on a Correct Microkernel's Design	28
3.2	Functionality	28
3.2.1	Inter-Process Communication	30
3.2.2	Process Scheduling	31
3.3	Overflow-Safe Implementation of the Timeout Management	31
3.4	Conclusion	34

The *kernel* of an operating system is the code that runs in the privileged system 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 peripheral 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 became 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 like Mach [RATY+88] or IBM's Workplace OS [FC98], however, suffered from a poor performance. When Jochen Liedtke analyzed these systems [Lie96], he pinned the problem down to a feature-overloaded mechanism for inter-process communication (IPC). Together with a light-weight, flexible IPC mechanism [Lie93], he proposed the minimality of hardware abstractions [Lie95] in the kernel and suggested that servers implement the traditional services of operating systems. Engler *et al.* [EKO95] 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.

The design of the VAMOS microkernel is substantially inspired by Liedtke but not as minimal as he or even Engler proposed. The microkernel hosts all functionality 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 VAMOS. Device drivers, which constitute the largest part of today’s monolithic kernels, are not part of the kernel. In the remaining chapter, we first regard the design aspect of provable correctness. Then, we provide an overview of the functionality of VAMOS and describe two important features of the kernel in more detail: the IPC mechanism and the scheduling policy. Finally, we consider an implementation issue for provable correctness: the overflow-safe management of timeouts in VAMOS.

Credits and Own Contribution. 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 design as well as its implementation. My key contributions to the kernel design are the capability-like concept of process identifiers and the abandonment of an idle process. My most important improvements of the implementation comprise the overflow-safe management of timeouts and a generic debug library usable in the kernel as well as in processes.

3.1 Requirements on a Correct Microkernel’s Design

Modern microkernels are strictly evaluated by their performance. While efficiency is clearly essential for the applicability of a microkernel, many implementations even accept a limited runtime in favor of utmost performance and declare, e.g., a finite range of numbers as “sufficiently large” such that overflows *should* not occur during the expected lifetime of the kernel. An old but even interface-visible example is Liedtke’s [Lie93] *generation counter*: In order to ensure a thread’s identity, the fixed internal format of thread identifiers reserves some bits for a counter, which is incremented on reincarnation. A possible overflow, however, is just neglected. Although such an approach might be acceptable for non-critical systems, it does certainly not suit a critical, verified system; moreover, a restriction on the lifetime contradicts the formal notion of liveness.

As a consequence, VAMOS has no such “expiration date”. For this to work, a capability-like management of process identifiers allows for an overflow-free and truly authentic process identification beyond reincarnations (see [Section 3.2.1](#)). Furthermore, we maintain a solely relative notion of time within the kernel (see [Section 3.3](#)).

Another design decision is related to provable correctness as well: the absence of a so-called *idle process*. In some microkernels, this distinguished process is scheduled with the lowest priority and should implement a busy wait. We circumvent assumptions on user-level processes if possible and have thus implemented the idle loop (which solely waits for incoming interrupts) directly in the kernel.

3.2 Functionality

The VAMOS kernel 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 3.1 lists the kernel calls that constitute the ABI.

The 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. Note that though this notion of privileging draws upon an analogy to the system mode of the processor, both concepts should not be confused. All processes run in user mode; process privileging is just a concept of the access-control mechanism of the VAMOS kernel and fully realized in software. We presume that the privileged processes constitute the user-mode parts of the operating system and implement a more sophisticated access-control mechanism. Non-privileged processes may then communicate with the privileged processes in order to inquire kernel services on their behalf.

When VAMOS boots, it launches one single process, the *init process*. This process is privileged and has to set up the required servers of the operating system, start and register the device drivers, and possibly spawn initial applications.

Kernel Call	Description
<i>Access Control</i>	
<code>set_privileged^p</code>	add a process to the set of privileged processes
<i>Process Management</i>	
<code>process_create^p</code>	create a new process from a memory image
<code>process_clone^p</code>	copy an already existing process
<code>process_kill^p</code>	kill a process
<i>Memory Management</i>	
<code>memory_add^p</code>	increase the amount of virtual memory for a process
<code>memory_free^p</code>	decrease the amount of virtual memory for a process
<i>Scheduling Mechanism</i>	
<code>chg_sched_params^p</code>	change scheduling parameters
<i>Device Driver Support</i>	
<code>change_driver^p</code>	(un)register a process as a driver for a set of devices
<code>enable_interrupts^d</code>	re-enable a set of interrupts after their successful handling
<code>dev_read^d / dev_write^d</code>	communicate with a certain device
<i>Inter-Process Communication</i>	
<code>ipc_send / ipc_receive</code>	unidirectionally communicate with another process
<code>ipc_request</code>	send a message and immediately wait for a reply
<code>change_rights</code>	manipulate IPC rights
<code>read_kernel_info</code>	receive information from the kernel

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

Table 3.1: Application binary interface of the VAMOS kernel

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 inquiries from or to that device, respectively. Moreover, the device driver is notified of interrupts from that device.

Inside the kernel, *time measurements* are based on a timer device, which periodically raises its interrupt line. In particular, we measure the runtime of processes and the waiting time for IPC in numbers of interrupt occurrences.

3.2.1 Inter-Process Communication

VAMOS supports exclusively synchronous IPC. This nowadays common restriction in microkernel designs has been introduced by Liedtke [Lie93] because it circumvents message queuing in the kernel and asynchronous IPC can as well be implemented at user level. In Table 3.1 on the previous page, we see the three different kinds of IPC calls for sending, receiving, and the combined send and receive from the same partner (request). The latter call guarantees that the invoking process cannot be scheduled between sending and receiving but immediately waits for its communication partner—typically a server. Together with the access-control mechanism for IPC, servers can ensure that their clients definitely wait for a reply.

Synchronization generally requires waiting. In VAMOS, processes may limit the maximal waiting time for an IPC operation by a finite number. If a process is not willing to wait at all, it specifies 0, which we also call *immediate timeout*. A positive timeout value indicates the number of timer interrupts until the IPC operation should be canceled. An unlimited waiting time is specified by a so-called *infinite timeout*.

We have implemented a capability-like [Lev84] access-control mechanism for IPC. While the kernel identifies processes by unique identifiers (PIDs), processes refer to each other via process-local alias names, so-called *handles*. The mapping from handles to PIDs is maintained by the kernel, which invalidates all handles to a process when it vanishes and sends an obituary to the handles' owners. That means, we have improved the authentic process identification in comparison to Liedtke's generation counter, which might overflow. Moreover, we circumvent guessing of PIDs through this additional layer of indirection. The only drawback of the indirection is the extra maintenance effort in VAMOS. A sparse handle database, however, permits a relatively small runtime overhead for the maintenance.

Additionally, IPC rights are associated with each handle. These rights are organized as a four-bit vector. The *request* bit permits solely combined send and receive calls while the *send* right grants permission to send in either way. The combined call furthermore requires an infinite receive timeout, unless the *finite* modifier bit is set. By this means, a server can constrain a client to wait infinitely long for a reply to its request. Moreover, the rights expire after the first successful send or request call unless the modifier bit *multiple* is set. Receiving from a known handle is always permitted.

Finally, the IPC mechanism is employed to convey *kernel notifications*. There are two conditions that VAMOS notifies processes about: An obituary bit informs whether formerly known processes have vanished, and occurred device interrupts are delivered

via interrupt bits to the corresponding drivers. Several notifications might be sent at the same time – possibly even together with a conventional IPC message – because all notification bits fit into a single register.

3.2.2 Process Scheduling

The basic policy underlying the scheduler in VAMOS is round-robin process selection. This basic policy, however, 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. This policy is also known as *static-priority scheduling* though technically, priorities might be adjusted through kernel calls at any time during runtime.

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.

3.3 Overflow-Safe Implementation of the Timeout Management

Recall that VAMOS supports finite IPC timeouts. From the design perspective, these timeouts must certainly be relative because specifying an absolute point in time for a timeout by a fixed-precision value would certainly limit the maximal runtime of the system. In the implementation, however, this relative notion of time requires some maintenance. Previously, VAMOS just neglected the maintenance (cumulating all timer interrupts since system start) and thus, became incorrect after some time because of arithmetic overflows. The naïve solution, to decrease the timeout values of all waiting processes whenever a timer interrupt occurs, is definitely most inefficient. Hence, we have evaluated two different approaches: an *occasional timeout adjustment* and an *overflow-aware timeout comparison*. Below, we explain the implementation problem in more detail and discuss our solutions. Jan Dörrenbächer [DDW09, Sect. 6] has formally verified that the occasional timeout adjustment does indeed not overflow. Note that our formal proof work (see Chapters 5 to 8) relies on implementation correctness.

Occasional Timeout Adjustment. This approach cumulates occurring timer interrupts and just occasionally, decreases the timeouts for waiting processes by the number of the accumulated timer interrupts. Here, we exploit the fact that processes specify their timeouts by signed 32-bit integers, where any negative value encodes the infinite timeout. Within the kernel, however, we use unsigned 32-bit integers to represent points in time and encode “never” by the maximum value ($2^{32} - 1$). Hence, we may defer the decrease of the timeouts for waiting processes for up to $2^{31} - 1$ timer interrupts.

More specifically, we maintain two global time variables: `current_time` accumulates occurred timer interrupts and `next_timeout` retains the expected next timeout. Additionally, there is a so-called *scheduling timeout* `sched_timeout[p]` for each waiting process p . Initially, the current time is set to zero, and the next timeout is set to “never”.

Whenever a process p issues an IPC call with a positive timeout argument t and is enqueued in the wait queue, its scheduling timeout is computed by adding the current time to the call’s timeout argument, i. e.,

```
    sched_timeout[p] = current_time + UNSIGNED(t);
```

Note that the value of t is less than 2^{31} because it is a signed 32-bit integer. Hence, the unsigned 32-bit integer `current_time + t` does not overflow unless `current_time` exceeds 2^{31} . Furthermore, the variable `next_timeout` is set to `sched_timeout[p]` if the latter is smaller than the former:

```
    if (sched_timeout[p] < next_timeout) {
        next_timeout = sched_timeout[p];
    }
```

If a process issues an IPC call with a negative timeout argument (meaning “infinite timeout”) and is enqueued in the wait queue, its scheduling timeout is simply set to $2^{32} - 1$ (“never”).

Upon each timer interrupt, the kernel checks whether there is a next timeout, i. e., the value of `next_timeout` is not “never”. In this case, the kernel increases the current time and checks whether it has reached the next timeout, i. e., `current_time == next_timeout`. If so, `next_timeout` is set to “never” and the wait queue is traversed: For each process p in the wait queue, the kernel checks whether the scheduling timeout has expired, i. e., `sched_timeout[p] == current_time`. Processes with expired scheduling timeouts are woken up; for the remaining processes p , the current time is subtracted from the scheduling timeout, i. e.,

```
    sched_timeout[p] = sched_timeout[p] - current_time;
```

and the variable `next_timeout` is set to the new value of `sched_timeout[p]` if the latter is smaller than the former. Finally, the current time is set to zero.

This implementation maintains several invariants. First, `current_time` is always the minimum value of all times (`next_timeout` and `sched_timeout[p]` for all waiting processes p). Second, `sched_timeout[p] - current_time` is always less than 2^{31} for all waiting processes p . Third, `current_time` is zero unless there is a next timeout. Hence, `next_timeout` is either “never” or less than 2^{31} : when the first process is scheduled to wait, its scheduling timeout is less than 2^{31} because `current_time` is zero; and after a timeout expired, the next timeout is set to the smallest `sched_timeout[p] - current_time`. Thus, `current_time` is less than 2^{31} .

Overflow-Aware Timeout Comparison. This approach maintains the same variables but never adjusts the scheduling timeouts. Instead, the check for elapsed timeouts relies on the modulo-arithmetical semantics of overflows in C0. While the occasional timeout adjustment maintains the invariant that `current_time` is always a minimum of all time variables, the overflow-aware comparison uses this value as a divide: values smaller than the current time are generally interpreted as larger than any value greater than or equal to the current time.

This interpretation follows the observation that the current time does not overtake timeouts but increases step by step until the timeouts elapse. If a scheduling timeout is nevertheless smaller than the current time, there must have been an overflow when the timeout was computed. Given that any timeout argument t is less than 2^{31} , the computed scheduling timeout is always less than the `current_time` in case of an overflow.

Conceptually, we define an overflow-aware order $x <^* y$ of timeouts when we technically replace comparisons of the form $x < y$ between two timeouts x and y by the following expression:

```
(x >= current_time && (y < current_time || x < y)
|| y < current_time && x < y)
```

Nevertheless, there remains a small problem: When a (finite) scheduling timeout is computed, it might now incidentally coincide with the value $2^{32} - 1$, decoding “never”. Hence, we refine our approach and use only even numbers for finite times, i. e., upon timer interrupts, the variable `current_time` is increased by two, and when a process p calls for IPC with the positive timeout argument t and is enqueued in the wait queue, the finite scheduling timeout is computed by

```
sched_timeout[p] = current_time + UNSIGNED(t)<<1;
```

Recall that t is less than 2^{31} such that there cannot be more than one wrap around.

Consequently, the variable `current_time` has always an even value; the other time variables (`next_timeout` and `sched_timeout[p]` for all waiting processes p) are either even or have the “never” value.

Evaluation. When comparing the compiled code, the overflow-aware solution is 128 instructions longer than the timeout adjustment. This static overhead is mainly caused by the more involved expression for the comparison of timeout values. As these expressions comprise lazy operators, the respective runtime overhead depends on the test case. Our regression-test suite contains two tests where we counted less instructions for the overflow-aware solution but when counting all tests, the timeout adjustment requires less instructions. Though we do not have a cycle-precise testing environment, we suspect that the overflow-aware solution would perform even worse because the branches for the lazy operators might stall the processor pipeline.

Note that the traversal of the wait queue is necessary in both solutions because the wait queue is not ordered by timeouts. Thus, a timeout-ordered wait tree could theoretically improve the approach of an overflow-aware comparison. Considering the relatively small

number of maximally running processes, however, we did not further investigate this approach.

3.4 Conclusion

We implemented a microkernel suitable for formal verification. An important prerequisite for the verification of liveness properties is a potentially unbounded runtime of the kernel. Most typically, a bounded runtime arises from the use of fixed-width data types for conceptually unbounded values. We discussed the problem based on two examples: process identifiers and time. For both examples, we realized sustainable solutions, namely the concept of handles, which guarantees the unique process identification even beyond termination, and a solely relative management of timeouts.

Though the solutions to both problems cause a performance overhead, we maintain that the costs are within reasonable bounds. The additional indirection for handles comes with the benefit of non-guessable PIDs; similar solutions are custom for reasons of security in high-performance kernels [SDN⁺04, Boy09]. Moreover, we kept the runtime overhead small by using a sparse handle database. Thus, the additional costs result in just four operations per look-up: a pointer dereference, an array access, a bitwise AND, and a comparison. Note that a kernel call may refer to up to two processes.

A similar low cost can be observed for the adjustment of timeout values: Whenever a timeout occurs, VAMOS decreases the scheduling timeout of each process remaining in the wait queue by the current time, and at last, it resets the global time. Thus, the total cost grows linearly with the length of the wait queue with an upper bound of 126 decrements and one plain assignment. Dequeuing a single process from the wait queue, on the contrary, involves calling a CVM primitive, two or three list manipulations and three updates of bookkeeping information.

Concluding, the design and implementation of a live microkernel is possible without giving up on performance.

4 Process Abstraction

I still believe in abstraction, but now I know that one ends with abstraction, not starts with it.

Alexander Stepanov, cited from B. Stroustrup: "Evolving C++"

Contents

4.1 Motivation	36
4.2 Process Models	37
4.3 The Assembly-Process Model	43
4.4 The C0-Process Model	47
4.5 Extended Compiler Correctness	55
4.6 Conclusion	62

The last chapter has presented the functionality of VAMOS on a comparatively abstract level. Most notably, the kernel calls have been described similar to function calls of a high-level programming language. Recall, however, the low-level perspective of the kernel programmer from [Section 2.1](#): Some user-mode computation might be interrupted at an arbitrary time, the kernel saves the user context and handles the interrupt. If the interrupt has been caused by a `trap` exception, the kernel might access registers and memory from the user's context and manipulate it to deliver results.

When formalizing the effects of a kernel call, we can either draw near the implementation level or introduce an abstraction level for processes, which separates the actual functionality from the implementation details. The first section contrasts both approaches and motivates our decision for the latter. The following sections describe the process model in general as well as its instantiation for VAMP assembly and C0. Finally, we extend the sequential compiler-correctness theorem to the process semantics.

Credits and Own Contribution. The first section illustrates Jan Dörrenbächer's initial approach to specify the VAMOS kernel together with the problems that I encountered when trying to introduce a kernel model with C0 processes. As a solution to these problems, I developed the generic definition of a process model as input-output automaton. While Jan Dörrenbächer instantiated the generic model for assembly processes, I verified its validity (see [Theorem 4.1](#) on page 47). Furthermore, the C0-process model and the extended compiler-correctness theorem stem from me. Sebastian Bogan [[Bog08](#)] specified the application interface of the SOS model based on the generic process interface. Hence, the SOS specification also benefits from the generic process model. Additionally, the general idea of this process modeling has been reused in a similar proof development

for the real-time kernel OLOS [DSS09, DSS10]. From this parallel proof development, I benefitted when I proved the correctness of the kernel library (part of the extended compiler-correctness theorem). Here, I could draw on previous experience of Mareike Schmidt.

4.1 Motivation

We consider the implementation of the `set_privileges` call. The currently running process p_{cup} might execute the instruction `trap 6` (meaning `set_privileges`). The kernel implementation then perceives the trap exception and examines register 11 of process p_{cup} in order to identify the process p_{target} that should gain privileges. If the register value is not a valid handle or the inquiry is not permitted, p_{cup} is informed about the specific problem by an update of its register 22 to the corresponding negative value. Otherwise, VAMOS adds process p_{target} to the set of privileged processes and sets register 22 of process p_{cup} to 0 (meaning success).

At first, we formalize the effects of the call directly at the implementation level. For a better readability, we encapsulate the update of a register value and the insertion of a process into the set of privileged processes into functions:

- Function `set_gpr` takes an assembly state, a register number, as well as the new value of this register, and then returns an assembly state with the updated register.
- Function `add_privilege` takes a component of the kernel state and a process number and then returns the updated component.

With these prerequisites in place, we can specify the call `set_privileges` as follows:

```

set_privileges_specV sV hn ≡
  let pcup = [cupV sV.schedds]; ptarget = [sV.rightsdb pcup].hdb hn
  in if ¬ privileged sV.rightsdb pcup
    then sV(|procs := sV.procs(pcup ↦ set_gpr (sV.procs pcup) 22 -4)|)
    else if ¬ valid_handle sV.rightsdb pcup hn
      then sV(|procs := sV.procs(pcup ↦ set_gpr (sV.procs pcup) 22 -1)|)
      else sV(|procs := sV.procs(pcup ↦ set_gpr (sV.procs pcup) 22 0),
              rightsdb := add_privilege sV.rightsdb ptarget)

```

This specification function takes the current VAMOS state s_V and hn , the (previously read) value of register 11. It extracts the process identifiers p_{cup} and p_{target} from the current state (whose components are detailed in [Section 5.1](#) but should not concern us at the moment) and distinguishes three cases:

- If the inquiry is not permitted, register 22 of p_{cup} is set to -4.
- If the specified handle is not valid, register 22 of p_{cup} is set to -1.
- Otherwise, register 22 of p_{cup} is set to 0, and the process p_{target} is added to the set of privileged processes.

This specification is certainly correct,¹ quite close to the implementation, and thus, code correctness is easy to show. It veils, however, the ABI and obstructs the concept of process privileging with details of register bindings and error codes. The call specification is intrinsically tied to assembly processes. The problem with this approach becomes apparent as soon as we try to abstract C0 processes: It is simply infeasible to take out a single assembly process and abstract it to a C0 process. We would always have to regard the complete VAMOS specification as a whole.

The remedy for this problem is the explicit specification of the interface between the kernel and the processes: We encapsulate the processes as self-contained automata and define the function `set_privileges_specV` using a generic process automaton. The exact interface of the process automaton is specified in the following section. For this example, we just assume that the generic process automaton fulfills the following requirements:

- It has a state space $\mathcal{S}_{\text{proc}}$,
- its output alphabet Ω_{proc} contains the values `SET_PRIVILEGED` hn (for all process handles hn),
- its input alphabet Σ_{proc} contains `ERR_UNPRIVILEGED`, `ERR_INVALID_HANDLE`, and `SUCCESS`,
- the transition function δ_{proc} is defined over $\mathcal{S}_{\text{proc}}$ under the inputs from Σ_{proc} , and
- the output function ω_{proc} computes an output for any process state.

Now, we specify the call `set_privileges` as:

```
set_privileges_specV sV hn ≡
let p_cup = [cupV sV.schedds]; p_target = [sV.rightsdb p_cup].hdb hn
in if ¬ privileged sV.rightsdb p_cup
  then sV(|procs := sV.procs(p_cup ↦ δ_proc ERR_UNPRIVILEGED [sV.procs p_cup])|)
  else if ¬ valid_handle sV.rightsdb p_cup hn
    then sV(|procs := sV.procs(p_cup ↦ δ_proc ERR_INVALID_HANDLE [sV.procs p_cup])|)
    else sV(|procs := sV.procs(p_cup ↦ δ_proc SUCCESS [sV.procs p_cup]),
             rightsdb := add_privilege sV.rightsdb p_target|)
```

This definition cleanly separates the concepts of the interface between processes and kernel (i. e., the ABI) and the actual specification of the process-privileging call (i. e., the kernel functionality). The separation permits to regard a process as a self-contained entity. Thus, we are able to abstract a C0 process from an assembly process and prove a simulation theorem that relates both process models independently from the kernel.

4.2 Process Models

The last section has emphasized the advantages of a self-containing input-output automaton for the process semantics, which allows us to hide the internal state and to just

¹We have formally shown that the given specification is equivalent to the one actually used for implementation correctness. Technically, the explanation is slightly imprecise because we have hidden an additional side-effect in `set_gpr`. More precisely, we can state: `set_gpr s_asm 22 (to_int i) = δ_proc i [s_asm]`. As we see later on, δ_{proc} additionally increments the program counters.

expose the generic interface. This section defines the class of automata that formally specify the semantics of VAMOS processes, thus characterizing an abstract kernel ABI. These automata can extend the semantics of sequential languages by primitives for the communication with the kernel. We define:

Definition 4.1 (Process Model). A process model is an input-output automaton specified by a tuple

$$(\mathcal{S}_{\text{proc}}, \text{is_valid}_{\text{proc}}, \text{is_init}_{\text{proc}}, \Sigma_{\text{proc}}, \Omega_{\text{proc}}, \delta_{\text{proc}}, \omega_{\text{proc}}, \text{size}_{\text{proc}})$$
 with

- a state space $\mathcal{S}_{\text{proc}}$,
- a validity predicate $\text{is_valid}_{\text{proc}}$,
- an initialization predicate $\text{is_init}_{\text{proc}}$,
- the alphabet Σ_{proc} of inputs from the kernel to the process,
- the alphabet Ω_{proc} of outputs from the process to the kernel,
- a transition function δ_{proc} , as well as
- output functions ω_{proc} and $\text{size}_{\text{proc}}$.

While the state space $\mathcal{S}_{\text{proc}}$ depends on the individual process model, the interface between kernel and processes is naturally shared by all process models. This interface is entirely defined by Σ_{proc} and Ω_{proc} .

Among other things, the output alphabet Ω_{proc} enumerates all possible kernel calls. Each kernel call has an identifier and a number of arguments, e.g., `SET_PRIVILEGED` hn for setting privileges. In the implementation, the kernel-call parameters are usually stored in registers. The C0 semantics, however, completely abstracts from registers. A translation from C0 expressions into register values would be quite artificial. Hence, the process semantics generally interpret the parameter values using a more abstract representation, which we explain below.

In many cases, not all possible register values have a meaningful interpretation. There are, for instance, only three priorities, eight devices, 1024 ports, and four IPC-right bits. If a process specifies a different value, the parameter is interpreted as \perp , meaning “out of range”.² Otherwise, the argument value is of the form $\lfloor x \rfloor$.

In two situations, however, a single value for “out of range” is not sufficient: The implementation of kernel calls like `DEV_READ` stores a word sequence into the memory of the process. In this case, one register specifies a pointer into the memory, and another one the length. Though the register values refer to bytes, only word-aligned values are valid. The kernel distinguishes such misaligned values (represented by an argument value `BufUndefined`) from well-specified memory regions that are situated beyond the available process memory (represented by `BufUnavailable`). A valid buffer is solely described by its length: `BufLength len`. Thus, the output `DEV_READ dev_id port (BufLength len)` describes the inquiry to read at most len words from a device.

Respectively the same holds for calls like `DEV_WRITE`, where the implementation reads a sequence of words out of the process memory. We call such a word sequence a *memory object*, and distinguish undefined memory objects `MObjUndefined`, unavailable objects

²Note that we reuse the same technique as for partial functions, see [Section 1.1](#).

Access Control

SET_PRIVILEGED *hn* — inquiry to privilege the process identified by handle *hn*

Process Management

PROCESS_CREATE *tsl prio img pages* — inquiry to create a new process with the timeslice *tsl* and the scheduling priority *prio* from the memory object *img*. The new process should occupy *pages* memory pages.

PROCESS_CLONE *hn* — inquiry to copy the process (identified by handle) *hn*

PROCESS_KILL *hn* — inquiry to kill the process (identified by) *hn*

Memory Management

MEMORY_ADD *hn pages* / MEMORY_FREE *hn pages* — inquiry to change the amount of virtual memory for the process *hn* by *pages* memory pages

Scheduling Mechanism

CHG_SCHED_PARAMS *hn tsl prio* — inquiry to change the timeslice and the scheduling priority of the process *hn* to *tsl* and *prio*, respectively.

Device Driver Support

CHANGE_DRIVER *hn devs register* — inquiry to either register (if *register = True*) or unregister the process *hn* as a device driver for the set *devs* of devices.

ENABLE_INTERRUPTS *devs* — inquiry to re-enable the interrupts for the devices *devs*

DEV_READ *dev_id port buffer* — inquiry to read from device with the ID *dev_id* at port *port* into the buffer *buffer*.

DEV_WRITE *dev_id port data* — inquiry to write the memory object *data* to port *port* of device *dev_id*.

Inter-Process Communication

IPC_SEND *hn_rcv rights_snd msg hn_add rights_add timeout_snd* — inquiry to send the memory object *msg* to process *hn_rcv* and grant *rights_snd* to the receiver. Additionally, process handle *hn_add* should be transferred, and *rights_add* should be granted for that process. The operation should run out of time after *timeout_snd* timer ticks.

IPC_RECEIVE *hn_snd buffer timeout_rcv* — inquiry to receive a message from process *hn_snd* into the buffer *buffer*. The operation should run out of time after *timeout_rcv* ticks.

IPC_REQUEST *hn_rcv rights_snd msg hn_add rights_add timeout_snd buffer timeout_rcv* — inquiry to send *msg* to *hn_rcv* and immediately wait for a reply from *hn_rcv*.

CHANGE_RIGHTS *hn_subj hn_obj grant rights* — inquiry to either grant (if *grant = True*) or revoke the rights *rights* for process *hn_subj* to send IPC to the process *hn_obj*.

READ_KERNEL_INFO *type* — gain information of type *type* from the kernel

No Kernel Call

UNDEFINED_TRAP — a trap instruction has occurred with an unused trap number

RUNTIME_ERROR — a runtime error like an illegal page fault has occurred

ϵ_Ω — there has been no exception; the process intends to do a local computation

Table 4.1: Output alphabet Ω_{proc} of processes

MObjUnavailable, and valid objects MObjSeq xs comprising a word sequence xs . The output `DEV_WRITE dev_id port MObjUnavailable`, for example, describes the inquiry to write a memory object to a device, though the concerned memory region does not entirely belong to the process memory (and hence, the kernel rejects this inquiry).

In addition to the kernel calls, processes may signal internal error conditions with Ω_{proc} . As the kernel calls are internally identified by a number, a process might specify a number, which is not associated with any call. This condition is signaled by the output value `UNDEFINED_TRAP`. Moreover, a process might generate exceptions like an overflow or an illegal page fault. These exceptions are collectively represented by value `RUNTIME_ERROR`. Finally, the output ε_{Ω} denotes the intention to perform a local computation as the next step. The complete output alphabet is shown in [Table 4.1](#) on the previous page.

The input alphabet Σ_{proc} reflects all kernel-initiated changes of a process. These comprise all possible responses to kernel calls, on the one hand, and kernel commands to change the amount of virtual memory, on the other hand. Furthermore, the kernel passes the input ε_{Σ} to the transition function δ_{proc} in order to request a local transition. [Table 4.2](#) on the facing page shows the complete alphabet for illustration.

The validity predicate `is_validproc` is a concession to the missing predicate subtypes in Isabelle/HOL: The set $\mathcal{S}_{\text{proc}}$ is represented as a type in Isabelle. In order to further constrain this type, we use the validity predicate.

The initialization predicate `is_initproc` is required for the call `PROCESS_CREATE`. This call is intended to start a program. The common scenario is that a C0 program is compiled by the verified C0 compiler and saved in a binary executable file. This file stores the initial memory content (code and data) of the assembly program. In this scenario, the operating system reads the file into memory and then calls the kernel in order to create a new process from the read file content. The kernel initializes the registers of the new process with fixed values and sets the memory content to the specified memory image. While the image uniquely describes the initial assembly configuration, the original C0 program cannot be reconstructed because the variable names are not preserved in the compiled code. Thus, the generic definition of process models uses a predicate relating memory images and initial process configurations.

For the sake of memory management, VAMOS needs to know the amount of virtual memory that is currently occupied by a process. Function `sizeproc` computes the necessary information from a given state.

Valid Process Models. So far, we have illustrated the intuition of our process automata. Our kernel models, however, rely on certain assumptions on the process models. Hence, we constrain the class of valid process models by a set of rules.³ As an intuitive example, we regard the rules that specify the intended semantics of `sizeproc` and its interaction with `ADD_MEMORY` and `FREE_MEMORY`.

³The above definition of process models is implemented as an axiomatic class [[Wen02](#)] in Isabelle. An *axiomatic class* specifies a subclass of types by a number of rules (or *axioms*) on operations defined over a class of types. Axiomatic classes are traditionally used to formalize algebraic structures.

Successful Responses to Kernel Calls

SUCCESS — a kernel call has been successfully carried out

SUCC_NEW_PROCESS hn_{new} — the kernel has launched a new process (in response to a PROCESS_CREATE or PROCESS_CLONE call) and handle hn_{new} points to it

SUCC_DEV_READ $data$ — responding to a DEV_READ call, the kernel has read the word list $data$ from a device

SUCC_RECEIVE hn_{snd} $reused_{\text{snd}}$ $rights_{\text{snd}}$ msg hn_{add} $reused_{\text{add}}$ $rights_{\text{add}}$ $stale$ ito $devs$
 — a kernel call IPC_RECEIVE or IPC_REQUEST has been successful. Handle hn_{snd} points to the IPC partner, this handle has been reused for a new process iff $reused_{\text{snd}} = \text{True}$, and the receiver has currently associated the IPC rights $rights_{\text{snd}}$ with hn_{snd} . The partner has sent the word list msg and additionally hn_{add} , which has been reused iff $reused_{\text{add}} = \text{True}$, and the receiver has currently $rights_{\text{add}}$ on handle hn_{add} . The parameters $stale$, ito , and $devs$ represent kernel notification bits: Iff $stale$ is set, there are stale handles, value ito signals that the sender waits with an infinite timeout, and the set $devs$ notifies about raised device interrupts.

SUCC_KINFO_STALEH hns — the kernel returns the set hns of stale handles in response to READ_KERNEL_INFO [StolenHandles]

Kernel Responses in Error Cases

ERR_OUT_OF_MEM — there is not enough memory to perform the inquired action

ERR_OUT_OF_PIDS — currently, no new processes can be launched because the upper bound of active processes has already been reached

ERR_PROCESS_NOT_READY — the specified process is not ready for a memory change

ERR_UNPRIVILEGED — the inquiry failed because the caller is not privileged or has insufficient IPC rights

ERR_INVALID_ARGS — the arguments specified together with the inquiry are invalid

ERR_INT_ALREADY_HANDLED — there was an attempt to register a process as a driver for a device, which is already driven by another process

ERR_INVALID_HANDLE — the specified handle is invalid. Some kernel calls take more than one handle. In these situations, we distinguish the invalid handle by a more specific error code: ERR_INVALID_SUBJ_HANDLE / ERR_INVALID_OBJ_HANDLE, ERR_SND_INVALID_HANDLE / ERR_RCV_INVALID_HANDLE.

ERR_SND_TIMEOUT / ERR_RCV_TIMEOUT — a timeout has occurred during the send/receive phase of the inquired IPC operation

ERR_MSG_OVERSIZED — the sending of a message failed because the message is too large to be accommodated in the buffer of the receiver

ERR_SND_SEGV / ERR_RCV_SEGV — the specified message or buffer, respectively, lives not completely in the accessible part of the memory.

Commands from the Kernel

ADD_MEMORY $pages$ — increase the amount of available memory by $pages$

FREE_MEMORY $pages$ — decrease the amount of available memory by $pages$

ϵ_{Σ} — perform a local step

Table 4.2: Input alphabet Σ_{proc} of processes

The memory size of a valid process state s is bounded, i. e.,

$$\frac{\text{is_valid}_{\text{proc}} s}{\text{size}_{\text{proc}} s \leq \text{TVM_MAXPAGES}} \quad (\text{size_bounded})$$

where TVM_MAXPAGES is an implementation-defined constant.

