Formal Verification of a Small Real-Time Operating System

Dissertation

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

Mareike Schmidt

mareike@wjpserver.cs.uni-saarland.de

Saarbrücken, 14. April 2011

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, 14. April 2011

Tag des Kolloquiums:	12. April 2011
Dekan:	Prof. Dr. Holger Hermanns
Vorsitzender des Prüfungsausschusses:	Prof. Dr. Sebastian Hack
1. Berichterstatter:	Prof. Dr. Wolfgang J. Paul
2. Berichterstatter:	Dr. habil. Werner Stephan
akademischer Mitarbeiter:	Dr. Alexey Pospelov

Ist man in kleinen Dingen nicht geduldig,
bringt man die grossen Vorhaben zum Scheitern.
(Konfuzius, chinesischer Philosoph, 551 - 479 v. Chr.)

Danksagung

An dieser Stelle möchte ich mich bei allen bedanken, die mich unterstützt und zum Gelingen dieser Arbeit beigetragen haben.

Insbesondere danke ich Herrn Prof. Wolfgang Paul, der mir die Möglichkeit zur Promotion an seinem Lehrstuhl gab, meine Arbeit wissenschaftlich betreute und darüber hinaus immer Verständnis für meine besondere Situation zeigte.

Ein weiterer Dank gilt allen Kollegen und Mitarbeitern des Lehrstuhls für die gute Zusammenarbeit und die angenehme Arbeitsatmosphäre. Besonders möchte ich Jan Dörrenbächer, Mark Hillebrand, Steffen Knapp, Dirk Leinenbach, Norbert Schirmer und Alexandra Tsyban erwähnen, die mir mit Rat und Tat zur Seite standen, wenn es um die Anbindung ihrer Theorien, die Verbesserung meiner Arbeit oder deren Publikation ging. Vor allem Matthias Daum möchte ich für seine stetige Unterstützung, die zahlreichen Diskussionen und seine konstruktive Kritik danken, die mich um vieles weitergebracht hat. Darüber hinausgehend ein großer Dank an die Damen des Sekretariats, für ihre Freundlichkeit und ihren Einsatz, um uns Mitarbeitern den Rücken für die 'wesentlichen Dinge' freizuhalten.

Ich möchte aber auch meine Freunde und meine Familie nicht vergessen, die in all der Zeit für mich da waren, mich ermutigt und unterstützt haben. Ein besonderer Dank gilt meinen Eltern, die mich auf meinem Weg immer begleitet haben und mir zur Seite standen. Ohne ihre Hilfsbereitschaft und ihren unermüdlichen Einsatz hätte ich es sehr viel schwerer gehabt, meine Ziele zu erreichen. Zu guter Letzt möchte ich mich bei meiner Tochter Sophie für ihr Verständnis und ihre Geduld mit mir ganz herzlich bedanken.

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

The foundation of this thesis is a distributed real-time system that connects several electronic control units (ECUs) with a communication bus. Each ECU consists of a processor executing the real-time operating system Olos and several applications on the one hand, and an interface to the bus (ABC) on the other hand. Olos provides application scheduling and controls the communication with the bus. The applications may communicate with Olos via so-called *system calls*. For applications written in high-level languages these calls are available in terms of library functions.

First, we present the design and the implementation of Olos and its necessary library functions. Thereafter, we introduce the abstract model of an entire ECU which specifies the interface to the bus (ABC), process models and the behaviour of Olos.

Then, we formulate a simulation theorem between the abstract ECU model and a model that embeds the concrete Olos implementation. The proof of this theorem provides us with the implementation correctness of Olos. Based on the formal correctness of our operating system, the last section of this thesis presents an approach to pervasively verify applications that are executed under Olos on a single ECU.

Kurze Zusammenfassung

Grundlage dieser Arbeit ist ein verteiltes Echtzeitsystem, welches mehrere elektronische Kontrolleinheiten (ECUs) mit einem Kommunikationsbus verbindet. Jede dieser Kontrolleinheiten besteht aus einer Schnittstelle zum Bus (ABC) und einem Prozessor, welcher das Echtzeitbetriebssystem Olos und mehrere Anwendungen ausführt. Olos organisiert die Ausführungszeit der Anwendungen auf dem Prozessor und steuert deren Kommunikation mit dem Bus. Die Anwendungen haben die Möglichkeit, über sogenannte *Systemaufrufe* mit dem Betriebssystem zu kommunizieren. Diese stehen den Anwendungen, die in höheren Programmiersprachen geschrieben sind, in Form von Bibliotheksfunktionen zur Verfügung.

Zuerst stellen wir den Entwurf und die Implementierung von Olos und den notwendigen Bibliotheksfunktionen vor. Danach beschreiben wir das abstrakte Modell einer vollständigen ECU, welches die Schnittstelle zum Bus (ABC), Prozessmodelle und das Verhalten des Betriebssystems Olos festlegt.

Wir formulieren ein Simulationstheorem zwischen dem abstrakten ECU Modell und einem Modell mit eingebetteter konkreter Olos-Implementierung. Der Beweis dieser Aussage liefert uns die Implementierungskorrektheit von Olos. Im letzten Teil der Arbeit benutzen wir dieses Ergebnis als Grundlage für einen Ansatz, mit dem durchgängig Anwendungen verifiziert werden können, die unter Olos auf einer elektronischen Kontrolleinheit ausgeführt werden.

Ausführliche Zusammenfassung

Computersysteme sind allgegenwärtig und beeinflussen unseren Alltag. Obwohl sich viele Menschen auf Computersysteme verlassen, hinterfragen nur wenige ihre Zuverlässigkeit. Besonders ihr Einsatz in sicherheitskritischen Bereichen, wie zum Beispiel in der Medizintechnik oder in Fahrzeugen, verlangt Robustheit, Sicherheit und Verlässlichkeit eines Systems. In diesem Zusammenhang könnte ein Fehlverhalten weitreichende Folgen haben. Gerade in letzer Zeit wurde die Fehlfunktion eines Bremskontrollsystems mit einem Softwarefehler in Verbindung gebracht [VB10, Guy10]. Allein das Gerücht einer solchen Schwachstelle der Software zieht einen wirtschaftlichen Schaden für den Hersteller nach sich. Weitaus schlimmer wäre jedoch die Vorstellung, dass schwere Unfälle mit ihren Konsequenzen die Folgen eines Softwarefehlers wären. Zudem ist in Autos ein kontinuierliches Ansteigen der Anzahl von elektronischen Steuergeräten, die für eine Vielzahl verschiedener Aufgaben zuständig sind, zu beobachten. Die steigende Komplexität verteilter Systeme stellt immer größere Anforderungen an die Verlässlichkeit und Fehlerfreiheit der Systeme und ihrer Komponenten. Der sicherste Ansatz, um schon beim Systementwurf konzeptionelle und menschliche Unzulänglichkeiten auszuschließen, ist *Verifikation*. Dazu wird die korrekte Funktionsweise einer Implementierung durch ein mathematisches Modell spezifiziert und ein Beweis dafür entwickelt, dass der Systementwurf dieser Spezifikation genügt. Wird dieser Beweis von einem Computer überprüft, sprechen wir von *formaler* Verifikation.

In der Praxis sind Computersysteme üblicherweise in mehreren Abstraktionsebenen implementiert: Schon die Hardware besteht aus unterschiedlich abstrakten Schichten, die von der Register-Transferebene über die Befehlssatzarchitektur bis hin zur Assemblerebene reicht. Das Betriebssystem stützt sich auf die abstrakteste Hardware-Schicht und stellt seinerseits eine Plattform zur Verfügung auf der Benutzerprogramme implementiert werden. Um systematische Fehler im gesamten Systementwurf ausschließen zu können, muss man deshalb alle Schichten des betrachteten Computersystems berücksichtigen. Wir nennen Verifikation, die sich über mehrere Schichten eines Systems erstreckt *durchgängig*.