As long as the resulting size is below this bound, an input $\text{ADD_MEMORY } pages$ should increase the size by $pages$, formally:

$$\frac{\text{size}_{\text{proc}} s + pages \leq \text{TVM_MAXPAGES} \quad \text{is_valid}_{\text{proc}} s}{\text{size}_{\text{proc}} (\delta_{\text{proc}} (\text{ADD_MEMORY } pages) s) = \text{size}_{\text{proc}} s + pages} \quad (\text{add_memory})$$

Respectively the same holds for FREE_MEMORY :

$$\frac{pages < \text{size}_{\text{proc}} s \quad \text{is_valid}_{\text{proc}} s}{\text{size}_{\text{proc}} (\delta_{\text{proc}} (\text{FREE_MEMORY } pages) s) = \text{size}_{\text{proc}} s - pages} \quad (\text{free_memory})$$

For all other inputs σ , the amount of available memory should remain constant, i. e.,

$$\frac{\forall x. \sigma \neq \text{ADD_MEMORY } x \quad \forall x. \sigma \neq \text{FREE_MEMORY } x \quad \text{is_valid}_{\text{proc}} s}{\text{size}_{\text{proc}} (\delta_{\text{proc}} \sigma s) = \text{size}_{\text{proc}} s} \quad (\text{size_const})$$

Furthermore, the output of a process should remain constant under memory changes. The only exception to that rule, however, are memory-related errors, e. g., the values MObjUnavailable or BufUnavailable in a process output: If a specified memory object or buffer has been completely inside the available memory, it might not be after FREE_MEMORY . Respectively, a memory object or buffer might have exceeded the available memory before ADD_MEMORY but does not afterwards. We express this relation between process outputs by the definition of a weak partial order such that, e. g.,

$$\begin{aligned} \text{DEV_READ } dev_id \text{ port } \text{BufUnavailable} &\leq \text{DEV_READ } dev_id \text{ port } \text{buffer} \\ \text{DEV_WRITE } dev_id \text{ port } \text{MObjUnavailable} &\leq \text{DEV_WRITE } dev_id \text{ port } \text{data} \\ \text{IPC_RECEIVE } hn_{\text{snd}} \text{ BufUnavailable } timeout_{\text{rcv}} &\leq \text{IPC_RECEIVE } hn_{\text{snd}} \text{ buffer } timeout_{\text{rcv}} \end{aligned}$$

Based on this order, we formalize the relation of process outputs under memory changes as follows:

$$\frac{\text{size}_{\text{proc}} s + pages \leq \text{TVM_MAXPAGES} \quad \text{is_valid}_{\text{proc}} s}{\omega_{\text{proc}} s \leq \omega_{\text{proc}} (\delta_{\text{proc}} (\text{ADD_MEMORY } pages) s)} \quad (\text{add_output})$$

$$\frac{pages < \text{size}_{\text{proc}} s \quad \text{is_valid}_{\text{proc}} s}{\omega_{\text{proc}} (\delta_{\text{proc}} (\text{FREE_MEMORY } pages) s) \leq \omega_{\text{proc}} s} \quad (\text{free_output})$$

There are far more constraints on valid process models. An extensive part of the specification concerns the invariance of $\text{is_valid}_{\text{proc}}$. The canon of the according rules is

that (a) the predicate $\text{is_valid}_{\text{proc}}$ holds for any initial state s if the initial memory content img and its size pages are in range, as well as (b) under valid inputs, the predicate $\text{is_valid}_{\text{proc}}$ is preserved by transitions. The rules are shown in [Table 4.3](#) on the next page. Furthermore, there are constraints on the process output. An important premise for the fairness proof in [Chapter 7](#) is that timeslices in VAMOS are bounded. This property can only be established if the timeslices specified with the calls `PROCESS_CREATE` and `CHG_SCHED_PARAMS` are bounded. We list all output-related rules in [Table 4.4](#) on page 45.

In the following two sections, we define two particular automata specifying assembly and C0 processes, which both satisfy the above constraints for a valid process model.

4.3 The Assembly-Process Model

The model of assembly processes extends the semantics of the sequential VAMP assembly language. For the communication with the kernel, we use the `TRAP` instruction, which is simply illegal in the sequential fragment of the assembly language.

The state space of assembly processes is identical to the one of the assembly semantics. In this section, we define the according predicates $\text{is_valid}_{\text{proc}}$ and $\text{is_init}_{\text{proc}}$ for this state space, its transition function δ_{proc} , as well as the according output functions ω_{proc} and $\text{size}_{\text{proc}}$.

Predicates. Recall that the validity predicate $\text{is_valid}_{\text{proc}}$ further constrains the state-space type of a process model to the well-formed states. For the sequential assembly semantics, the necessary well-formedness constraints are collected in predicate valid_asm . We add two more constraints for assembly processes: The system mode is forbidden and the main-memory size is bounded by zero and the total size (`TVM_MAXPAGES`) of virtual memory in the system. Formally, we define:

$$\text{is_valid}_{\text{proc}} \ s_{\text{asm}} \equiv \text{valid_asm} \ s_{\text{asm}} \wedge 0 \leq s_{\text{asm}}.\text{sprs} \ \& \ \text{PTL} < \text{int TVM_MAXPAGES} \wedge s_{\text{asm}}.\text{sprs} \ \& \ \text{MODE} \neq 0$$

As mentioned in [Section 4.2](#), we use the initialization predicate for the call `PROCESS_CREATE` $\text{tsl} \ \text{prio} \ \text{img} \ \text{pages}$. If an assembly process is created, the initial state is uniquely determined by the parameters img and pages , the memory image and the process size in pages, respectively. We define a function that computes the initial state from both parameters:

$$\text{init}_{\text{asm}} \ \text{img} \ \text{pages} \equiv \begin{aligned} & (\text{dpc} = 0, \text{pcp} = 4, \text{gprs} = \text{replicate } 32 \ 0, \\ & \text{sprs} = \text{replicate } 32 \ 0[\text{PTL} := \text{to_int32 } \text{pages} - 1, \text{MODE} := 1], \\ & \text{mm} = \lambda ad. \text{if } ad < |\text{img}| \ \text{then } \text{img} \ ! \ ad \ \text{else } 0) \end{aligned}$$

The program counters point to the first two addresses in the main memory. The register files are initialized with 32 registers, each; the register values are set to zero with the exception of `MODE` and `PTL`.⁴ The mode register is set to 1 indicating that the process

⁴The value of most special-purpose registers is irrelevant. The assembly semantics does not permit the access in the user mode, anyways.

$\frac{0 < pages \leq TVM_MAXPAGES \quad img \leq pages \cdot PAGE_SIZE}{\forall d \in \text{set } img. -2^{31} \leq d < 2^{31} \quad \text{is_init_proc } img \text{ pages } s} \quad \text{is_valid_proc } s$	(is_init_proc_valid)
$\frac{\omega_{\text{proc } s} \neq \epsilon_{\Omega} \quad \text{is_valid_proc } s}{\text{is_valid_proc } (\delta_{\text{proc}} \text{ SUCCESS } s)}$	(success_valid)
$\frac{\omega_{\text{proc } s} = \text{DEV_READ } dev_id \text{ port } (\text{BufLength } len) \quad data = len \quad \forall d \in \text{set } data. -2^{31} \leq d < 2^{31} \quad \text{is_valid_proc } s}{\text{is_valid_proc } (\delta_{\text{proc}} (\text{SUCC_DEV_READ } data) s)}$	(succ_dev_read_valid)
$\frac{\text{is_request } (\omega_{\text{proc } s}) \vee \text{is_receive } (\omega_{\text{proc } s}) \quad -2^{31} \leq hn_{\text{snd}} < 2^{31} \quad -2^{31} \leq hn_{\text{add}} < 2^{31} \quad \forall d \in \text{set } msg. -2^{31} \leq d < 2^{31} \quad msg \cdot 4 < 2^{32} \quad \text{is_valid_proc } s}{\text{is_valid_proc } (\delta_{\text{proc}} (\text{SUCC_RECEIVE } hn_{\text{snd}} \ u_s \ g_s \ msg \ hn_{\text{add}} \ u_a \ g_a \ stl \ ito \ devs) s)}$	(succ_receive_valid)
$\frac{\omega_{\text{proc } s} \neq \epsilon_{\Omega} \quad -2^{31} \leq hn < 2^{31} \quad \text{is_valid_proc } s}{\text{is_valid_proc } (\delta_{\text{proc}} (\text{SUCC_NEW_PROCESS } hn) s)}$	(succ_new_process_valid)
$\frac{\omega_{\text{proc } s} = \text{READ_KERNEL_INFO } kinfo \quad \text{is_valid_proc } s}{\text{is_valid_proc } (\delta_{\text{proc}} (\text{SUCC_KINFO_STALEH } hns) s)}$	(succ_kinfo_staleh_valid)
$\frac{\text{size_proc } s + pages \leq TVM_MAXPAGES \quad \text{is_valid_proc } s}{\text{is_valid_proc } (\delta_{\text{proc}} (\text{ADD_MEMORY } pages) s)}$	(add_memory_valid)
$\frac{pages < \text{size_proc } s \quad \text{is_valid_proc } s}{\text{is_valid_proc } (\delta_{\text{proc}} (\text{FREE_MEMORY } pages) s)}$	(free_memory_valid)
$\frac{\omega_{\text{proc } s} \neq \epsilon_{\Omega} \quad \text{is_error } \sigma \quad \text{is_valid_proc } s}{\text{is_valid_proc } (\delta_{\text{proc}} \sigma s)}$	(is_error_valid)
$\frac{\omega_{\text{proc } s} = \epsilon_{\Omega} \quad \text{is_valid_proc } s}{\text{is_valid_proc } (\delta_{\text{proc}} \epsilon_{\Sigma} s)}$	(no_input_valid)

Table 4.3: Rules on valid process models concerning the invariance of is_valid_proc

$\frac{\omega_{\text{proc}} s = \text{PROCESS_CREATE } \textit{tsl} \ \textit{prio} \ \textit{img} \ \textit{pages} \quad \textit{is_valid}_{\text{proc}} s}{\textit{tsl} \leq \text{MAXTSL}} \quad (\text{maxtsl_create})$
$\frac{\omega_{\text{proc}} s = \text{CHG_SCHED_PARAMS } \textit{hn} \ \textit{tsl} \ \textit{prio} \quad \textit{is_valid}_{\text{proc}} s}{\textit{tsl} \leq \text{MAXTSL}} \quad (\text{maxtsl_sched})$
$\frac{\omega_{\text{proc}} s = \text{PROCESS_CREATE } \textit{tsl} \ \textit{prio} \ (\text{MObjSeq } \textit{img}) \ \textit{pages} \quad \textit{is_valid}_{\text{proc}} s}{\forall d \in \text{set } \textit{img}. -2^{31} \leq d < 2^{31}} \quad (\text{img_create})$
$\frac{\omega_{\text{proc}} s = \text{IPC_SEND } \textit{hn}_{\text{rcv}} \ \textit{rights}_{\text{snd}} \ (\text{MObjSeq } \textit{msg}) \ \textit{hn}_{\text{add}} \ \textit{rights}_{\text{add}} \ \textit{to}_{\text{snd}} \quad \textit{is_valid}_{\text{proc}} s}{(\forall d \in \text{set } \textit{msg}. -2^{31} \leq d < 2^{31}) \wedge \textit{msg} \cdot 4 < 2^{32}} \quad (\text{msg_send})$
$\frac{\omega_{\text{proc}} s = \text{IPC_REQUEST } \textit{hn}_{\text{rcv}} \ \textit{rgh}_{\text{snd}} \ (\text{MObjSeq } \textit{msg}) \ \textit{hn}_{\text{add}} \ \textit{rgh}_{\text{add}} \ \textit{to}_{\text{snd}} \ \textit{buff} \ \textit{to}_{\text{rcv}} \quad \textit{is_valid}_{\text{proc}} s}{(\forall d \in \text{set } \textit{msg}. -2^{31} \leq d < 2^{31}) \wedge \textit{msg} \cdot 4 < 2^{32}} \quad (\text{msg_request})$

 Table 4.4: Rules on valid process models constraining the output function ω_{proc}

runs in user mode, and the PTL register reflects the intended size \textit{pages} of the process, such that $\text{mm_size}(\text{init}_{\text{asm}} \ \textit{img} \ \textit{pages}) = \textit{pages}$.

Finally, we define the initialization predicate for assembly processes simply as an equality test:

$$\text{is_init}_{\text{proc}} \ \textit{img} \ \textit{pages} \ s_{\text{asm}} \equiv s_{\text{asm}} = \text{init}_{\text{asm}} \ \textit{img} \ \textit{pages}$$

Transitions. The transition function δ_{proc} is specified by a case distinction on the process inputs. For all error inputs σ , the transition function updates the result register 22 with the corresponding error code and increases the program counters. Formally, we define:

$$\delta_{\text{proc}} \ \sigma \ s_{\text{asm}} = s_{\text{asm}}(\text{gprs} := s_{\text{asm}}.\text{gprs}[22 := \text{to_int } \sigma], \text{dpc} := s_{\text{asm}}.\text{pcp}, \text{pcp} := (s_{\text{asm}}.\text{pcp} + 4) \bmod 2^{32})$$

where to_int converts the symbolic error inputs into the corresponding plain integer number, e. g., $\text{to_int } \text{ERR_UNPRIVILEGED} = -4$. Respectively the same holds for SUCCESS and $\text{SUCC_NEW_PROCESS } \textit{hn}_{\text{new}}$, with a numerical result value of 0 and \textit{hn}_{new} , respectively. For the other successful kernel replies, δ_{proc} is defined in a similar fashion but other general-purpose registers or the main memory are changed in addition to the register 22. Iff the input is $\text{ADD_MEMORY } \textit{pages}$, we increase the PTL register and zero-initialize the additional memory. Most notably, the program counters are left unchanged because they do not complete a pending kernel call. We specify

$$\delta_{\text{proc}} (\text{ADD_MEMORY } \textit{pages}) s_{\text{asm}} =$$

$$s_{\text{asm}}(\text{spr} := s_{\text{asm}}.\text{spr}[\text{PTL} := s_{\text{asm}}.\text{spr} ! \text{PTL} + \text{to_int32 } \textit{pages}],$$

$$\text{mm} := \lambda i. \text{if } \text{mm_size } s_{\text{asm}} \cdot 1024 \leq i < (\text{mm_size } s_{\text{asm}} + \textit{pages}) \cdot 1024 \text{ then } 0$$

$$\text{else } s_{\text{asm}}.\text{mm } i)$$

and respectively for `FREE_MEMORY` *pages*:

$$\delta_{\text{proc}} (\text{FREE_MEMORY } \textit{pages}) s_{\text{asm}} =$$

$$s_{\text{asm}}(\text{spr} := s_{\text{asm}}.\text{spr}[\text{PTL} := s_{\text{asm}}.\text{spr} ! \text{PTL} - \text{to_int32 } \textit{pages}])$$

Finally, we inherit the transition from the sequential assembly semantics for the empty input:

$$\delta_{\text{proc}} \varepsilon_{\Sigma} s_{\text{asm}} = \delta_{\text{asm}} s_{\text{asm}}$$

Output Functions. The output function ω_{proc} takes a process state s_{asm} and computes the corresponding output ω . In short, the function determines whether a process generates an exception during the next step and classifies the various exceptions. A `RUNTIME_ERROR` subsumes all programming errors, namely illegal instructions, misaligned or out-of-range memory accesses, or unintended arithmetic overflows.⁵ If the process is about to execute a `TRAP` instruction, we further differentiate the various kernel calls (see [Table 4.1](#) on page 39) and an `UNDEFINED_TRAP`, which signals that a trap instruction occurs with an unused trap number. Finally, if no exception is generated in the real system, the function returns ε_{Ω} . We specify:

$$\omega_{\text{proc}} s_{\text{asm}} \equiv$$

$$\text{if } \text{is_illegal_instr } s_{\text{asm}} \vee \text{is_misaligned_pc } s_{\text{asm}} \vee \text{is_misaligned_data } s_{\text{asm}} \vee$$

$$\text{is_outranged_pc } s_{\text{asm}} \vee \text{is_outranged_data } s_{\text{asm}} \vee \text{is_overflow } s_{\text{asm}}$$

$$\text{then } \text{RUNTIME_ERROR}$$

$$\text{else if } \exists \textit{imm}. \text{current_instr } s_{\text{asm}} = \text{TRAP } \textit{imm} \text{ then } \text{trap_sel } s_{\text{asm}} \text{ else } \varepsilon_{\Omega}$$

The auxiliary function `trap_sel` is essentially an elaborate case distinction on the immediate constant of `current_instr` s_{asm} , i. e.,

$$\text{current_instr } s_{\text{asm}} = \text{TRAP } 1 \implies$$

$$\text{trap_sel } s_{\text{asm}} =$$

$$\text{PROCESS_CREATE } (\text{to_nat32 } (s_{\text{asm}}.\text{gpr} ! 11)) (\text{to_prio } (s_{\text{asm}}.\text{gpr} ! 12))$$

$$(\text{to_memobj } s_{\text{asm}} (s_{\text{asm}}.\text{gpr} ! 13) (s_{\text{asm}}.\text{gpr} ! 14)) (\text{to_nat32 } (s_{\text{asm}}.\text{gpr} ! 15))$$

$$\text{current_instr } s_{\text{asm}} = \text{TRAP } 2 \implies \text{trap_sel } s_{\text{asm}} = \text{PROCESS_CLONE } (s_{\text{asm}}.\text{gpr} ! 11)$$

$$\text{current_instr } s_{\text{asm}} = \text{TRAP } 3 \implies \text{trap_sel } s_{\text{asm}} = \text{PROCESS_KILL } (s_{\text{asm}}.\text{gpr} ! 11)$$

etc. and ultimately:

$$\llbracket \text{current_instr } s_{\text{asm}} = \text{TRAP } \textit{imm}; \textit{imm} \notin \text{used_trapnrs} \rrbracket$$

$$\implies \text{trap_sel } s_{\text{asm}} = \text{UNDEFINED_TRAP}$$

⁵The arithmetic operations of the VAMP come in two flavors: one set ignores overflows and the other one generates an exception. We call an overflow *unintended* iff it occurs on an exception-generating operation.

The output function $\text{size}_{\text{proc}}$ is much easier to specify: It is simply defined to be mm_size .

Validity. With the above definitions, we have specified a model of assembly processes. Now, we show that this model is indeed a valid process model such that we can later use it in our kernel models.

Theorem 4.1 (Validity of the Assembly-Process Model). *Our assembly-process model is valid.*

Proof. A process model is called valid if all 21 validity rules introduced in [Section 4.2](#) hold for this model. From the definitions of $\text{is_valid}_{\text{proc } s_{\text{asm}}}$ and $\text{size}_{\text{proc } s_{\text{asm}}}$, we can easily conclude ([size.bounded](#)). The proof of ([add.memory](#)) mainly involves a careful inspection of the definitions for $\delta_{\text{proc}}(\text{ADD_MEMORY pages}) s_{\text{asm}}$, $\text{size}_{\text{proc } s_{\text{asm}}}$, and $\text{mm_size } s_{\text{asm}}$. Furthermore, we can conclude ([add.output](#)), i. e., that the output after $\delta_{\text{proc}}(\text{ADD_MEMORY pages}) s_{\text{asm}}$ is greater or equal, because the program counters are not altered and a possibly vanished memory-related error still satisfies our order relation. Respectively the same can be said about ([free.memory](#)) and ([free.output](#)). In order to prove ([size.const](#)), we observe that $\text{size}_{\text{proc } s_{\text{asm}}}$ is equal to $\text{mm_size } s_{\text{asm}}$ and the latter only accesses the special-purpose register PTL, which is only changed by $\delta_{\text{proc}} \sigma s_{\text{asm}}$ if the input σ is `ADD_MEMORY pages` or `FREE_MEMORY pages`. Similarly, we prove that the remaining 15 rules hold. We save us the tedious walk through all those rules in this document. In the work for this thesis, the validity of assembly processes has been formally shown in Isabelle/HOL. \square

4.4 The C0-Process Model

C0 processes extend the sequential C0 language. There is, however, no built-in language construct in C0 that permits the communication with the kernel. A common practice in C programming is the implementation of language extensions by specialized libraries, which internally use assembly code. We have used this technique for the desired kernel-communication facilities: A special *kernel library* provides the kernel-call semantics to C0 programs via functions, which are implemented in inlined assembly code. Using this library, user programs can be written in C0 without extra portions of assembly. A kernel call in the user program simply becomes a function call to one of the library functions.

As an example, [Figure 4.1](#) on the following page shows the implementation of the function `vc_process_clone`, which provides the kernel call `PROCESS_CLONE hn` to C0 user programs. The implementation loads the parameter `hn`, which identifies the process to clone, into register 11, performs a `trap` instruction passing constant 2, and returns with the value of register 22, which contains the response from the kernel.

The implementation of some kernel calls, however, is more involved than the above example suggests because our kernel library supports generic types. Typically, generic types are used for data containers like doubly-linked lists. We use this concept primarily to permit sending and receiving of arbitrarily typed C0 values between processes. Unfortunately, C0 does not support a generic programming concept like C++ templates

```
int vc_process_clone(unsigned int hn) {
    int result;
    asm { lw(r11, r30, asm_offset(hn));
          trap(2);
          sw(r22, r30, asm_offset(result));
        };
    return result;
}
```

Figure 4.1: Implementation of the function `vc_process_clone` from the kernel library

[Int98, § 14]. Hence, we use C preprocessing macros [Ame99, § 6.10.3] to specify generic patterns for individually typed library functions.

More specifically, the actual C0 program code and the library implementation of a C0 process are linked together on the source-code level. Thus, the programmer can specify via macros within the C0 program code, for which concrete types the C0 program requires which of the generic library functionality. The C preprocessor then expands the macros from the pattern specified in the library to a particular function implementation. The following macro, for instance, expands to the definition of the function `vc_ipc_send_int`, which is used to send a single integer to another process:

```
DEFINE_VAMOS_IPC_SEND(int)
```

This general approach has first been used within Verisoft for the C0 library of doubly-linked lists [Ngu05].

Furthermore, our library supports the exchange of arbitrarily typed data via IPC. On the kernel level, there are no means to control that the corresponding data types match. Thus, a C0 process might expect to receive a longer message from another process than the latter actually sends.⁶

Even worse, a process might receive a value that is out of range for the concerned type: All C0 values of a fundamental type are represented by 32-bit integers on the assembly layer but a Boolean value is represented by either 0 or 1; the value 2 should just never occur (and might confuse the C0 semantics). Hence, the kernel library features special assembly code that sanitizes the received data by converting possibly illegal data into a valid value. For a Boolean value, its representation is set to 1 unless it had been zero, for a character value, all bits above the eighth are set to zero, and for a pointer value, the representation is simply turned into 0 (meaning Null).

We now turn our attention towards the definition of the actual process model.

⁶Note that the kernel rejects to send a message if it exceeds the buffer of the intended receiver. The possibility to receive a shorter message, on the contrary, we regard as a distinctive feature. The SOS, for instance, supports a wide range of various requests, some requiring more, others less parameters. Hence, the SOS library features a short and a long message type for sending and always calls receive with the long message type.

State Space. Unlike assembly processes, we cannot simply reuse the state space of the sequential C0 semantics for C0 processes.

First, the sequential transition function is partial, i. e., δ_{C0} returns \perp in case of a programming error like null-pointer dereferencing. In order to keep track of such an error, we extend the state space of the sequential semantics by the value \perp . The sequential program state is kept in the record component $s_{C0}.pstate$ of the C0-process state s_{C0} . Thus, $s_{C0}.pstate$ is either \perp , or $[s_{C0}.pstate].prog$ stores the remaining statements and $[s_{C0}.pstate].mem$ the current state of the program variables and the heap objects.

Second, we need to retain the static program definition within the process state because though the program definition is static during the lifetime of a C0 program in execution, a process might be replaced by another one during the lifetime of the computer system. Hence, a C0-process state s_{C0} features the record components $s_{C0}.typetab$ and $s_{C0}.funtab$, which represent the type-name table and the function table. Note that the symbol table of the global variables can be reconstructed from the memory configuration $[s_{C0}.pstate].mem$ and does not need to be stored additionally (for $s_{C0}.pstate = \perp$, the program definition is irrelevant).

Finally, the sequential C0 semantics implicitly assumes sufficient memory, which is inappropriate for C0 processes. Hence, we store the main-memory size of the process in the record component $s_{C0}.mmsize$.

Summarizing, a state s_{C0} of a C0 process comprises the following four components:

- $s_{C0}.pstate$ holds the current program state of the C0 program in execution,
- $s_{C0}.typetab$ retains the type-name table,
- $s_{C0}.funtab$ stores the function table, and
- $s_{C0}.mmsize$ contains the current main-memory size of the process.

In the remainder of the section, we define the according predicates is_valid_{proc} and is_init_{proc} for C0-process states, their transition function δ_{proc} , as well as the corresponding output functions ω_{proc} and $size_{proc}$.

Predicates. The validity predicate is_valid_{proc} formulates well-formedness constraints on the states of a process model. For C0, we inherit those from the sequential semantics. These are collected in the predicate $valid_C0\ tt\ ft\ c$, which takes three parameters: the type-name table tt , the function table ft , and the state c of the program in execution. Additionally, we require two conditions for well-formed C0 processes.

First, the main-memory size is bounded by the total size ($TVM_MAXPAGES$) of virtual memory in the system.

Second, the program has to comply with the kernel library. More specifically, if items of the kernel library are declared in the program definition, the items have to match with the library's one.

We illustrate the latter condition by an example: If there is a function called `vc_process_clone` then it has exactly one parameter, which has the name `hn` and the type unsigned 32-bit integer, and the return type of the function is signed 32-bit integer. It does not suffice, however, to constrain the function's calling interface, i. e., the parameters and the return type. In order to figure out whether an assembly memory image can be

abstracted to a C0 process, we have to know in the process model how the library functions are implemented.

From [Figure 4.1](#) on page 48, we learn that function `vc_process_clone` has exactly one stack variable, which is called `result` and its type is signed 32-bit integer. Furthermore, we constrain the function body with a predicate `is_clone_body`. Note that for a generic function pattern, we cannot define a constant with its function body in the C0 semantics, even if the implementations of the function body (as expanded by the C preprocessor) are identical: Recall that a statement in the C0 semantics features an identifier, which is unique in the whole program code. Thus, two functions generated from the same pattern, like `vc_ipc_send_int` and `vc_ipc_send_bool`, feature distinct statement identifiers. For the same reason, we refrain from defining constants for the body of fixed functions like `vc_process_clone`.

We specify our requirements on `vc_process_clone` as follows:

$$\begin{aligned} & (“vc_process_clone”, fdef) \in \text{set } ft \implies \\ & \text{is_clone_body } fdef.\text{body} \wedge \\ & fdef.\text{stack_vars} = [(“result”, Integer)] \wedge \\ & fdef.\text{params} = [(“hn”, UnsgndT)] \wedge fdef.\text{rettype} = Integer \end{aligned}$$

Similarly, we proceed for all functions of the kernel library. Finally, we collect all these requirements on the program definition in the predicate `libvamos_compatible tt ft gst`.

Note the slight difference of our approach compared to the one that In der Rieden & Tsyban [IT08, In 09] pursue when linking a library with the C0 code of a kernel: They assume two distinct programs, which are then linked together, while we only regard the linked C0 program. Where they require certain preconditions on the two programs to link, we simply specify the class of C0 programs that is compatible with our library. This different perspective might be influenced by the different nature of the libraries: While In der Rieden & Tsyban just regard a library with fixed functions, we deal with generic function patterns for generic types.

Now, we formally specify the validity of C0-process states:

$$\begin{aligned} \text{is_valid}_{\text{proc}} s_{C0} &\equiv \\ & s_{C0}.\text{mmsize} \leq \text{TVM_MAXPAGES} \wedge \\ & (\text{case } s_{C0}.\text{pstate} \text{ of } \perp \Rightarrow \text{True} \\ & \quad | [c] \Rightarrow \text{valid_C0 } s_{C0}.\text{typetab } s_{C0}.\text{funtab } c \wedge \\ & \quad \quad \text{libvamos_compatible } s_{C0}.\text{typetab } s_{C0}.\text{funtab } (\text{gm_st } c.\text{mem})) \end{aligned}$$

The initialization predicate `is_initproc img pages sC0` is used for the call `PROCESS_CREATE tsl prio img pages`. It determines whether the C0 state `sC0` is a valid initial state of size `pages` with a C0 program that can be compiled into the memory image `img`. At first, the process should have the correct size. Furthermore, the C0 program has to be compilable and compatible with the kernel library. The program state should be the initial state of this program. Finally, the code of the compiled program should be stored in `img` and the address range for the global variables should be zero-initialized (we summarize both conditions in the auxiliary predicate `is_compilation tt ft gst img`). Formally:

$$\begin{aligned}
\text{is_init}_{\text{proc}} \text{ img pages } s_{C0} &\equiv \\
& s_{C0}.\text{mmsize} = \text{pages} \wedge \\
& (\exists \text{gst. is_compilable } s_{C0}.\text{typetab } s_{C0}.\text{funtab } \text{gst} \wedge \\
& \quad \text{libvamos_compatible } s_{C0}.\text{typetab } s_{C0}.\text{funtab } \text{gst} \wedge \\
& \quad s_{C0}.\text{pstate} = [\text{init_C0 } s_{C0}.\text{funtab } \text{gst}] \wedge \\
& \quad \text{is_compilation } s_{C0}.\text{typetab } s_{C0}.\text{funtab } \text{gst } \text{img})
\end{aligned}$$

Transitions. The transition function of C0 processes is defined as a case distinction on the process input σ . Three such inputs are kernel responses that do not fit into a single result value: `SUCC_DEV_READ` data , `SUCC_RECEIVE` $\text{hn}_{\text{snd}} \text{reused}_{\text{snd}} \text{rights}_{\text{snd}} \text{msg } \text{hn}_{\text{add}} \text{reused}_{\text{add}} \text{rights}_{\text{add}} \text{stale } \text{ito } \text{devs}$, and `SUCC_KINFO_STALEH` hns . As the corresponding updates are quite involved, we encapsulate them in the functions `c0_updt_succ_read`, `c0_updt_succ_receive`, and `c0_updt_succ_kinfo`, respectively. These functions compute the next program state from the type-name table, the program state, and (some of) the input parameters.

The update for the inputs `ADD_MEMORY` pages and `FREE_MEMORY` pages , in contrast, is straightforward: We just add or subtract the specified number of pages from the component $s_{C0}.\text{mmsize}$.

If the input is ϵ_{Σ} , we use the sequential transition function δ_{C0} in order to compute the program state of the next step. For all other inputs, namely `SUCCESS`, `SUCC_NEW_PROCESS` hn_{new} , and the error inputs, we simply update the result variable by `to_int` σ . This update is encapsulated in the function `set_result` $\text{tt } c \ i$.

Formally, we specify the transitions of C0 processes as follows:

$$\begin{aligned}
\delta_{\text{proc}} \sigma s_{C0} &\equiv \\
& \text{case } \sigma \text{ of} \\
& \text{SUCC_DEV_READ } \text{data} \Rightarrow \\
& \quad s_{C0}(\text{pstate} := \text{let}_{\perp} c = s_{C0}.\text{pstate} \text{ in } \text{c0_updt_succ_read } s_{C0}.\text{typetab } c \ \text{data}) \\
& | \text{SUCC_RECEIVE } \text{hn}_{\text{snd}} \ \text{reused}_{\text{snd}} \ \text{rights}_{\text{snd}} \ \text{msg } \ \text{hn}_{\text{add}} \ \text{reused}_{\text{add}} \ \text{rights}_{\text{add}} \ \text{stale } \ \text{ito} \\
& \quad \text{devs} \Rightarrow \\
& \quad s_{C0}(\text{pstate} := \text{let}_{\perp} c = s_{C0}.\text{pstate} \\
& \quad \quad \text{in } \text{c0_updt_succ_receive } s_{C0}.\text{typetab } c \ \text{hn}_{\text{snd}} \ \text{rights}_{\text{snd}} \ \text{msg } \ \text{hn}_{\text{add}} \\
& \quad \quad \text{rights}_{\text{add}} \ (\text{to_int } \sigma)) \\
& | \text{SUCC_KINFO_STALEH } \text{hns} \Rightarrow \\
& \quad s_{C0}(\text{pstate} := \text{let}_{\perp} c = s_{C0}.\text{pstate} \text{ in } \text{c0_updt_succ_kinfo } s_{C0}.\text{typetab } c \ \text{hns}) \\
& | \text{ADD_MEMORY } \text{pages} \Rightarrow s_{C0}(\text{mmsize} := s_{C0}.\text{mmsize} + \text{pages}) \\
& | \text{FREE_MEMORY } \text{pages} \Rightarrow s_{C0}(\text{mmsize} := s_{C0}.\text{mmsize} - \text{pages}) \\
& | \epsilon_{\Sigma} \Rightarrow s_{C0}(\text{pstate} := \text{let}_{\perp} c = s_{C0}.\text{pstate} \text{ in } \delta_{C0} \ s_{C0}.\text{typetab } s_{C0}.\text{funtab } c) \\
& | _ \Rightarrow s_{C0}(\text{pstate} := \text{let}_{\perp} c = s_{C0}.\text{pstate} \text{ in } \text{set_result } s_{C0}.\text{typetab } c \ (\text{to_int } \sigma))
\end{aligned}$$

where we write $\text{let}_{\perp} x = a \text{ in } e$ as shorthand for $\text{case } a \text{ of } \perp \Rightarrow \perp \mid [x] \Rightarrow e$.

If we use the function `set_result` $\text{tt } c \ i$, we implicitly assume that the process has signaled a kernel call via ω_{proc} . We may do so because of the constraints on valid process

models (see Table 4.3 on page 44): If the process output is empty, i. e., $\omega_{\text{proc } SC0} = \varepsilon_{\Omega}$, a process transition is only known to yield a valid successor state with one of the following inputs: ε_{Σ} , ADD_MEMORY pages, or FREE_MEMORY pages. In neither case, we use the function `set_result`. As we will see below, the other outputs are either kernel calls, in which case the C0 process calls one of the kernel-library functions, or a runtime error.

In case of a runtime error, we explicitly set the program state to \perp ; otherwise, we update the result variable lv of the current function-call statement `fst_stmt c.prog = SCALL lv fn ps sid` by the specified new value i . Moreover, we remove the call statement from the program using function `rm_scall`, which replaces the first statement by `SKIP` if it is a function call:

$$\text{fst_stmt } c.\text{prog} = \text{SCALL } lv \text{ } fn \text{ } ps \text{ } sid \implies \text{fst_stmt } (\text{rm_scall } c.\text{prog}) = \text{SKIP}$$

We specify function `set_result` as follows:

```
set_result tt c i ≡
  case fst_stmt c.prog of
  SCALL lv fn ps sid ⇒
    let⊥ m' = mem_write tt c.mem (VARACC lv) i
    in [(mem = m', prog = rm_scall c.prog)]
  | _ ⇒ ⊥
```

Using this function, we can now specify the remaining update functions `c0_updt_succ_read`, `c0_updt_succ_receive`, and `c0_updt_succ_kinfo`. All three functions are defined in a very similar fashion, hence, we only present the specification of `c0_updt_succ_receive` as a representative:

```
c0_updt_succ_receive tt c hn_snd rights_snd msg hn_add rights_add i ≡
  case fst_stmt c.prog of
  SCALL lv fn ps sid ⇒
    if ∃ys. fn = "vc_ipc_receive_" ⊙ ys
    then if hn_add = HN_NONE
      then let⊥ m' = mem_writes tt c.mem
        [(DEREF (ps ! 0), [hn_snd]),
         (DEREF (ps ! 1), [rights2num rights_snd]),
         (DEREF (ps ! 2), msg), (DEREF (ps ! 3), [HN_NONE])]
        in set_result tt (c(mem := m')) i
      else let⊥ m' = mem_writes tt c.mem
        [(DEREF (ps ! 0), [hn_snd]),
         (DEREF (ps ! 1), [rights2num rights_snd]),
         (DEREF (ps ! 2), msg), (DEREF (ps ! 3), [hn_add]),
         (DEREF (ps ! 4), [rights2num rights_add])]
        in set_result tt (c(mem := m')) i
    else if hn_add = HN_NONE
      then let⊥ m' = mem_writes tt c.mem
        [(DEREF (ps ! 6), [rights2num rights_snd]),
```



```

      (DEREF ( $\rho s$  ! 7),  $msg$ ), (DEREF ( $\rho s$  ! 8), [HN_NONE]))
  in set_result  $tt$  ( $c(|mem := m'|)$ )  $i$ 
else let  $\perp$   $m' = mem\_writes$   $tt$   $c.mem$ 
      [(DEREF ( $\rho s$  ! 6), [rights2num  $rights_{snd}$ ]),
       (DEREF ( $\rho s$  ! 7),  $msg$ ), (DEREF ( $\rho s$  ! 8), [ $hn_{add}$ ]),
       (DEREF ( $\rho s$  ! 9), [rights2num  $rights_{add}$ ])]
  in set_result  $tt$  ( $c(|mem := m'|)$ )  $i$ 
|  $\_ \Rightarrow \perp$ 

```

This function specification is the most elaborate one of our update functions. We distinguish, whether the kernel responds to an `IPC_RECEIVE` or an `IPC_REQUEST` call. In the former case, we update the first parameters, in the latter, we skip the first seven parameters that belong to the send part and update the subsequent parameters. Moreover, a `IPC_REQUEST` call receives from the same communication partner that it did send to, and hence, there is no need to store a handle to the sender, which is necessary for `IPC_RECEIVE`. In both cases, we update the parameter for the additional rights only if the sender has provided an additional handle (i. e., the additional handle is not `HN_NONE`).