Das Forschungsprojekt Verisoft verfolgt genau diesen Ansatz, um die Qualität integrierter Computersysteme schon im Entwurf zu sichern. Genauer gesagt, entstand diese Arbeit im Rahmen des Teilprojekts Automotive, das die durchgängige formale Verifikation des automatischen Automobilnotrufs *eCall*[Eur03] zum Ziel hat. Das zugrundeliegende Echtzeitsystem verbindet mehrere elektronische Kontrolleinheiten (ECUs) mit einem Kommunikationsbus. Ein Übertragungsprotokoll ordnet dann jeder angeschlossenen Komponente feste Sendezeiten zu. Dieser Ansatz führt auf das von Kopetz und Grünstreidl [KG94] entwickelte *time-triggered protocol* zurück und findet heutzutage weite Akzeptanz und vielfachen Einsatz in der Industrie.

Jede ECU besteht aus einem Prozessor und einer Schnittstelle zum Bus (ABC). Auf dem Prozessor läuft das Echtzeitbetriebssystem Olos und darauf mehrere interagierende Anwendungen, die bestimmte Aufgaben ausführen. Die Funktionalität von Olos umfasst im Wesentlichen die Interaktion zwischen Kommunikationshardware und Anwendersoftware.

Olos stellt den Anwendungen eine Prozessorabstraktion mit

(a) exklusivem Zugang zu Ressourcen wie Speicher und Registern und
(b) Kommunikationsmitteln, um Dienste des Betriebssystems anzufordern,
zur Verfügung. Eine solche Abstraktion nennt man im allgemeinen *Prozess* und die Kommunikationsmittel *Systemaufrufe*. Da die meisten Programme kompiliert werden, um als Prozess unter einem Betriebssystem zu laufen, dehnen wir den Prozessbegriff auf eine Programmsemantik mit Systemaufrufen aus. Anwendungen in höheren Programmiersprachen können Systemaufrufe nutzen, die Assemblerbefehle kapseln. Diese Aufrufe stehen dem Anwender in Form von Bibliotheksfunktionen zur Verfügung.

Olos wurde mit Rücksicht auf seine spätere Verifikation entwickelt und implementiert. Ein mathematisches Modell liefert die Spezifikation einer kompletten elektronischen Steuereinheit mit eingebettetem Betriebssystem. Mit Hilfe eines Simulationstheorems beweisen wir dann die Korrektheit der Olos Implementierung. Dazu benötigen wir die Definition einer Abstraktionsfunktion, die die Datenstrukturen der Implementierung auf die des Modells abbildet. Eine weitere Grundlage für den Beweis ist eine Implementierungsinvariante, die alle notwendigen Gültigkeitsaussagen über die Datenstrukturen zusammenfasst. Die Implementierungskorrektheit liefert uns die Sicherheit, dass wir unser abstraktes Modell auch auf höheren Ebenen wiederverwenden können. Unter bestimmten Annahmen können wir es beispielsweise in das verteilte Modell von Steffen Knapp [Kna08] integrieren.

Diese Annahmen benutzen wir auch für die Präsentation eines Ansatzes, der es erlaubt kommunizierende Anwendungen durchgängig zu verifizieren. Zu diesem Zweck benutzen wir den Ansatz von Daum et al. [DDB08, DDWS08], der für den Vamos-Kernel den Compiler-Satz für sequentielle Sprachen auf die Semantik von Prozessen erweitert. Dieses Ergebnis basiert auf dem von Leinenbach & Petrova [LP08] formal verifizierten Compiler, der Programme des C-Dialekts C0 in Vamp Assembler Befehle übersetzt. Wir passen diese Idee an unser Betriebssystem Olos an und (i) formalisieren die Systemaufrufe von Olos, (ii) führen eine Spezifikation von C0 und Assembler-Prozessen ein, die die sequentiellen Sprachen um die Systemaufrufe erweitern, und (iii) beweisen die Erweiterung des sequentiellen Compiler-Satzes für unsere Prozessmodelle.

Schließlich können wir das Ergebnis in unser abstraktes ECU Modell einbetten. Diese Arbeit erlaubt es interagierende Anwendungen auf abstrakter Ebene zu verifizieren, die abgeleiteten Ergebnisse bis zur Assemblerebene zu transferieren und sogar Schlüsse über das Verhalten einer ganzen ECU zu ziehen, auf der die Anwendungen laufen.

Im Wesentlichen umfasst diese Arbeit drei Teile:

(i) die Implementierung des Echtzeitbetriebssystems Olos und das mathematische Modell einer gesamten ECU
(ii) den Simulationsbeweis der Implementierungskorrektheit und
(iii) einen Ansatz, der es erlaubt kommunizierende Anwendungen, die unter Olos laufen, durchgängig zu verifizieren

Alle Definitionen, Modelle und Beweise, die im Verlauf dieser Arbeit präsentiert werden, wurden mit dem interaktiven Theorembeweiser Isabelle/HOL formalisiert.

Contents

1 Introduction

Take time to deliberate,
but when the time for action arrives, stop thinking and go in.

Andrew Jackson, quoted in: "No Ordinary Moments: A Peaceful Warrior's Guide to Daily Life", 1992 by Dan Millman

Contents

Computer systems are omnipresent and affect our daily lives. Although many people confide in computer systems only a few scrutinize their trustworthiness. Especially, their employment in safety-critical areas e. g., in medical engineering or vehicles demands for their robustness, safety and reliability. In this context a failure may cost human lives. Recently, a failing accelerator control was associated to a software bug [VB10, Guy10]. The mere rumour of such a software flaw is economically troublesome, not to mention the tragedy of possibly resulting fatal accidents. Moreover, the number of electronic devices in cars dealing with a variety of different tasks increases steadily. The growing complexity of distributed systems reveals the urgency to ensure their reliability and correct functionality. The most secure approach to rule out all systematic design errors is *verification*. This approach specifies the correct functionality of an implementation by a mathematical model and develops a proof that the system design satisfies this specification. Verification is called *formal* if the proofs are checked by a computer.

Usually, computer systems are implemented in several abstraction layers: Even the hardware is modelled at different levels of abstraction from the register-transfer layer to the instruction set architecture to the assembly layer. The operating system is based on the highest abstract hardware layer, and provides in turn a platform on which applications are implemented. In order to safely exclude all systematic errors in the design of the complete system, all layers must be regarded. We obtain *pervasiveness* of a verification attempt when it spans over multiple layers of a system. Usually, several layers are coupled by formal soundness and simulation theorems. Then, results that are derived in an abstract layer can be transferred down to a correctness theorem on a lower level.

The research project Verisoft aims at the pervasive formal verification of integrated computer systems to assure their quality already at the design stage. More specifically, the topic of this thesis belongs to the automotive subproject that aims to pervasively verify an automatic emergency call system *eCall*[Eur03].

We consider a distributed real-time system that consists of several *electronic control units* (ECUs) connected by a communication bus. Then, a network protocol schedules fixed transmission times to each connected component. This approach traces back to the *time-triggered protocol* developed by Kopetz and Grünstreidl [KG94]. Variations of this protocol are nowadays widely accepted and used in industry.

Each ECU comprises a bus interface called automotive bus controller (ABC) and a processor. The former decouples the processor from the communication bus and takes care of the timely transmission and reception of messages that are sent over the bus. The processor executes the real-time operating system Olos and several interacting applications that perform specific tasks. Basically, the functionality of Olos comprises the interaction between the communication hardware and the application software. Olos provides a processor abstraction to the applications with
(a) the exclusive access to resources like registers and memory and
(b) means for the communication with the operating system to request further services.
This abstraction is commonly referred to as a *process* and the means to communicate with the operating system are called *system calls*. As even most high-level programs are eventually compiled to run as a process, we consider any program semantics with system calls as a process. System calls are available to high-level applications in terms of library functions which encapsulate instructions of lower-levels to access low-level functionality. A system call library provides a programming interface for these calls. Thus, a programmer may use these functions without knowing the concrete implementation details of such a call.

The real-time operating system Olos was developed verification in mind. We finalized its initial implementation. A mathematical model serves as specification of an entire ECU comprising the ABC device, Olos functionality and applications. Then, we prove a simulation theorem to reveal that the Olos implementation satisfies this model. Therefore, we define an abstraction function to map data structures of the implementation to corresponding ones in the model. A further basis for the proof is an implementation invariant specifying all necessary validity constraints for these data structures. The implementation correctness ensures that our abstract ECU model can be reused on higher levels of abstraction. Under several assumptions we can integrate this model to the distributed model of Steffen Knapp [Kna08]. We also rely on this argument when we present an approach that allows to pervasively verify communicating applications.

For this purpose we use the approach of Daum et al. [DDB08, DDWS08] for the Vamos kernel that extends the sequential compiler-correctness theorem to the semantics of processes. This result relies on the formal verified compiler of Leinenbach & Petrova [LP08], which translates programs written in the C-dialect C0 to Vamp assembly instructions. We adapt this idea to our operating system Olos and (i) formalize the Olos system calls (ii) introduce the specifications of C0 and assembly processes that extend the sequential languages by the system calls, and (iii) prove the extension of the sequential compiler-correctness theorem for our process models. Finally, we embed this result in our abstract ECU model. This work is the foundation that permits to verify communicating applications on an abstract level, transfer the derived results down to the assembly layer and even infer properies about the entire ECU.

Essentially, this thesis comprises three parts:
(i) the implementation of the real-time operating system Olos and the specification of an entire ECU
(ii) the simulation theorem of the implementation correctness, and
(iii) an approach to pervasively verify communicating applications running on top of Olos

All definitions, models and proofs presented in this thesis have been formalized with the interactive theorem prover Isabelle/HOL[1].

Outline. The remainder of this chapter introduces notation, presents general approaches of verification, and presents related work. Chapter 2 describes in more detail the specific context and all necessary prerequisites of this thesis. The main part of this work starts in Chapter 3, which introduces the design principles and the implementation of the real-time operating system Olos. The formal specification of an ECU including all its subcomponents is described in Chapter 4. Chapter 5 reports on the implementation correctness proof. A simulation theorem proving the correctness of the Olos system-call library is presented in Chapter 6. Finally, in Chapter 7 we conclude and propose potential future work.

1.1 Related Work[2]

There have been numerous attempts to increase confidence in system software by means of formal methods.

An article of Klein [Kle09b] comprehensively summarizes all past and present approaches to the verification of operating-system kernels. He reports that already in the 1970s the Provable Secure Operating System (Psos) introduced concepts for OS verification and aimed at applying formal methods throughout the entire implementation of the OS. The retrospective report [NF03] from 2003 describes that Psos comprised hardware and software and strives for a useful general-purpose operating system with demonstrable security properties. Despite some simple illustrative proofs were carried out, it would be false to claim that Psos was a proved secure operating-system. Nevertheless, the approach clearly demonstrates how properties such as security could be formally proved.

The aim of the UCLA Secure Unix project [WKP80] was the verification of a kernel supporting threads, a capability-based access control, virtual memory, and device accesses. It relied on the assumptions that the compiler and the hardware work correctly. They reported that about 20% of the code proofs and 90% of the specifications were completed.

In the end of the 1980s, Bevier [Bev89] reports on the first fully verified kernel: KIT,

[1] An overview of the formal work and the corresponding directories is given in Section A.3. All proofs will soon be published in the Repository: `http://www.verisoft.de/VerisoftRepository.html`

[2] I thank Jan Dörrenbächer and Matthias Daum for the valuable knowledge about related work in their publications [Dör10, Dau10]

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 cornerstone theorem of this work is limited to memory separation of processes. Olos, in contrast, is implemented in C and has been developed with an industrial use case in mind.

Several past verification projects concentrated on the specification but fell short on the actual verification. The VFiasco project [HTS02] aims at the verification of the microkernel Fiasco implemented in a subset of C++. Besides model checking of basic safety properties in Spin [End05] for a strongly limited and simplified version of Fiasco IPC, three properties concerning ready and waiting lists have been verified in PVS [Sch07]. However, this work is less than full code verification since C++ code was transformed (only for this purpose) into PVS without defining a formal semantics of the used C++ subset. Even some functions were only described by their semantical effects, and hence the resulting implementation model is rather a specification. The VFiasco approach to C++ source code verification was continued in the Robin Project [Tew07] that investigated the verification of the Nova hypervisor. After the completion of Robin in 2008, about 70% of the high-level specification of the Nova kernel was produced. Neither, VFiasco nor Robin did manage to verifiy significant portions of the implementation.

With the capability-based microkernel Eros and its successor the Coyotos kernel [SDD+04], Shapiro and Weber defined a capability based kernel system and prove certain security policies to hold. The security model of the Eros kernel was not formally connected to the implementation. The successor project Coyotos finished a complete formal specification of the microkernel and the used programming language BitC. Some early experiments were carried out but no formal proofs have been reported yet.

Several projects regard the operating system kernel decoupled from the remaining system. Then, formal methods are applied to the isolated kernel layer with the aim to show the correctness of a given specification and some properties. More specifically, the implementation of the underlying layers are assumed to be correct. In contrast to the pervasive approach we are aiming at, it can not be ensured that the specification can be integrated or used in the upper layers of the system. This method is used e. g., of the Mask project [MWTG00]. The project's name stands for mathematically analyzed separation kernel. It guarantees a separation property that is provided by construction in the highest-level model, and by multiple formal refinement proofs down to a low-level design, which is close to an implementation. Code proofs, however, were not attempted.

The Flint project [FSDG08] does not directly aim at operating-system verification but it made a number of important contributions in the area. In the project an assembly code verification framework was developed and code for context switching on a x86 machine was formally proved to be correct. Although a verification logic for assembler code is presented, no integration of results into high level programming languages has been undertaken yet. Embedded Device [HALM06] report on the verification and Common Criteria certification of a software-based embedded device featuring a seperation kernel.

Their proof should qualify for EAL7, the highest Common Criteria evaluation level. They were successful in the verification but in contrast to a fully verified system they did not reach down to the implementation level.

Verification has a long tradition for improving the confidence in very complex operation-systems functionality but seldom attempts full code coverage. There are projects that focus on selected system components and prove some high level properties. A fruitful target for verification have been, for instance, file systems [AZKR04] and network protocols [AS97]. Moreover, the verified property and the abstraction level vary widely. Bevier et al. [BCT04], for instance, have specified the interface of the Synergy file system on a very abstract level while Yang et al. [YTE04] used model checking to systematically check implementations for specific file-system errors.

An important aspect for security-sensitive applications is the reliability of software. Secure system architectures are specifically developed for such applications. These architectures are usually based on a microkernel that faciliates the separation of legacy operating-system functionality from security-sensitive services. Therefore, they provide complete functionality of common general-purpose operating systems and reduce the trusted computing base for security-sensitive applications. 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'[3]. The Perseus framework is the basis for the Turaya security platform [LP06].

Furthermore, Green Hills Software has announced that its Integrity-178B[4] real-time operating system was certified as EAL6 which is the highest rating given to an OS so far. Nevertheless, this level of assurance only guarantees the formal treatment of software requirements whereas the functional specification, the high- and low-level design of the system are treated semi-formal and the implementation is affirmed in an informal way.