Output Functions. The output function ω_{proc} takes a process state s_{C0} and computes the corresponding output ω . Mainly, we distinguish three cases:

1. A runtime error is indicated, e. g., by an undefined state,
2. the program is about to call one of the kernel-library functions, or
3. the program intends a local step without kernel communication.

Note that runtime errors might show at different stages of the inspection of s_{C0} : If during previous program execution, an uninitialized variable has been read or the Null-pointer dereferenced, the function δ_{C0} is undefined, i. e., $s_{C0}.pstate$ has been set to \perp . This value directly encodes a previous runtime error. Another runtime error is insufficient memory, i. e., if predicate `sufficient_memory` does not hold.

Furthermore, there must be sufficient stack memory to invoke the function, i. e., predicate `sufficient_stack` holds, if a kernel-library function has been called. In addition, a runtime error may be revealed during the expression evaluation of kernel-call arguments.

Technically, we first exclude an undefined program state and insufficient memory. Both conditions immediately lead to a runtime error. Otherwise, we distinguish the first statement of the remaining program. If it is a function call, we check for sufficient stack memory. Insufficient stack memory leads to a runtime error. Otherwise, the corresponding output depends on the function name and the specified arguments – we hide the further case distinction on the function’s name and arguments in the constant `get_output`. If the first remaining statement is not a function call, we always return ϵ_{Σ} .

Thus, we specify:

```

 $\omega_{proc} s_{C0} \equiv$ 
  if  $s_{C0}.pstate = \perp \vee$ 
     $\neg$  sufficient_memory ( $s_{C0}.mmsize \cdot PAGE\_SIZE$ )  $s_{C0}.typetab$   $s_{C0}.funtab$  [ $s_{C0}.pstate$ ]
  then RUNTIME_ERROR

```

```

else case fst_stmt [sC0.pstate].prog of
  SCALL lv fn ps sid  $\Rightarrow$ 
    if  $\neg$  sufficient_stack sC0.typetab sC0.funtab [sC0.pstate] fn then RUNTIME_ERROR
    else get_output sC0.typetab [sC0.pstate] fn ps
  | _  $\Rightarrow$   $\varepsilon_\Omega$ 

```

The auxiliary function `get_output` is essentially an elaborate case distinction on the function name `fn`, e. g.,

```

fn = "vc_process_clone"  $\Rightarrow$ 
  get_output tt c fn es =
    (case mem_read tt c.mem (es ! 0) of  $\perp$   $\Rightarrow$  RUNTIME_ERROR
     | [v]  $\Rightarrow$  PROCESS_CLONE v)
fn = "vc_process_kill"  $\Rightarrow$ 
  get_output tt c fn es =
    (case mem_read tt c.mem (es ! 0) of  $\perp$   $\Rightarrow$  RUNTIME_ERROR
     | [v]  $\Rightarrow$  PROCESS_KILL v)
fn = "vc_process_change_scheduling"  $\Rightarrow$ 
  get_output tt c fn es =
    (let hn = mem_read tt c.mem (es ! 0); tsl = mem_read tt c.mem (es ! 1);
      prio = mem_read tt c.mem (es ! 2)
     in if |es|  $\neq$  3  $\vee$  hn =  $\perp$   $\vee$  tsl =  $\perp$   $\vee$  prio =  $\perp$  then RUNTIME_ERROR
     else CHG_SCHED_PARAMS [hn] [tsl] (nat2prio [prio]))

```

etc. and ultimately:

```

 $\neg$  is_kcall fn  $\Rightarrow$  get_output tt c fn es =  $\varepsilon_\Omega$ 

```

The output function `sizeproc` is much easier to define: It simply returns the current value of component `sC0.mmsize`, i. e., `sizeproc sC0 \equiv sC0.mmsize`.

Validity. With the above definitions, we have specified a model of C0 processes. As for assembly processes, we can show that this model is valid.

Theorem 4.2 (Validity of the C0-Process Model). *Our C0-process model is valid.*

Proof. A process model is called valid if all 21 validity rules introduced in [Section 4.2](#) hold for this model. In principle, we prove these rules for C0 processes in the same manner as for the assembly processes. As an example, the equations ([size_bounded](#)), ([add_memory](#)), and ([free_memory](#)) easily follow from the definitions of `is_validproc sC0`, `sizeproc sC0`, `δ_{proc} (ADD_MEMORY pages) sC0`, and `δ_{proc} (FREE_MEMORY pages) sC0`.

Despite these simple examples, the overall proof effort for the validity of the C0-process model is considerably higher than the one for the assembly-process model. One reason for the increased effort is the more abstract (and highly redundant) memory model. The explicitly typed memory leads to a much more complex, nested invariant on C0 states. This invariant needs to be maintained for all changes induced by kernel calls.

Usually, the kernel calls change the process state similarly to normal program behavior. IPC, however, is an exception to that rule: A process might receive a message, which is shorter than the size of the corresponding buffer variable. In the sequential C0 semantics, on the contrary, there is no means to partially update a variable. Such special cases in the process semantics add to the effort of showing that the invariant is maintained.

Another difficulty results from the evaluation of expressions: In assembly, the values for a computation are directly read from a register or a single memory cell. In C0, on the contrary, such values can be expressed by complex expressions. Consequently, there is a considerably longer chain of reasoning involving the complex invariant to conclude, for instance, that a particular argument is in a specified range.

The validity of the C0-process model has been formally shown in Isabelle/HOL. \square

4.5 Extended Compiler Correctness

Recall that Leinenbach & Petrova [LP08] have shown compiler correctness for the sequential C0 semantics with respect to the sequential part of assembly (cf. [Theorem 2.1](#) on page 23). In this section, we extend their result to the two corresponding process semantics, which have been defined in the last two sections. In particular, we extend the existing simulation relation for processes, define predicates for the successful execution of processes, and formally state that C0 processes simulate assembly processes.

At first, we extend the existing C0 simulation relation. While assembly processes share their state space with the sequential assembly semantics, C0 processes additionally hold information on occurred runtime errors, the program definition, and the available memory size. In essence, our extension is a merely syntactic adaptation, relating the program state $s_{C0}.pstate$ of a C0-process state s_{C0} and an assembly state s_{asm} with the sequential C0 simulation relation `consistent`. Certainly, this relation is only meaningful if there has been no runtime error. In this case, there is simply no corresponding assembly process. Furthermore, the memory size in C0 and in assembly should coincide. We define:

Definition 4.2 (Process-Simulation Relation). The simulation relation `consistentproc` states that an assembly-process state s_{asm} encodes a C0-process state s_{C0} .

Formally, we specify:

$$\begin{aligned} \text{consistent}_{\text{proc}} \ s_{C0} \ \text{alloc} \ s_{asm} &\equiv \\ (s_{C0}.pstate \neq \perp \wedge \text{consistent} \ s_{C0}.typetab \ s_{C0}.funtab \ [s_{C0}.pstate] \ \text{alloc} \ s_{asm}) \wedge \\ s_{C0}.mmsize = \text{mm_size} \ s_{asm} \end{aligned}$$

Now, we define the successful execution of C0 and assembly processes in analogy to the successful execution of sequential assembly code (see [Definition 2.3](#) on page 22). Intuitively, a successful execution is characterized by the absence of runtime errors ($\omega_{\text{proc}} \ s \neq \text{RUNTIME_ERROR}$).

Recall, however, that the compiler-correctness theorem does not hold if statements with inlined assembly are executed in the C0 semantics, i. e.,

$$\forall i < n. \neg \text{is_Asm } (\text{fst_stmt } [\delta_{C0}^i \text{ tt ft } (\text{init_C0 ft gst})].\text{prog})$$

Though there are different techniques [GHLPO5, ST08] to overcome this restriction, we exclude inlined assembly in C0 processes because it is simply dispensable: The kernel library has especially been implemented to spare additional assembly code.

Note that in contrast to the sequential semantics, process transitions take inputs from the kernel. We require that the inputs are valid with respect to the current process state. The rules presented in Table 4.3 on page 44 specify that process validity should be preserved under transitions with certain inputs. Indirectly, these rules formulate requirements on the valid inputs. We gather the valid process inputs for a certain process state s in the set `valid_proc_inputs s`.

Finally, we define the successful execution of C0 processes as follows:

Definition 4.3 (Successful Execution of a C0 Process). We call a computation of a C0 process a *successful execution* iff

- all process inputs are valid with respect to the current process state,
- there is no runtime error in any state of the computation, and
- no inlined assembly statement is to be executed during the computation.

Formally, we specify the successful C0-process execution inductively over input sequences:

$$\begin{aligned} & \vdash_{C0}^{\text{proc}} s_{C0} \xrightarrow{[]} s'_{C0} \equiv s_{C0} = s'_{C0} \wedge \omega_{\text{proc}} s_{C0} \neq \text{RUNTIME_ERROR} \\ & \vdash_{C0}^{\text{proc}} s_{C0} \xrightarrow{(i \odot \text{is})} s'_{C0} \equiv \vdash_{C0}^{\text{proc}} \delta_{\text{proc}} i s_{C0} \xrightarrow{\text{is}} s'_{C0} \wedge i \in \text{valid_proc_inputs } s_{C0} \wedge \\ & \quad \omega_{\text{proc}} s_{C0} \neq \text{RUNTIME_ERROR} \wedge \neg \text{is_Asm } (\text{fst_stmt } [s_{C0}.\text{pstate}].\text{prog}) \end{aligned}$$

Conceptually, the successful execution of assembly processes is defined analogously. The sequential compiler correctness, however, additionally guarantees that the assembly code generated by the compilation of C0 programs does not modify itself. More formally, assembly code should neither write into the code range `crange` of the memory ($\neg \text{mem_write_inside_range } s_{\text{asm}} \text{ crange}$), nor should the program counters point outside this range (`inside_range crange sasm.dpc`).

Furthermore, there is a slight difference between C0 and assembly executions regarding runtime errors: A C0 computation is not a successful execution if the state s'_{C0} reached at the end features a runtime error while for successful assembly computations, the last state s'_{asm} is unconstrained. This difference stems from a different recognition of runtime errors. In C0, we recognize it after a transition by the invalid program state \perp , while in assembly, we employ predicates specifying whether an exception occurs during the next step. Thus, a runtime error diagnosed in the current assembly state occurs in the subsequent step of the C0 semantics. For the same reason, we do not check in assembly for runtime errors, illegal write accesses, and the range of the program counter iff the input is `ADD_MEMORY pages` or `FREE_MEMORY pages`.

Finally, we define the successful execution of assembly processes:

Definition 4.4 (Successful Execution of an Assembly Process). We call a computation of an assembly process a *successful execution* with respect to a code range `crange` iff

- all process inputs are valid with respect to the current process state and
- unless prior to an input `ADD_MEMORY pages` or `FREE_MEMORY pages`:
 - no runtime errors occur during a transition for the computation,
 - the memory in `crange` is not written, and
 - all instructions are only fetched from `crange`.

Formally, we specify the successful assembly-process execution inductively over input sequences:

$$\begin{aligned}
 & crange \vdash_{asm}^{proc} s_{asm} \xrightarrow{[]} s'_{asm} \equiv s_{asm} = s'_{asm} \\
 & crange \vdash_{asm}^{proc} s_{asm} \xrightarrow{(i \odot is)} s'_{asm} \equiv crange \vdash_{asm}^{proc} \delta_{proc} i s_{asm} \xrightarrow{is} s'_{asm} \wedge \\
 & i \in \text{valid_proc_inputs } s_{asm} \wedge \\
 & ((\forall \text{pages}. i \neq \text{ADD_MEMORY pages} \wedge i \neq \text{FREE_MEMORY pages}) \longrightarrow \\
 & \omega_{proc} s_{asm} \neq \text{RUNTIME_ERROR} \wedge \neg \text{mem_write_inside_range } s_{asm} crange \wedge \\
 & \text{inside_range } crange s_{asm}.dpc)
 \end{aligned}$$

In principle, we can formally state the process-simulation theorem based on these definitions. For convenience, however, we additionally define an abbreviation: Recall that the sequential simulation theorem might require several transitions in the assembly semantics for a single transition in the C0 semantics. With our notions of successful process execution, we reflect this circumstance by input sequences of different length. Although an assembly process might perform more transitions than a C0 process, we should certainly require that the actual kernel interaction, i. e., all inputs except for ε_Σ , remain equal. For this purpose, we define a normalization function $\gg is \ll$ over input sequences is that simply removes all empty input. Formally: $\gg is \ll \equiv [i \in is . i \neq \varepsilon_\Sigma]$

Theorem 4.3 (Process Simulation). *We assume that the parameters `img` and `pages` are well-formed wrt. `(is_init_proc_valid)` on page 44, there is an initial C0-process state s_{C0} satisfying these parameters, and a successful execution leads from s_{C0} using the input sequence is into the state s'_{C0} .*

If the above assumptions hold, there exists an input sequence is' , an allocation function `alloc`, and a final assembly state s'_{asm} such that

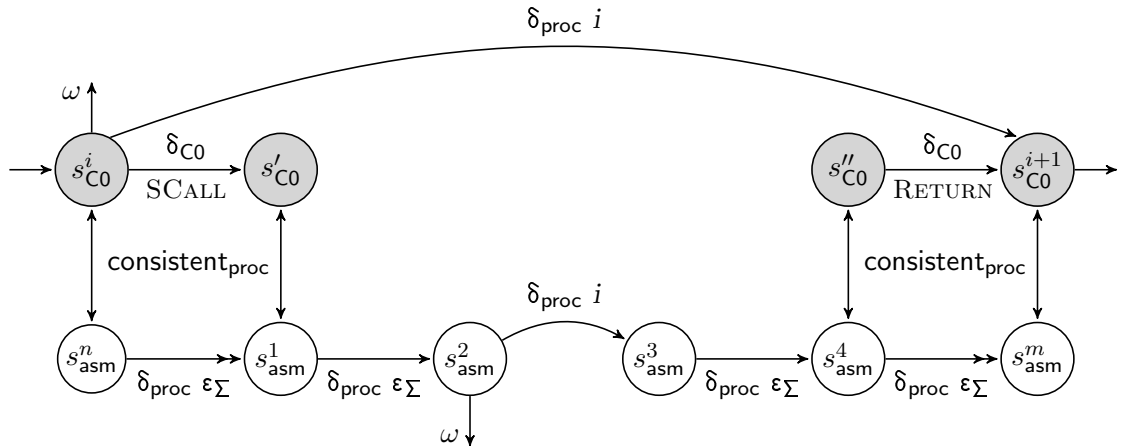
- the normalized input sequences $\gg is \ll$ and $\gg is' \ll$ are equal,
- starting in the initial assembly state that is uniquely identified by `img` and `pages`, the assembly process successfully advances under is' to s'_{asm} , and
- the final C0 state s'_{C0} simulates s'_{asm} under the allocation function `alloc`.

Formally:⁷

$$\begin{aligned}
 & \ll \forall d \in \text{set } img. -2^{31} \leq d < 2^{31}; 0 < \text{pages} \leq \text{TVM_MAXPAGES}; \\
 & \text{is_init_proc } img \text{ pages } s_{C0}; \vdash_{C0}^{proc} s_{C0} \xrightarrow{is} s'_{C0} \ll \\
 \implies & \exists is' \text{ alloc } s'_{asm}. \gg is \ll = \gg is' \ll \wedge \\
 & \text{code_range } s_{C0} \vdash_{asm}^{proc} \text{init_asm } img \text{ pages} \xrightarrow{is'} s'_{asm} \wedge \\
 & \text{consistent_proc } s'_{C0} \text{ alloc } s'_{asm}
 \end{aligned}$$

⁷Note that for succinctness, we overload the function `code_range` for C0 processes:

`code_range sC0 ≡ code_range sC0.typetab (gm_st [sC0.pstate].mem) sC0.funtab`

Figure 4.2: Verification scheme for the `vc_process_clone` function

Proof. The proof scheme for extended compiler correctness largely follows its sequential counterpart. We verify the theorem in two steps:

Using [Theorem 2.1](#) on page 23 (sequential compiler correctness), we can establish the sequential consistency relation for a successor of the initial assembly state, which is well-formed and its special-purpose registers have remained constant. Hence, the process-simulation relation can be established as well.

In a second step, we show inductively, that (a) the well-formedness and (b) the process-simulation relation are preserved under input sequences. In the induction step of this proof, we distinguish the possible process inputs. Well-formedness means the invariance of `is_validproc` under a transition with any valid input. Using [Theorem 4.1](#) on page 47 and [Theorem 4.2](#) on page 54, we can easily establish this claim for each valid input.

The second claim, that the process-simulation relation is preserved, is much more involved. For an empty input ϵ_Σ , we employ [Lemma 2.2](#) on page 24 (the sequential induction step). For the other inputs, we examine the implementation of the corresponding kernel-library functions.

[Figure 4.2](#) shows the case that the library function `vc_process_clone` is called in a C0-process state s_{C0}^i . From the induction hypothesis, we know that there exists a corresponding assembly state s_{asm}^n that satisfies the simulation relation `consistentproc`. Using the sequential C0 semantics, we execute the function-call statement. From the sequential compiler theorem, we know that the execution⁸ ($\delta_{proc} \epsilon_\Sigma$) of the corresponding, compiled code yields an assembly state s_{asm}^1 satisfying the simulation relation `consistentproc`.

Starting in this state, we execute the inlined assembly code (cf. [Figure 4.1](#) on page 48) of the function body. After a transition, the code reaches the trap instruction in state s_{asm}^2 . At this stage, the transition function δ_{proc} uses an input i from VAMOS to proceed to s_{asm}^3 . After a further step, we arrive at the end of the inlined assembly code. From the final assembly state s_{asm}^4 and the C0 state s'_{C0} immediately before the execution of

⁸Recall that a transition δ_{asm} is equal to $\delta_{proc} \epsilon_\Sigma$.

the inlined assembly statement, we construct the corresponding s''_{C0} immediately after the assembly statement. For this to work, we have to show that the assembly code did not disrupt the C0 execution environment (code, stack pointer, etc.). If the assembly code preserves the integrity of the execution environment, we can again establish the $\text{consistent}_{\text{proc}}$ relation.

In state s''_{C0} , we employ the sequential C0 semantics to execute the return statement and arrive in the state s^{i+1}_{C0} . From the compiler theorem, we know that there exists a corresponding assembly process such that the simulation relation holds. If the primitive implementation is correct, the state s^{i+1}_{C0} is equal to the state computed from state s^i_{C0} by δ_{proc} with the input i .

The proof for the other primitives proceeds similarly. Recall that our library is polymorphic. We have formalized the proof in Isabelle/HOL for 10 of the 12 fixed library functions and for 2 of the 5 function patterns that constitute the kernel library. \square

Proof Experiences and Challenges. When we applied [Lemma 2.2](#) on page 24 (the sequential induction step) for the case of an empty input ε_{Σ} , we identified a problem with arithmetic-overflow exceptions. Previously, compiler correctness had only been used for programs running in system mode. While overflow exceptions are turned off in system mode, they might occur in user mode. Upon our request, Dirk Leinenbach has strengthened his theorems accordingly. Furthermore, we could improve an earlier version of this lemma during our formal proof work and abandon or weaken some not strictly necessary preconditions (this thesis presents the improved version).

For reasoning about inlined VAMP assembly, we have been able to partially reuse previous work [[ST08](#)]. Note, however, that some generally helpful lemmata of this work, like the absence of exceptions during the execution of an inlined assembly statement, assume that only sequential assembly transitions are executed and formally break when we introduce our process-specific trap semantics.

Furthermore, the verification of inlined assembly within C0 processes is more complicated than within sequential C0 programs. One complication originates from the additional input of transitions and the necessity to show that this input is valid. Most notably, the empty input ε_{Σ} is only valid if the process output is ε_{Ω} , cf. ([no_input_valid](#)) on page 44. In this context, we deal with additional interrupt sources⁹ as well as with the memory consumption. The latter might be statically over-estimated to an upper bound (cf. [[Alk09](#), Section 3.3.8]) in the verification of sequential C0 functions with inlined assembly code.

Another obstacle is the often redundant error-checking in the sequential C0 transitions. In the C0 process semantics, we evaluate the expressions involved in a kernel call already when we compute the output for the current state. Hence it is often clear from the context that a memory update cannot fail. The sequential C0 transitions, however, check every time for failure and thus, we repeatedly provide evidence for its absence. A related problem is not process-specific: C0 transitions frequently check type correctness,

⁹Namely, arithmetic-overflow exceptions may occur in user mode while they are disabled in the system mode—the same reason why the original compiler-correctness theorem had been too weak.

which is always redundant because valid C0 programs are generally type-safe.

Fortunately, we could draw on synergies with a parallel proof development: Mareike Schmidt [DSS10] pioneered in verifying correctness for a similar library of the real-time kernel OLOS. Nevertheless, the verification effort for the correctness proof of our library is considerable because our library is much more extensive. The VAMOS library supports 16 kernel calls, which are implemented in 12 fixed functions and 5 function patterns, compared to 3 calls implemented in 3 fixed functions for OLOS. In addition, there remain some technical issues.

First, our kernel library supports generic types. Though this feature has previously been implemented with the C0 library of doubly-linked lists [Ngu05], we are the first to aim at the verification of generic C0 functions with inlined assembly. Additionally, the correctness proof of the list-manipulation functions formally depends (for historic reasons) on the particular type instance. Just recently, Schirmer & Wenzel [SW09] presented an improved representation for state spaces in Simpl to overcome this problem. For inlined assembly code, we unfortunately cannot use the Simpl framework. Still, we certainly rely on a generic proof for all possible type instances of the generic function patterns in the library.

Second, our library supports – in contrast to the OLOS library – the exchange of arbitrarily typed data. Hence, our library features special assembly code that sanitizes possibly invalid, received data. This assembly code comprises 32 instructions containing jumps and branches, and is thus comparatively complicated.

Third, a particularly complex problem concerns changes of larger memory chunks during inlined assembly computations. This problem has previously appeared in several case studies. Alkassar [Alk09, Section 3.3.9], for instance, complains about the memory representation in the C0 semantics, which scatters C0 values of an aggregate type into different values of a fundamental type. For the construction of the C0 state immediately after the assembly statement (denoted by s''_{C0} in Figure 4.2 on page 58), the changed memory addresses have to be related to C0 left-expressions. Due to the scattered memory representation, the current formalization requires that these left-expressions have to be of fundamental types. With aggregate types, we can deal by decomposition into its fundamental types. This approach, however, induced a considerable overhead during the verification of the OLOS library.¹⁰ Schmidt naturally focussed on the particular aggregate type, which is exchanged in OLOS, rather than developing a general, reusable technology that might systematically deal with aggregate types. Certainly, we cannot repeat the verification of IPC-`receive` and `-request` kernel calls for each type. We should rather generalize the existing formalization to support arbitrary types.

Existence of an Initial State. Recall that both, the sequential compiler-correctness theorem as well as the process-simulation theorem, rely on the existence of an initial assembly state representing the compiled C0 program. Below, we substantiate the validity of this assumption for processes. More precisely, the process-simulation theorem

¹⁰Unfortunately, the proof details do not fit into a conference paper. The interested reader might soon find them in Mareike Schmidt’s PhD thesis.

assumes the predicate $\text{is_init}_{\text{proc}} \text{img pages } s_{C0}$ to hold, where the parameters img and pages uniquely identify the initial assembly state.

In essence, $\text{is_init}_{\text{proc}}$ conjoins the predicates is_compilable , $\text{libvamos_compatible}$, and is_compilation (see Section 4.4). We inherit is_compilable from sequential compiler correctness. The predicate $\text{libvamos_compatible}$ constrains the definition of functions with particular names only if they are present in the function table. Thus, any C0 program can be made compatible with the kernel library by renaming its functions.

It just remains to show that a *suitable* memory image img can be constructed for each compilable program. Besides the predicate is_compilation , there are two indirect requirements on a suitable image: When an existing process issues a kernel call $\text{PROCESS_CREATE } \text{tsl prio img pages}$, the parameter img must be well-formed, see (img_create) on page 45. Furthermore, the kernel rejects the process creation if an image exceeds the memory size of the new process. The latter, however, is bounded by the constant TVM_MAXPAGES , see ($\text{is_init_proc_valid}$) on page 44. We state:

Lemma 4.4. *For each compilable program, there is a suitable memory image.*

$$\begin{aligned} &\text{is_compilable } tt \text{ ft } gst \implies \\ &\exists \text{img. is_compilation } tt \text{ ft } gst \text{ img} \wedge (\forall d \in \text{set } \text{img. } -2^{31} \leq d < 2^{31}) \wedge \\ &|\text{img}| \leq \text{TVM_MAXPAGES} \cdot \text{PAGE_SIZE} \end{aligned}$$

Proof. Using the conversion function to_int32 , we construct a witness img for the existential quantifier by converting the compiled program $\text{codegen_program } tt \text{ ft } gst$ – a list of assembly instructions – instruction-wise into an integer list. From this construction method, we can easily conclude that img stores the compiled program code, i. e., $\text{is_compilation } tt \text{ ft } gst \text{ img}$ holds. Furthermore, the range of the conversion function is a subset of the desired interval $\{-2^{31}..<2^{31}\}$. Finally, is_compilable comprises static resource restrictions, which especially limit the overall program size. Hence, the program code always fits into TVM_MAXPAGES memory pages. \square

With this insight, we can easily construct a suitable initial process:

Lemma 4.5 (Existence of an Initial C0-Process State). *For each compilable, libvamos-compatible C0 program, we can find suitable parameters img and pages for the construction of an initial C0-process state. Formally:*

$$\begin{aligned} &\llbracket \text{is_compilable } tt \text{ pt } gst; \text{libvamos_compatible } tt \text{ pt } gst \rrbracket \\ &\implies \exists \text{img pages. } (\forall d \in \text{set } \text{img. } -2^{31} \leq d < 2^{31}) \wedge 0 < \text{pages} \leq \text{TVM_MAXPAGES} \wedge \\ &\quad \text{is_init}_{\text{proc}} \text{img pages} \\ &\quad (\text{pstate} = \llbracket \text{init_C0 } pt \text{ gst} \rrbracket, \text{typetab} = tt, \text{funtab} = pt, \text{mmsize} = \text{pages}) \end{aligned}$$

Proof. Using Lemma 4.4, the statement follows after unfolding the definition of $\text{is_init}_{\text{proc}}$ on page 51. \square

Possible Further Extensions. We like to point out a limitation of the present simulation theorems: When the C0 execution finishes, i. e., the remaining program comprises a

single SKIP statement, the program state reaches a fixpoint. In this case, the present simulation relation consistent is weak, e. g., the program counters of the corresponding assembly state are unconstrained. Consequently, we can, for example, not conclude from the successful execution of a C0 process, that a simulated assembly process cannot fail. The assembly process successfully executes until the remaining C0 program is empty – but then, it might crash.

It is possible to strengthen the simulation relation and require in this case that the program counters point to the beginning of a special exit code. Certainly, this exit code is platform-specific. It can be supplied to the compiler as an additional parameter similar to, e. g., the program base. With these prerequisites in place, we could extend the C0-process model and the process-simulation theorem to the correct termination of processes.

4.6 Conclusion

Our process abstraction is an important witness on how the coordinated effort of pervasive system verification differs from a verification approach that considers only a single layer at a time. Strictly speaking, the encapsulation of processes into a self-contained entity slightly complicates the code verification [DDW09, Sect. 6]. At the same time, it is a necessary precondition for the abstraction of processes.

If only the implementation of the kernel is verified, the correctness statement might focus on the kernel manipulations of the user-visible processor parts. More specifically, this approach directly identifies the process state by the processor state as far as visible from the user mode. Transitions of the kernel specification might even completely abstract from the semantics of user-mode instructions and represent an arbitrary execution sequence of instructions in the user mode as a single, non-deterministic state update.¹¹ For pervasive verification, in contrast, the exact semantics of the user-mode instructions is crucial because it is the foundation for the verification of application programs.

When processes are specified as self-contained entities, the general process abstraction should be carefully crafted such that it fits different languages. Recall that a memory image uniquely defines the initial state of an assembly process but not the one of a C0 process. In fact, our first general process model assumed a uniquely defined initial state and had to be changed for the introduction of C0 processes.

¹¹In a private conversation, Gerwin Klein reported that in the seL4 project [KEH⁺09], user-mode instructions are indeed specified by non-deterministic state updates.

5 Adding Concurrency: The Kernel Models

Das also war des Pudels Kern!

Johann Wolfgang von Goethe, in: "Faust I"

Contents

5.1 Overview of the Model Components	64
5.2 A Simple Kernel Call: <code>SET_PRIVILEGED</code> Revisited	67
5.3 Process Termination	68
5.4 Inter-Process Communication	71
5.4.1 IPC Rights Management	71
5.4.2 Specifying IPC in the VAMOS Model	72
5.4.3 IPC in CoUP	76
5.5 The Scheduler Specification	78
5.6 Invariants over Execution Traces	81
5.7 Conclusion	85

In the previous chapter, we have described our process abstraction. Now, we embed the processes into two concurrent kernel models, the VAMOS specification, and the CoUP model. The former specifies the exact behavior of our microkernel with a particular scheduler. In an ongoing refinement proof, this specification is linked to the actual C0 implementation [DDW09, Sect. 6]. When this verification effort succeeds, we know that the specification indeed simulates the implementation of the kernel on the hardware.

The latter model abstracts the scheduler and focusses on the interaction of the processes with the microkernel. We employ this more abstract model in order to describe the reordering of interleaved sequences and to introduce C0 processes.

Both models are Moore machines. We formally define the kernel specification with the tuple $\mathcal{A}_V = (\mathcal{S}_V, \mathcal{S}_V^0, \hat{\Sigma}, \hat{\Omega}, \omega_V, \delta_V)$ and the CoUP model with $\mathcal{A}_{VC} = (\mathcal{S}_{VC}, \mathcal{S}_{VC}^0, \hat{\Sigma}, \hat{\Omega}, \omega_{VC}, \Delta_{VC})$. The state spaces \mathcal{S}_V and \mathcal{S}_{VC} are supersets of the initial-state sets \mathcal{S}_V^0 and \mathcal{S}_{VC}^0 , respectively. For the communication with the peripheral devices, we use the input alphabet $\hat{\Sigma}$ and the output alphabet $\hat{\Omega}$. The functions ω_V and ω_{VC} determine the output in a current state. Finally, δ_V and Δ_{VC} describe the transitions of the models. A notable difference between these machines regards the determinism: While the VAMOS specification is fully deterministic, CoUP features a non-determinism in its scheduling decisions. Consequently, the transition relation δ_V is functional while Δ_{VC} is not. Besides that, the VAMOS specification and the CoUP model mainly differ in the process abstraction.

Certainly, it is desirable to share common concepts, which minimizes the specification work as well as the need for equivalence proofs. A vital role for reusable specifications has been our process abstraction: It proved to be beneficial that many definitions for the VAMOS specification have been generalized for arbitrary process abstractions (as described in [Section 4.1](#)) and thus, can be reused in CoUP as well. From this perspective, the generalization of the whole CoUP model to arbitrary process abstractions is just a logical continuation because it paves the way for the integration of other language semantics.

Below, we describe the different components of the models side by side. A complete formal definition of both models, however, is beyond the scope of this thesis. Thus, we content ourselves with exemplary insights that become relevant in proofs of the remaining document: Our main focus regards the differences in the formalization of both models with an emphasis on reusable specification functions for similar parts. These differences concern us in the simulation proof between VAMOS and CoUP in [Chapter 6](#). Furthermore, we describe the scheduler specification, which the fairness proof in [Chapter 7](#) relies on. Finally, we formulate and prove an invariant on the execution traces of the VAMOS specification, which is used in both proofs.

Credits and Own Contribution. Jan Dörrenbächer has developed an initial kernel specification before the process abstraction had been invented; he has generalized the initial specification to arbitrary process models and continuously maintained the VAMOS specification. Furthermore, he has formulated an initial VAMOS invariant and proved its invariance for a few exemplary cases.

For the general idea of a computational VAMOS model with C0 processes, I am indebted to Eyad Alkassar [[Alk05](#)]. The CoUP model as such, however, is my own contribution. A joint effort of Jan Dörrenbächer and me have been adaptations to many VAMOS specification functions improving its reusability. During my verification work, I have fixed many flaws and oversights in both models, on the one hand, and considerably strengthened the VAMOS invariant, on the other hand. Furthermore, I showed the actual invariance of the formulated VAMOS invariant.

5.1 Overview of the Model Components

Communication Alphabets. Our kernel uses memory-mapped I/O for device communication. Hence, the output alphabet $\hat{\Omega}$ comprises read and write accesses to device addresses. The input alphabet $\hat{\Sigma}$ is a set of pairs, consisting of the raised interrupt lines (a set *ints* of device identifiers) and incoming data (a list *mifo* of 32-bit integers that have been read from a device). Hillebrand *et al.* [[HIP05](#), [AH08](#)] have described the device interface in detail.

State Spaces. A state $s_V \in \mathcal{S}_V$ comprises the following components:

$s_V.\text{procs}$ is a partial function that maps process identifiers (PIDs) to their assembly states s_{asm} . For inactive processes, this function is undefined.

`sv.priodb` maps the PIDs of the active processes to their priorities.

`sv.schedds` contains the scheduling data structures like wait and ready queues as well as process-specific accounting information like the length of the timeslice. In [Section 5.5](#), we discuss the details of the scheduler.

`sv.rightsdb` comprises the access-control rights. More specifically, this component maps PIDs to information for the IPC rights management (see [Section 5.4](#)), on the one hand, and a Boolean flag determining whether the corresponding process is privileged, on the other hand.

`sv.sndstatdb` stores the so-called send status, a remnant of the process status in the implementation. Usually, the membership in a ready or wait queue 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. In this case, the send status `True` specifies that the process has successfully sent and now waits for a reply.

`sv.devds` contains data for device communication.

At first sight, the separation of priorities from the remaining scheduling data might surprise. The reason for this partitioning is that CoUP abstracts from the particular scheduling algorithm, but the priorities remain visible. We modularized the states and the corresponding transition functions in order to share the definition of similar parts between the VAMOS specification, CoUP, and the SOS model [[Bog08](#), §4].