Most recent verification projects aiming at the complete code-level verification of operating system kernels are L4.verified, Verisoft [Pau05, HP07] and its successor Verisoft XT.

L4.verified. The L4.verified project [HEK+07, KEH+09] completed the world's first refinement proof for a general-purpose microkernel. L4.verified focuses on the seL4 kernel, which is based on the ARM11 platform. 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. Relying on [KEH+09, Kle09a], the L4.verified project is providing a machine-checked proof of the functional correctness of the seL4 microkernel with respect to a high level, formal

[3] The announcement has been found on the project's homepage, http://www.perseus-os.org/content/pages/Evaluation.htm.

[4] For more information, see http://www.ghs.com/products/safety_critical/integrity-do-178b.html

description of its expected behaviour. The lowest level of the verification and of the refinement is a high- performance C and ARM assembly implementation of the seL4 microkernel. The next level up, the low-level design, is a detailed, executable specification of the intended behaviour of the C implementation. This executable specification is derived automatically from a prototype of the kernel which was written in the high-level, functional programming language Haskell. While trying to avoid messy specifics of how data structures and code are optimized in C, the executable specification still represents a pretty concrete view on the kernel implementation. For instance, it still contains doubly-linked lists i. e., pointer structures. Furthermore, user transitions are completely abstracted from the semantics of user-mode instructions and specified as non-deterministically changing arbitrary user-accessible parts of the state space. The highest refinement layer is the high-level design, an abstract, operational specification of the kernel. Usually, this is accompanied with a high level of abstraction. It is reported, however, that this layer precisely describes argument formats, encodings and error reporting. Thus, for instance, some of the C-level size restrictions become visible on this level. On the other hand, the specification uses non-determinism in order to avoid an explicit description of the system. For example, neither a scheduling policy is defined at the abstract level nor the correctness of the interrupt controller management is modelled in detail. The refinement proofs are machine-checked in the interactive theorem prover Isabelle/HOL. In [Boy09, EKE08], they also defined an abstract access control model of seL4 that captures how capabilities (the kernel s access control mechanism) are distributed in the system and showed the isolation of security domains based on this model. The access control model, however, is not formally connected to the high-level design of seL4 which is used in the refinement proof. Without doubt, the results of the L4.verified project are very promissing.

However, there are several issues to be considered: The approach assumes the correctness of the C compiler, the assembly code, and the hardware[5]. Then, the abstraction level of the high-level design seems to be comparatively low (which reduces the effort regarding the refinement proof) and unsuitable to show, for instance, properties of the access control mechanism. Instead, these proofs rely on a (probably) more abstract model which is not formally connected to the high-level design in the refinement proof. Furthermore, the abstract models are not yet applied, in the sense that they are used to specify an operating system, for instance. In the future, the project plans to use their correctness proof for proving the correctness of applications running on top of the seL4 microkernel. Finally, the proof relies on a certain standard behavior of memory.

In contrast to the abstract models of seL4 microkernel, the formal model of an ECU is completely integrated into the model stack that covers all layers from applications down to hardware. The ECU model literally uses definitions from the lower layers and can be integrated to higher layers on the other hand. The seamless integration into the model stack is not the only remarkable difference between the high-level design of seL4 and the ECU model. The ECU model is used to prove functional correctness of the Olos implementation. Moreover, properties of verified applications running on an ECU can

[5]For proof assumptions, see also http://ertos.nicta.com.au/research/l4.verified/proof.pml

be integrated into properties of an entire ECU. In contrast to the non-deterministic user processes in the seL4 model, we specify the exact semantics of applications running on an ECU. This fact is crucial for pervasive verification because only an exactly defined semantics is the basis for the verification of application programs. And finally, as opposed to us the L4.verified project neither verified boot-up nor any assembly code.

Verisoft. Besides the verification of Olos, the Verisoft project also dealt with the correctness of the microkernel Vamos. Jan Dörrenbächer [Dör10] has shown large parts of the Vamos functional correctness proof and presented a model of the Vamos kernel. This model of the Vamos kernel is completely integrated into a model stack reaching from applications down to the hardware. This model establishes an abstraction level that serves as basis for the simple operating system Sos of Sebastian Bogan [Bog08] and the model of communicating user processes Coup of Matthias Daum [Dau10]. Furthermore, it is used to show the fairness of the Vamos scheduler [Dau10, DDW09]. Compared to Olos the complexity of Vamos is much higher. Most notably, the verification of the microkernel Vamos [DDW09] has reached a mature state. While this kernel has more features than Olos, the Vamos functional correctness proof does not cover the entire code.

Verisoft XT. The Verisoft XT[6] project aims at the complete code-level verification of the PikeOS kernel, the virtualization environment Hyper-V and the Baby Hyper-V. The first two verification targets are part of commercial products, whereas the latter one is seen in academic context.

- PikeOS. PikeOS [Kai07, BBBB09b] is available for Intel's x86, PowerPC, and ARM, and has an L4-like kernel with about 6,000 lines of code. The proofs are carried out by the VCC verification environment [CDH+09] (a descendant of the Spec# program-verification environment of Microsoft Research), which uses a trusted tool chain comprising the automatic verifier for concurrent C code VCC, the verification condition generator Boogie [BLW, BMSW], and the automated theorem prover Z3 [DMB08]. The supported C fragment is a large fragment of ANSI C. Recent publications [BBBB09a, BBBB09b] mainly introduce the tool chain and methodology. Apart from the verification of a simple system call which changes the priority of a thread, there are no reports yet on the portion of verified code.

- Hyper-V. [LS09, Sam08]. The kernel comprises 100,000 lines of C and 5,000 lines of assembly code and is part of a commercial virtualization environment. The hypervisor turns a single real multi-processor x64 machine (with AMD SVM or Intel VMX virtualization extensions) into a number of virtual multi-processor x64 machines. The verification is also realized by the VCC verification environment (see above); the supported C fragment is a large fragment of ANSI C. Recent

[6] More information about this project is available at http://verisoftxt.de/.

publications [CAB+09, CMST09, CMST10a, CMST10b] in this context mainly focus on the methodology of VCC and its application.

- Baby Hyper-V. Alkassar et al. [AHPP10], report on the successful verification of the so-called baby hypervisor. In comparison to the Hyper-V, the baby hypervisor and the architecture it virtualizes are very simple because it only includes the initialization of the guest partitions and a simple shadow page table algorithm for memory virtualization. However, it played an important role of driving the development of the VCC technology and applying it to system verification.

The VCC technology [CMST09] demands a high level of trust: The current VCC version uses an axiomatization of the execution model consisting of various axioms, which introduce some rather abstract concepts like concurrency and ownership of references. Though critical subproblems of the foundation are tackled by informal as well as formal proof methods, the integration into a uniform foundational theory is significantly less prioritized. In this respect, the tool chains, methodologies etc. are driven by the need to deal with the existing code that can only be changed, if errors have been revealed.

1.2 Methodology

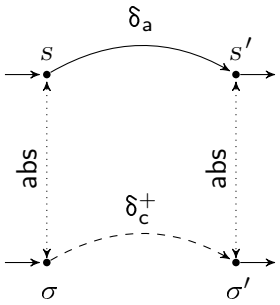

Figure 1.1: Simulation proof

In order to obtain pervasive formal verification of a system we have to consider several adjacent layers of a system. We have learned, that these layers usually are coupled by formal soundness and simulation theorems. The entire model stack of the Verisoft project is formalized in the theorem prover Isabelle/HOL. The main results of this work, namely the implementation correctness theorem (presented in Chapter 5) and the extended compiler correctness theorem (shown in Chapter 6), are both simulation proofs. Figure 1.1 depicts the technique that is used to prove simulation proofs. Roughly speaking we relate a concrete state σ to a corresponding abstract state s via an abstraction relation or function abs. Correctness is given when all possible transitions δ_c on the concrete layer result into successor states σ' that can be abstracted to corresponding states of the specification s' after a correlating transition in the model δ_a. Note that an abstract transition usually simulates a number of steps in the concrete layer.