For scheduling, VAMOS maintains different process queues, which are represented as PID lists in the specification. In particular, there is a ready queue `sv.schedds.ready prio` of schedulable processes for each priority $prio \in \{\text{LOW}, \text{MEDIUM}, \text{HIGH}\}$. 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:

```
cupV sv.schedds ≡
  case concat (map sv.schedds.ready [HIGH, MEDIUM, LOW]) of [] ⇒ ⊥ | x ⊙ xs ⇒ [x]
```

The corresponding CoUP state s_{VC} inherits most components of s_V . Only two components change: The process abstraction becomes a model parameter, i. e., component `svC.procs` of a CoUP state s_{VC} is a partial function from PIDs to the generic state space $\mathcal{S}_{\text{proc}}$ of an arbitrary, valid process model. Moreover, the scheduling data structures are replaced by a current-process indicator. We retain the current process in the state in order to compute the output from the current state. The output function ω_{VC} signals the demand for device communication *in the next step*. To determine this demand, we employ the output function ω_{proc} of the current process. Consequently, we need to fix the current process in the CoUP state.

Transitions. The transition function δ_V takes two parameters: (a) an input $(ints, mifo)$ from the device subsystem, and (b) a VAMOS state s_V . It returns a successor state s'_V . Within a transition, we distinguish up to three phases:

1. If there is a current process p_{cup} such that $cup_V s_V.schedds = \lfloor p_{cup} \rfloor$, we consult its output $\omega_{proc} \lfloor s_V.procs p_{cup} \rfloor$ and compute the response according to the current VAMOS state. For instance, if a process calls `DEV_READ`, we check for sufficient privileges and choose the corresponding response res for success or failure. Upon success, res comprises the device data $mifo$, and otherwise an error value. With this response as input, we advance the current process by calling its transition function: $s_V(\lfloor procs := s_V.procs(p_{cup} \mapsto \delta_{proc} res \lfloor s_V.procs p_{cup} \rfloor) \rfloor)$
2. If the timer-interrupt line is raised, formally determined by the predicate `is_timer_on` $(ints, mifo)$, the scheduler is called. The scheduler charges the currently running process for its computation and possibly elects another process to compute. Furthermore, it checks whether timeouts of waiting processes have elapsed and wakes up the corresponding processes. We defer the detailed specification to [Section 5.5](#).
3. Finally, VAMOS delivers the occurred interrupts to the attached drivers (assuming that the latter are waiting; otherwise, the interrupts are saved in $s_V.devds$ for a later delivery).

The CoUP transitions behave very similarly. A transition $s_V \xrightarrow{(ints, mifo)}_{VC} s'_V$ in $\Delta_{VC} ints mifo$, however, obtains p_{cup} directly from $s_V.cup$. Moreover, the effects of the second phase become non-deterministic: An arbitrary, active process (or none at all) is elected to compute in the next cycle, and some of the possibly waiting processes with finite timeouts are woken up. Both effects are specified independently from the timer interrupt. Formally, we split the second phase into two and then specify each transition phase in its own relation:

$\Delta_{VC}^{cup} mifo$ advances the current process using the input $mifo$ from the devices,
 Δ_{VC}^{sched} elects the next current process,
 Δ_{VC}^{to} wakes up waiting processes, and
 Δ_{VC}^{ints} $ints$ delivers the interrupts in $ints$.

The transition relation Δ_{VC} is now simply the composition of these four relations. Note, however, that when processes are woken up in the second or third phase of δ_V , a former empty ready queue might become non-empty such that the next current process changes again. In the CoUP model, there are no ready queues and the current process is solely changed by Δ_{VC}^{sched} . Hence, we place this relation at the very end of the composition:

$$\Delta_{VC} ints mifo \equiv \Delta_{VC}^{cup} mifo \circ \Delta_{VC}^{to} \circ \Delta_{VC}^{ints} ints \circ \Delta_{VC}^{sched}$$

where the relational composition $R \circ S$ of two relations R and S is defined as $\{(r, s). \exists t. (r, t) \in R \wedge (t, s) \in S\}$.

Thus, the major specification work is encapsulated in the sub relations. The first phase is the most elaborate one. We define the relation $\Delta_{VC}^{\text{cup}} \text{ mifo}$ by 33 introduction rules.¹ The majority of these rules specify the semantics of kernel calls. These rules mostly follow a schematic pattern while reusing definitions from the VAMOS specification. The next sections explain this pattern by the examples of the SET_PRIVILEGED and the PROCESS_KILL calls. In Section 5.4, we show the specification of IPC as an interesting exception to this general pattern. Furthermore, there are three rules not belonging to kernel calls. The most prominent one is an internal transition if the current process does not signal any output. Formally:

$$\begin{aligned} & \llbracket s_{VC}.cup = \lfloor p_{cup} \rfloor; \omega_{proc} \llbracket s_{VC}.procs \ p_{cup} \rrbracket = \varepsilon_{\Omega} \rrbracket \\ \implies & (s_{VC}, s_{VC}(\lfloor procs := s_{VC}.procs(p_{cup} \mapsto \delta_{proc} \ \varepsilon_{\Sigma} \llbracket s_{VC}.procs \ p_{cup} \rrbracket \rfloor))) \\ & \in \Delta_{VC}^{\text{cup}} \text{ mifo} \end{aligned} \quad (\text{internal})$$

Another rule specifies the lack of any progress, e. g., because there is no current process or the process issues an IPC call and waits for a suitable communication partner. We specify:

$$(s, s) \in \Delta_{VC}^{\text{cup}} \text{ mifo} \quad (\text{reflexivity})$$

Finally, a process might experience an internal runtime error. This case is treated as if the process would have killed itself (see Section 5.3).

5.2 A Simple Kernel Call: SET_PRIVILEGED Revisited

Recall the specification of the SET_PRIVILEGED call of VAMOS in Section 4.1. In essence, we distinguish unsuccessful cases, on the one hand, where only the caller is notified about the reason leading to the failure, and a successful case, on the other hand, where internal kernel data structures are changed in addition to the caller notification. Hence, we may slightly rephrase the earlier specification to:

```
set_privileges_specV sV hn ≡
  let pcup = ⌊cupV sV.schedds⌋; ptarget = ⌊sV.rightsdb pcup⌋.hdb hn;
      res = set_privileged_response sV.rightsdb pcup hn
  in if is_error res then sV(⌊procs := sV.procs(pcup ↦ δproc res ⌊sV.procs pcup⌋⌋)
    else sV(⌊procs := sV.procs(pcup ↦ δproc SUCCESS ⌊sV.procs pcup⌋⌋),
            rightsdb := add_privilege sV.rightsdb ptarget)
```

where the function `set_privileged_response` computes the response to the caller from the current kernel state:

```
set_privileged_response rightsdb pcup hn ≡
  if ¬ privileged rightsdb pcup then ERR_UNPRIVILEGED
  else if ¬ valid_handle rightsdb pcup hn then ERR_INVALID_HANDLE else SUCCESS
```

¹The term *introduction rule* originates from natural deduction [Gen35]. It refers to an inference rule with a predicate in the conclusion. Thus, the predicate can be “introduced” by forward reasoning. You may perceive these rules as an inductive definition of the predicate. We regard $x \in \Delta_{VC}^{\text{cup}} \text{ mifo}$ as such a predicate and define $\Delta_{VC}^{\text{cup}} \text{ mifo}$ by rules of the form $P \ x \implies x \in \Delta_{VC}^{\text{cup}} \text{ mifo}$.

The VAMOS specification defines similar functions for each kernel call, which are then reused in the specification of CoUP.

Note that the above call specification has an implicit assumption: The current process is defined and issues a SET_PRIVILEGED call with the function's argument hn as parameter, i. e.,

$$\exists p_{\text{cup}}. \text{cup}_V \text{ sVC.schedds} = \lfloor p_{\text{cup}} \rfloor \wedge \omega_{\text{proc}} \lceil \text{sVC.procs } p_{\text{cup}} \rceil = \text{SET_PRIVILEGED } hn$$

In the VAMOS specification, the central function `call_dispatcher_specV` serves as dispatcher between the individual kernel-call specifications ensuring this assumption. In CoUP, however, all assumptions are made explicit in the introduction rules. We specify the unsuccessful cases by

$$\begin{aligned} & \llbracket \text{sVC.cup} = \lfloor p_{\text{cup}} \rfloor; \omega_{\text{proc}} \lceil \text{sVC.procs } p_{\text{cup}} \rceil = \text{SET_PRIVILEGED } hn; \\ & \text{res} = \text{set_privileged_response } \text{sVC.rightsdb } p_{\text{cup}} \text{ } hn; \text{is_error } \text{res} \rrbracket \\ \implies & (\text{sVC}, \text{sVC}(\lfloor \text{procs} := \text{sVC.procs}(p_{\text{cup}} \mapsto \delta_{\text{proc}} \text{res } \lceil \text{sVC.procs } p_{\text{cup}} \rceil) \rfloor)) \\ & \in \Delta_{\text{VC}}^{\text{cup}} \text{ mifo} \end{aligned} \quad (\text{privilege_err})$$

and the successful case by

$$\begin{aligned} & \llbracket \text{sVC.cup} = \lfloor p_{\text{cup}} \rfloor; \omega_{\text{proc}} \lceil \text{sVC.procs } p_{\text{cup}} \rceil = \text{SET_PRIVILEGED } hn; \\ & \neg \text{is_error}(\text{set_privileged_response } \text{sVC.rightsdb } p_{\text{cup}} \text{ } hn); \\ & p_{\text{target}} = \lceil \text{sVC.rightsdb } p_{\text{cup}} \rceil.\text{hdb } hn \rrbracket \\ \implies & (\text{sVC}, \\ & \text{sVC}(\lfloor \text{procs} := \text{sVC.procs}(p_{\text{cup}} \mapsto \delta_{\text{proc}} \text{SUCCESS } \lceil \text{sVC.procs } p_{\text{cup}} \rceil), \\ & \text{rightsdb} := \text{add_privilege } \text{sVC.rightsdb } p_{\text{target}} \rfloor)) \\ & \in \Delta_{\text{VC}}^{\text{cup}} \text{ mifo} \end{aligned} \quad (\text{privilege_succ})$$

Note that the rules introduce abbreviations like p_{cup} or res in form of free variables as opposed to the let expressions employed in VAMOS functions.

Most kernel calls in CoUP are specified following exactly this pattern. For a few calls, there is just a single rule, because even the successful case only changes processes.

5.3 Process Termination

A process might cease to exist in two situations: Either, it signals an internal runtime error as output to the kernel, or a process explicitly invokes the PROCESS_KILL call. In the former case, the kernel immediately terminates the faulty process. In the latter, the caller specifies the victim by a process handle. Only a privileged caller is permitted to terminate another active process. Consequently, there are two unsuccessful cases of a PROCESS_KILL call: missing privileges and an invalid handle. We specify the kernel's response to the calling process by:

```
process_kill_response rightsdb p_cup hn ≡
  if hn ≠ HN_SELF ∧ ¬ privileged rightsdb p_cup then ERR_UNPRIVILEGED
  else if ¬ valid_handle rightsdb p_cup hn then ERR_INVALID_HANDLE else SUCCESS
```


Note that suicide is permitted without privileges because the process could alternatively provoke a runtime error. In fact, the transition for runtime errors in the VAMOS specification is defined as a special case of process kill:

$$\omega_{\text{proc}} \lceil s_V.\text{procs} \lceil \text{cup}_V s_V.\text{schedds} \rceil \rceil = \text{RUNTIME_ERROR} \implies \\ \text{call_dispatcher_spec}_V s_V \text{ mifo} = \text{process_kill_spec}_V s_V \text{ HN_SELF}$$

where the actual effect of `PROCESS_KILL` is as usually encapsulated in a function:

```
process_kill_spec_V s_V hn ≡
  let p_cup = ⌈cup_V s_V.schedds⌉; p_victim = ⌈s_V.rightsdb p_cup⌉.hdb hn;
      res = process_kill_response s_V.rightsdb p_cup hn
  in if is_error res then s_V(⌈procs := s_V.procs(p_cup ↦ δ_proc res ⌈s_V.procs p_cup⌉)⌉)
     else s_V(⌈procs := v_kill_updt_procs s_V.procs s_V.rightsdb s_V.devds
              (set s_V.schedds.wait) p_cup ⌈p_victim⌉,
              schedds := v_kill_updt_schedds s_V.schedds s_V.procs s_V.priodb s_V.rightsdb
                          ⌈p_victim⌉,
              priodb := s_V.priodb(⌈p_victim⌉ := ⊥),
              sndstatdb := v_kill_updt_sndstatdb s_V.sndstatdb s_V.procs s_V.rightsdb
                          ⌈p_victim⌉,
              rightsdb := v_kill_updt_rightsdb s_V.rightsdb ⌈p_victim⌉,
              devds := v_kill_updt_devds s_V.devds ⌈p_victim⌉)⌉)
```

The number of effected components is an indicator for the complexity of this specification. The update of component `priodb` is stated directly because it is sufficiently simple. The remaining updates are more elaborate and thus, encapsulated in extra functions.

The termination of a process concerns, for instance, processes that await an IPC operation. Furthermore, processes might have a handle to the victim, which becomes invalid through the termination. If a process currently accepts kernel notifications, the death notification is delivered immediately. We only provide the definition of `v_kill_updt_procs` as an example:

```
v_kill_updt_procs procs rightsdb devds waits p_cup p_victim ≡
  λx. if x = p_victim then ⊥
      else if x = p_cup then ⌈δ_proc SUCCESS ⌈procs p_cup⌉⌉
          else if x ∈ waits ∧ sends_to procs rightsdb p_victim x
              then ⌈δ_proc ERR_SND_INVALID_HANDLE ⌈procs x⌉⌉
          else if x ∈ waits ∧ conveys_hn_of procs rightsdb p_victim x
              then ⌈δ_proc ERR_INVALID_HANDLE ⌈procs x⌉⌉
          else if x ∈ waits ∧ receives_from procs rightsdb p_victim x
              then ⌈δ_proc ERR_RCV_INVALID_HANDLE ⌈procs x⌉⌉
          else if x ∈ waits ∧ accepts_knotes procs rightsdb p_victim x
              then ⌈δ_proc
                  (deliver_knotes
                   (v_kill_updt_rightsdb rightsdb p_victim) devds
                   devds.saved x)⌉
```

$$\begin{aligned} & \llbracket \text{procs } x \rrbracket \\ \text{else } & \text{procs } x \end{aligned}$$

where the predicate `sends_to procs rightsdb p_{victim} p_{snd}` holds iff p_{snd} aspires to send a message to p_{victim} , the predicate `conveys_hn_of procs rightsdb p_{victim} p_{snd}` holds iff p_{snd} aspires to send a process handle of p_{victim} , the predicate `receives_from procs rightsdb p_{victim} p_{rcv}` holds iff p_{rcv} aspires to receive a message from p_{victim} , and the predicate `accepts_knotes procs rightsdb p_{victim} p_{rcv}` holds iff p_{rcv} currently accepts kernel notifications and has a handle to p_{victim} .

Note that `v_kill_updt_procs` does not take the whole scheduling data as parameter (like done with all other components) but only the set of waiting processes. Thus, we can reuse the function in CoUP, which does not maintain scheduling data.

For CoUP, we specify the process termination in a function `vc_terminate_proc` analogously to `process_kill_specv` for the VAMOS specification. More specifically, the function `vc_terminate_proc waits p_{victim} p_{cup} s_{vc}` specifies the termination of process p_{victim} induced by p_{cup} with a set `waits` of waiting processes in state s_{vc} as follows:

$$\begin{aligned} \text{vc_terminate_proc } \text{waits } p_{\text{victim}} p_{\text{cup}} s_{\text{vc}} \equiv & \\ s_{\text{vc}}(\text{procs } := & \text{v_kill_updt_procs } s_{\text{vc}}.\text{procs } s_{\text{vc}}.\text{rightsdb } s_{\text{vc}}.\text{devds } \text{waits } p_{\text{cup}} p_{\text{victim}}, \\ \text{priodb } := & s_{\text{vc}}.\text{priodb}(p_{\text{victim}} := \perp), \\ \text{sndstatdb } := & \text{v_kill_updt_sndstatdb } s_{\text{vc}}.\text{sndstatdb } s_{\text{vc}}.\text{procs } s_{\text{vc}}.\text{rightsdb } p_{\text{victim}}, \\ \text{rightsdb } := & \text{v_kill_updt_rightsdb } s_{\text{vc}}.\text{rightsdb } p_{\text{victim}}, \\ \text{devds } := & \text{v_kill_updt_devds } s_{\text{vc}}.\text{devds } p_{\text{victim}}) \end{aligned}$$

In contrast to `process_kill_specv`, the function `vc_terminate_proc` does not treat any error cases but implicitly assumes a successful `PROCESS_KILL` call. Note the variable `waits`, which is the remnant of VAMOS' wait queue. As we have abandoned this queue in CoUP, we choose non-deterministically whether the kernel should regard a process to be waiting.

We constrain `waits` from both sides: On the one hand, we are certain that all processes p wait, which have completed only the send phase of a request, i. e., $s_{\text{vc}}.\text{sndstatdb } p = \llbracket \text{True} \rrbracket$. On the other hand, a process might only wait for IPC, i. e., its output should be one of `IPC_SEND`, `IPC_RECEIVE`, or `IPC_REQUEST`. Furthermore, error conditions like valid handles are checked before the process is enqueued in the wait queue—a process cannot be waiting in case of such errors. The predicate `vc_pending_ipc` over-estimates the set of waiting processes.

Within these boundaries, all values of `waits` are possible. If a process is in `waits`, function `vc_terminate_proc` notifies the process about a possibly vanished handle. Otherwise, the process learns about the vanished handle as soon as it is elected as the current process.

We specify the two possibilities of process termination, a successful `PROCESS_KILL` call and a runtime error, by the following rules:

$$\begin{aligned} \llbracket s_{\text{vc}}.\text{cup} = \llbracket p_{\text{cup}} \rrbracket; \omega_{\text{proc}} \llbracket s_{\text{vc}}.\text{procs } p_{\text{cup}} \rrbracket = \text{PROCESS_KILL } hn; \\ \{p. s_{\text{vc}}.\text{sndstatdb } p = \llbracket \text{True} \rrbracket\} \subseteq \text{waits}; \end{aligned}$$

$$\begin{aligned}
& \text{waits} \subseteq \{p. \text{vc_pending_ipc } s_{VC}. \text{procs } s_{VC}. \text{sndstatdb } s_{VC}. \text{rightsdb } s_{VC}. \text{devds } p\}; \\
& \neg \text{is_error } (\text{process_kill_response } s_{VC}. \text{rightsdb } p_{\text{cup}} \text{ hn}); \\
& p_{\text{victim}} = \llbracket [s_{VC}. \text{rightsdb } p_{\text{cup}}]. \text{hdb } \text{hn} \rrbracket \\
& \implies (s_{VC}, \text{vc_terminate_proc } \text{waits } p_{\text{victim}} p_{\text{cup}} s_{VC}) \in \Delta_{VC}^{\text{cup}} \text{ mifo} \quad (\text{pkill_succ}) \\
& \llbracket [s_{VC}. \text{cup} = [p_{\text{cup}}]; \omega_{\text{proc}} [s_{VC}. \text{procs } p_{\text{cup}}] = \text{RUNTIME_ERROR}; \\
& \{p. s_{VC}. \text{sndstatdb } p = [\text{True}]\} \subseteq \text{waits}; \\
& \text{waits} \subseteq \{p. \text{vc_pending_ipc } s_{VC}. \text{procs } s_{VC}. \text{sndstatdb } s_{VC}. \text{rightsdb } s_{VC}. \text{devds } p\} \rrbracket \\
& \implies (s_{VC}, \text{vc_terminate_proc } \text{waits } p_{\text{cup}} p_{\text{cup}} s_{VC}) \in \Delta_{VC}^{\text{cup}} \text{ mifo} \quad (\text{exception})
\end{aligned}$$

5.4 Inter-Process Communication

This section describes the specification of the kernel's mechanism for synchronous message passing between processes. It is divided into three parts. The first one presents data structures especially introduced for IPC, namely, the component `rightsdb` containing the access-control rights. This component is part of the CoUP states as well as of the VAMOS specification's states. The second part explains in detail how the IPC calls are specified in VAMOS. The last part elaborates on how the specification of the same calls in CoUP benefits from the abandoned scheduling data structures.

5.4.1 IPC Rights Management

The largest part of the access-control rights belongs to the IPC rights management. [Section 3.2.1](#) briefly introduced the capability-like access control for IPC. The basic concept are handles with assigned IPC rights. In short, a handle is a process-local name for a particular process. For each active process, the kernel stores a mapping from handles to the kernel-global process identifiers together with a set of rights that constrain the ability to send to the identified process.

In particular, the component `s.rightsdb` stores for each active process `pid` a record with the following components:

hdb is the handle database, a partial function mapping handles to process identifiers. Basically, handles are positive numbers. A few distinguished handles have special semantics. First, `HN_SELF` is a self-reference, i. e., $[s.\text{rightsdb } pid].\text{hdb } \text{HN_SELF} = [pid]$. Second, `HN_KERNEL` refers to the kernel, e. g., for limiting an `IPC_RECEIVE` call to kernel notifications. Third, `HN_NONE` refers to no process. This handle is used for a so-called *open receive*, i. e., an `IPC_RECEIVE` call not limited to a particular process. Last, a newly created process does not yet know other processes but may refer to its creator using `HN_CREATOR`.

creator if `pid` has been launched by a `PROCESS_CREATE` call, this field stores the PID of the caller. A cloned process inherits the creator of its original. The creator of the initial process is undefined.

irdb is the IPC-rights database, assigning a set of rights to process identifiers. The possible rights are `REQUESTRIGHT`, `SENDRIGHT`, `FINITERIGHT`, and `MULTIPLERIGHT`.

stales is a set of *stale handles*, i. e., handles that have once pointed to meanwhile terminated processes. Note that stale handles can no longer be used even if a process later inherits the same identifier. The kernel notifies processes of their non-empty stale set whenever they accept kernel notifications.

5.4.2 Specifying IPC in the VAMOS Model

The distinctive feature of IPC is synchronization. While all other kernel calls are necessarily handled immediately when issued, a process might choose to wait until an appropriate IPC partner becomes available. More precisely, a message transfer is only possible if one of the communication partners has waited. For the actual message transfer, however, it is not relevant whether the sender or the receiver has been waiting. Hence, we specify message transfers independently from process states (current or waiting).

We encapsulate the specification in the per-component update functions. Most of these functions are technically quite involved and their formal definition is irrelevant for the simulation proof between the kernel models in [Chapter 6](#), simply because we use these functions in both models. Hence, we content ourselves with an informal description of these functions.

Function `v_ipc_trans_updt_procs` computes the new state of component `procs`. More specifically, it determines the according kernel response for the receiver (including process handles, IPC-right sets, the conveyed message, and the kernel notification bits), advances the receiving process with this response, and unless the sending process has called for an `IPC_REQUEST`, it advances the latter with `SUCCESS`.

Function `v_wkp_updt_schedds` updates the scheduling data structures `schedds`. More specifically, all processes that have completed their IPC operation, are removed from the wait queue and re-added to the according ready queue. Note that for a completed send phase of an `IPC_REQUEST` call, there are additional actions to be taken: If the sender has been the current process, it should be scheduled to wait for a reply, otherwise, its scheduling timeout should be adjusted according to the specified receive timeout.

Furthermore, function `v_ipc_trans_updt_sndstatdb` computes the new state of the component `sndstatdb`. Its formal definition is quite simple:

$$\begin{aligned} \text{v_ipc_trans_updt_sndstatdb } \textit{sndstatdb } p_{\text{snd}} p_{\text{rcv}} \textit{is_request} \equiv \\ \textit{sndstatdb}(p_{\text{snd}} \mapsto \textit{is_request}, p_{\text{rcv}} \mapsto \text{False}) \end{aligned}$$

The send status of the sender p_{snd} is set to the value of variable *is_request*, which determines whether the sender has initiated an `IPC_REQUEST` call. Additionally, the receiver's send status is set to `False` because its IPC operation completes.

Analogously, we define the functions `v_ipc_trans_updt_rightsdb`, which updates `rightsdb` with respect to the IPC rights that the sender might have granted to the receiver, and `v_ipc_trans_updt_devds`, which updates `devds` if the kernel has notified the receiver about pending interrupts.

Note that we exclude several faults in the call specification before a message transfer can actually take place. First, we guard against faults that are independent from a possibly available communication partner—a missing IPC right, for instance, or a misaligned

buffer. We call such faults *invocation errors*. Second, we distinguish whether a suitable communication partner is already waiting. We refer to this situation as a *rendezvous*.

If there is no rendezvous, we regard the timeout of the considered IPC call. A timeout of zero means that the process is not willing to wait. In this case, the kernel cancels the IPC call immediately with a timeout error (ERR_SND_TIMEOUT or ERR_RCV_TIMEOUT). Otherwise, the calling process is moved to the wait queue aspiring a later message transfer. The latter operation is encapsulated in function `v_ipc_wait_updt_schedds`.

Furthermore, there might be a suitable communication partner but the designated sender intends to transfer a message that exceeds the receiver's provided buffer. In this case, the sender is informed about this problem (ERR_MSG_OVERSIZED). Otherwise, the message transfer takes place.

For the IPC_SEND call, we define function `ipc_send_err` that guards against all faults, including invocation errors, immediate timeouts, and oversized messages. In contrast to previous call specifications, we distinguish three cases, here: An error response, the movement of the process to the wait queue, and immediate message transfer. As a special convention, the function returns ε_Σ for the second and SUCCESS for the third case. Formally, we define:

```
ipc_send_err procs schedds sndstatdb rightsdb hn_rcv rights_snd msg hn_add rights_add
  timeout_snd  $\equiv$ 
  let  $p_{\text{snd}} = \lceil \text{cup}_V \text{ schedds} \rceil$ ;
      res = ipc_send_invoc_err rightsdb  $p_{\text{snd}}$  hn_rcv rights_snd msg hn_add rights_add
  in if is_error res then res
      else if ipc_send_rendez_vous procs schedds sndstatdb rightsdb  $p_{\text{snd}}$  hn_rcv
           then if ipc_send_buffer_ovfl procs rightsdb  $p_{\text{snd}}$  hn_rcv msg
                then ERR_MSG_OVERSIZED else SUCCESS
           else if timeout_snd = 0 then ERR_SND_TIMEOUT else  $\varepsilon_\Sigma$ 
```

where the function `ipc_send_invoc_err` guards against invocation errors, the predicate `ipc_send_rendez_vous procs schedds sndstatdb rightsdb p_{snd} hn_rcv` holds iff the process indicated by `hn_rcv` is already waiting and willing to receive from p_{snd} , and the predicate `ipc_send_buffer_ovfl procs rightsdb p_{snd} hn_rcv msg` checks whether the message `msg` exceeds the receiver's buffer.

With these prerequisites in place, we formally specify IPC_SEND in the VAMOS model:

```
ipc_send_spec_V s_V hn_rcv rights_snd msg hn_add rights_add timeout_snd  $\equiv$ 
  let res = ipc_send_err s_V.procs s_V.schedds s_V.sndstatdb s_V.rightsdb hn_rcv rights_snd
      msg hn_add rights_add timeout_snd;
       $p_{\text{snd}} = \lceil \text{cup}_V s_V.schedds \rceil$ ;  $p_{\text{rcv}} = \lceil \lceil s_V.rightsdb p_{\text{snd}} \rceil.hdb hn_rcv \rceil$ 
  in if is_error res then s_V( $\lceil \text{procs} := s_V.procs(p_{\text{snd}} \mapsto \delta_{\text{proc}} res \lceil s_V.procs p_{\text{snd}} \rceil \rceil$ )
  else if res =  $\varepsilon_\Sigma$ 
      then s_V( $\lceil \text{schedds} := v\_ipc\_wait\_updt\_schedds s_V.schedds p_{\text{snd}} \lceil s_V.priodb p_{\text{snd}} \rceil$ 
              timeout_snd  $\rceil$ )
  else s_V( $\lceil \text{procs} := v\_ipc\_trans\_updt\_procs s_V.procs s_V.rightsdb s_V.devds p_{\text{snd}}$  False
          hn_rcv rights_snd msg hn_add rights_add,
```

```

schedds := v_wkp_updt_schedds sV.schedds [prcv] sV.priodb,
sndstatdb := v_ipc_trans_updt_sndstatdb sV.sndstatdb psnd prcv False,
rightsdb := v_ipc_trans_updt_rightsdb sV.rightsdb psnd hnrcv hnadd
             rightssnd rightsadd,
devds := v_ipc_trans_updt_devds sV.devds prcv)

```

Similarly, we specify the `IPC_REQUEST` call. The only difference regards the effects of a message transfer: While the message transfer completes an `IPC_SEND` call, the caller of `IPC_REQUEST` immediately awaits its partner's reply after a successful send. Thus, the caller of `IPC_REQUEST` is appended to the wait queue, its send status is changed to `True`, and—in contrast to an `IPC_SEND` call—the process is not yet informed about the successful send phase because the receive phase might still fail. For this reason, most update functions for the message transfer have a parameter `is_request` as we have seen it for `v_ipc_trans_updt_sndstatdb`.

The specification of `IPC_RECEIVE`, however, is considerably more complex. While messages are always sent to a particular process, receiving is possible without specifying the communication partner in advance. In consequence, a single `IPC_RECEIVE` call might cause several waiting senders to fail because they tried to transfer an oversized message. Furthermore, the kernel may deliver notifications about pending interrupts or stale handles without an actual message.

Despite the call's complexity, we employ the common schematic pattern for the overall specification of `IPC_RECEIVE` as we usually do for calls in the `VAMOS` model:

```

ipc_receive_specV sV hnsnd buffer timeoutrcv ≡
  let pcup = [cupV sV.schedds];
      sendQ =
        [p ∈ sV.schedds.wait .
         v_is_sending_to sV.procs sV.sndstatdb sV.rightsdb p pcup];
      res = ipc_rcv_invoc_err sV.rightsdb pcup hnsnd buffer
  in if is_error res then sV(|procs := sV.procs(pcup ↦ δproc res [sV.procs pcup]))|)
  else sV(|procs := v_ipc_rcv_updt_procs sV.procs sV.rightsdb sV.devds sendQ pcup
           hnsnd buffer timeoutrcv,
           schedds := v_ipc_rcv_updt_schedds sV.schedds sV.procs sV.rightsdb
                     sV.priodb sV.sndstatdb sV.devds sendQ pcup hnsnd buffer
                     timeoutrcv,
           sndstatdb := v_ipc_rcv_updt_sndstatdb sV.sndstatdb sV.procs sV.rightsdb
                      sendQ pcup hnsnd buffer,
           rightsdb := v_ipc_rcv_updt_rightsdb sV.rightsdb sV.procs sendQ pcup hnsnd
                      buffer,
           devds := v_ipc_rcv_updt_devds sV.devds sV.rightsdb sV.procs sendQ pcup
                   hnsnd buffer)|)

```

The function definition starts introducing some abbreviations, namely the current process `pcup`, the queue `sendQ` of processes waiting to send to `pcup` (filtered out from the wait queue), and finally the indicator `res` for invocation errors. The essential insight,

however, is that we only guard against invocation errors and encapsulate the effects of a successful call in per-component update functions. Certainly, the actual functionality is only revealed by the definition of these functions.

We explain the effect of an IPC_RECEIVE call at the example of the function `v_ipc_rcv_updt_procs`:

```

v_ipc_rcv_updt_procs procs rightsdb devds sendQ p_rcv hn_snd buffer timeout_rcv ≡
  let (sendQ_bovfl, sendQ') =
    ipc_rcv_split_sq procs [rightsdb p_rcv] sendQ hn_snd buffer;
    procs_updt_bovfl =
      λp. if p ∈ set sendQ_bovfl then [δ_proc ERR_MSG_OVERSIZED [procs p]]
        else procs p
  in case sendQ' of
    [] ⇒ if accepts_knotes rightsdb devds hn_snd p_rcv
      then procs_updt_bovfl(p_rcv ↦
        δ_proc (deliver_knotes rightsdb devds devds.saved p_rcv) [procs p_rcv])
      else if timeout_rcv = 0
        then procs_updt_bovfl(p_rcv ↦ δ_proc ERR_RCV_TIMEOUT [procs p_rcv])
        else procs_updt_bovfl
  | p ⊙ ps ⇒
    case ω_proc [procs p] of
      IPC_SEND hn_rcv rights_snd msg hn_add rights_add timeout_snd ⇒
        v_ipc_trans_updt_procs procs_updt_bovfl rightsdb devds p False hn_rcv
          rights_snd msg hn_add rights_add
      | IPC_REQUEST hn_rcv rights_snd msg hn_add rights_add timeout_snd buffer
        timeout_rcv ⇒
          v_ipc_trans_updt_procs procs_updt_bovfl rightsdb devds p True hn_rcv
            rights_snd msg hn_add rights_add
      | _ ⇒ default

```

At first, this function defines three abbreviations. The pair $(\text{sendQ}_{\text{bovfl}}, \text{sendQ}')$ is set to the result of function `ipc_rcv_split_sq`, which partitions the send queue `sendQ`, i. e., the list of all processes willing to send a message to `p_rcv`, into two lists. The former, `sendQ_bovfl`, is the longest prefix of the send queue exclusively containing processes that try to send oversized messages, the latter, `sendQ'`, is the remaining list after stripping that prefix. In the abbreviation `procs_updt_bovfl`, the error `ERR_MSG_OVERSIZED` is signaled to all processes in the prefix `sendQ_bovfl`. With these abbreviations in place, there is a case distinction on the remaining send queue `sendQ'`: If empty, we check for kernel notifications. If there are no pending notifications or the process does not accept them, there is no suitable communication partner. If the process is not willing to wait, we send a timeout error, or otherwise append it to the wait queue (i. e., the process state remains unchanged). If processes remain in the new send queue `sendQ'`, the message of the first process `p` in `sendQ'` is transferred to the receiver. The case distinction on $\omega_{\text{proc}} [procs p]$ is merely used to identify the call parameters. Note that the function definition implicitly assumes that all processes in `sendQ` are calling for `IPC_SEND` or

IPC_REQUEST.