1.3 Notation[7]

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 work is written using Isabelle's document-generating facilities, which guarantees that the presented theorems correspond to formally proved 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 intuitive higher-order logic, where universal quantification is $\bigwedge$, implication is $\Longrightarrow$, and equality is $\equiv$. Nested implications like $P_1 \Longrightarrow P_2 \Longrightarrow P_3 \Longrightarrow C$ are abbreviated with $[\![P_1; P_2; P_3]\!] \Longrightarrow C$, where we refer to P_1, P_2, and P_3 as the premises and to C as the conclusion. We may also write:

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

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.

Functions. In HOL all functions are total, e. g., nat $\Rightarrow$ nat is a total function on natural numbers. 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

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

function application by $f^0 = (\lambda x.\ x)$ and $f^{n+1} = f \circ f^n$. Function update is $f(y := v) \equiv \lambda x.$ **if** $x = y$ **then** v **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., $\lfloor x \rfloor$, and the other one is simply the additional element $\bot$. A base value can be projected by $\lceil x \rceil$, which is defined by the sole equation $\lceil \lfloor x \rfloor \rceil = x$. As HOL is a total logic, the term $\lceil \bot \rceil$ is nevertheless a well-defined yet unspecified value. For data types, we write the structural case distinction over some value v as **case** v **of** $\bot \Rightarrow g \mid \lfloor x \rfloor \Rightarrow f\ x$. Some data types might have many constructors, which are all treated in the same fashion in a certain situation. In this situation, we may write **case** v **of** Foo $\Rightarrow x \mid \_ \Rightarrow y$, which means value x if v has the value Foo, and otherwise value y. Finally, we abbreviate **case** a **of** $\bot \Rightarrow \bot \mid \lfloor x \rfloor \Rightarrow e$ with $\mathbf{let}_\bot\ x = a$ **in** e.

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. The head of list xs is computed with $\mathsf{hd}\ xs$ and the remainder, its tail, is $\mathsf{tl}\ xs$. Function last returns the last element of a list. We write $[a, b, c]$ instead of $a \odot b \odot c \odot [\,]$. With $xs \circledcirc ys$, 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 $\mathsf{set}\ xs$ 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 $\mathsf{replicate}\ n\ e$ we denote a list that consists of n elements e. Function $\mathsf{rotate1}$ takes the head of a list and appends it on its tail $\mathsf{rotate1}\ xs = \mathsf{tl}\ xs \circledcirc [\mathsf{hd}\ xs]$. We drop the last element of a non-empty list xs by applying the function $\mathsf{butlast}\ xs$. With $\mathsf{map}\ f\ xs$, the function f is applied to all elements in xs. We can extract a sublist of j elements starting from the i-th element of a given list xs by using function $\mathsf{extr}\ i\ j\ xs$. With $[i..{<}j]$ we denote a list of consequitive natural numbers from i to j (exclusively). Function $\mathsf{zip}\ xs\ ys$ takes two lists xs and ys and returns a new list by combining the list elements to pairs. We use function fold in order to apply function f with a list of inputs to a state s. The recursively defined function returns state s when the input list is empty, otherwise it starts its computation with the head of the input list: $\mathsf{fold}\ f\ (x \odot xs)\ s = \mathsf{fold}\ f\ xs\ (f\ x\ s)$. Function $\mathsf{list\_sum}\ xs$ returns the sum of all numbers stored in list xs.

Records and Tuples. A record is constructed by assigning all of its fields: $(\!|\mathsf{fld}_1 = v_1, \mathsf{fld}_2 = v_2|\!)$. Field fld_1 of record r is selected by $r.\mathsf{fld}_1$ and updated with value x via $r(\!|\mathsf{fld}_1 := x|\!)$. The first and second component of a pair $p = (p_1, p_2)$ can be accessed with the

functions $\mathsf{fst}\ p = p_1$ and $\mathsf{snd}\ p = p_2$. Tuples with more than two components are pairs nested to the right.

2 Background

"What is the right way to listen to a fugue:
as a whole, or as the sum of its parts?"

D. Hofstadter, quoted in: Gödel, Escher, Bach: An Eternal Golden Braid

Contents

The context of this work is the German Verisoft project, a large-scale research project with partners from industry and academia, funded by the German government. The main goal of the project is the *pervasive formal verification* [BHMY89] of computer systems.

In order to prevent errors and gain compatible verification results, all theories are developed in the logical framework of the interactive theorem prover Isabelle/HOL.

Verisoft focuses on three goals: (i) the creation of methods and tools permitting the pervasive formal verification of computer-system designs (ii) an increase of the industrial productivity and quality, and (iii) the prototypical realization of four concrete computer systems, where three are from the industrial sector.

More precisely, this work belongs to the automotive subproject where we consider a distributed real-time system that consists of several *electronic control units* (ECUs)

connected by a bus. The ECUs comprise an automotive bus controller (ABC) and a processor executing the real-time operating system Olos and several application programs.

This chapter presents a general view of the context of this thesis and introduces all required fundamentals used for the implementation, specification and verification of our operating system Olos.

In Section 2.1, we introduce the scope of this work and classify its dependencies. Therefore, we give a short overview of the implementation layers in our software system. The behaviour of each layer is specified by a computational model. We summarize the general verification approach that has been developed within Verisoft for sequential programs. This approach can deal with mixed-language implementations. In our case, we rely on two different languages: Vamp assembly [MP00] and C0 [Lei08], a fragment of C. We briefly present both languages in Section 2.2 and Section 2.3, respectively. Then, we outline the compiler-correctness theorem (Section 2.4) of Leinenbach & Petrova [LPP05, Pet07, LP08].[1] Section 2.5 briefly describes an approach of Starostin & Tsyban [ST08, Tsy09] to reason formally about inline-assembly portions in C0 programs. More specifically, we show how to lift the effects of the assembly instructions back to the C0 level.

Furthermore, we present the concept of interrupts [MP00] (Section 2.6) and the device model [AH08] (Section 2.7). Olos is implemented on the verified RISC processor Vamp [BJK+06] using a programming framework called communicating virtual machines (Cvm) [GHLP05, IT08]. We describe this framework in Section 2.8. Finally, we introduce general concepts of the verification environment Isabelle/Simpl [Sch06] in Section 2.9.

2.1 System and Proof Architecture

In this section, we briefly introduce the implementation layers of our system. More specifically, we describe how the operating system Olos is integrated in the system-stack. Moreover, we present the general verification approach for sequential programs that has been developed within Verisoft and specify the semantic stack layers we used in the following work.

System Stack. The system software layers used in the automotive subproject are depicted in Figure 2.1 on the next page. Our work is restricted to the software layers (depicted as white boxes) built on top of a model 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 real-time operating system Olos is implemented in C0 without extra portions of inlined assembly. Both layers interact via C0 function calls: The Cvm framework calls the top-level function `kdispatch` of Olos, and Olos itself uses Cvm primi-

[1] For major parts of Section 2.2,Section 2.3, and Section 2.4, I am indebted to Matthias Daum [Dau10]

tives to access low-level functionality. While these two layers run in system mode of the processor, applications are executed in user mode on top of our operating system. The user applications may request services from OLOS using so-called *system 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. 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 application is to resume when the kernel execution finishes. The functionality of OLOS is accessible for applications system calls. Technically, a system call is implemented using the special instruction trap: This instruction generates an exception similar to e. g., an arithmetic overflow. Exceptions are a 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 OLOS. OLOS can examine and manipulate the state of the application using CVM primitives. That means, the calling application can store parameters to a system call in registers and memory in order to describe its specific demand from OLOS. The operating system, in turn, interprets the inquiry by examining the applications state and in response, it alters this state accordingly. A well-defined interface precisely defines this interaction between the operating system and applications by assigning a system-call semantics to register values and memory contents.