5.4.3 IPC in CoUP

In CoUP, we can simplify the IPC specification. As a consequence of the abandoned scheduling data structures, there is no notion of waiting. We say that a process is *pending* in send or receive, iff the process might possibly be in the wait queue. More specifically, a pending process issues an IPC call and there are no invocation errors. Furthermore, we distinguish the phase, in which a process is pending. Accordingly, we specify:

$$\begin{aligned}
 \text{is_pending_send } & \text{procs sndstatdb rightsdb } p_{\text{cup}} \text{ hn}_{\text{rcv}} \text{ rights}_{\text{snd}} \text{ msg hn}_{\text{add}} \text{ rights}_{\text{add}} \\
 & \text{timeout}_{\text{snd}} \equiv \\
 & \omega_{\text{proc}} [\text{procs } p_{\text{cup}}] = \text{IPC_SEND } \text{hn}_{\text{rcv}} \text{ rights}_{\text{snd}} \text{ msg hn}_{\text{add}} \text{ rights}_{\text{add}} \text{ timeout}_{\text{snd}} \wedge \\
 & \neg \text{is_error } (\text{ipc_send_invoc_err } \text{rightsdb } p_{\text{cup}} \text{ hn}_{\text{rcv}} \text{ rights}_{\text{snd}} \text{ msg hn}_{\text{add}} \text{ rights}_{\text{add}}) \vee \\
 & (\exists \text{buffer } \text{timeout}_{\text{rcv}}. \\
 & \quad \omega_{\text{proc}} [\text{procs } p_{\text{cup}}] = \\
 & \quad \text{IPC_REQUEST } \text{hn}_{\text{rcv}} \text{ rights}_{\text{snd}} \text{ msg hn}_{\text{add}} \text{ rights}_{\text{add}} \text{ timeout}_{\text{snd}} \text{ buffer } \text{timeout}_{\text{rcv}} \wedge \\
 & \quad \text{sndstatdb } p_{\text{cup}} = [\text{False}] \wedge \\
 & \quad \neg \text{is_error} \\
 & \quad (\text{ipc_request_invoc_err } \text{rightsdb } p_{\text{cup}} \text{ hn}_{\text{rcv}} \text{ rights}_{\text{snd}} \text{ msg hn}_{\text{add}} \text{ rights}_{\text{add}} \\
 & \quad \text{buffer } \text{timeout}_{\text{rcv}}))
 \end{aligned}$$

expressing that process p_{cup} is pending in the send phase, and:

$$\begin{aligned}
 \text{is_pending_rcv } & \text{procs sndstatdb rightsdb } p_{\text{cup}} \text{ hn}_{\text{snd}} \text{ buffer } \text{timeout}_{\text{rcv}} \equiv \\
 & \omega_{\text{proc}} [\text{procs } p_{\text{cup}}] = \text{IPC_RECEIVE } \text{hn}_{\text{snd}} \text{ buffer } \text{timeout}_{\text{rcv}} \wedge \\
 & \neg \text{is_error } (\text{ipc_rcv_invoc_err } \text{rightsdb } p_{\text{cup}} \text{ hn}_{\text{snd}} \text{ buffer}) \vee \\
 & (\exists \text{rights}_{\text{snd}} \text{ msg hn}_{\text{add}} \text{ rights}_{\text{add}} \text{ timeout}_{\text{snd}}. \\
 & \quad \omega_{\text{proc}} [\text{procs } p_{\text{cup}}] = \\
 & \quad \text{IPC_REQUEST } \text{hn}_{\text{snd}} \text{ rights}_{\text{snd}} \text{ msg hn}_{\text{add}} \text{ rights}_{\text{add}} \text{ timeout}_{\text{snd}} \text{ buffer } \text{timeout}_{\text{rcv}}) \wedge \\
 & \quad \text{sndstatdb } p_{\text{cup}} = [\text{True}]
 \end{aligned}$$

for process p_{cup} pending in the receive phase.

With this new notion, we gain more symmetry in the CoUP model. Recall that in the VAMOS specification, two situations lead to a rendezvous: (a) There is a receiver in the wait queue, which aspires a message from the current process, and the current process issues IPC_SEND or IPC_REQUEST addressing this process. (b) A process in the wait queue aspires to send to the current process, and the current process issues IPC_RECEIVE, accepting a message from the sender. In CoUP, both situations fall together: There is a process pending in the receive phase, accepting messages from another process, and the latter is pending in the send phase aspiring to send to the former.

Thus, we summarize all possible effects of a rendezvous, including errors of oversized messages, kernel notifications, and message transfers, by a single rule:

$$\llbracket s.\text{cup} = [p_{\text{cup}}];$$

$$\begin{aligned}
 & \text{is_pending_rcv } s.\text{procs } s.\text{sndstatdb } s.\text{rightsdb } p_{\text{rcv}} \text{ hn}_{\text{snd}} \text{ buffer } \text{timeout}_{\text{rcv}}; \\
 & \forall p_{\text{snd}} \in \text{set } \text{sendQ}. \\
 & \quad s.\text{procs } p_{\text{snd}} \neq \perp \wedge \\
 & \quad \text{vc_is_sending_to } s.\text{procs } s.\text{sndstatdb } s.\text{rightsdb } p_{\text{snd}} p_{\text{rcv}}; \\
 & \llbracket p_{\text{cup}} = p_{\text{rcv}} \vee p_{\text{cup}} \in \text{set } \text{sendQ} \rrbracket \\
 \implies & (s, s(\llbracket \text{procs} := \text{v_ipc_rcv_updt_procs } s.\text{procs } s.\text{rightsdb } s.\text{devds } \text{sendQ } p_{\text{rcv}} \text{ hn}_{\text{snd}} \\
 & \quad \text{buffer } \text{timeout}_{\text{rcv}}, \\
 & \quad \text{sndstatdb} := \text{v_ipc_rcv_updt_sndstatdb } s.\text{sndstatdb } s.\text{procs } s.\text{rightsdb } \text{sendQ } p_{\text{rcv}} \\
 & \quad \text{hn}_{\text{snd}} \text{ buffer}, \\
 & \quad \text{rightsdb} := \text{v_ipc_rcv_updt_rightsdb } s.\text{rightsdb } s.\text{procs } \text{sendQ } p_{\text{rcv}} \text{ hn}_{\text{snd}} \\
 & \quad \text{buffer}, \\
 & \quad \text{devds} := \text{v_ipc_rcv_updt_devds } s.\text{devds } s.\text{rightsdb } s.\text{procs } \text{sendQ } p_{\text{rcv}} \text{ hn}_{\text{snd}} \\
 & \quad \text{buffer} \rrbracket)) \\
 & \in \Delta_{\text{VC}}^{\text{cup}} \text{ mifo} \qquad \text{(rendez_vous)}
 \end{aligned}$$

where the free variable sendQ determines the assumed send queue of the receiver. It is constrained by the fact that all processes in this queue have to be active and furthermore, the predicate vc_is_sending_to has to hold, i.e., each process p_{snd} in sendQ is pending in the send phase and aspires sending to p_{rcv} . Formally, we define:

$$\begin{aligned}
 & \text{vc_is_sending_to } \text{procs } \text{sndstatdb } \text{rightsdb } p_{\text{snd}} p_{\text{rcv}} \equiv \\
 & \quad \exists \text{hn}_{\text{rcv}} \text{ rights}_{\text{snd}} \text{ msg } \text{hn}_{\text{add}} \text{ rights}_{\text{add}} \text{ timeout}_{\text{snd}}. \\
 & \quad \text{is_pending_send } \text{procs } \text{sndstatdb } \text{rightsdb } p_{\text{snd}} \text{hn}_{\text{rcv}} \text{ rights}_{\text{snd}} \text{ msg } \text{hn}_{\text{add}} \text{ rights}_{\text{add}} \\
 & \quad \text{timeout}_{\text{snd}} \wedge \\
 & \quad \llbracket \text{rightsdb } p_{\text{snd}} \rrbracket.\text{hdb } \text{hn}_{\text{rcv}} = \llbracket p_{\text{rcv}} \rrbracket
 \end{aligned}$$

Note that as a side-effect of reusing the definitions from the VAMOS specification for IPC_RECEIVE , the rule (rendez_vous) also specifies the behavior of COUP for an immediate timeout error of a process pending in the receive phase. For all other non- rendez_vous situations, we define additional rules. Namely, these situations are invocation errors for all three IPC calls and an immediate timeout during a pending send. Formally, we specify:

$$\begin{aligned}
 & \llbracket s.\text{cup} = \llbracket p_{\text{cup}} \rrbracket; \\
 & \quad \omega_{\text{proc}} \llbracket s.\text{procs } p_{\text{cup}} \rrbracket = \text{IPC_SEND } \text{hn}_{\text{rcv}} \text{ rights}_{\text{snd}} \text{ msg } \text{hn}_{\text{add}} \text{ rights}_{\text{add}} \text{ timeout}_{\text{snd}}; \\
 & \quad \text{res} = \text{ipc_send_invoc_err } s.\text{rightsdb } p_{\text{cup}} \text{hn}_{\text{rcv}} \text{ rights}_{\text{snd}} \text{ msg } \text{hn}_{\text{add}} \text{ rights}_{\text{add}}; \\
 & \quad \text{is_error } \text{res} \rrbracket \\
 \implies & (s, s(\llbracket \text{procs} := s.\text{procs}(p_{\text{cup}} \mapsto \delta_{\text{proc}} \text{res } \llbracket s.\text{procs } p_{\text{cup}} \rrbracket) \rrbracket)) \in \Delta_{\text{VC}}^{\text{cup}} \text{ mifo} \\
 & \qquad \text{(send_invoc_err)}
 \end{aligned}$$

for invocation errors upon send (accordingly, there are rules for IPC_REQUEST and IPC_RECEIVE) as well as

$$\begin{aligned}
 & \llbracket s.\text{cup} = \llbracket p_{\text{cup}} \rrbracket; \\
 & \quad \text{is_pending_send } s.\text{procs } s.\text{sndstatdb } s.\text{rightsdb } p_{\text{cup}} \text{hn}_{\text{rcv}} \text{ rights}_{\text{snd}} \text{ msg } \\
 & \quad \text{hn}_{\text{add}} \text{ rights}_{\text{add}} 0 \rrbracket
 \end{aligned}$$

$$\begin{aligned} \Rightarrow & (s, s(\text{procs} := s.\text{procs}(p_{\text{cup}} \mapsto \delta_{\text{proc}} \text{ERR_SND_TIMEOUT } [s.\text{procs } p_{\text{cup}}]))) \\ & \in \Delta_{\text{VC}}^{\text{cup}} \text{ mifo} \qquad \qquad \qquad (\text{send_imm_timeout}) \end{aligned}$$

for an immediate timeout upon send. Unsurprisingly, these definitions just resemble the corresponding error cases that we already know from the VAMOS specification and simply advance the current process with the according error response.

In conclusion, the auxiliary functions that we have defined and presented for the VAMOS specification permit a relatively succinct and simple specification of IPC in CoUP.

5.5 The Scheduler Specification

The scheduler deals with two tasks, both related to time: The accounting of process runtime, which is informally described in [Section 3.2](#), and the management of timeouts, which [Section 3.3](#) reports on. The scheduling data structures $s_V.\text{schedds}$ of the VAMOS state space comprise several sub components to support these tasks.

First, there are different process queues (lists of process identifiers), in particular:

- one ready queue $s_V.\text{schedds}.\text{ready } prio$ of schedulable processes per priority $prio \in \{\text{LOW}, \text{MEDIUM}, \text{HIGH}\}$,
- a queue $s_V.\text{schedds}.\text{wait}$ of currently not schedulable but active processes, which have issued an IPC call and are waiting for a suitable communication partner, and
- a list of currently unused PIDs $s_V.\text{schedds}.\text{inactive}$.

Second, process-specific scheduling information for active processes is collected in the partial function $s_V.\text{schedds}.\text{procdb}$ that maps PIDs to a record of (a) the timeslice tsl , (b) the amount of consumed runtime ctsl , and (c) the scheduling timeout timeout .

Third, there is a current-time variable $s_V.\text{schedds}.\text{time}$. Recall that the kernel measures time based on the periodic timer interrupts. Variable $s_V.\text{schedds}.\text{time}$ counts the overall number of timer interrupts in the system. Note that in contrast to the implementation, this variable is never decreased in the specification. Thus, there is a monotonically increasing offset between the variables in the implementation and the specification.²

As we have seen in [Section 5.1](#), the ready queues determine the process to compute. If the current process p_{cup} computes when a timer interrupt is raised, the component ctsl for p_{cup} is increased until the process has finally run for tsl timer interrupts. In the latter case, ctsl is reset and the corresponding ready queue is rotated by one place such that the next process in this queue is scheduled.

Formally, we describe this task by the following function:

```
v_charge_process s_V p_cup ≡
  let procdb = [s_V.schedds.procdb [p_cup]];
      ready = s_V.schedds.ready [s_V.priodb [p_cup]]
  in if p_cup = ⊥ ∨ [p_cup] ∉ set ready then s_V
      else s_V([schedds := if procdb.tsl ≤ procdb.ctsl
```

²More information on the abstraction relation coupling implementation and specification can be found in [\[DDW09, Sect. 6.1\]](#).

```

then  $s_V$ .schedds
  (ready :=  $s_V$ .schedds.ready
   ([ $s_V$ .priodb [ $p_{cup}$ ]] :=
    [ $x \in ready . x \neq [p_{cup}] \odot [[p_{cup}]]$ ],
   procdb :=  $s_V$ .schedds.procdb( $[p_{cup}] \mapsto procdb(ctl := 0)$ ))
else  $s_V$ .schedds
  (procdb :=  $s_V$ .schedds.procdb( $[p_{cup}] \mapsto
   procdb(ctl := procdb.ctl + 1)$ ))

```

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 cup_V s_V has been computing in the last step. The function specification defines two abbreviations *procdb* and *ready* for the process-specific information and the ready queue of the process p_{cup} , respectively. If no process has been computing in the last cycle or this process is no longer ready, the state remains untouched. Otherwise, we compare the timeslice with the consumed time. If the consumed time *procdb.ctl* exceeds the timeslice *procdb.tsl*, process p_{cup} is filtered out from the current ready queue *ready* and appended at the end. Furthermore, the *ctl* for p_{cup} is reset. If the timeslice is not exceeded, the *ctl* for p_{cup} is just increased.

If a process issues an IPC call and no partner is ready for communication, it is removed from its ready queue and enqueued in s_V .schedds.wait, the scheduling timeout *timeout* is computed from the current time s_V .schedds.time and the relative timeout that has been specified with the call.

With every timer interrupt, the current time is increased. Afterwards, the scheduler checks whether the scheduling timeout of any waiting process p has thereby expired, i. e., $[s_V$.schedds.procdb p].*timeout* = s_V .schedds.time. If so, the scheduler sends the according error message to the process and sets it back to the ready state.

Formally, we describe this task by the following function:

```

v_manage_timeouts  $s_V \equiv
  s_V(\text{procs} := \lambda p. \text{if } \text{timeout\_elapsed } s_V.\text{schedds } p
    \text{ then if } \text{is\_sending } s_V.\text{procs } s_V.\text{sndstatdb } p
      \text{ then } [\delta_{proc} \text{ ERR\_SND\_TIMEOUT } [s_V.\text{procs } p]]
      \text{ else } [\delta_{proc} \text{ ERR\_RCV\_TIMEOUT } [s_V.\text{procs } p]]
    \text{ else } s_V.\text{procs } p,
  \text{schedds} := s_V.\text{schedds}
  (\text{time} := s_V.\text{schedds.time} + 1,
   \text{ready} := \lambda prio.
     s_V.\text{schedds.ready } prio \odot
     [p \in s_V.\text{schedds.wait} .
      \text{timeout\_elapsed } s_V.\text{schedds } p \wedge [s_V.\text{priodb } p] = prio],
   \text{wait} := [p \in s_V.\text{schedds.wait} . \neg \text{timeout\_elapsed } s_V.\text{schedds } p],
   \text{procdb} := \lambda p. \text{if } \text{timeout\_elapsed } s_V.\text{schedds } p
     \text{ then } [s_V.\text{schedds.procdb } p](ctl := 0)
     \text{ else } s_V.\text{schedds.procdb } p),$ 
```

```

sndstatdb := λp. if timeout_elapsed sv.schedds p then [False] else sv.sndstatdb p)
    
```

where in component `procs`, an error message is signaled to all processes p with an elapsed timeout, i. e., when predicate `timeout_elapsed` holds. For the appropriate error message, it is distinguished whether the process has been waiting in the send or in the receive phase. Furthermore, these processes are set back to ready: To the ready queue of each priority $prio$, we append the wait queue filtered for processes with an elapsed timeout and the according priority. Moreover, we remove all processes with an elapsed timeout from the wait queue. For these processes, we reset the consumed time. Finally, the send status for processes with an elapsed timeout is set to `False` because they do not longer wait for a reply.

Summarizing, two different tasks are carried out in VAMOS when the timer interrupt is raised: The scheduler checks whether scheduling timeouts of waiting processes have expired, on the one hand, and it accounts for the runtime of the process that has computed in the last step, on the other hand. Hence, the specification function of the scheduler is the composition of the according two specifications:

```

scheduler_specv sv pcup ≡ v_charge_process (v_manage_timeouts sv) pcup
    
```

In CoUP, there is no scheduling information. Thus, we can only specify the scheduler effects non-deterministically. At first, some of the processes that are pending with a finite timeout are woken up. We define:

$$\begin{aligned}
 \Delta_{\text{VC}}^{\text{to}} \equiv & \{(s, s') . \\
 & \exists up' sdb'. s' = s(\text{procs} := up', \text{sndstatdb} := sdb') \wedge \\
 & (\forall p. up' p = s.\text{procs } p \wedge sdb' p = s.\text{sndstatdb } p \vee \\
 & \text{does_finite_ipc } s.\text{procs } s.\text{sndstatdb } s.\text{rightsdb } p \wedge \\
 & up' p = \\
 & \text{(if is_sending } s.\text{procs } s.\text{sndstatdb } p \\
 & \text{ then } [\delta_{\text{proc}} \text{ ERR_SND_TIMEOUT } [s.\text{procs } p]] \\
 & \text{ else } [\delta_{\text{proc}} \text{ ERR_RCV_TIMEOUT } [s.\text{procs } p]]) \wedge \\
 & sdb' p = [\text{False}])\}
 \end{aligned}$$

meaning that all pairs (s, s') in the relation only differ in `procs` and `sndstatdb`. Additionally, for each process p , the value of the components must either be equal, or in s , the process has issued an IPC call with a finite timeout (we encapsulate this property in `does_finite_ipc`) and in s' , the process has received a timeout-error message (corresponding to the current IPC phase of the process) and its send status is reset.

In a second step, either no, or an arbitrary, active process is elected to compute next:

$$\Delta_{\text{VC}}^{\text{sched}} \equiv \{(s, s') . \exists cup'. s' = s(\text{cup} := cup') \wedge (\forall p. cup' = \perp] \longrightarrow s.\text{procs } p \neq \perp)\}$$

Note that we specify the scheduler effects in CoUP independently from the timer interrupt because the correctness of the processes should not rely on a particular timing.

5.6 Invariants over Execution Traces

Not all states are reachable in the execution traces of the kernel models. During our survey through the kernel models, we have learned about several assumptions on states, like that the priority of every active process is defined. So far, these assumptions have just been informal statements but when reasoning about execution traces, we like to employ this implicit knowledge. In this section, we formally define the notion *execution trace* for the VAMOS specification, gather properties on states into a predicate inv_V , and show that this predicate holds for all reachable states.

We represent VAMOS execution traces as two functions *states* and *inputs* that map the step number to the current state $s_V \in \mathcal{S}_V$ and to the input $i \in \hat{\Sigma}$ for the next step, respectively.

Definition 5.1 (VAMOS Execution Trace). Two functions *states* and *inputs* describe an execution trace of the VAMOS specification iff they meet the following two conditions:

- $\text{states } 0 \in \mathcal{S}_V^0$
- $\forall n. \text{states } (n + 1) = \delta_V (\text{inputs } n) (\text{states } n)$

When reasoning on the behavior of VAMOS, we rely on several properties that are invariant over all execution traces. We collect these facts in the predicate inv_V . This invariant, however, is quite elaborate. Hence, we introduce various predicates on several VAMOS state components and finally define the invariant as the conjunct of these subpredicates.

First of all, we know from [Section 5.1](#) that VAMOS maintains various data for active processes. We call the process *pid* active iff the function $s_V.\text{procs}$ is defined for *pid*, i. e., $s_V.\text{procs } pid \neq \perp$. In this case, the process-accounting information, the priority, the send status, and the access-control rights for *pid* should be defined. Furthermore, we assume that these functions are undefined for inactive processes. Formally:

$$\begin{aligned} \text{is_proc_info}_V s_V &\equiv \\ &\forall p. (s_V.\text{schedds}.\text{procdB } p = \perp \longleftrightarrow s_V.\text{procs } p = \perp) \wedge \\ &\quad (s_V.\text{priodb } p = \perp \longleftrightarrow s_V.\text{procs } p = \perp) \wedge \\ &\quad (s_V.\text{sndstatdb } p = \perp \longleftrightarrow s_V.\text{procs } p = \perp) \wedge \\ &\quad (s_V.\text{rightsdb } p = \perp \longleftrightarrow s_V.\text{procs } p = \perp) \end{aligned}$$

Additionally, [Section 5.1](#) states that the send status keeps track of the phase of an `IPC_REQUEST` call. Consequently, the status bit of a process *pid* should only be set if *pid* issues an `IPC_REQUEST` call. We state:

$$\begin{aligned} \text{sndstat_imp_request } s.\text{procs } s.\text{sndstatdb} &\equiv \\ \forall p. s.\text{sndstatdb } p = \text{True} &\longrightarrow \\ (\exists hn_{\text{rcv}} \text{rights}_{\text{snd}} \text{msg } hn_{\text{add}} \text{rights}_{\text{add}} \text{timeout}_{\text{snd}} \text{buffer } \text{timeout}_{\text{rcv}}. \\ \omega_{\text{proc}} [s.\text{procs } p] = & \\ \text{IPC_REQUEST } hn_{\text{rcv}} \text{rights}_{\text{snd}} \text{msg } hn_{\text{add}} \text{rights}_{\text{add}} \text{timeout}_{\text{snd}} \text{buffer } \text{timeout}_{\text{rcv}}) & \end{aligned}$$

Similarly, a send status of `True` implies that the process is waiting, formally:

$$\text{sndstat_imp_wait } s_V.\text{schedds } s_V.\text{sndstatdb} \equiv$$

$$\forall p. s_V.\text{sndstatdb } p = \lfloor \text{True} \rfloor \longrightarrow p \in \text{set } s_V.\text{schedds.wait}$$

In fact, most of our assumptions on the states of execution traces are concerned with the scheduling data structures (see [Section 5.5](#)). For instance, inactive processes should reside in the inactive queue and neither in the wait nor any ready queue. Accordingly, active processes may not be in the inactive queue but in either the wait queue or a ready queue. Formally:

$$\text{valid_queue_members } s_V.\text{procs } s_V.\text{schedds} \equiv$$

$$\forall p. \text{case } s_V.\text{procs } p \text{ of}$$

$$\quad \perp \Rightarrow p \in \text{set } s_V.\text{schedds.inactive} \wedge$$

$$\quad \quad p \notin \text{set } s_V.\text{schedds.wait} \wedge (\forall i. p \notin \text{set } (s_V.\text{schedds.ready } i))$$

$$\quad \lfloor a \rfloor \Rightarrow p \notin \text{set } s_V.\text{schedds.inactive} \wedge$$

$$\quad \quad (p \in \text{set } s_V.\text{schedds.wait}) \neq (\exists i. p \in \text{set } (s_V.\text{schedds.ready } i))$$

Moreover, all processes in a ready queue should have the queue's associated priority:

$$\text{prio_of_readyQ } s_V.\text{schedds } s_V.\text{priodb} \equiv$$

$$\forall i. \forall p \in \text{set } (s_V.\text{schedds.ready } i). \lceil s_V.\text{priodb } p \rceil = i$$

Furthermore, the entries in each queue should be pairwise distinct:

$$\text{distinct_queues } s_V.\text{schedds} \equiv$$

$$(\forall m < |s_V.\text{schedds.inactive}|.$$

$$\quad \forall n < m. s_V.\text{schedds.inactive} ! m \neq s_V.\text{schedds.inactive} ! n) \wedge$$

$$(\forall m < |s_V.\text{schedds.wait}|. \forall n < m. s_V.\text{schedds.wait} ! m \neq s_V.\text{schedds.wait} ! n) \wedge$$

$$(\forall i. \forall m < |s_V.\text{schedds.ready } i|.$$

$$\quad \forall n < m. s_V.\text{schedds.ready } i ! m \neq s_V.\text{schedds.ready } i ! n)$$

[Section 4.2](#) mentions another scheduling-related invariant: For the fairness proof, it is important that timeslices are bounded:

$$\text{bounded_timeslices } s_V.\text{schedds} \equiv$$

$$\forall p \text{ procdata}. s_V.\text{schedds.procdb } p = \lfloor \text{procdata} \rfloor \longrightarrow \text{procdata.tsl} \leq \text{MAXTSL}$$

When carefully examining the definition of `v_charge_process` on page 79, we recognize that `ctsl` should be zero if a ready process is not the first in its queue. This subtlety, however, is irrelevant for temporal fairness. If we would nevertheless specify it, we had to prove the invariance. Hence, we are content with the weaker formulation.

A further scheduler invariant has already been suggested by [Section 3.3](#): All scheduling timeouts should be greater than or equal to the global time. Certainly, the scheduling timeout should be infinite if the process has issued an IPC call with an infinite timeout. Formally:

$$\text{valid_timeouts } s_V.\text{procs } s_V.\text{schedds } s_V.\text{sndstatdb } s_V.\text{rightsdb} \equiv$$

$$\forall p \in \text{set } s_V.\text{schedds.wait}.$$

$$s_V.\text{schedds.time} \leq \lceil s_V.\text{schedds.procdb } p \rceil.\text{timeout} \wedge$$

$$(\lceil s_V.\text{schedds.procdb } p \rceil.\text{timeout} = \infty \iff$$

$$\quad \text{current_timeout } \lceil s_V.\text{procs } p \rceil \lceil s_V.\text{sndstatdb } p \rceil = \infty)$$

where `current_timeout` s_{proc} `sndstat` examines the call issued by the process in state s_{proc} and selects the send or receive timeout depending on the send status `sndstat` in case of `IPC_REQUEST`.

Section 5.4 states that processes may only wait for IPC. Besides that, the process must agree to wait by specifying a positive timeout value with the call. Certainly, a process should not wait in a rendezvous situation. Additionally, it is not desirable that a process waits though some error impedes a rendezvous: We know that the kernel checks for invocation errors of IPC calls immediately when the call is issued. If changes of the kernel state induce such an error while the process waits, the process is immediately notified of the error, as we have seen with terminating processes in **Section 5.3**. Consequently, we may assume that there are no invocation errors for waiting processes. We collect all this knowledge in the predicate `wait_imp_IPC`. Similarly, the predicate `valid_handles` s_{procs} `s.rightsdb` is concerned with IPC. It summarizes properties on valid handles like those specified in **Section 5.4**.

There are two more predicates, which are independent from the scheduling data structures. First, all active processes should certainly be well-formed according to our process abstraction (see **Section 4.2**):

$$\text{valid_procs } s_{\text{procs}} \equiv \forall pid. s_{\text{procs}} \text{ } pid = \perp \vee \text{is_valid_proc } [s_{\text{procs}} \text{ } pid]$$

Second, the predicate `valid_devds` s_{procs} `s.devds` summarizes constraints on device data structures.

Finally, we define the overall invariant as a conjunct of all previous predicates:

Definition 5.2 (VAMOS Trace Invariant). The predicate `invV` is defined as follows:

$$\begin{aligned} \text{inv}_V s_V \equiv & \text{is_proc_info}_V s_V \wedge \text{valid_procs } s_V.\text{procs} \wedge \\ & \text{valid_queue_members } s_V.\text{procs } s_V.\text{schedds} \wedge \text{distinct_queues } s_V.\text{schedds} \wedge \\ & \text{prio_of_readyQ } s_V.\text{schedds } s_V.\text{priodb} \wedge \\ & \text{wait_imp_IPC } s_V.\text{procs } s_V.\text{schedds } s_V.\text{sndstatdb } s_V.\text{rightsdb } s_V.\text{devds} \wedge \\ & \text{valid_timeouts } s_V.\text{procs } s_V.\text{schedds } s_V.\text{sndstatdb } s_V.\text{rightsdb} \wedge \\ & \text{bounded_timeslices } s_V.\text{schedds} \wedge \text{sndstat_imp_wait } s_V.\text{schedds } s_V.\text{sndstatdb} \wedge \\ & \text{sndstat_imp_request } s_V.\text{procs } s_V.\text{sndstatdb} \wedge \text{valid_handles } s_V.\text{procs } s_V.\text{rightsdb} \wedge \\ & \text{valid_devds } s_V.\text{procs } s_V.\text{devds} \end{aligned}$$

Ultimately, we show that this predicate is indeed invariant over execution traces:

Theorem 5.1 (VAMOS Trace Invariant). *The predicate `invV` holds on all states of an arbitrary execution trace, formally:*

`invV` (states m) holds for arbitrary m in any VAMOS execution trace (states, inputs).

Proof. We prove this statement by induction. At first, we establish `invV` for the initial states: $s_V \in \mathcal{S}_V^0 \implies \text{inv}_V s_V$. This proof is comparatively simple because the set of initial states is small: From **Section 3.2**, we now that VAMOS launches a single, privileged process at the beginning; the initial states hence only vary in the size and the image of this init process.

Then, we show that inv_V is maintained by the transition function, i. e., $\text{inv}_V s_V \implies \text{inv}_V (\delta_V i s_V)$. This proof, in contrast, is very elaborate because of the complexity of the kernel transitions and the invariant. Note that the invariant holds after each transition phase. Thus, we have verified the overall statement for each transition phase separately. The first transition phase, which advances the current process, is the most complex one. Just for this phase, we have proven the invariant predicate by predicate, and each time by a case distinction over the 18 possible process outputs. Within this proof, we have used [Theorem 4.1](#) on page 47. The formal proof has taken more than two person months.

Finally, we derive our claim $\text{inv}_V (\text{states } m)$ by the definition of traces. \square

In the same fashion, we can define an invariant over the CoUP states. First, we define the validity constraints for the process-specific information without the scheduling data:

$$\begin{aligned} \text{is_proc_info}_{VC} s_{VC} &\equiv \\ &\forall p. (s_{VC}.\text{priodb } p = \perp \longleftrightarrow s_{VC}.\text{procs } p = \perp) \wedge \\ &\quad (s_{VC}.\text{sndstatdb } p = \perp \longleftrightarrow s_{VC}.\text{procs } p = \perp) \wedge \\ &\quad (s_{VC}.\text{rightsdb } p = \perp \longleftrightarrow s_{VC}.\text{procs } p = \perp) \end{aligned}$$

Second, we specify the validity of the current process—the only remains from the scheduling data structures in CoUP. In [Section 5.5](#), the transition $\Delta_{VC}^{\text{sched}}$ explicitly requires that if the current process is defined, it refers to an active process. Formally:

$$\text{valid_cup}_{VC} s_{VC}.\text{procs } s_{VC}.\text{cup} \equiv \forall p. s_{VC}.\text{cup} = [p] \longrightarrow s_{VC}.\text{procs } p \neq \perp$$

Third, we reuse all predicates from VAMOS that do not involve the scheduling data. Fourth, we can conclude from the combination of the predicates `sndstat_imp_wait` and `wait_imp_IPC` that if the send status of a process is `True`, the timeout value specified with the IPC call is positive and there are no errors impeding a rendezvous. We encapsulate this fact in the predicate `vc_sndstat_imp_no_error`.

Finally, we specify:

$$\begin{aligned} \text{inv}_{VC} s_{VC} &\equiv \text{is_proc_info}_{VC} s_{VC} \wedge \text{valid_procs } s_{VC}.\text{procs} \wedge \text{valid_cup}_{VC} s_{VC}.\text{procs } s_{VC}.\text{cup} \wedge \\ &\quad \text{sndstat_imp_request } s_{VC}.\text{procs } s_{VC}.\text{sndstatdb} \wedge \\ &\quad \text{vc_sndstat_imp_no_error } s_{VC}.\text{procs } s_{VC}.\text{sndstatdb } s_{VC}.\text{rightsdb} \wedge \\ &\quad \text{valid_handles } s_{VC}.\text{procs } s_{VC}.\text{rightsdb} \wedge \text{valid_devds } s_{VC}.\text{procs } s_{VC}.\text{devds} \end{aligned}$$

and prove the according invariance theorem:

Theorem 5.2 (CoUP Trace Invariant). *We assume that two functions states and inputs describe an execution trace of the CoUP model, i. e., states $0 \in \mathcal{S}_{VC}^0$ and for all steps n , there is a transition $\text{states } n \xrightarrow{\text{inputs } n}_{VC} \text{states } (n + 1)$. Then, the predicate inv_{VC} holds for every state $\text{states } n$ in the trace.*

Proof. Just as with inv_V , we prove this statement by induction. This proof is substantially simpler because of the abandoned scheduling data structures. Note, however, that the invariant does not hold after each transition phase of Δ_{VC} : If the current process terminates, the current-process indicator is dangling. Fortunately, only Δ_{VC}^{cup} is

concerned with the current process. Hence, we introduce a slightly weakened invariant inv'_{VC} that omits $\text{valid_cup}_{\text{VC}}$. Assuming inv_{VC} , the predicate inv'_{VC} holds after $\Delta_{\text{VC}}^{\text{cup}}$ and is preserved by $\Delta_{\text{VC}}^{\text{to}}$ and $\Delta_{\text{VC}}^{\text{ints}}$. Finally, the transition $\Delta_{\text{VC}}^{\text{sched}}$ re-establishes $\text{valid_cup}_{\text{VC}}$ and thus inv_{VC} .

Note that for the proof about $\Delta_{\text{VC}}^{\text{cup}}$, we rely on the rules of process-model validity. Thus, the CoUP invariant only holds if the underlying process model is valid (cf. the according constraint on the CoUP state space in [Section 5.1](#)).

We have formally proven the theorem in about two person weeks. \square

Note that the actually required invariant heavily depends on the verification task. As our proofs about the VAMOS specification center around the scheduling mechanism, most assumptions are concerned with the scheduling data structures.

5.7 Conclusion

This chapter defined two automata, the VAMOS specification \mathcal{A}_V and the CoUP model \mathcal{A}_{VC} , which both formally describe the behavior of VAMOS. While the former constitutes a low-level kernel specification for code verification with the exact scheduling behavior, the latter abstracts from scheduling details and focusses on the interaction of processes.

An important but sometimes challenging task [[DDB08](#)] was the adjustment of the models to their respective purposes: While a kernel model specified close to the implementation facilitates the code verification, the verification of kernel properties like scheduler fairness (see [Chapter 7](#)) benefit from a fairly abstract kernel model. Both models share many common concepts, which we specified once and for all. For reusable specifications, we generalized many definitions for the VAMOS specification to use an arbitrary process model. Thus, we could easily reuse them in CoUP as well. This foresight was the key to succinct and manageable simulation proofs. Constant adjustments of such definitions during the development of the models show that a high degree of reusability can best be achieved if the models are developed in parallel.

It turned out that a major obstacle for our proofs was the lack of an early, systematic validation for specifications. During the formal proof development for the invariance of inv_V and for scheduler fairness, we found many trivial oversights like misinterpretations of the code or failures to update all relevant specification parts after changes. We experienced that especially trivial flaws are hard to locate during interactive theorem proving: While the human way of thinking abstracts from trivialities, the proof automation in Isabelle only works for correct statements.

6 Simulation: Formally Relating the Kernel Models

Contents

6.1 Kernel-Model Simulation	87
6.2 Abstracting Concurrent Assembly Processes to C0 Processes	90
6.2.1 On the Commutability of Transitions: From Trace Theory to Kleene Algebra	91
6.2.2 Process Simulation Meets True Concurrency	92
6.2.3 Commuting Transitions in CoUP	93
6.3 Conclusion	95

In this chapter, we formally establish a simulation between the VAMOS specification and a CoUP model with C0 processes. Note that not every assembly process can be abstracted to a C0 process. The compiled C0 programs are a strict subset of the assembly programs: For a given assembly program, there might not exist any C0 program that compiles to this assembly program. Though the operating system and its applications are usually implemented in C0, our verification framework should not rely on this assumption. Quite on the contrary, it should be possible to verify that the operating system works correctly even in the presence of malicious assembly processes.

Hence, assembly processes coexist with C0 processes in our CoUP model. We formally express this coexistence by a combined process model: Its process states are the disjoint union of the assembly and the C0 state spaces while the corresponding predicates $\text{is_valid}_{\text{proc}}$ and $\text{is_init}_{\text{proc}}$ as well as the transition and output functions δ_{proc} , ω_{proc} , and $\text{size}_{\text{proc}}$ are simply defined as mediators, selecting the corresponding function depending on the kind of state.

We establish the simulation in two steps: In [Section 6.1](#), we narrow our consideration to CoUP states with solely assembly processes. We define a function abstracting VAMOS states to those CoUP states, and show that the CoUP transitions simulate all possible VAMOS transitions. In [Section 6.2](#), we abstract assembly processes within CoUP to C0 processes. Note that the latter abstraction is not strictly necessary for the verification of the SOS in Verisoft (see [Section 8.3](#)).

6.1 Kernel-Model Simulation

We start with the definition of the abstraction function:

Definition 6.1 (Kernel-State Abstraction). The function abs_{VC} constructs a CoUP state from a VAMOS state by reducing the scheduling information to the current-process indicator. Formally:

$$\begin{aligned} \text{abs}_{\text{VC}} s_V \equiv & \\ & (\text{procs} = s_V.\text{procs}, \text{cup} = \text{cup}_V s_V.\text{schedds}, \text{priodb} = s_V.\text{priodb}, \\ & \text{sndstatdb} = s_V.\text{sndstatdb}, \text{rightsdb} = s_V.\text{rightsdb}, \text{devds} = s_V.\text{devds}) \end{aligned}$$

Note that in the resulting CoUP state, all processes are necessarily assembly processes. With the abstraction function in place, we can directly formulate the required simulation theorem:

Theorem 6.1 (Kernel-Model Simulation). *Provided that the VAMOS invariant holds, CoUP transitions simulate VAMOS transitions wrt. abs_{VC} . Formally:*

$$\text{inv}_V s_V \implies \text{abs}_{\text{VC}} s_V \xrightarrow{i}_{\text{VC}} \text{abs}_{\text{VC}} (\delta_V i s_V)$$

Proof. Similar to the verification of the VAMOS invariant (see [Theorem 5.1](#) on page 83), we separately regard the four transition relations (see the definition of Δ_{VC} on page 66).

Within the proof for the first transition relation, we distinguish whether there is a current process. If there is none, the first phase is not present in VAMOS, i. e., the call dispatcher is not invoked at all. We model this behavior by the rule (reflexivity) of $\Delta_{\text{VC}}^{\text{cup}}$ *mifo* on page 67.

For an existing current process, we state that the transitions of $\Delta_{\text{VC}}^{\text{cup}}$ *mifo* simulate the transitions of the call dispatcher. Note, however, that the component `cup` remains unchanged under $\Delta_{\text{VC}}^{\text{cup}}$ *mifo* while the call dispatcher might cause a change of the current process by altering the scheduling data structures. Thus, we explicitly reinforce the old current process for simulation:

$$\begin{aligned} & \llbracket \text{inv}_V s_V; \text{cup}_V s_V.\text{schedds} = \lfloor p_{\text{cup}} \rfloor \rrbracket \\ & \implies (\text{abs}_{\text{VC}} s_V, (\text{abs}_{\text{VC}} (\text{call_dispatcher_spec}_V s_V \text{ mifo})) (\text{cup} := \lfloor p_{\text{cup}} \rfloor)) \in \Delta_{\text{VC}}^{\text{cup}} \text{ mifo} \end{aligned}$$

The proof of this fact is quite elaborate; hence, we defer its discussion to [Lemma 6.2](#) on the next page.

In the same fashion, we encapsulate the simulation of `scheduler_specV` by $\Delta_{\text{VC}}^{\text{to}}$ (cf. [Section 5.5](#)). This simulation follows easily from the definition of `scheduler_specV` and the corresponding relation $\Delta_{\text{VC}}^{\text{to}}$: The outer function `v_charge_process` of `scheduler_specV` solely alters the scheduling data structures. Hence, the changes are no longer visible when the state is abstracted to CoUP and the current process is reset.

The inner function `v_manage_timeouts` additionally updates processes and send statuses for processes with an elapsed timeout. In $\Delta_{\text{VC}}^{\text{to}}$, a process and its send status is either unchanged, or the process issues a finite IPC call and the updates are exactly the same as for processes with an elapsed timeout in `v_manage_timeouts`. Note that an elapsed timeout requires that the concerned process has issued a finite IPC call. Thus, $\Delta_{\text{VC}}^{\text{to}}$ simulates the transitions of `v_manage_timeouts` and hence those of `scheduler_specV`.

Similarly, we show the simulation of the third VAMOS phase by $\Delta_{\text{VC}}^{\text{ints}}$. In a last step, we combine our results. Each VAMOS transition phase is simulated by the relations

$\Delta_{\text{VC}}^{\text{cup}}$, $\Delta_{\text{VC}}^{\text{to}}$, and $\Delta_{\text{VC}}^{\text{ints}}$, under a constant current process. Note that the current process is irrelevant in $\Delta_{\text{VC}}^{\text{to}}$ and $\Delta_{\text{VC}}^{\text{ints}}$ because they focus on waking up waiting processes.

Finally, we establish the overall simulation relation using $\Delta_{\text{VC}}^{\text{sched}}$ to simulate any changes of the current process during the individual transition phases. Recall that the definition of $\Delta_{\text{VC}}^{\text{sched}}$ on page 80 requires that only an active process might become the current one. We employ the VAMOS invariant to show that this requirement is not violated by the new current process.

The formalization of this proof in Isabelle/HOL took about two person months, mostly for the verification of [Lemma 6.2](#). \square

Lemma 6.2. *The transitions of $\Delta_{\text{VC}}^{\text{cup}}$ simulate those of the call dispatcher:*

$$\begin{aligned} & \llbracket \text{inv}_{\text{V}} s_{\text{V}}; \text{cup}_{\text{V}} s_{\text{V}}.\text{schedds} = \lfloor p_{\text{cup}} \rfloor \rrbracket \\ \implies & (\text{abs}_{\text{VC}} s_{\text{V}}, (\text{abs}_{\text{VC}} (\text{call_dispatcher_spec}_{\text{V}} s_{\text{V}} \text{ mifo})) (\text{cup} := \lfloor p_{\text{cup}} \rfloor)) \in \Delta_{\text{VC}}^{\text{cup}} \text{ mifo} \end{aligned}$$

Proof. In essence, we prove this lemma by unfolding `call_dispatcher_specV` and employing the introduction rules that define $\Delta_{\text{VC}}^{\text{cup}}$ (cf. [Section 5.2](#)). The definition of `call_dispatcher_specV`, however, is a case distinction on the output $\omega_{\text{proc}} \llbracket s_{\text{V}}.\text{procs } p_{\text{cup}} \rrbracket$ of the current process. Consequently, our proof structure follows this case distinction, regarding the different kernel calls one by one.

For most kernel calls, it is sufficient to distinguish between the error cases, on the one hand, and the successful case on the other hand. Just recall the specification of the `SET_PRIVILEGED` call in [Section 5.2](#): In the VAMOS specification, the call is specified by the function `set_privileges_specV`, which merely differentiates whether `is_error` (`set_privileged_response sV.rightsdb pcup hn`) holds. For each case, we have a corresponding rule of $\Delta_{\text{VC}}^{\text{cup}}$: (`privilege_err`) and (`privilege_succ`), respectively.

The only exception to that rule is IPC, or more specifically: the `IPC_SEND` and `IPC_REQUEST` calls. About seventy percent of the formal proof script (including auxiliary lemmata) are concerned with these two calls. This verification effort is the prize for the gained symmetry in the Coup model (see [Section 5.4](#)): In contrast to all other calls, we have spared extra rules for successful `IPC_SEND` and `IPC_REQUEST` calls because we claimed that the effects are already covered by a successful `IPC_RECEIVE`. The simulation proof formally legitimates this claim.

Thus, for the `IPC_SEND` and `IPC_REQUEST` calls, we first sort out the cases with invocation errors or immediate timeouts, which $\Delta_{\text{VC}}^{\text{cup}}$ has extra rules for. Then, we show that the rule (`rendez_vous`) covers all remaining cases. In the course of the latter proof, we verify propositions like:

- If predicate `ipc_send_rendez_vous` holds for the current process p_{cup} , we can conclude `is_pending_rcv` for another process p_{rcv} and `vc_is_sending_to` for p_{cup} and p_{rcv} . This fact follows from the definitions of the predicates and the VAMOS invariant. The latter is required, e. g., for the definedness of the rights data base (see predicate `is_proc_infoV`) or that there are no invocation errors for waiting processes (predicate `wait_imp_IPC`).

- If both predicates `ipc_send_buffer_ovfl` and `ipc_send_rendez_vous` hold for p_{cup} , the function `ipc_rcv_split_sq` sorts p_{cup} into the first list. Again, we unfold the definitions and employ the invariant.
- Assuming p_{rcv} is waiting, neither predicate `accepts_knotes` holds, nor can the specified timeout be zero. This fact follows directly from the VAMOS invariant.

□

6.2 Abstracting Concurrent Assembly Processes to C0 Processes

Our original motive for the introduction of CoUP has been the verification of C0 programs in the concurrent environment of our microkernel. Yet, our consideration in the previous section has been limited to a CoUP model with solely assembly processes. In [Section 4.5](#), however, we have established the simulation between C0 and assembly processes. This section presents the capstone that combines both results and finally establishes a CoUP model with C0 processes.

The difficulty in combining the previous results stems from the fact that a transition of a C0 process may span multiple transitions of an assembly process while VAMOS might schedule another process in each transition. Thus, we have to rearrange transition sequences before we can combine adjacent assembly transitions to a C0 transition. [Figure 6.1](#) illustrates the situation:

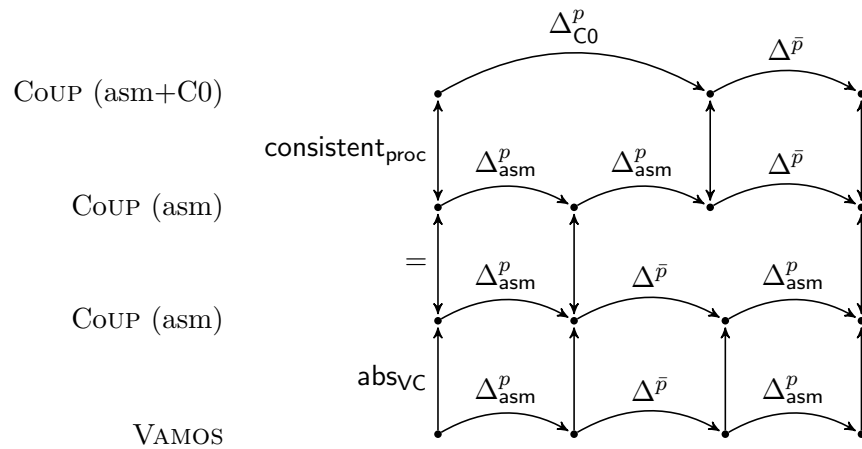


Figure 6.1: Abstracting assembly processes in a concurrent setting to C0 processes

We see exemplary extracts from kernel traces, stacked on top of each other. At the top, there are two transitions of the CoUP model with assembly and C0 processes. The first one is labeled Δ_{C0}^p , denoting that a C0 process p performs a transition. The second transition is labeled $\Delta_{\bar{p}}$ for a transition that does not involve p . Below that, we see the

execution trace of the CoUP model with solely assembly processes. From the process-simulation theorem ([Theorem 4.3](#) on page 57), we know that Δ_{C0}^p abstracts a number of assembly transitions – in the illustration, this number is two.

At the bottom of the figure, we see an extract from a VAMOS trace. In the first transition, the assembly process p advances, the second transition does not involve p , and the third one advances p again. From [Theorem 6.1](#) on page 88, we know that the CoUP model with solely assembly processes simulates these VAMOS transitions, as shown above. The yet missing link is a rearrangement of transitions in CoUP such that the transitions Δ_{asm}^p are adjacent. Below, we work out the technical details that allow us to establish this link.

6.2.1 On the Commutability of Transitions: From Trace Theory to Kleene Algebra

Usually, trace theory [[Maz87](#)] is employed to describe the commutability of transitions in the execution sequences of concurrent state transition systems. In short, trace theory defines equivalence classes over transition sequences based on an independency relation. This binary relation I contains commutable transition pairs, i. e., $\forall(\Delta_1, \Delta_2) \in I. \Delta_1 \circ \Delta_2 = \Delta_2 \circ \Delta_1$. Non-commutable or dependent transitions typically include locking operations or synchronization points. Trace theory simply states equivalence for all traces that result from permutations of commutable transitions.

The immediately arising question is whether the transitions Δ_{asm}^p and $\Delta^{\bar{p}}$ in [Figure 6.1](#) on the preceding page are commutable. Unfortunately, this is not necessarily the case. In general, the process and kernel transitions are dependent because the input value for a process transition resembles information about the kernel state, on the one hand, and the kernel transitions use the output functions of the processes to gain information about the process states, on the other hand.

For a process transition, we can determine from the process output whether the transition requires information on the current kernel state. An internal transition $\delta_{proc} \varepsilon_{\Sigma}$ is independent from the kernel state and requires an output of ε_{Ω} (see the process-validity rule ([no_input_valid](#)) on page 44). All other process transitions require some information about the kernel state.

Respectively, the kernel gains information about a process solely via its output functions ω_{proc} and $size_{proc}$. From ([size_const](#)) on page 42, we know that the size remains constant under all inputs except for `ADD_MEMORY pages` and `FREE_MEMORY pages`. The output ω_{proc} , on the contrary, might change under any process transition.

Consequently, the suggested rearrangement in the illustration cannot be substantiated with trace theory. A closer look, however, reveals a resort: The key observation is that the process output ε_{Ω} inhibits kernel interaction while other outputs open up the *possibility* for interaction – but cannot enforce it. In other words, bringing an internal process transition forward, may extend but never restricts the set of possible traces.

Trace theory establishes an equivalence or bisimilarity relation though in our context, a plain simulation is sufficient. Recall that we have shown that CoUP simulates VAMOS while we know that VAMOS does not simulate CoUP. Hence, it is sufficient to

show that permutations preserve CoUP's simulation of VAMOS while it does not harm if additional CoUP transitions become possible after permutation. Conceptually, we define an independency relation I such that $\forall(\Delta_1, \Delta_2) \in I. \Delta_1 \circ \Delta_2 \subseteq \Delta_2 \circ \Delta_1$.¹ If we swap a transition Δ_1 and a consecutive transition Δ_2 in a CoUP trace, we preserve the simulation of VAMOS. Note that the commutations are not reversible, i. e., without loss of generality, we may not swap Δ_2 and Δ_1 if Δ_2 precedes Δ_1 .

6.2.2 Process Simulation Meets True Concurrency

After we have clarified the theoretical background for the permutation of transitions, we take a closer look on the transitions to be commuted for process simulation.

From a process-isolated perspective, the situation has been depicted in Figure 4.2 on page 58. As we have seen at the beginning of this section, the situation becomes more involved when we regard the transitions of a process in the context of the true-concurrent kernel execution. The kernel-centered perspective of this situation has been visualized by the upper two transition sequences of Figure 6.1 on page 90. Figure 6.2 connects both illustrations, extending the former to complete kernel states and transitions, and refining the latter with details on the involved process transitions:

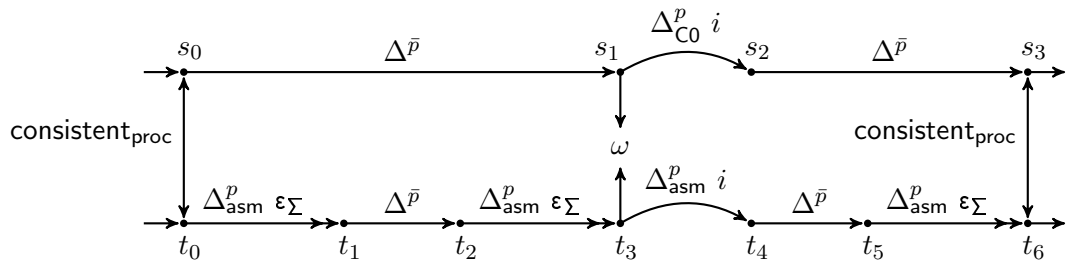


Figure 6.2: Process simulation in a concurrent CoUP trace

The illustration shows two CoUP traces. The states s_i contain an active C0 process with PID p while the states t_i contain a corresponding assembly process at p . The state of process p in s_0 simulates the one of p in t_0 . We assume that p yields some output ω in s_0 . The process state remains constant under all subsequent CoUP transitions $\Delta^{\bar{p}}$ that do not involve p . From Figure 4.2 on page 58, we know that the assembly process p in t_0 might require several internal transitions (here labeled $\Delta_{asm}^p \varepsilon \Sigma$) until it reaches in t_3 a state yielding the same output. In a CoUP trace, transitions $\Delta^{\bar{p}}$ might interleave with the internal process transitions. From the states s_1 and t_3 , respectively, the process p advances under an input i suitable to the output ω . Note that without loss of generality, the depicted transitions $\Delta_{C0}^p i$ and $\Delta_{asm}^p i$ might alter the whole kernel state, not just

¹Our approach draws upon Kleene algebras (see Section 1.2.1). We define the according algebra over the universe $\mathcal{S}_{VC} \times \mathcal{S}_{VC}$ where addition is set union, multiplication is the relational composition, 0 is the empty set and 1 is the identity relation. We adopt the subset relation as partial ordering and define the Kleene star as the reflexive transitive closure.

the process-local state. Further internal assembly transitions might again interleave with transitions $\Delta^{\bar{p}}$ until simulation of process p is established between s_3 and t_6 .

Concluding, we identify two situations where an interleaving of process and kernel transitions might occur: Between internal transitions of an assembly process between $\text{consistent}_{\text{proc}}$ in t_0 and the state yielding the appropriate output ω in t_3 as well as after the process-kernel interaction and before $\text{consistent}_{\text{proc}}$ is established again, i. e., between the states t_4 and t_6 . Below, we show that internal process transitions can safely be commuted with subsequent kernel transitions not involving p .

6.2.3 Commuting Transitions in CoUP

So far, we have been quite unspecific about the kernel transitions $\Delta^{\bar{p}}$ that do not involve the considered process p . Now, we draw our attention to this neglect and examine the CoUP transitions more closely to identify commutable transitions. A first observation is that a process might advance several times during one Δ_{VC} transition, for instance in $\Delta_{\text{VC}}^{\text{cup}}$ and in $\Delta_{\text{VC}}^{\text{to}}$. Hence, we inspect the individual transition phases $\Delta_{\text{VC}}^{\text{cup}}$, $\Delta_{\text{VC}}^{\text{to}}$, $\Delta_{\text{VC}}^{\text{ints}}$ and $\Delta_{\text{VC}}^{\text{sched}}$. Furthermore, we implicitly assume in the remaining subsection that the invariant inv_{VC} holds.

An internal process transition might only take place as part of $\Delta_{\text{VC}}^{\text{cup}}$ if (a) the process is active and (b) its output is ε_{Ω} . In the remainder, we implicitly assume that these preconditions hold for process p . Additionally, $\Delta_{\text{VC}}^{\text{cup}}$ requires that the process is the current one. We simplify matters and define a transition relation Δ_{ε}^p that extends an internal process transition to the whole kernel state independently from the current process. This relation resembles the transitions labeled $\Delta_{\text{asm}}^p \varepsilon_{\Sigma}$ from [Figure 6.2](#) on the facing page:

$$\Delta_{\varepsilon}^p \equiv \{(s, s') \mid \exists s_{\text{proc}}. s.\text{procs } p = \lfloor s_{\text{proc}} \rfloor \wedge \omega_{\text{proc}} s_{\text{proc}} = \varepsilon_{\Omega} \wedge s' = s(\text{procs} := s.\text{procs}(p \mapsto \delta_{\text{proc}} \varepsilon_{\Sigma} s_{\text{proc}}))\}$$

This relation refines $\Delta_{\text{VC}}^{\text{cup}}$ if the process p is the current one:

$$s.\text{cup} = \lfloor p \rfloor \implies (s, s') \in \Delta_{\text{VC}}^{\text{cup}} \text{ mifo} \iff (s, s') \in (\Delta_{\varepsilon}^p)^{=}$$

where $R^=$ denotes the reflexive closure of relation R . For the equality of $\Delta_{\text{VC}}^{\text{cup}}$ and Δ_{ε}^p , we cannot neglect the reflexivity of $\Delta_{\text{VC}}^{\text{cup}}$ but when considering permutations below, we ignore it because the identity relation is trivially commutable. Note that $\Delta_{\text{VC}}^{\text{cup}}$ uses the parameter *mifo* only for device communication, hence we omit it for internal steps Δ_{ε}^p .

If, in contrast, the current process is not p , an internal transition of p may be swapped with a subsequent $\Delta_{\text{VC}}^{\text{cup}}$:

$$\llbracket s.\text{cup} \neq \lfloor p \rfloor; (s, s') \in \Delta_{\text{VC}}^{\text{cup}} \text{ mifo} \circ \Delta_{\varepsilon}^p \rrbracket \implies (s, s') \in \Delta_{\varepsilon}^p \circ \Delta_{\text{VC}}^{\text{cup}} \text{ mifo}$$

Informally, this claim follows from a simple observation: The VAMOS specification considers the output of a process in two situations: The output of the current process determines the kind of its computation, and in case of an IPC call, VAMOS additionally

checks the output of the waiting processes for a possible rendezvous. Immediately after an internal transition, however, a process does not wait. From [Lemma 6.2](#) on page 89, we know that $\Delta_{\text{VC}}^{\text{cup}}$ simulates this behavior of the VAMOS call dispatcher. Hence, $\Delta_{\text{VC}}^{\text{cup}}$ includes the possibility to disregard a potentially changed output of p . For the formal proof of this claim, however, we have regarded the 33 rules introducing $\Delta_{\text{VC}}^{\text{cup}}$ one by one.

Additionally, we have formalized similar facts about swapping internal process transitions with subsequent other transition relations. We claim

$$(s, s') \in \Delta_{\text{VC}}^{\text{sched}} \circ \Delta_{\epsilon}^P \implies (s, s') \in \Delta_{\epsilon}^P \circ \Delta_{\text{VC}}^{\text{sched}}$$

because $\Delta_{\text{VC}}^{\text{sched}}$ is solely concerned with the current process while Δ_{ϵ}^P completely neglects it. The transitions $\Delta_{\text{VC}}^{\text{to}}$ and $\Delta_{\text{VC}}^{\text{ints}}$ only affect processes pending in an IPC operation. Though after an internal transition, a process might potentially be pending in IPC, the definitions of $\Delta_{\text{VC}}^{\text{to}}$ and $\Delta_{\text{VC}}^{\text{ints}}$ include the possibility of an unchanged process. Hence:

$$\begin{aligned} (s, s') \in \Delta_{\text{VC}}^{\text{ints}} \text{ ints} \circ \Delta_{\epsilon}^P &\implies (s, s') \in \Delta_{\epsilon}^P \circ \Delta_{\text{VC}}^{\text{ints}} \text{ ints} \\ (s, s') \in \Delta_{\text{VC}}^{\text{to}} \circ \Delta_{\epsilon}^P &\implies (s, s') \in \Delta_{\epsilon}^P \circ \Delta_{\text{VC}}^{\text{to}} \end{aligned}$$

Consequently, we may rearrange the internal process transitions in an interleaved sequence of kernel transitions such that the internal transitions precede all remaining transitions. Recall, however, that this conclusion relies on our implicit assumption that the process p remains active and its output ϵ_{Ω} . Fortunately, most transitions completely disregard processes with an output of ϵ_{Ω} :

$$\begin{aligned} (s, s') \in \Delta_{\text{VC}}^{\text{ints}} \text{ ints} &\implies s'.\text{procs } p = s.\text{procs } p \\ (s, s') \in \Delta_{\text{VC}}^{\text{to}} &\implies s'.\text{procs } p = s.\text{procs } p \\ (s, s') \in \Delta_{\text{VC}}^{\text{sched}} &\implies s'.\text{procs } p = s.\text{procs } p \end{aligned}$$

We know these facts because $\Delta_{\text{VC}}^{\text{ints}}$ and $\Delta_{\text{VC}}^{\text{to}}$ only modify processes calling for IPC while $\Delta_{\text{VC}}^{\text{sched}}$ is solely concerned with `cup`. In spite of that, process p might certainly be killed by another process (see [Section 5.3](#)). Otherwise, however, $\Delta_{\text{VC}}^{\text{cup}}$ does not alter the process state with an output of ϵ_{Ω} . Formally:

$$\llbracket [s.\text{cup} \neq [p]]; (s, s') \in \Delta_{\text{VC}}^{\text{cup}} \text{ mifo} \rrbracket \implies s'.\text{procs } p = \perp \vee s'.\text{procs } p = s.\text{procs } p$$

We prove this claim by a case distinction over the output of the current process and expect the changed process states in each case. Recall that there are only two situations, where a process state might change: when being the current process or while issuing an IPC call.

The possibility of process termination gives rise to another consideration: When a C0 process is compiled, the resulting assembly process might be terminated while executing the instructions of a single C0 statement. In [Figure 6.2](#) on page 92, p features the output ω already in state s_0 while the compiled process at the bottom could be terminated in t_2 , thus never featuring that output. Thus, we investigate the insertion of internal process transitions. Formally, we state:

$$\begin{aligned} (s, s') \in \Delta_{\text{VC}}^{\text{cup}} \text{ mifo} \cap \{(x, y). y.\text{procs } p = \perp\} &\implies \\ (s, s') \in \Delta_{\epsilon}^P \circ \Delta_{\text{VC}}^{\text{cup}} \text{ mifo} \cap \{(x, y). y.\text{procs } p = \perp\} &\end{aligned}$$

i. e., we constrain $\Delta_{\text{VC}}^{\text{cup}}$ such that it terminates p . If so, we may insert an internal process transition because process termination is generally independent from the process output (see [Section 5.3](#)).

6.3 Conclusion

Summarizing, this chapter presented two results: first, a simulation between the kernel models with assembly processes, and second, the necessary lemmata about possible reorderings of internal process transitions that allow us to finally embed C0 processes into the kernel models.

The simulation proof between the kernel models ([Theorem 6.1](#) on page 88) substantially benefitted from the prior harmonization of the involved models (cf. [Chapter 5](#)). The remaining proof effort was mainly caused by the gained symmetry for IPC in CoUP. In the retrospective, we suspect further improvement opportunities in both, the proof development and the specification of the kernel models.

Our proof development suffered from repeated set-backs when the VAMOS invariant inv_{V} proved to be too weak. Instead of starting with a minimal invariant, which is strengthened by need, one could start with a stronger formulation and withdraw unneeded propositions after the simulation proof (and before the proof of invariance). Originally, we assumed that in the latter approach, the risk of overestimating the invariant would be too high. Our experience, however, is contrary: The risk of under-estimating the minimally required invariant is much higher than overestimating the maximal invariant. A recently appeared case study [[CKS08](#)] in kernel verification suffered from the same problem and came to the same conclusion.

Additionally, we suspect that further optimization of the IPC specification in both VAMOS models might have reduced the proof effort. An important, first step in this direction has been the specification of the message transfer independently from the current process (see [Section 5.4.2](#)). This optimization has greatly simplified the proof. Unfortunately, there is no case study that could evaluate the gained verification speed.

Ultimately, we can take the idea of harmonizing the two kernel models even a step further and ask whether a single kernel model could have sufficed for both purposes. In the hindsight, we acknowledge that this approach would have been generally possible. Mainly project-pragmatic reasons led to two different kernel models. At the same time, we like to point out that specifying a single model serving different purposes requires utmost care. From the amount of adjustments at both models during our work, we can imagine that the same effort as for the simulation proof could have easily been spent in the specification work for a single model serving both purposes.

Though initially motivated by the embedding of C0 processes, the lemmata about reorderings of process transitions ultimately allow us to prove properties over all possible kernel traces while confining our consideration to a substantially smaller, representative subset, where rescheduling only takes place at kernel calls. The general course of action has been described by Alkassar *et al.* [[ABP09](#), [Alk09](#)] for a similar setting though in our

case, the reorderings are not reversible, i. e., we cannot establish equivalence classes of permutations.

7 A Temporal Property: Scheduler Fairness

Contents

7.1 An Introductory Scheduling Example	97
7.2 Common Notions of Fairness	98
7.3 Developing and Proving Prioritized Fairness	99
7.4 Rephrasing the Temporal Property in LTL	103
7.5 Conclusion	104

We start this chapter by an introductory example illustrating the operation of the scheduler in a VAMOS trace. We continue with a comparison of different fairness notions found in the literature. In the major part of this chapter, we develop our notion of *prioritized fairness*, which precisely reflects the policy of the VAMOS scheduler, and prove that our scheduler conforms with this definition. Concluding, we lift our result to a thin layer of temporal logic that provides the succinct notation usually employed for temporal properties.

7.1 An Introductory Scheduling Example

Before we descend down into the formal details, we illustrate the operation of our scheduler. [Figure 7.1](#) on the following page shows an example trace. The matrices at the top represent the ready queues and the wait queue of VAMOS. Our scheduler always elects the first process in the highest, non-empty ready queue for computation. In the first shown step, two processes are in the highest priority class, and process 1 is computing (as shown at the bottom) because it is the first in the highest ready queue. The elected process runs (at least) until it generates an exception or an external interrupt occurs. In the example, we assume that process 1 issues an IPC call and then has to wait for a suitable partner such that process 2 is computing in the second step.

A timer interrupt occurs (indicated by \downarrow) and process 2 is charged for its computation in step 2, i. e., the counter `[s.schedds.procdb (process 2)].ctsl` is increased. Process 2, however, is the only one in its ready queue and continues to compute in step 3. Now, we assume that process 2 issues an open IPC receive and is found waiting in step 4 while process 3 computes. In step 5, a timer interrupt occurs, which charges process 3. For this example, we assume that all processes have a single timeslice—hence, process 3 will be preempted as its timeslice ran out. Moreover, we assume that the scheduler wakes

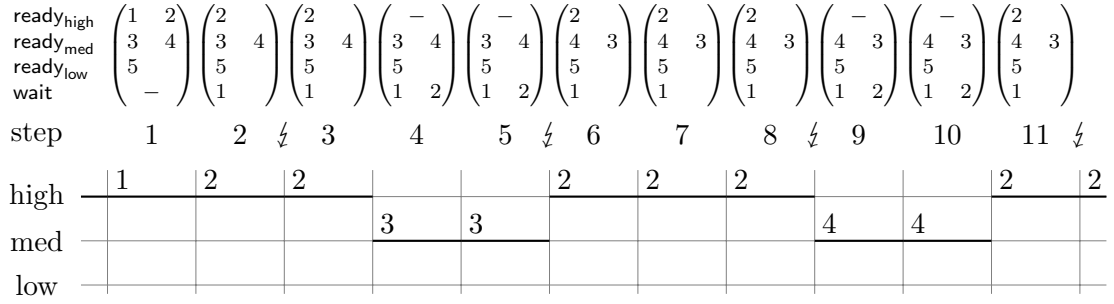


Figure 7.1: An exemplary execution trace of VAMOS together with the scheduling queues

up process 2 because of an elapsed timeout. Now, process 2 continues to compute until step 8, where it again issues an IPC call and starts waiting. Later on, we return to this example.

7.2 Common Notions of Fairness

There are various notions of fairness. In the operating-system community, fairness of static-priority scheduling is typically measured by the ratio of the CPU usage among equally prioritized processes. Fagin & Williams [FW83] illustrate quantitative fairness by the *carpool problem*: A number of persons form a carpool. Each day, some of them arrive to share in the ride. A fair scheduling algorithm is required that determines for any given day, which one of the arrived persons should drive the car.

Fagin & Williams propose that each person sharing in a ride should owe the driver the reciprocal of the number of sharers. They call this debt the *degree of unfairness* and claim that it is fair to always schedule the poorest sharer as the driver. Applied to process scheduling, the degree of unfairness is computed as follows: If a process is chosen from a ready queue with length n , we subtract $1 - 1/n$ from its debt and add $1/n$ to the debt of each unchosen process in the queue. Alternatively, Naor [Nao05] suggests a shuttle-bus fare, i. e., a fixed price is assigned to the CPU usage, regardless of the current demand. The VAMOS scheduler adopts this notion and measures unfairness by the time that a process has resided in the ready queue since it were most recently chosen.

In temporal logics, fairness refers to a liveness property. In his taxonomy, Francez [Fra86] differentiates three forms of temporal fairness: *Unconditional fairness* requires that each process is infinitely often scheduled. Formally: $\Box \Diamond \text{scheduled}$. *Strong fairness*, in contrast, formulates the same requirement only for processes that are infinitely often ready to compute, i. e., $\Box \Diamond \text{enabled} \longrightarrow \Box \Diamond \text{scheduled}$. Finally, *weak fairness* only claims that processes are infinitely often scheduled if eventually, they forever remain ready, i. e., $\Diamond \Box \text{enabled} \longrightarrow \Box \Diamond \text{scheduled}$. As we show below, the VAMOS scheduler features a strong fairness property.

Certainly, it is desirable to predict at least the waiting time of a certain process until it finally computes but this task is hard to achieve. In principle, the waiting time of the

processes in the highest priority class is linear in the number of processes and an upper bound for the waiting time could be given. For real-time operating systems [KP07] with a fixed process schedule, static program analysis [HF04] can reliably predict an upper bound for the worst-case execution time of programs. Recollect, however, that we use VAMOS in a computer system for e-mail exchange, where keyboard inputs and network packages arrive non-deterministically. Depending on the pressed key, even a new process might be launched. In this scenario, static program analysis cannot show its advantages. Measurements [SP07, PZH07] can better cope with non-determinism and produce considerably lower time bounds but are not exhaustive and might thus miss an unlikely worst case.

Besides the difficulty to prove a quantitative fairness property, the temporal property simply suffices for our original motivation: We can lift statements about the termination of user programs into the concurrent environment of kernel executions. Thus, even total correctness of a process in the concurrent environment can be established without a detailed timing analysis.

7.3 Developing and Proving Prioritized Fairness

Liveness properties are defined over traces. Thus, based on the definition of VAMOS traces on page 81, an unconditional fairness property is:

$$\exists l \geq k. \text{progress} [(states\ l).procs\ pid] [(states\ (l + 1)).procs\ pid]$$

where k is an arbitrary, fixed step number, pid is an arbitrary, fixed process ID, and the predicate $\text{progress } p\ q$ holds iff q can be produced by a single transition¹ from p , i. e., $\exists i. q = \delta_{\text{proc } i}\ p$. Note that the requirement of a single transition for progress is significant: An arbitrary transition sequence might pass twice through the same state. If progress were defined as the transitive closure over transitions, the progress relation would become reflexive for all such states.

This statement, however, is too strong for the VAMOS scheduler because it neglects several aspects of our system:

- Processes are dynamic entities and PIDs may be reused. Thus, the process pid in step l might not be the same as the one in step k (see Section 5.3).
- A process pid might start an IPC operation with an infinite timeout (see Section 3.3). If there is never another process willing to communicate with process pid , the latter starves.
- Prioritization leads to the preemption of less prioritized process classes – without any time bound (according to our scheduling policy as described in Section 3.2.2).
- The implementation relies on a live timer device because the scheduler is activated only by the timer interrupt (see Section 5.1).

¹Technically, the matter is complicated by the fact that the amount of process memory is held with the processes. Transitions changing only this amount, are certainly not considered as progress.