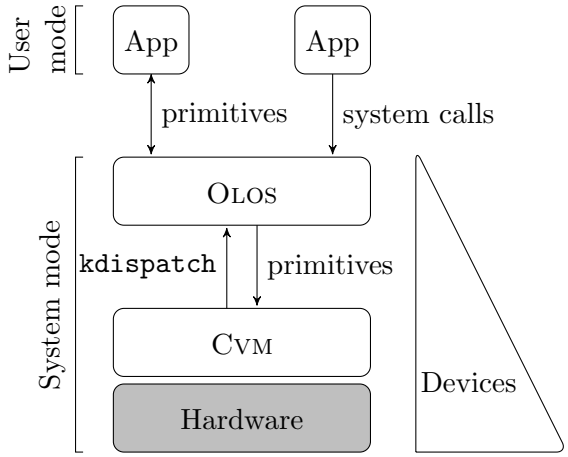

Figure 2.1: System stack

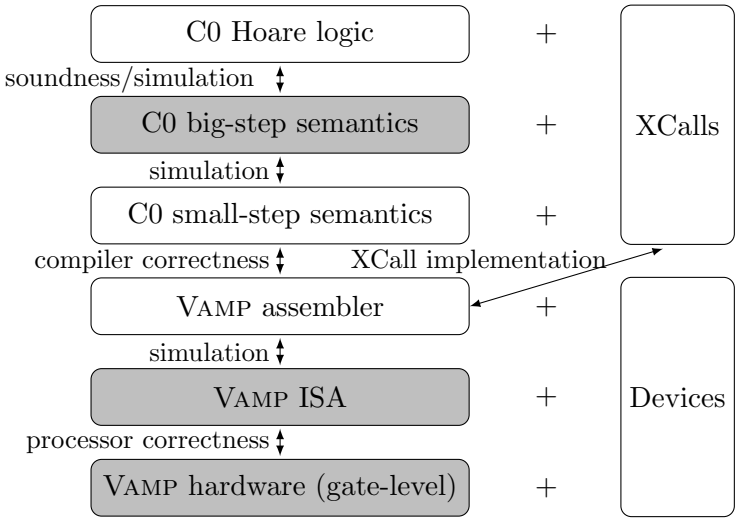

Figure 2.2: Semantics Stack

Semantics Stack. An overview of the semantics stack in depicted in Figure 2.2. The C0-semantics stack consists of a Hoare Logic on the highest level, a big-step and a small-step semantics. The underlying VAMP machine level includes an assembly layer, an instruction set architecture, and at the lowest layer a gate-level hardware. All results gained on a specific layer can be transferred down to the lowest layer by applying simulation, soundness and correctness proofs that connect all adjacent levels with each other. The lowest layer used in this thesis to prove correctness theorems is the VAMP assembly layer.

In the Hoare logic we can reason about pre- and postconditions of sequential, type-safe, and assembly-free C0 programs. Compiler correctness, in contrast, is formulated at the small-step semanctics layer. Both layers are bridged by the big-step semantics which is suited to express results of the Hoare logic operationally. The differences of these layers reflect their purpose: Whereas the Hoare logic was designed to support verification of individual programs, the small-step semantics supports considerations about interleaving programs at the system level. In the context of system code, we often have to deal with portions of inlined assembly code. The effects on low-level computations cannot be expressed at the C0 layers. There is an approach of abstracting these computations into atomic functions. These functions, referred to as extended calls (*XCalls*), encapsulate the portion of inlined assembler code. For this purpose, a C0 state is extended with an additional component representing external states e. g., devices or applications. Then, the XCall is a procedure call that performs a transition on the external state and communicates with C0 via parameter passing and return values. A more detailed description of XCalls is presented in [ASS08]. This model can be integrated into all C0-semantics layers. XCalls can be transferred down to the VAMP assembler semantics where they are finally discharged by an implementation proof.

Similar to the XCalls in the upper layers, devices can be integrated into the lower

layers. Their state and transition functions are shared between all semantic layers of the VAMP. The device transition functions as well as the VAMP semantics describe interleaved small-step computations to obtain concurrent execution of the combined system. A fundamental prerequisite to prove a global property of the combined system is to disentangle the different computations by means of reordering [AH08, Alk09] in order to use individual transfer results then. A comprehensive report on the semantic stack is given in [AHL+09].

All layers used in context of this thesis are shown as white boxes. Our operating system OLOS is written in the sequential language C0. The implementation-correctness theorem is specified and carried out at the Hoare logic layer using XCalls to manipulate integrated application and device states. The correctness-proofs of the system-call library are carried out on lower levels. We prove a simulation theorem stating that all C0 functions implementing system calls simulate a certain number of steps of the VAMP assembly machine executing the compiled code. The augmentation of the compiler correctness theorem involves the C0 small-step as well as the VAMP-assembly level.

2.2 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 was 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 in such a way 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 an 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 modelled 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 4.2.2.

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 either 32-bit integers to naturals or vice versa.

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

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

$$\begin{aligned}&\mathsf{is\_valid\_asm}\ s_{\mathsf{asm}} \equiv\\ &\ s_{\mathsf{asm}}.\mathsf{dpc} < 2^{32} \wedge s_{\mathsf{asm}}.\mathsf{pcp} < 2^{32} \wedge |s_{\mathsf{asm}}.\mathsf{gprs}| = 32 \wedge |s_{\mathsf{asm}}.\mathsf{sprs}| = 32 \wedge\\ &\ (\forall i{\in}\{0{<}..{<}32\}.\ -2^{31} \leq s_{\mathsf{asm}}.\mathsf{gprs}\ !\ i < 2^{31}) \wedge\\ &\ (\forall i{\in}\mathsf{used\_sprs}.\ -2^{31} \leq s_{\mathsf{asm}}.\mathsf{sprs}\ !\ i < 2^{31}) \wedge (\forall \mathit{addr}.\ -2^{31} \leq s_{\mathsf{asm}}.\mathsf{mm}\ \mathit{addr} < 2^{31})\end{aligned}$$

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

$$\mathsf{is\_system\_mode}\ s_{\mathsf{asm}} \equiv s_{\mathsf{asm}}.\mathsf{sprs}\ !\ \mathsf{MODE} = 0$$

In the course of this thesis, we do not deal with 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:

$$\mathsf{mm\_size}\ s_{\mathsf{asm}} \equiv \mathsf{to\_nat32}\ (s_{\mathsf{asm}}.\mathsf{sprs}\ !\ \mathsf{PTL} + 1)$$

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.[2] Otherwise, the transition is specified by a simple case distinction on current_instr s_{asm}. We use the notation $\delta^n_{asm}\ s_{asm}$ to denote the result of executing n steps of the assembly machine starting in state s_{asm}.

Memory Access Instructions. In the course of this thesis, two memory access instructions are of special interest. They are used to transfer data from memory into a general purpose register and vice versa. Both operations have three parameters:

- *RD* is the register in which the memory data is stored or of which register it is loaded
- *RS1* is the register containing the memory source or destination address in the memory
- *i* is the offset of the memory address

The *load-word instruction* llw *RD RS1 i* loads a word from the memory to a specified general purpose register, whereas the *store-word instruction* lsw *RD RS1 i* stores a word from a general purpose register into the memory.