Consequently, we have to weaken our notion of fairness in this context. First, we recall that liveness conditions are formulated over infinite transition sequences. Hence, we exclude all terminating processes, i. e., we assume $\forall n \geq k. (\text{states } n).\text{procs } pid \neq \perp$.

Second, we define a predicate `pending_infinite_ipc s pid`, which examines a state s and holds iff the process pid is pending in an IPC operation with an infinite timeout. If a process remains forever in this state, it is certainly not the fault of the scheduler but a programming or protocol error. Thus, we do not consider such starving processes, i. e., we require $\forall n \geq k. \exists m \geq n. \neg \text{pending_infinite_ipc } (\text{states } m) pid$.

Third, the priority of a process is runtime-configurable by the kernel call `CHG_SCHED_PARAMS`. Upon a priority change, the process will be removed from its old priority's ready queue and *appended* to the new one. As long as the priority is only configured at the beginning (or more precisely: changed only a finite number of times), we can find an infinite subtrace without a priority change. If, however, there are infinitely many changes, the process may starve. We exclude this case because it contradicts the concept of static-priority scheduling, i. e., we require: $\forall n \geq k. (\text{states } (n + 1)).\text{priodb } pid = (\text{states } n).\text{priodb } pid$.

Fourth, we assume that the timer device produces infinitely often timer interrupts as part of the inputs, i. e., $\forall n \geq k. \exists m \geq n. \text{is_timer_on } (\text{inputs } m)$ where predicate `is_timer_on` indicates a raised timer interrupt.

So far, our restrictions have been evident adjustments to the considered system. The remaining problem, which regards priorities, is somewhat more involved. Let us consider a process pid in the highest priority class, i. e., $(\text{states } k).\text{priodb } pid = \text{[HIGH]}$. With the assumptions above, we can show that this process pid will eventually make progress, formally:

$$\exists l \geq k. \text{progress } [(\text{states } l).\text{procs } pid] [(\text{states } (l + 1)).\text{procs } pid]$$

This formulation of fairness, however, states nothing about the processes residing in lower priority classes. A statement over fairness for the latter necessarily depends on the (intermittent) absence of a process in a higher priority class, which is ready to compute. All processes, which are willing to compute, are enqueued in the ready queue of its priority. Thus, we can determine the absence of a prioritized, ready process by examining the corresponding ready queues, i. e., process pid has the maximal priority in state s if

$$\forall prio > [s.\text{priodb } pid]. s.\text{schedds}.\text{ready } prio = []$$

We encapsulate this property in a predicate `has_maxprio s pid`.

In order to extend our fairness statement to all processes, we examine our exact scheduling mechanism more carefully: The currently computing process p_{cup} is charged (by increasing its consumed time $[s.\text{schedds}.\text{procdb } p_{\text{cup}}].\text{ctsl}$) and eventually preempted if *and only if* the timer interrupt occurs. In other words, a process must have the maximal priority infinitely often *while* the timer interrupt occurs, i. e.,

$$\forall n \geq k. \exists m \geq n. \text{has_maxprio } (\text{states } m) pid \wedge \text{is_timer_on } (\text{inputs } m)$$

Note that this assumption implies timer liveness.

The execution trace shown in Figure 7.2 illustrates this problem: In step 8, process 2 computes and issues an IPC receive. When the timer interrupt occurs, the scheduler recognizes that process 2 has been computing at the last step. Process 4 starts to compute in step 9 and sends a message to process 2 in step 10, which causes the latter to wake up. Thus, process 2 is found computing when the timer arrives in step 11. If the sequence of events between step 8 and 11 continues, process 4 will never get charged and process 3 starves although it has frequently the maximal priority.

ready _{high}	(2)	(-)	(-)	(2)
ready _{med}	(4 3)	(4 3)	(4 3)	(4 3)
ready _{low}	(5)	(5)	(5)	(5)
wait	(1)	(1 2)	(1 2)	(1)
step	8	↯ 9	10	11 ↯
high	2			2
med		4	4	
low				

Figure 7.2: A VAMOS trace with a race condition

Of course, it is possible to change the scheduling mechanism such that all processes found running in between two timer interrupts are charged. This code change, however, would require extra bookkeeping during the IPC path, which is the most critical one for system performance. Most L4 kernels even completely skip the scheduler for performance reasons after some IPC operations [Ruo06, EGR07]. Hence, we favored the shorter path over an optimized scheduling.

It is crucial to understand the implications of this design choice, in particular, the possible exploits by malicious processes and the mechanisms to its prevention by prioritized processes. For a working exploit, the user process needs a clock of higher precision than the timer tick, which can only be achieved by another external timer device. If we exclude this unlikely setting, the process cannot predict the time when the tick arrives and the consistent recurrence of the race condition becomes exponentially unlikely.

An informal statement of likeliness, however, is not satisfactory to an assumption of a formal proof. Hence, we examine the necessary preconditions for the race condition. There are three possibilities for a switch to higher priority between timer ticks: (a) A process of higher priority might be created, (b) the scheduling priority of an existing process might be raised, or (c) an IPC call might resume a prioritized, waiting process. The former two are caused by kernel calls reserved for privileged processes. These processes are typically most prioritized and have to be trusted anyways. The latter requires the collaboration of the prioritized process. We regard this situation as a special form of priority inheritance [Ruo06].

Concluding, we state:

Lemma 7.1 (Prioritized Fairness). *The VAMOS scheduler is fair with respect to priority classes, i. e., for all suffixes of an infinite trace, all processes will eventually make progress, iff they (a) never terminate, (b) remain forever in the same priority class, (c) do not starve in an IPC operation with an infinite timeout, and (d) have infinitely often the maximal priority while the timer interrupt is raised.*

We formalize this claim as:

$$\llbracket \text{states } 0 \in \mathcal{S}_V^0; \forall n. \text{states } (n + 1) = \delta_V(\text{inputs } n) (\text{states } n); \quad (\text{trace})$$

- $$\begin{aligned} \forall n \geq k. (\text{states } n).\text{procs } pid &\neq \perp; & (a) \\ \forall n \geq k. (\text{states } (n + 1)).\text{priodb } pid &= (\text{states } n).\text{priodb } pid; & (b) \\ \forall n \geq k. \exists m \geq n. \neg \text{pending_infinite_ipc } &(\text{states } m) pid; & (c) \\ \forall n \geq k. \exists m \geq n. \text{has_maxprio } (\text{states } m) &pid \wedge \text{is_timer_on } (\text{inputs } m)] & (d) \\ \implies \exists l \geq k. \text{progress } [(\text{states } l).\text{procs } &pid] [(\text{states } (l + 1)).\text{procs } pid] \end{aligned}$$

Proof. We prove the theorem by case distinction. From the invariant inv_V (cf. [Section 5.6](#)) we know that a process pid can either be inactive, waiting, or ready. The first case immediately contradicts [Assumption \(a\)](#).

If a process is waiting, we can infer from the invariant inv_V that the process is aspiring an IPC operation and that there is no suitable partner ready for communication (cf. predicate wait_imp_IPC).

Let us assume that the process forever remains in the wait queue. If the process has issued an infinite IPC call, we have a contradiction with [Assumption \(c\)](#). If, however, a finite timeout has been specified with the call, the timeout will eventually elapse. We know this because of timer liveness and the fact that δ_V invokes the scheduler (see [Section 5.5](#)) whenever the timer interrupt occurs. Upon each invocation, the scheduler increases the variable time and checks for elapsed timeouts. Once, the timeout is elapsed, the process is dequeued from the wait queue (see v_manage_timeouts on page 80).

Thus, we know that the process cannot remain forever in the wait queue. The transition function δ_V specifies exactly two cases, where a process is dequeued from the wait queue: (a) there is a partner issuing a kernel call for IPC or (b) the operation times out. In both cases, VAMOS responds to the process with a result value that indicates success or failure. Formally: The transition function δ_V involves an update of the process $s.\text{procs } pid$ by $\delta_{asm} \text{ res } [s.\text{procs } pid]$ with the result value res —our definition of progress .

The remaining case is that process pid is ready at step k . We know from inv_V that ready processes reside in the ready queue corresponding to their priority. When examining δ_V , we can observe that a process will be dequeued from a ready queue only if (a) it becomes inactive, (b) its priority changes, or (c) it has been computing. The first two cases lead to a contradiction with our Assumptions [\(a\)](#) and [\(b\)](#).

Computing usually means that the process immediately makes progress. Most notably, our implementation guarantees liveness, i. e., a started user process performs at least one step between two subsequent kernel entries (first phase in δ_V). Still, there might be no immediate progress if a computing process calls the kernel for an IPC operation. In this case, however, the kernel will enqueue the process in the wait queue, and we have shown fairness for all processes in this queue.

Finally, we regard the case that a process is never dequeued from the ready queue. Together with [Assumption \(d\)](#), we can infer that the process will move forward in the ready queue until it is the first one: The assumption ensures that the process has the current maximum priority infinitely often while the timer interrupt is active. When the timer interrupt occurs, the scheduler is invoked, and it charges the current process p_{cup} , i. e., the first process in pid 's ready queue (as specified by v_charge_process on page 79).

Charging a process means increasing the consumed time $[s.\text{schedds.procdb } p_{cup}].\text{ctsl}$ unless it exceeds the timeslice, and moving the process to the end of its ready queue

if the timeslice is exceeded. Thus, the consumed time value of p_{cup} increases strictly monotonic as long as the current process remains the first in its ready queue. Moreover, timeslices are bounded by a fixed value because they are stored in a 32-bit number in the implementation (cf. predicate `bounded_timeslices` on page 82). Hence, the consumed time eventually exceeds the timeslice, and p_{cup} is moved to the end of its ready queue. That means, p_{cup} appears *after* pid in the ready queue.

By induction, we can infer that process pid will eventually be the first in its ready queue and thus, when it eventually has the maximum priority, it is the current process. The current process computes in the next step and we have already shown that a computing process eventually makes progress. \square

7.4 Rephrasing the Temporal Property in LTL

So far, the formal statement requires explicit quantification on step numbers over the states and inputs functions. We have proven the theorem on this layer for methodical reasons: Recall that the larger context of this work is pervasive systems verification with the long-term goal of a coherent theory in a single theorem prover. Consequently, a specialized theorem prover for temporal logic is insufficient in this context. Nevertheless, temporal properties like our prioritized fairness are usually described in temporal logics. The advantage of such a logic is succinctness while the Isabelle/HOL formalization (Lemma 7.1 on page 101) is crucial for the property transfer. We have combined the best of both worlds by an embedding of future-time linear temporal logic (LTL) with action Kripke structures into Isabelle/HOL (see Section 1.2.2). Thus, we can integrate our main theorem into the overall context while presenting it in LTL:

Theorem 7.2 (Prioritized Fairness). *The VAMOS scheduler is fair with respect to priority classes, i. e., if a process pid (a) finally never terminates, (b) finally remains forever in the same priority class, (c) is infinitely often not pending in an IPC operation with an infinite timeout, and (d) has infinitely often the maximal priority while the timer interrupt is raised, it will always eventually progress.*

Formally:

$$\begin{aligned}
 \langle \mathcal{S}_V^0, \hat{\Sigma}, \delta_V \rangle_A \models & \diamond \square (\lambda(i, s, n). \text{s.procs } pid \neq \perp) \longrightarrow & (a) \\
 & \diamond \square (\lambda(i, s, n). \text{n.priodb } pid = \text{s.priodb } pid) \longrightarrow & (b) \\
 & \square \diamond (\lambda(i, s, n). \neg \text{pending_infinite_ipc } s \text{ } pid) \longrightarrow & (c) \\
 & \square \diamond (\lambda(i, s, n). \text{has_maxprio } s \text{ } pid \wedge \text{is_timer_on } i) \longrightarrow & (d) \\
 & \square \diamond (\lambda(i, s, n). \text{progress } [s.\text{procs } pid] \ [n.\text{procs } pid])
 \end{aligned}$$

Proof. We deduce the theorem from Lemma 7.1 on page 101 mainly by the expansion of the LTL definitions. Note, however, that Lemma 7.1 on page 101 universally quantifies over a single suffix while in LTL, Assumption (a), Assumption (b), and the conclusion might hold for different suffixes. We instantiate the position k in the lemma with the maximal position of the three suffixes and derive our claim. \square

According to Francez [Fra86], strong fairness is formalized in LTL as: $\square \diamond \text{enabled} \longrightarrow \square \diamond \text{scheduled}$, i. e., if a certain process is infinitely often ready to compute (LTL-predicate

enabled), it will always eventually compute (*scheduled*). Technically, our theorem has a different structure.

If we examine the Assumptions (a) and (b), however, we observe that these assumptions are of a very basic kind: It is certainly impossible to prove temporal fairness for a terminating process, and a changed priority contradicts the notion of *static*-priority scheduling. We maintain that any real system has similar basic assumptions and do not consider these conditions as part of the *enabledness*.

We may confine our consideration to a system with a static number of processes and without priority changes, i. e., after a time of system set up, the kernel calls `PROCESS_KILL` and `CHG_SCHED_PARAMS` are no longer used. In this scenario, the Assumptions (a) and (b) become *set-up conditions* and we just consider the state after set up as the initial state. In this system, we have only two enabledness assumptions of the kind $\Box\Diamond\textit{enabled}$. This artifact is important because the conjunction of both enabledness assumptions would be considerably stronger—resulting in a significantly weaker theorem.

Though temporal properties are naturally based on infinite traces, the complete exclusion of terminating processes might irritate – most notably, because a process might terminate itself. In this case, termination should certainly be considered as some sort of progress. Fortunately, we can rephrase [Theorem 7.2](#) on the previous page as follows:

$$\begin{aligned}
 \langle \mathcal{S}_V^0, \hat{\Sigma}, \delta_V \rangle_A \models \Diamond\Box(\lambda(i, s, n). n.\text{priodb } pid = s.\text{priodb } pid) &\longrightarrow (b) \\
 \Box\Diamond(\lambda(i, s, n). \neg \text{pending_infinite_ipc } s \text{ } pid) &\longrightarrow (c) \\
 \Box\Diamond(\lambda(i, s, n). \text{has_maxprio } s \text{ } pid \wedge \text{is_timer_on } i) &\longrightarrow (d) \\
 \Box\Diamond(\lambda(i, s, n). s.\text{procs } pid = \perp \vee \text{progress } [s.\text{procs } pid] [n.\text{procs } pid]) &
 \end{aligned}$$

Note that this formulation is logically equivalent to the former: The inequality in [Assumption \(a\)](#) can be written as negated equality. By definition, $\Diamond\Box\neg\varphi$ is semantically equivalent to $\neg\Box\Diamond\varphi$, an implication with a negated premise is equivalent to a disjunct, and $\Box\Diamond\varphi \vee \Box\Diamond\psi$ is equivalent to $\Box\Diamond(\varphi \vee \psi)$.

7.5 Conclusion

During verification, we found (and fixed) a bug in the temporal behavior of VAMOS, which can easily be overlooked in a step-wise refinement proof: Previously, CVM handled page faults and device interrupts at the same time. In that case, a frequently arriving timer interrupt might inhibit any progress of processes.

The reason for this problem is found in the system’s architecture: Recall that CVM handles page faults transparently, i. e., not even the C0 layer of the kernel notices a page fault. Device interrupts, on the contrary, are passed on to the C0 layer. Furthermore, note that a page fault is a so-called repeat interrupt, i. e., the last instruction could not successfully be executed and should be repeated when the necessary page has been swapped in. Device interrupts, in contrast, are so-called continue interrupts, i. e., they do not interfere with the execution of instructions. If a device interrupt and a page fault occur simultaneously and are handled at the same time, the processor might not

have successfully executed any user-mode instruction since the last device interrupt was handled.

Recall that the VAMOS scheduler uses the timer interrupt to measure process run-times. Thus, if the timer interrupt occurs too frequently, a process might be charged for the computation of zero steps. The exact meaning of “too frequent” occurrences, however, depends on a detailed timing analysis. Handling a page fault, for instance, takes about 14,000 system-mode instructions and up to two page faults might occur during the execution of a single user-mode instruction. Furthermore, a scheduler invocation might involve almost 75,000 instructions (depending on the length of the wait queue). Certainly, the individual execution time of an instruction heavily depends on other hardware properties like the current cache status.

We have decided to solve this problem by a much simpler solution: We require that CVM is live, i. e., between two calls into the C0 layer, the processor must execute at least one step in user mode. This requirement is realized by ignoring device interrupts if they occur simultaneously with a page fault.

In conclusion, this finding demonstrates that bugs are not necessarily revealed during the refinement proof of a single layer.

Furthermore, our proof can be integrated with a refinement proof of Jan Dörrenbächer [DDW09, Sect. 6], which links the VAMOS specification to the C0 implementation. To our knowledge, this is the first verification result that establishes a temporal property on a fairly realistic microkernel implementation.

8 Towards Verifying a User-Mode Operating System

I know that I do not know.

Socrates, according to Platon: "The Apology of Socrates"

Contents

8.1 Non-Termination	108
8.2 Fair Scheduling of Applications	110
8.2.1 Priorities in the SOS	110
8.2.2 Applying Scheduler Fairness to Applications	110
8.3 Conclusion	112

Microkernel-based software systems source out significant parts of the usual operating-system functionality into user mode. Pervasive verification of an operating system (OS) naturally extends to the user processes providing this functionality to applications. The system software employs the client-server design pattern: Clients are the application programs that request services via inter-process communication (IPC) from the OS servers. Depending on whether the OS functionality is spread over several servers, we call this system a *multi-* or a *single-server operating system*.

From the verification point of view, multi-server systems are the more demanding ones. Usually, an operating system provides a set of interrelating services: Access control, for instance, is involved in any interaction with the applications. The underlying access policies, in turn, should survive a system restart, which raises a dependency on permanent storage facilities. In general, invariants typically span the states of several processes in a multi-server OS, expressing dependencies in the behavior of different OS processes. A single-server OS, in contrast, cannot rely on the specific behavior of any other process—they are all applications and certainly, the OS should be able to cope with malicious application programs.

The most natural approach for the verification of a multi-server system is a concurrent Hoare logic that regards the states of all dependent processes at once and formalizes the possible transitions of this process system. Suitable logics have long been proposed in the literature, for example Lamport’s *Generalized Hoare Logic* [Lam80a, LS84]. A more recent approach is the employment of separation logic for concurrent programs as, for instance, Hobor *et al.* [HAN08] have formalized in Coq. In Isabelle/HOL, Prensa Nieto [Pre02] has formalized a Hoare logic for a concurrent while language, which could be extended to a concurrent Simpl. In analogy to the embedding of C0 into Simpl, we could

develop theorems about the embedding of the CoUP model into this concurrent Simpl language. Certainly, the effort for this approach is considerable.

For a single-server OS, in contrast, we may narrow our consideration to the behavior of a single process and its interaction with the concurrent environment. In this scenario, we can reuse most of the original Isabelle/Simpl framework with only slight extensions for processes. Note, however, that any reasoning about a process in Isabelle/Simpl relies on at least one implicit premise about the concurrent environment: The process must remain active during the whole computation that is reasoned about.

Depending on the property to prove, there might be further assumptions about the environment. Thus, we initiate a duality in arguments: Relying on basic invariants of the kernel, we are able to show certain process properties, and as long as the kernel can rely on these properties, it preserves certain system invariants.

Below, we exemplify this general idea by a specific single-server OS, namely the *Simple Operating System* (SOS) of Bogan [Bog08]. More specifically, we sketch the proof for two SOS properties: first, that the SOS never terminates, and second, that the prioritized SOS process cooperates with the VAMOS scheduler, thereby facilitating a fair scheduling among applications. Concluding, we relate these properties to the overall goal of a refinement proof linking the implementation with the specification of the SOS.

8.1 Non-Termination

The SOS provides essential services like access to keyboard, screen, and permanent storage to the application programs. Thus, the existence of the SOS process is vital for the application processes. In this section, we argue why the SOS process does not terminate in our system. For this purpose, we first identify the conditions that lead to the termination of a process. Second, we argue why these conditions can never apply to the SOS process. We conclude the section with a sketch of the necessary steps to formalize our arguments.

From [Section 5.3](#), we know that process termination can either be self-induced by an internal runtime error, or caused by a `PROCESS_KILL` call. In the second case, further restrictions apply: First, the calling process must be privileged (unless committing suicide) or the call fails and no process is terminated. Second, the call argument *hn* has to be a valid handle – referring to the SOS process, in our case.

Assuming that applications are not privileged, we can easily show that no application can terminate the SOS process. We state:

Corollary. *An unprivileged process cannot terminate another process. Formally:*

$$\begin{aligned} & \llbracket \text{inv}_{VC} \ s_{VC}; \ s_{VC}.\text{cup} = \lfloor p_{\text{cup}} \rfloor; \neg \text{privileged } s_{VC}.\text{rightsdb } p_{\text{cup}}; \ p_{\text{victim}} \neq p_{\text{cup}}; \\ & \ s_{VC}.\text{procs } p_{\text{victim}} \neq \perp; \ s_{VC} \xrightarrow{i}_{VC} s'_{VC} \rrbracket \\ \implies & \ s'_{VC}.\text{procs } p_{\text{victim}} \neq \perp \end{aligned}$$

Proof. We unfold Δ_{VC} and prove the statement for each transition phase. The phases Δ_{VC}^{to} , $\Delta_{VC}^{\text{ints}}$, and $\Delta_{VC}^{\text{sched}}$ do neither spawn new processes nor terminate existing ones.

Only in $\Delta_{\text{VC}}^{\text{cup}}$, an active process might become inactive. When distinguishing the 33 introduction rules, we recognize that only the successful `PROCESS_KILL` call is relevant (rule `(pkill.succ)` on page 71). Together with the definitions of `process_kill_response` on page 69 and `invVC` on page 84, we can finally conclude that a non-privileged process can only kill itself. \square

Certainly, it remains to show that indeed only the SOS process is privileged. We may infer this fact from the behavior of the SOS. Directly after boot-up, the only existing process is the SOS. Any spawned process is initially unprivileged, and this situation only changes if a privileged process – namely, the SOS – issues a `SET_PRIVILEGED` call. A static check of the SOS implementation excludes this possibility.

Consequently, the SOS is not killed by another process. It could, however, still terminate itself – either directly with a call `PROCESS_KILL HN_SELF`, or indirectly by a runtime error. A static check is insufficient this time because the SOS calls `PROCESS_KILL` at one place; but we can deploy Isabelle/Simpl for the SOS process to ensure that the argument’s value does not evaluate to `HN_SELF`. Similarly, we can exclude runtime errors resulting, e. g., from a buffer overflow, by program verification.

Note that the necessary adaptation of Isabelle/Simpl for this program verification is straightforward: It amounts to specify the involved library functions similar to normal C0 functions. For `vc_process_kill`, however, there are two peculiarities: First, an argument evaluating to `HN_SELF` is illegal and formally treated as a runtime error. Second, the function’s return value naturally depends on the kernel state. We assume that the process has no information about the kernel state and should thus be able to deal with any valid input value (see Table 4.3 on page 44). Hence, we would specify the return value non-deterministically. In the same manner, we can specify the other library functions. With these prerequisites in place, the program verification proceeds as usual: An automatic syntax translator reads the SOS program and generates Simpl code from it, and function specifications are written and verified in the Hoare logics.

Summarizing the insights from this example, we identify three kinds of properties: (a) the target property T , in our example the existence of the SOS process, (b) safety properties P_V about the kernel, like that the only privileged process is the SOS, and (c) safety properties P_S about the SOS process, e. g., that the process does not feature an output of `RUNTIME_ERROR`, `SET_PRIVILEGED hn`, or `PROCESS_KILL HN_SELF`.

We organize the proof of our ultimate goal into smaller subgoals, which establish or preserve these properties over VAMOS states: First, we examine \mathcal{S}_V^0 to ensure that T and P_V hold initially.¹ Second, we employ Isabelle/Simpl for the verification of P_S assuming that the SOS process exists. Third, we show that T and P_V are preserved under a transition assuming that P_S holds in the first place. We can succinctly express these facts using LTL:

1. $\langle \mathcal{S}_V^0, \hat{\Sigma}, \delta_V \rangle_A \models T \wedge P_V$
2. $\langle \mathcal{S}_V^0, \hat{\Sigma}, \delta_V \rangle_A \models P_S \mathcal{W} \neg T$

¹For the remaining section, we assume that \mathcal{S}_V^0 contains only states, where the SOS is the init process.

$$3. \langle \mathcal{S}_V^0, \hat{\Sigma}, \delta_V \rangle_A \models \Box(T \wedge P_V \wedge P_S \longrightarrow \mathcal{N}(T \wedge P_V))$$

From these facts, we ultimately show by a simple induction proof that the SOS process never terminates, i. e., $\langle \mathcal{S}_V^0, \hat{\Sigma}, \delta_V \rangle_A \models \Box T$.


8.2 Fair Scheduling of Applications

In this section, we apply the fairness theorem about the VAMOS scheduler to the application processes that compete for computation time with the SOS process and against each other. More specifically, we simplify the assumptions of [Theorem 7.2](#) on page 103. The simplifications are possible because the SOS ensures that all applications are members of the same scheduling-priority class, and the SOS process itself possesses a higher priority than the applications. Below, we first formalize the priority assignment within the SOS, and then examine the implications for fair application scheduling.

8.2.1 Priorities in the SOS

Initially, the SOS process OS_PID has the priority OS_PRIO \equiv HIGH, which VAMOS assigns to the init process. Moreover, the SOS process is the only active process. Formally, we infer from the definition of \mathcal{S}_V^0 :

$$s_V \in \mathcal{S}_V^0 \implies s_V.\text{priodb OS_PID} = \lfloor \text{OS_PRIO} \rfloor \wedge (\forall p_{\text{app}} \neq \text{OS_PID}. s_V.\text{priodb } p_{\text{app}} = \perp)$$

When the SOS creates an application process with the call `PROCESS_CREATE tsl prio pages`, the second parameter is always $\lfloor \text{APP_PRIO} \rfloor$ (with $\text{APP_PRIO} \equiv \text{MEDIUM}$). Moreover, the SOS never issues a `CHG_SCHED_PARAMS` call. These conditions are necessary safety properties about the SOS process.

Together with safety properties about the kernel state, e. g., that the SOS process is active and the only privileged process, we might infer that the SOS process forever has priority OS_PRIO, and all application processes are in the same priority class of APP_PRIO. Formally, we aim at the following statement:

$$s_V.\text{priodb OS_PID} = \lfloor \text{OS_PRIO} \rfloor \wedge (\forall p_{\text{app}} \neq \text{OS_PID}. s_V.\text{priodb } p_{\text{app}} \in \{\perp, \lfloor \text{APP_PRIO} \rfloor\})$$

as an invariant over kernel traces with the SOS as init process.

8.2.2 Applying Scheduler Fairness to Applications

As an immediate consequence from static priorities, we can omit the corresponding assumption of the fairness theorem:

$$\langle \mathcal{S}_V^0, \hat{\Sigma}, \delta_V \rangle_A \models \Diamond \Box (\lambda(i, s, n). n.\text{priodb } pid = s.\text{priodb } pid)$$

Furthermore, recollect from [Section 7.3](#) that VAMOS requires

$$\langle \mathcal{S}_V^0, \hat{\Sigma}, \delta_V \rangle_A \models \Box \Diamond (\lambda(i, s, n). \text{has_maxprio } s \text{ } pid \wedge \text{is_timer_on } i)$$

for the fair scheduling of process *pid*. We certainly continue to rely on timer liveness, but we can eliminate the predicate `has_maxprio`. This predicate holds iff the ready queues of all higher priority classes are empty. Naturally, this condition is always met for the SOS process and for applications, iff the SOS is not ready.

Given that the SOS does not terminate, a fair application scheduling requires that any SOS computation eventually results in an IPC operation, which is (besides process termination) the only way to yield for other processes. As suggested by Bogan [Bog08, §6.2], we should be able to infer this property from the SOS implementation.

The implementation is organized as shown in Figure 8.1: In the main function, there is some set-up code followed by an infinite loop. The loop body starts with an open IPC receive and subsequently handles the incoming interrupts and a possibly arrived application request. If all the handlers for interrupts and application requests terminate, the SOS will infinitely often issue the open receive.

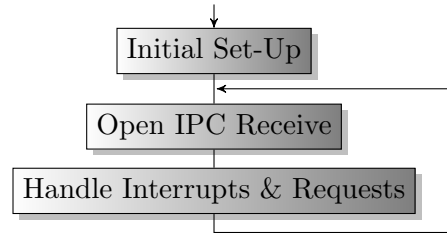


Figure 8.1: SOS control-flow (simplified)

Despite that, we cannot immediately conclude that application processes are scheduled. The current maximal priority only declines if the SOS process indeed waits, which is not the case in two situations: There might be a rendezvous with an already waiting process, on the one hand, or a pending kernel notification, on the other hand.

Fortunately, there can only be a bounded number of immediate rendezvous because the length of the wait queue is naturally finite. Thus, the SOS complies with the requests until there is no one left. Note that because of the prioritized SOS, no application can interfere with the SOS. This circumstance dependably prevents that new requests arrive while the SOS proceeds with the previous ones.

There are two sources for kernel notifications: Either, there is an external device interrupt or a known process has ceased to exist. Bogan describes how the SOS just disables too fast arriving device interrupts. Furthermore, as the number of active processes is naturally bounded, the number of terminating processes is consequently bounded. If the SOS only acknowledges the incoming obituaries, this source of kernel notifications eventually runs dry. A newly created process, however, would always be appended to the ready queue, and thus, scheduled after the already existing ones.

Our argument hitherto substantiates that applications are scheduled at all; but unfortunately, *fair* scheduling requires that the applications have the current maximal priority *while* the timer interrupt occurs. The current implementation of the SOS does not guarantee this requirement. The simplest remedy is a change of the SOS implementation, which explicitly enforces a waiting time before the handling of each application request. For this purpose, we could insert a blocking IPC call with a finite, positive timeout. Unfortunately, an explicit wait decreases the response time of the SOS.

The current implementation, in contrast, ensures fairness by enforcing the applications to wait for a reply to their requests. From Section 7.3, we know that for an unprivileged process, the only means to provoke a race condition with the timer is an IPC call to

a prioritized process. Thus, if an application continuously sends messages to the SOS process, the SOS process might always comply with the query while the timer interrupt arrives. Note however, that applications may not send messages to the SOS unless they also wait for a reply. Whenever the SOS finally sends its reply, the requesting application will be appended to the ready queue upon the completion of the IPC request. Consequently, any other application in the ready queue will be scheduled before this application is scheduled again.

In order to enforce that applications wait for a reply, the SOS employs VAMOS' access-control mechanism for IPC (see Section 3.2.1). More specifically, the SOS forbids `IPC_SEND` calls by unsetting the send bit and only permits `IPC_REQUEST` calls through the request bit. Thus, if an application were trying to send a message via `IPC_SEND`, VAMOS would immediately turn this query down because of lacking permissions. Furthermore, the SOS does never set the finite modifier, which prohibits that an application sets a finite receive timeout. Consequently, an application has to wait until the SOS services its request.

When proving that the SOS indeed sets the IPC rights as discussed, we can split our goal as usual into properties about the SOS process and about the kernel. The kernel property is that IPC rights are granted in the following situations: When a new process is created, VAMOS solely grants the request right to its creator. When an active process is cloned, the new process inherits the IPC rights from the original. Furthermore, a process can grant IPC rights with the calls `IPC_SEND`, `IPC_REQUEST`, and `CHANGE_RIGHTS`. In particular, only a privileged process may grant rights to communicate with another process.

Consequently, it suffices to constrain the SOS process: We require that this process never issues any of the above calls accompanied with the send bit or the finite modifier for its own PID. Then, the applications can never gain these bits for the SOS process.

Ultimately, we can state that VAMOS fairly schedules applications, i. e.,

$$\begin{aligned} \langle \mathcal{S}_V^0, \hat{\Sigma}, \delta_V \rangle_A \models \Box \Diamond (\lambda(i, s, n). \neg \text{pending_infinite_ipc } s \text{ pid}) \longrightarrow \\ \Box \Diamond (\lambda(i, s, n). \text{is_timer_on } i) \longrightarrow \\ \Box \Diamond (\lambda(i, s, n). s.\text{procs } pid = \perp \vee \text{progress } [s.\text{procs } pid] [n.\text{procs } pid]) \end{aligned}$$

The formal proof for this statement remains future work.

8.3 Conclusion

This chapter gave a prospect on the verification of user-mode operating systems. After an introductory, general discussion, we focussed on the SOS, the operating system actually used for the academic computer system in the Verisoft project. A key observation has been that the SOS is a single-server OS, and hence, we may regard just the SOS process in isolation and its interaction with the kernel but neglecting the other processes. This approach is not only simpler; with just a slight extension, we can largely reuse existing Verisoft technology that has been developed and multiply approved for sequential C0 programs. At the example of two particular SOS properties, we outlined the general

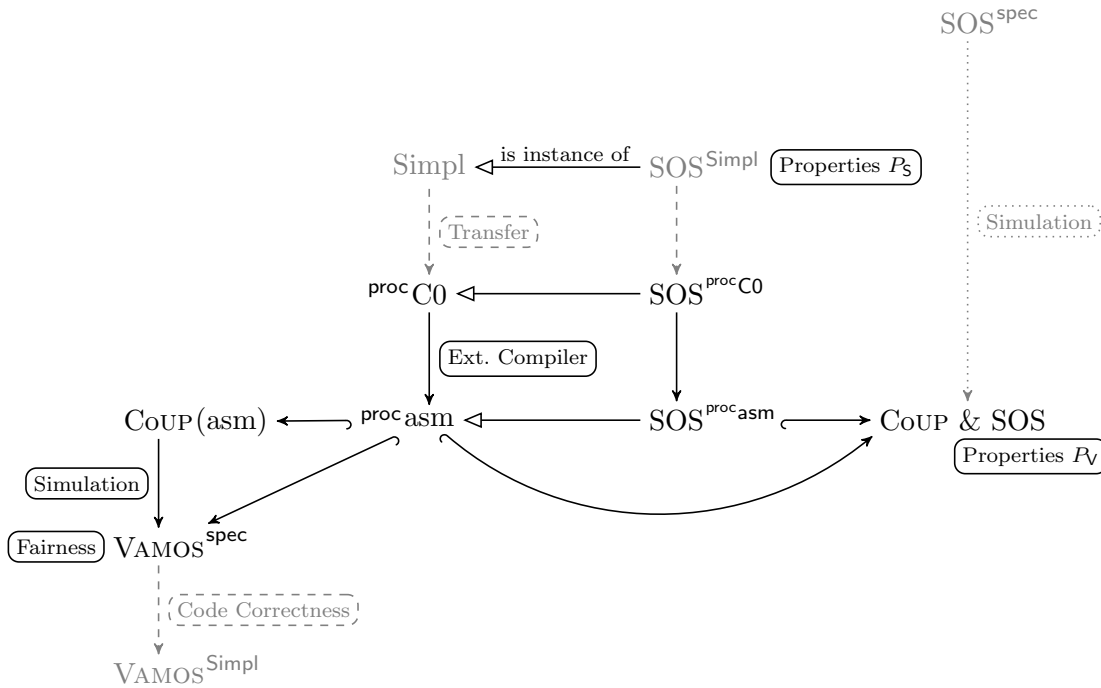


Figure 8.2: Proof plan for the pervasive verification of the SOS

verification procedure, on the one hand, and clarified the combination of the existing technology with our models, on the other hand.

Summarizing the previous discussion, [Figure 8.2](#) sketches a verification plan for a refinement proof that links the implementation with the specification of the SOS, and relates this effort to the overall goal of pervasive system verification. The modeling layers and proofs adjacent to this thesis are depicted in gray with dashed lines, and future work in gray with dotted lines. The figure is vertically split: The left-hand side shows the general model stack while the right-hand side depicts models with the particular SOS process.

On the bottom left of [Figure 8.2](#), $\text{VAMOS}^{\text{Simpl}}$ denotes the implementation of the VAMOS microkernel represented in the Simpl language. In an ongoing verification attempt, Jan Dörrenbächer establishes code correctness, thus linking the implementation with the specification $\text{VAMOS}^{\text{spec}}$ above. This specification uses our process abstraction, which is marked by the arrow from the assembly process model proc^{asm} to $\text{VAMOS}^{\text{spec}}$. At the specification level, we have shown fair process scheduling. From this scheduling property, we can infer that the SOS process advances, on the one hand, and that applications are fairly scheduled,² on the other hand. A simulation theorem relates the specification with our CoUP model. This model is parametrized over a process model, and is here used with assembly processes.

²We assume here that the SOS implementation has been changed as suggested in [Section 8.2](#).

If an assembly process has been compiled from a C0 program, we can use our extended compiler-correctness theorem to establish a simulation with the corresponding C0 process procC0 . Furthermore, we can conveniently verify properties about a C0 process in the Hoare logics of the Isabelle/Simpl framework [Sch05, Sch06]. These properties can then be transferred down to the C0-process semantics using the transfer theorems [Sch06] for the embedding of C0 into Simpl. Alkassar *et al.* [ASS08] applied this approach for the pervasive verification of a paging mechanism. Note that in our context, we additionally need the fairness theorem (Theorem 7.2 on page 103) to transfer termination.

On the right-hand side of Figure 8.2 on the previous page, this general approach is exemplified with the SOS process. The C0 implementation of Bogan *et al.* [Bog08] can automatically be translated to Simpl (denoted as $\text{SOS}^{\text{Simpl}}$) just like Dörrenbächer [DDW09] did it with the VAMOS implementation. On this layer, safety properties P_S about the SOS process can be proven and transferred down via the C0 process $\text{SOS}^{\text{procC0}}$ (using the transfer theorems from Simpl to C0) to the assembly process $\text{SOS}^{\text{procasm}}$ (employing the extended compiler correctness). Based on the proven properties about the SOS process, VAMOS properties P_V like that it never terminates the SOS process can be verified in the CoUP model when the init process is instantiated with $\text{SOS}^{\text{procasm}}$ (denoted as CoUP & SOS^{proc}). Note that compared to a verification directly on the VAMOS model, this approach benefits from the simpler CoUP model with respect to scheduling. Eventually, such a verified kernel property could even be the simulation between the SOS specification SOS^{spec} [Bog08] and the instantiated kernel. Following our terminology, simulation is then our *target property* P_T .

9 Conclusion and Future Work

I am very optimistic.

Wolfgang J. Paul on the chances to complete a proof

Contents

9.1 Formal Results	115
9.2 Measuring the Formal Verification Effort	117
9.3 Outlook	118

The key result of this thesis is the formal justification of an abstraction from a kernel model with explicit, deterministic scheduling to a concurrent process system with non-deterministic but temporally fair scheduling. An important ingredient of our model architecture is a process abstraction that is independent from a particular programming language. We instantiated the process abstraction for two process models extending sequential programming-language semantics. This extension forms the basis of an approach to reason about processes, which can draw to a large extent on previous results of the Verisoft project. In the sections below, we summarize our formal results, quantify the verification effort, and give an outlook for future work.

9.1 Formal Results

Figure 9.1 on the following page illustrates our formal results in detail. All theorem names surrounded by a double line are formally proven; for those framed with a single line, the proof is not yet completed. Previous, related results are drawn in gray with dashed lines while gray color, dotted lines and a slanted font depict future work.

We defined a generic process abstraction (**Section 4.2**) and instantiated it for VAMP assembly and C0 (Sections **4.3** and **4.4**) as depicted by procasm and procC0 on the left side. We formulated a number of well-formedness constraints for process models specifying process-model validity (**Section 4.2**) and showed that these rules are fulfilled by both instances (**Theorem 4.1** on page 47 and **Theorem 4.2** on page 54). The validity of the process models is the premise for embedding them into kernel models.

At the bottom left of **Figure 9.1** on the next page, $\text{VAMOS}^{\text{spec}}$ denotes the specification of the VAMOS kernel. Though Jan Dörrenbächer [**DDB08**] always maintained the specification, we substantially influenced its overall design and corrected many oversights. In particular, the specification now uses our process abstraction, which cleanly separates the concepts of the application binary interface (i. e., the interface between processes and the kernel) and the actual kernel functionality (e. g., the global effect of

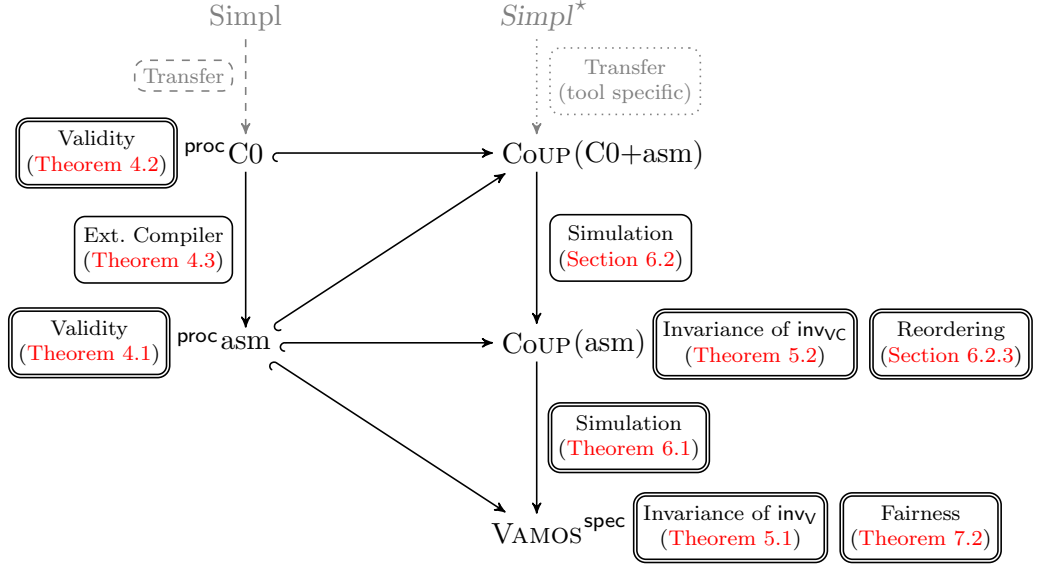


Figure 9.1: Overview of the formal results

kernel calls). For the states of $\text{VAMOS}^{\text{spec}}$, we defined the predicate inv_V and showed that it is invariant over a VAMOS execution trace ([Theorem 5.1](#) on page 83). With the help of this invariance, we proved prioritized fairness for the VAMOS scheduler ([Theorem 7.2](#) on page 103).

Furthermore, we abstracted from the explicit scheduler in $\text{VAMOS}^{\text{spec}}$ and developed the CoUP model with non-deterministic scheduling. This kernel model is parametrized over a process model. In [Figure 9.1](#), two instances are shown: $\text{CoUP}(\text{asm})$ features only assembly processes (proc_{asm}) while $\text{CoUP}(\text{C0+asm})$ additionally features C0 processes (proc_{C0}). Using the invariance of inv_V over VAMOS traces, we formally proved that the model $\text{CoUP}(\text{asm})$ simulates $\text{VAMOS}^{\text{spec}}$ ([Theorem 6.1](#) on page 88). In addition, we defined the predicate inv_{V_C} and formally proved its invariance over execution traces of the CoUP model ([Theorem 5.2](#) on page 84). This proof relies on a valid, underlying process model. Based on the invariance of inv_{V_C} , we have shown that some transitions may be reordered (as described in [Section 6.2.3](#)).

For the verification of C0 processes, we extended the compiler-correctness theorem of Dirk Leinenbach [[Lei08](#)] to processes ([Theorem 4.3](#) on page 57). Though the formal proof of this theorem is not yet completed, we have developed it to a large extent (cf. [Section 4.5](#)). Combining our extended compiler theorem with Norbert Schirmer’s transfer theorems [[Sch06](#)] from *Simpl* to C0, we can eventually transfer properties proven in Isabelle/*Simpl* down to the C0 semantics.

While Isabelle/*Simpl* can only deal with a single process, our results can also be used to justify the abstractions of a verification tool that allows for the concurrent execution of multiple processes. In [Figure 9.1](#), we use *Simpl** as a placeholder for such a tool. Most notably, our model $\text{CoUP}(\text{C0+asm})$ becomes relevant in this context. It

constitutes a concurrent computation model for C0 processes and assembly processes on top of the VAMOS microkernel. While the transfer of proven properties from *Simpl** to $\text{CoUP}(\text{C0+asm})$ certainly depends on the intrinsics of the verification tool, we made inroads into the formal foundation for $\text{CoUP}(\text{C0+asm})$. In essence, this foundation is established by a simulation between $\text{CoUP}(\text{C0+asm})$ and $\text{CoUP}(\text{asm})$, as we described it in [Section 6.2](#). This simulation just combines our corollaries about the reordering of transitions in [Section 6.2.3](#) with the extended compiler theorem ([Theorem 4.3](#) on page 57).

9.2 Measuring the Formal Verification Effort

When quantifying the formal specification effort, we can only roughly estimate our share in the development time for the various models with about a year. There are four theorems, where the formal proof effort exceeded a person month: Establishing the invariance of inv_V for $\text{VAMOS}^{\text{spec}}$ ([Theorem 5.1](#) on page 83) as well as the simulation between $\text{VAMOS}^{\text{spec}}$ and $\text{CoUP}(\text{asm})$ ([Theorem 6.1](#) on page 88) required about two person months, each. The fairness proof ([Theorem 7.2](#) on page 103), on the contrary, took about eight person months. Furthermore, we worked for over a person month on the extended compiler theorem ([Theorem 4.3](#) on page 57) although it is not yet completed.

Note that the actual verification effort does not correlate with the lines of the final proof script: The invariance proof takes over 7.000 lines, the simulation proof counts less than 1.000 lines, fairness has been proven in about 4.000 lines, and the yet incomplete proof for extended compiler correctness already exceeds 22.000 lines.

Partially, this surprising effect may result from the order, in which we verified: at first the simulation, then the fairness, the invariance, and finally the extended compiler-correctness proof. A major verification obstacle was the lack of an early, systematic validation of specifications. Thus, especially at the beginning, we experienced frequent set-backs because of many trivial oversights and flaws in the kernel models. While these set-backs slowed down the productivity, they increased the number of reviews of the proof script. As a side-effect of these numerous reviews, the proof quality increased, which eventually resulted in shorter proof scripts. Furthermore, we proved especially at the beginning many general, auxiliary facts about the kernel models. As these facts are not specific to the actual verification problem, we stored them together with the kernel model and did thus not count them as part of the final proof script. Besides that, the increasing experience with the Isabelle proof assistant might have induced some proof acceleration over time.

In addition, the nature of the verification goals varied greatly. At the one end, there is the extended compiler theorem, where the largest part concerns library correctness. Here, we proved many times correctness for often very similar code. Likewise, the proof of invariance has essentially been proven by a huge case distinction with many similar cases: We just guided the proof assistant to unfold the right definitions. At the other end, there is a temporal property, which requires much more creativity, e. g., to find applicable instances for existential quantifiers. Additionally, a more challenging proof

goal requires a cleaner proof style. In particular, we worked much harder for a clean, understandable (and hence, substantially shorter) proof script of the fairness proof than we did for library correctness or the invariance proof. After all, our primary aim was to complete the formal proofs in due time rather than to optimize them for best human readability.

9.3 Outlook

Our work has been motivated by the verification of the *simple operating system* (SOS), which has been developed and specified by Sebastian Bogan [Bog08]. A formal proof is certainly far beyond the scope of this thesis and remains future work. In spite of that, we developed our formal work with this use case in mind and substantiated its applicability by fairly detailed and partially formalized proof sketches. Particularly for this purpose, we invented the process abstraction.

In general, the process abstraction opens up two directions for the development of verification techniques for process systems: Either, we may regard only one process at a time and its interaction with the kernel (as in Simpl), or we might consider the concurrent execution of multiple processes at the same time (assuming a concurrent verification environment).

In [Chapter 8](#), we discussed the first approach, which suffices to verify the correctness of a single-server user-mode operating system like the SOS. Summarizing, process properties can conveniently be verified in the Isabelle/Simpl framework and then transferred down to the C0 process semantics. The extended compiler theorem furthermore ensures that corresponding properties hold on the assembly semantics. We can combine such process properties with those proven about the CoUP(asm) model. Finally, we know because of the corresponding simulation theorem that these properties indeed hold on the VAMOS^{spec} level.

For a multi-server operating system, however, you might prefer a concurrent verification tool. Our formal work may also be used to formally justify the abstraction of such a tool. Given that a concurrent verification tool has neither been developed nor used in the Verisoft project, this formal justification remains future work.

Bibliography

- [ABP09] Eyad Alkassar, Sebastian Bogan, and Wolfgang Paul. Proving the correctness of client/server software. *Sādhanā: Academy Proceedings in Engineering Sciences*, 34(1):145–191, February 2009.
- [AH08] Eyad Alkassar and Mark A. Hillebrand. Formal functional verification of device drivers. In Jim Woodcock and Natarajan Shankar, editors, *Verified Software: Theories, Tools, and Experiments*, volume 5295 of *LNCS*, pages 225–239. Springer, October 2008.
- [AHK⁺07] Eyad Alkassar, Mark Hillebrand, Steffen Knapp, Rostislav Rusev, and Sergey Tverdyshev. Formal device and programming model for a serial interface. In Bernhard Beckert, editor, *VERIFY*, volume 259 of *CEUR Workshop Proceedings*, pages 4–20. CEUR-WS.org, 2007.
- [AHL⁺08] Eyad Alkassar, Mark A. Hillebrand, Dirk Leinenbach, Norbert W. Schirmer, and Artem Starostin. The Verisoft approach to systems verification. In Natarajan Shankar and Jim Woodcock, editors, *Verified Software: Theories, Tools, and Experiments*, volume 5295 of *LNCS*, pages 209–224. Springer, 2008.
- [AHL⁺09] Eyad Alkassar, Mark A. Hillebrand, Dirk C. Leinenbach, Norbert W. Schirmer, Artem Starostin, and Alexandra Tsyban. Balancing the load: Leveraging a semantics stack for systems verification. *J. Autom. Reasoning: Operating System Verification*, 42(2–4):389–454, 2009.
- [Alk05] Eyad Alkassar. Constructing a formal framework for modeling and verifying a real operating system. Master’s thesis, Saarland University, 2005.
- [Alk09] Eyad Alkassar. *OS Verification Extended – On the Formal Verification of Device Drivers and the Correctness of Client/Server Software*. PhD thesis, Saarland University, 2009.
- [Ame99] American National Standards Institute. *ANSI/ISO/IEC 9899-1999: Programming Languages — C*. American National Standards Institute, New York, USA, December 1999.
- [Ash75] Edward A. Ashcroft. Proving assertions about parallel programs. *J. Comput. Syst. Sci.*, 10(1):110–135, 1975.

- [ASS08] Eyad Alkassar, Norbert Schirmer, and Artem Starostin. Formal pervasive verification of a paging mechanism. In *TACAS*, volume 4963 of *LNCS*, pages 109–123. Springer, 2008.
- [AZKR04] Konstantine Arkoudas, Karen Zee, Viktor Kuncak, and Martin C. Rinard. Verifying a file system implementation. In Jim Davies, Wolfram Schulte, and Michael Barnett, editors, *ICFEM*, volume 3308 of *LNCS*, pages 373–390. Springer, 2004.
- [BB04] Bernhard Beckert and Gerd Beuster. Formal specification of security-relevant properties of user interfaces. In *Workshop on Critical Systems Development with UML*, pages 139–146, Lisbon, Portugal, 2004. TU Munich Tech. Rep. TUM-I0415.
- [BBBB09] Christoph Baumann, Bernhard Beckert, Holger Blasum, and Thorsten Bormer. Formal verification of a microkernel used in dependable software systems. In Bettina Buth, Gerd Rabe, and Till Seyfarth, editors, *SAFE-COMP*, volume 5775 of *LNCS*, pages 187–200. Springer, 2009.
- [BCT95] William R. Bevier, Richard Cohen, and Jeff Turner. A specification for the synergy file system. Technical Report 120, Computational Logic Inc., 1995.
- [BHMY89] William R. Bevier, Warren A. Hunt, Jr., J. Strother Moore, and William D. Young. An approach to systems verification. *J. Autom. Reasoning*, 5(4):411–428, 1989.
- [BHW06] Gerd Beuster, Niklas Henrich, and Markus Wagner. Real world verification – experiences from the Verisoft email client. In Geoff Sutcliffe, Renate Schmidt, and Stephan Schulz, editors, *Empirically Successful Computerized Reasoning*, volume 192 of *CEUR Workshop Proceedings*, page 112. CEUR-WS.org, 2006.
- [BJK⁺03] Sven Beyer, Christian Jacobi, Daniel Kroening, Dirk Leinenbach, and Wolfgang Paul. Instantiating uninterpreted functional units and memory system: Functional verification of the VAMP. In Daniel Geist and Enrico Tronci, editors, *CHARME*, volume 2860 of *LNCS*, pages 51–65. Springer, 2003.
- [BJK⁺06] Sven Beyer, Christian Jacobi, Daniel Kröning, Dirk Leinenbach, and Wolfgang J. Paul. Putting it all together: Formal verification of the VAMP. *STTT*, 8(4–5):411–430, 2006.
- [Bog08] Sebastian Bogan. *Formal Specification of a Simple Operating System*. PhD thesis, Saarland University, 2008.
- [Boy09] Andrew Boyton. A verified shared capability model. In Gerwin Klein, Ralf Huuck, and Bastian Schlich, editors, *Systems Software Verification*, volume 254 of *ENTCS*, pages 25–44. Elsevier Science B.V., 2009.

-
- [CDH⁺09] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A practical system for verifying concurrent C. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Markus Wenzel, editors, *TPHOLs*, volume 5674 of *LNCS*, pages 23–42. Springer, 2009. Invited paper.
- [CKS08] David Cock, Gerwin Klein, and Thomas Sewell. Secure microkernels, state monads and scalable refinement. In Otmane Aït Mohamed, César Muñoz, and Sofiène Tahar, editors, *TPHOLs*, volume 5170 of *LNCS*, pages 167–182. Springer, 2008.
- [CMST09] Ernie Cohen, Michał Moskal, Wolfram Schulte, and Stephan Tobies. A precise yet efficient memory model for C. In *Systems Software Verification*, volume 254 of *ENTCS*, pages 85–103. Elsevier Science B.V., 2009.
- [DDB08] Matthias Daum, Jan Dörrenbächer, and Sebastian Bogan. Model stack for the pervasive verification of a microkernel-based operating system. In Bernhard Beckert and Gerwin Klein, editors, *VERIFY*, volume 372 of *CEUR Workshop Proceedings*, pages 56–70. CEUR-WS.org, August 2008.
- [DDW09] Matthias Daum, Jan Dörrenbächer, and Burkhart Wolff. Proving fairness and implementation correctness of a microkernel scheduler. *J. Autom. Reasoning: Operating System Verification*, 42(2–4):349–388, 2009.
- [DDWS08] Matthias Daum, Jan Dörrenbächer, Burkhart Wolff, and Mareike Schmidt. A verification approach for system-level concurrent programs. In Jim Woodcock and Natarajan Shankar, editors, *Verified Software: Theories, Tools, and Experiments*, volume 5295 of *LNCS*, pages 161–176, Toronto, Canada, October 2008. Springer.
- [DHP05] Iakov Dalinger, Mark Hillebrand, and Wolfgang Paul. On the verification of memory management mechanisms. In Dominique Borrione and Wolfgang Paul, editors, *CHARME*, volume 3725 of *LNCS*, pages 301–316. Springer, 2005.
- [DMS⁺09] Markus Dahlweid, Michał Moskal, Thomas Santen, Stephan Tobies, and Wolfram Schulte. VCC: Contract-based modular verification of concurrent C. In *ICSE*, pages 429–430. IEEE Computer Society, 2009.
- [DSS09] Matthias Daum, Norbert W. Schirmer, and Mareike Schmidt. Implementation correctness of a real-time operating system. In *SEFM*, pages 23–32. IEEE Computer Society, 2009.
- [DSS10] Matthias Daum, Norbert W. Schirmer, and Mareike Schmidt. From operating-system correctness to pervasively verified applications. In *IFM*, LNCS. Springer, 2010. To appear.

- [EGR07] Kevin Elphinstone, David Greenaway, and Sergio Ruocco. Lazy scheduling and direct process switch – merit or myths? In *Workshop on Operating System Platforms for Embedded Real-Time Applications*, Pisa, Italy, July 2007. Available at http://www.ertos.nicta.com.au/publications/papers/Elphinstone_GR_07.pdf.
- [EH86] E. Allen Emerson and Joseph Y. Halpern. “sometimes” and “not never” revisited: On branching versus linear time temporal logic. *J. ACM*, 33(1):151–178, 1986.
- [EKO95] Dawson R. Engler, M. Frans Kaashoek, and James O’Toole. Exokernel: An operating system architecture for application-level resource management. In *SOSP*, pages 251–266. ACM, 1995.
- [FC98] Brett D. Fleisch and Mark Allan A. Co. Workplace microkernel and OS: a case study. *Softw. Pract. Exper.*, 28(6):569–591, 1998.
- [Fra86] Nissim Francez. *Fairness*. Springer, September 1986.
- [FSDG08] Xinyu Feng, Zhong Shao, Yuan Dong, and Yu Guo. Certifying low-level programs with hardware interrupts and preemptive threads. In *PLDI*, pages 170–182. ACM, 2008.
- [FW83] Ronald Fagin and John H. Williams. A fair carpool scheduling algorithm. *IBM Journal of Research and Development*, 27(2):133–139, 1983.
- [Gen35] Gerhard Gentzen. Untersuchungen über das logische Schließen I. *Mathematische Zeitschrift*, 39:176–210, 1935.
- [GHLP05] Mauro Gargano, Mark Hillebrand, Dirk Leinenbach, and Wolfgang Paul. On the correctness of operating system kernels. In J. Hurd and T. F. Melham, editors, *TPHOLs 2005*, volume 3603 of *LNCS*, pages 1–16. Springer, 2005.
- [HALM06] Constance L. Heitmeyer, Myla Archer, Elizabeth I. Leonard, and John D. McLean. Formal specification and verification of data separation in a separation kernel for an embedded system. In *ACM Conference on Computer and Communications Security*, pages 346–355. ACM, 2006.
- [HAN08] Aquinas Hobor, Andrew W. Appel, and Francesco Zappa Nardelli. Oracle semantics for concurrent separation logic. In Sophia Drossopoulou, editor, *ESOP*, volume 4960 of *LNCS*, pages 353–367. Springer, 2008.
- [HEK⁺07] Gernot Heiser, Kevin Elphinstone, Ihor Kuz, Gerwin Klein, and Stefan M. Petters. Towards trustworthy computing systems: taking microkernels to the next level. *Operating Systems Review*, 41(4):3–11, July 2007.
- [HF04] Reinhold Heckmann and Christian Ferdinand. Worst-case execution time prediction by static program analysis. White paper, AbsInt Angewandte Informatik GmbH, 2004. Available via <http://www.absint.com/wcet.htm>.

-
- [HHF⁺05] Hermann Härtig, Michael Hohmuth, Norman Feske, Christian Helmuth, Adam Lackorzynski, Frank Mehnert, and Michael Peter. The Nizza secure-system architecture. In *CollaborateCom*. IEEE Computer Society, 2005.
- [HIP05] Mark A. Hillebrand, Thomas In der Rieden, and Wolfgang J. Paul. Dealing with I/O devices in the context of pervasive system verification. In *ICCD*, pages 309–316. IEEE Computer Society, 2005.
- [Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
- [HP96] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Mateo, CA, second edition, 1996.
- [HP07] Mark A. Hillebrand and Wolfgang J. Paul. On the architecture of system verification environments. In *Haifa Verification Conference*, pages 153–168. Springer, 2007.
- [In 09] Thomas In der Rieden. *Verified Linking for Modular Kernel Verification*. PhD thesis, Saarland University, November 2009.
- [Int98] International Organization for Standardization. *ISO/IEC 14882-1998: Programming Languages — C++*. International Organization for Standardization, Geneva, Switzerland, September 1998.
- [IT08] Tom In der Rieden and Alexandra Tsyban. CVM – a verified framework for microkernel programmers. In *Systems Software Verification*, volume 217 of *ENTCS*, pages 151–168. Elsevier Science B.V., 2008.
- [Jon83] Cliff B. Jones. Specification and design of (parallel) programs. In *IFIP Congress*, pages 321–332, 1983.
- [Kai07] Robert Kaiser. Combining partitioning and virtualization for safety-critical systems. White Paper WP_CPV_10_A4_R10, SYSGO AG, 2007. Available via <http://www.sysgo.com/news-events/whitepapers/>.
- [KEH⁺09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *SOSP*, pages 207–220. ACM, 2009.
- [Kle09] Gerwin Klein. Operating system verification — an overview. *Sādhanā: Academy Proceedings in Engineering Sciences*, 34(1):27–69, February 2009.
- [Kno74] Gary D. Knott. A proposal for certain process management and intercommunication primitives. *Operating Systems Review*, 8(4):7–44, 1974.
- [Koz90] Dexter Kozen. On Kleene algebras and closed semirings. In *MFCS*, volume 452 of *LNCS*, pages 26–47. Springer, 1990.

- [KP07] Steffen Knapp and Wolfgang Paul. Realistic worst case execution time analysis in the context of pervasive system verification. In Thomas Reps, Mooly Sagiv, and Jörg Bauer, editors, *Program Analysis and Compilation, Theory and Practice: Essays Dedicated to Reinhard Wilhelm on the Occasion of His 60th Birthday*, volume 4444 of *LNCS*, pages 53–81. Springer, 2007.
- [Kri63] Saul A. Kripke. Semantical considerations on modal logic. *Acta Philosophica Fennica*, 16:83–94, 1963.
- [Lam80a] Leslie Lamport. The “Hoare logic” of concurrent programs. *Acta Inf.*, 14:21–37, 1980.
- [Lam80b] Leslie Lamport. “sometime” is sometimes “not never” - on the temporal logic of programs. In *POPL*, pages 174–185, 1980.
- [Lei08] Dirk Carsten Leinenbach. *Compiler Verification in the Context of Pervasive System Verification*. PhD thesis, Saarland University, 2008.
- [Lev84] Henry M. Levy. *Capability-Based Computer Systems*. Butterworth-Heinemann, Newton, MA, USA, 1984.
- [Lie93] Jochen Liedtke. Improving IPC by kernel design. In *SOSP*, pages 175–188. ACM, 1993.
- [Lie95] Jochen Liedtke. On μ -kernel construction. In *SOSP*, pages 237–250. ACM, 1995.
- [Lie96] Jochen Liedtke. Towards real microkernels. *Commun. ACM*, 39(9):70–77, 1996.
- [LNRS07a] Bruno Langenstein, Andreas Nonnengart, Georg Rock, and Werner Stephan. A history-based verification of distributed applications. In Bernhard Beckert, editor, *VERIFY*, volume 259 of *CEUR Workshop Proceedings*, pages 70–84. CEUR-WS.org, 2007.
- [LNRS07b] Bruno Langenstein, Andreas Nonnengart, Georg Rock, and Werner Stephan. Verification of distributed applications. In Francesca Saglietti and Norbert Oster, editors, *SAFECOMP*, volume 4680 of *LNCS*, pages 315–328. Springer, 2007.
- [LP06] Markus Linnemann and Norbert Pohlmann. Schöne neue Welt?! – Die vertrauenswürdige Sicherheitsplattform Turaya. *IT-Sicherheit*, 3-4, 2006.
- [LP08] Dirk Leinenbach and Elena Petrova. Pervasive compiler verification: From verified programs to verified systems. In *Systems Software Verification*, volume 217 of *ENTCS*, pages 23–40. Elsevier Science B.V., 2008.

-
- [LPP05] Dirk Leinenbach, Wolfgang J. Paul, and Elena Petrova. Towards the formal verification of a C0 compiler: Code generation and implementation correctness. In *SEFM*, pages 2–12. IEEE Computer Society, 2005. Invited paper.
- [LS84] Leslie Lamport and Fred B. Schneider. The “Hoare logic” of CSP, and all that. *ACM Trans. Program. Lang. Syst.*, 6(2):281–296, 1984.
- [LS09] Dirk Leinenbach and Thomas Santen. Verifying the Microsoft Hyper-V Hypervisor with VCC. In *FM*, volume 5850 of *LNCS*, pages 806–809. Springer, 2009. Invited paper.
- [Maz87] Antoni W. Mazurkiewicz. Trace theory. In Wilfried Brauer, Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Advances in Petri Nets, 1986*, volume 255 of *LNCS*, pages 279–324. Springer, 1987.
- [Mil82] Robin Milner. *A Calculus of Communicating Systems*. Springer, 1982.
- [Moo02] J. Strother Moore. A grand challenge proposal for formal methods: A verified stack. In *10th Anniversary Colloquium of UNU/IIST*, pages 161–172. Springer, 2002.
- [MP00] Silvia M. Mueller and Wolfgang J. Paul. *Computer Architecture: Complexity and Correctness*. Springer, 2000.
- [Nao05] Moni Naor. On fairness in the carpool problem. *J. Algorithms*, 55(1):93–98, 2005.
- [Ngu05] Veronique Gonsu Nguiekom. Verifikation von doppelt verketteten Listen auf Pointerebene. Diploma thesis, Saarland University, 2005. Available at <http://www-wjp.cs.uni-sb.de/publikationen/Ng05.pdf>.
- [NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
- [OG76] Susan S. Owicki and David Gries. Verifying properties of parallel programs: An axiomatic approach. *Commun. ACM*, 19(5):279–285, 1976.
- [O’H04] Peter W. O’Hearn. Resources, concurrency and local reasoning. In Philippa Gardner and Nobuko Yoshida, editors, *CONCUR*, volume 3170 of *LNCS*, pages 49–67. Springer, 2004.
- [Pau94] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *LNCS*. Springer, 1994.
- [Pau08] Wolfgang Paul. Towards a worldwide verification technology. In *Verified Software: Theories, Tools, and Experiments, October 2005, Revised Selected Papers and Discussions*, volume 4171 of *LNCS*, pages 19–25, Zürich, Switzerland, June 2008. Springer.

- [Pet07] Elena Petrova. *Verification of the C0 Compiler Implementation on the Source Code Level*. PhD thesis, Saarland University, 2007.
- [Pre02] Leonor Prensa Nieto. *Verification of Parallel Programs with the Owicki-Gries and Rely-Guarantee Methods in Isabelle/HOL*. PhD thesis, TU Munich, 2002.
- [Pri57] Arthur N. Prior. *Time and Modality*. Oxford Univ. Press, 1957.
- [PRS⁺01] Birgit Pfitzmann, James Riordan, Christian Stübke, Michael Waidner, and Arnd Weber. The PERSEUS system architecture. Technical Report RZ 3335 (#93381), IBM Research Division, 2001.
- [PZH07] Stefan M. Petters, Patryk Zadarnowski, and Gernot Heiser. Measurements or static analysis or both? In Christine Rochange, editor, *WCET*, volume 07002 of *Dagstuhl Seminar Proceedings*, Schloss Dagstuhl, Germany, 2007. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI).
- [RATY⁺88] Richard Rashid, Jr. Avadis Tevanin, Michael Young, David Golub, and Robert Baron. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. *IEEE Trans. Comput.*, 37(8):896–908, 1988.
- [Rey02] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74. IEEE Computer Society, 2002.
- [Ruo06] Sergio Ruocco. Real-time programming and L4 microkernels. In *Workshop on Operating System Platforms for Embedded Real-Time Applications*, Dresden, Germany, July 2006. Available at http://www.ertos.nicta.com.au/publications/papers/Ruocco_06.pdf.
- [Sam08] Thaima Samman. Verifying 50,000 lines of C code. *Futures, Microsoft's European Innovation Magazine*, 21, June 2008.
- [Sch05] Norbert Schirmer. A verification environment for sequential imperative programs in Isabelle/HOL. In *LPAR 04*, LNCS, pages 398–414. Springer, 2005.
- [Sch06] Norbert Schirmer. *Verification of Sequential Imperative Programs in Isabelle/HOL*. PhD thesis, TU Munich, 2006.
- [SDN⁺04] J. Shapiro, M. S. Doerrie, E. Northup, S. Sridhar, and M. Miller. Towards a verified, general-purpose operating system kernel. In *FM Workshop on OS Verification, Tech. Rep. 0401005T-1*, pages 1–19. National ICT Australia, 2004.
- [Smi96] Mark A. Smith. *Formal Verification of TCP and T/TCP*. PhD thesis, Massachusetts Institute of Technology, 1996.

- [SP07] Mohit Singal and Stefan M. Petters. Issues in analysing L4 for its WCET. In *Workshop on Microkernels for Embedded Systems*, Sydney, Australia, January 2007. Available at http://www.ertos.nicta.com.au/publications/papers/Singal_Petters_07.pdf.
- [ST08] Artem Starostin and Alexandra Tsyban. Correct microkernel primitives. In *Systems Software Verification*, volume 217 of *ENTCS*, pages 169–185. Elsevier Science B.V., 2008.
- [SW09] Norbert Schirmer and Makarius Wenzel. State spaces – The locale way. In *Systems Software Verification*, volume 254 of *ENTCS*, pages 161–179. Elsevier Science B.V., 2009.
- [Tew07] Hendrik Tews. Formal methods in the Robin project: Specification and verification of the Nova microhypervisor. In *C/C++ Verification Workshop, Tech. Rep. ICIS–R07015*, pages 59–68. Radboud University Nijmegen, June 2007.
- [TS08] Sergey Tverdyshev and Andrey Shadrin. Formal verification of gate-level computer systems. In Kristin Yvonne Rozier, editor, *LFM 2008: Sixth NASA Langley Formal Methods Workshop*, NASA Scientific and Technical Information (STI), pages 56–58. NASA, 2008.
- [Ver09] Verisoft Project. Verisoft repository. Available at <http://www.verisoft.de/VerisoftRepository.html>, 2009.
- [VP07] Viktor Vafeiadis and Matthew J. Parkinson. A marriage of rely/guarantee and separation logic. In Luís Caires and Vasco Thudichum Vasconcelos, editors, *CONCUR*, volume 4703 of *LNCS*, pages 256–271. Springer, 2007.
- [Wen02] Markus Wenzel. *Using Axiomatic Type Classes in Isabelle*. TU Munich, 2002.
- [WKP80] B. Walker, R. Kemmerer, and G. Popek. Specification and verification of the UCLA Unix security kernel. *Commun. ACM*, 23(2):118–131, 1980.
- [YTEM06] Junfeng Yang, Paul Twohey, Dawson R. Engler, and Madanlal Musuvathi. Using model checking to find serious file system errors. *ACM Trans. Comput. Syst.*, 24(4):393–423, 2006.