For both instructions, we compute the target address in the memory from the address stored in register *RS1* and an offset given as immediate value *i*. The result is called the *effective address* and is formally defined as:

$$\text{ea}\ s_{asm} \equiv (\text{to\_nat32}\ (s_{asm}.\text{gprs}\ !\ RS1) + \text{to\_nat32}\ i)\ \text{mod}\ 2^{32}$$

A number n is *divisible* by k if k divides n without leaving a remainder i. e.,

$$(k\ \text{dvd}\ n) = (n\ \text{mod}\ k = 0)$$

The semantics of a load-word instruction where the current assembly state is valid and the effective address is well-formed (i. e., it is in range and word aligned) is given with:

$$\begin{aligned}
&\text{current_instr}\ s_{asm} = \text{llw}\ RD\ RS1\ i \wedge \\
&\text{is_valid_asm}\ s_{asm} \wedge 4\ \text{dvd ea}\ s_{asm} \wedge \text{ea}\ s_{asm} < 2^{32} \Longrightarrow \\
&\delta_{asm}\ s_{asm} = \\
&s_{asm} \\
&(\!|\text{dpc} := s_{asm}.\text{pcp},\ \text{pcp} := (s_{asm}.\text{pcp} + 4)\ \text{mod}\ 2^{32}, \\
&\ \text{gprs} := s_{asm}.\text{gprs}[RD := s_{asm}.\text{mm}\ (\text{ea}\ s_{asm}\ \text{div}\ 4)]|\!)
\end{aligned}$$

The data stored in the memory at the effective address is copied to general purpose register *RD*. Moreover, the program counters are increased.

[2] Technically, we avoid extra case distinctions for the detection of all possible invalid cases but sort out those cases before we apply the transition function. In some illegal cases the transition function is meaningless, although it would not get stuck.

The store-word instruction applied on a valid assembly state with a well-formed effective address and an existing register index RD has the following effects: The value of the general purpose register RD is stored to the effective address in the memory and the program counters are increased. Reading from general purpose register 0 is defined as 0. Formally:

$$
\begin{aligned}
&\mathsf{current\_instr}\ s_{\mathsf{asm}} = \mathsf{lsw}\ RD\ RS1\ i \wedge \\
&\mathsf{is\_valid\_asm}\ s_{\mathsf{asm}} \wedge 4\ \mathsf{dvd}\ \mathsf{ea}\ s_{\mathsf{asm}} \wedge \mathsf{ea}\ s_{\mathsf{asm}} < 2^{32} \wedge RD < 32 \Longrightarrow \\
&\delta_{\mathsf{asm}}\ s_{\mathsf{asm}} = \\
&s_{\mathsf{asm}} \\
&(\!|\mathsf{dpc} := s_{\mathsf{asm}}.\mathsf{pcp},\ \mathsf{pcp} := (s_{\mathsf{asm}}.\mathsf{pcp} + 4)\ \mathsf{mod}\ 2^{32}, \\
&\ \mathsf{mm} := s_{\mathsf{asm}}.\mathsf{mm}(\mathsf{ea}\ s_{\mathsf{asm}}\ \mathsf{div}\ 4 := \textbf{if}\ RD = 0\ \textbf{then}\ 0\ \textbf{else}\ s_{\mathsf{asm}}.\mathsf{gprs}\ !\ RD)|\!)
\end{aligned}
$$

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 have restricted 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 must 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 `~`, logical negation `!`, and conversion operations between integral values. Binary operations are arithmetic operations (`+`, `-`, `*`, `/`, `%`), bitwise operations (`|`, `&`, `^`, `<<`, `>>`), and comparisons (`>`, `<`, `==`, `!=`, `>=`, `<=`) as well as the lazy binary operations (Boolean conjunction `&&` and disjunction `||`). Left expressions are expressions that re-

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
LazyBinOp $lbop$ e_1 e_2	lazy binary operation
UnOp uop e	unary operation
AddrOf e_l	address of left-expression e_l
Deref e	dereferencing e

Table 2.1: C0 expressions e

fer to memory objects, namely variable, array and structure accesses as well as pointer dereferencing.

The statements in C0 permit assignments, dynamic memory allocation, sequential composition, conditional and repeated execution, inlined assembly, function calls, and returns from functions. Some of the statements are tagged with unique statement identifiers of type *sid*. These identifiers are mainly used in the compiler correctness proof to examine the relationship between statements and to determine their original order within a program. The empty statement and the sequential composition are called *structural* and are the only statements that are not tagged with statement identifiers.

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.
- Return statements with return expression e and statement identifier sid are mod-

Skip	the empty statement
Comp s_1 s_2	sequential composition
Ass e_l e sid	assignment of expression e to left-expression e_l
PAlloc e_l tn sid	allocation of an object of type name tn and assignment to e_l
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

elled with RETURN *e sid*.
- A list of VAMP assembly instructions *ls* can be inlined with statement ASM *ls sid*. This statement extends C0 to $C0_A$ and the semantics of $C0_A$ programs is defined as alternating C0 small-steps semantics and VAMP assembly semantics.
- The sequential composition COMP s_1 s_2 presents the consecutive execution of the statements s_1 and s_2.
- Finally, the empty statement SKIP does nothing.

C0 Memory Configuration. The C0 memory configuration stores information about variables of a C0 program together with their values. As C0 is perfectly type-safe, the memory is typed and only stores fundamental types.

A *memory frame* consists of three parts:
- The content that maps addresses to memory cells, where elementary values are stored,
- a symbol table that holds all variables of the memory frame together with their types and,
- finally a set of already initialized variables.

The memory itself includes one memory frame for global variables and one for heap objects. The local variables are stored in several stack frames. Each of them consists of a single memory frame stored together with the return destination of the calling function. The local memory is modelled as a list of stack frames where the topmost stck frame is stored at the *end* of this list. Thus, the index of stack frame remains constant during its lifetime. The recursion depth recursion_depth returns the current number of stack frames in the memory. The topmost local memory frame has consequently the index recursion_depth $s_{C0} - 1$. We can access the topmost stack frame with function toplm and the return destination with function toprd. The symbol table of the global variables can be extracted from a memory state with the function gm_st, whereas the symbol table of the top most stack frame can be computed with function toplm_symbols. Function extract_symbolconf extracts the symbol tables of the global, the local and the heap memory.

Generalized Variables. Pointers in the small-step semantics are represented in a structural way using so-called *generalized variables* short g-variables. They are defined inductively and are structurally similar to left-expressions. There are three base cases for g-variables:
- gvar_gm x: global variables of name x,
- gvar_lm x i: local variables with name x located in the i-th local memory frame of the stack, and
- gvar_hm i: nameless heap variables that are referenced by an index i

The inductive case defines g-variables for array and structure access:
- gvar_arr g i: the i-th array element is a g-variable when g is an array type g-variable
- gvar_str g n: the component with name n of a structure type g-variable g is also a g-variable

The predicates is_global_gvar, is_local_gvar, and is_heap_gvar hold when the corresponding g-variable is global, local or a heap variable. Note that these predicates are dichotomous. Local g-variables that are located in the topmost stack frame fulfill the predicate is_top_local_gvar.

The set of subvariables sub_gvars of a given g-variable is defined inductively: Initially, a g-variable itself is contained in this set and all array elements and components of a structure typed g-variable are contained. For a given g-variable x that belongs to the set of subvariables of g-variable g, we denote: $x \in \mathsf{sub\_gvars}\ g$. For composed g-variables, we call the highest ancestor of g-variables the root and then predicate is_root_gvar holds [Lei08, Def.4.11]. A g-variable is initialized if its root g-variable belongs to the set of initialized variables of the corresponding memory frame; g-variables in the heap are initialized by definition. For an initialized g-variable *gvar* predicate gvar_initialized *m gvar* holds [Lei08, Def.4.20]. The set of all reachable g-variables is denoted with reachable_gvars [Lei08, Sec. 8.2.2]. We call a g-variable g reachable when it is a global or local variable, another reachable pointer g-variable points to g or g is a sub-variable of a reachable g-variable. The function vlookup searches in the global memory or the topmost local memory frame of memory state m a g-variable that corresponds to a given variable name *vn*. If it finds no corresponding g-variable, it returns $\bot$. Finally, function addrof_gvar takes a type table *tt*, a symbol table *sc* and a g-variable and returns a record consisting of the type ad_tn, the start address ad_base of the memory object as well as the name of the memory where the object is stored ad_memname.

Expression Evaluation. Expression evaluation in the C0 small-step semantics uses a function eval that takes the type table, a memory configuration, and an expression. When the evaluation succeeds it returns a data slice *ds*. A data slice is represented as a record with five components:

1. the left value of the expression *ds*.ds_lval,
2. its type *ds*.ds_type,
3. a flag determining whether the expression represents a memory object *ds*.intermediate,
4. a flag indicating whether the expression is initialized *ds*.ds_initialized and,
5. the expression value *ds*.ds_data

With a given g-variable, a memory configuration and a type table we can compute the referenced data in form of a data slice by using function get_dataslice. When the type of the referenced data is elementary i. e., no array or structural type, we call the corresponding g-variable elementary and predicate elementary_gvar holds.

We use the function mem_update *tt m g d* in order to manipulate memory objects. This function updates the object denoted by the g-variable g in memory m by the new data d. 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, this function is partial.

Program Rest. The program rest contains statements that still have to be executed. More specifically, the program rest is one of the statements as depicted in Table 2.2 on

page 21. Initially, it holds the body of the main function and grows or shrinks, depending on the execution of the program. At this place, we define two functions that manipulate the program rest. If the first statement of the program rest is a function call, function rm_scall replaces it with a Skip or does not change anything otherwise. Formally:

```
rm_scall stmt =
(case stmt of Comp s s' ⇒ Comp (rm_scall s) s'
 | SCall lv fn es sid ⇒ Skip | _ ⇒ stmt)
```

Function remove_fst_stmt is used to remove the first statement of a program rest. A sequential composed statement recursively call the function until its first sub-statement is not composed. In this case, remove_fst_stmt returns the second sub-statement of the sequential composition. Single statements are simply replaced with a Skip statement. We define function remove_fst_stmt as follows:

```
remove_fst_stmt stmt =
(case stmt of
 Comp s s' ⇒
   if is_Comp s then Comp (remove_fst_stmt s) s' else s'
 | _ ⇒ Skip)
```

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

Some of the proofs employ an extended C0 state that does not only consist of the dynamic program state but additionally includes the static program definition. The extended C0 state is called *monolithic* and consists of the following components:

- s_{C0}.pstate a C0 program state including a program rest and the memory state as defined above,
- s_{C0}.ttab a type table and,
- s_{C0}.ftab a function table.

An evaluation function eval_m computes the evaluation of a given expression together with a given monolithic state s_{C0}. Internally, it extracts the type table and the memory

state of s_{C0} and invokes eval to compute the corresponding data slice. We embed the transition function δ_{C0} into a partial transition function for monolithic states δ_{C0m}. Here, we simply change component s_{C0}.pstate and leave the static program definitions unchanged. In principle, all proofs using the monolithic type can be formulated with a C0 program state and the static program definitions. Nevertheless, hiding the tables often simplifies reading our formulae.

Function Call and Return. To cut a long story short, the semantics of a function call is the following: the stack is extended by a new stack frame on the top, the function parameters kept in the parameter list *es* are evaluated and copied to the frame and the return destination stores the g-variable of the function call's left variable *lv*. These operations are encapsulated in the function extend_stack. In the program rest the function call is substituted by the function body. The return statement should be the last instruction in a function body. It updates the left variable of the function call *lv* with the evaluated return expression, deletes the topmost stack frame and sets the new program rest to SKIP. Function remove_toplm *m* computes the changed memory state after the removal of the topmost stack frame. A more detailed description of the C0 small-step semantics concerning function calls will be given in Chapter 6.

Here, we define a useful function set_primres that directly updates the left variable of a function call with a primitive value (i. e., values of an elementary type). It takes a monolithic C0-state and a primitive return value *rv* and returns a modified C0 memory state. In the case that the first statement of the program rest is a function call, it directly updates the memory at the address of the lefthand variable of the function call with the return value. Therefore, eval_m computes a data slice from the literal expression of the primitive value *rv*. If the first statement is not a function call or the memory update fails, function set_primres returns $\bot$. Formally:

```
set_primres rv sC0 ≡
  case fst_stmt sC0.pstate.prog of
  SCALL lv fn es sid ⇒
    mem_update sC0.ttab sC0.pstate.mem ⌈evalm sC0 (VARACC lv)⌉.ds_lval
     ⌈evalm sC0 (LIT (Prim rv))⌉
  | _ ⇒ ⊥
```

Validity of C0 States. We briefly summarize the constraints of a valid C0 state s_{C0} with respect to a type table *tt* and a function table *ft*. The requirements are preserved under all small-step transitions δ_{C0} and comprise the following claims:

1. the function table is valid: (*tt*, gm_st s_{C0}.mem, *ft*) ∈ valid_ft
2. the program rest of s_{C0} is valid
3. the number of return statements in the program rest is less than the recursion depth of the program
4. the stack of s_{C0} is valid
5. the type table *tt* is valid valid_tenv *tt*;

6. the symbol table of the global variables is valid valid_symboltable tt (gm_st s_{C0}.mem)
7. all local symbol tables correspond to a function in the function table ft
8. the types of all heap variables are valid
9. all memory frames are type correct and,
10. the return destinations are valid.

All these requirements define the set of valid C0 states. We denote this set with: $s_{C0} \in$ valid_C0confs tt ft

As we focused only on definitions used in this thesis, we refer to [Lei08] for a comprehensive and detailed description of the C0 language.

2.4 Compiler Correctness

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. The compiler correctness theorem always starts in an initial state and can only deal with sequential C0 semantics. More specifically, the theorem does not hold when inline assembly is executed. In the course of this thesis we introduce function calls that are used by applications to communicate with the operating system. These C0 functions embed inline assembly code. We strengthen the induction step of the compiler correctness theorem in order to deal with these language extensions. Moreover, we can start from any C0 state that fulfills the preconditions.

We first describe the memory layout of compiled programs. Briefly summarized, there are three memory regions allocated:

i a code region containing the compiled code,
ii the stack, where all stack frames are stored and,
iii the heap.

For each local memory frame the stack stores the content of its C0 counterpart and additionally a frame header. The latter stores the address of the result variable and the return address of the function call in the code. The start address of the code is denoted by *code_base* whereas the heap starts at address *heap_base*. Several general purpose registers keep track of addresses pointing to

- the stack start (i. e., sbase s_{asm} = to_nat32 (s_{asm}.gprs ! 28)),
- the current top element of the heap (i. e., toph s_{asm} = to_nat32 (s_{asm}.gprs ! 29))

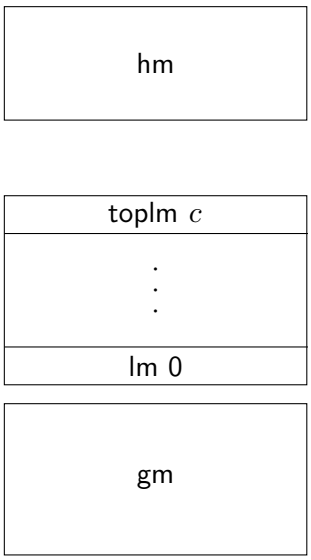

Figure 2.3: C0 memory layout

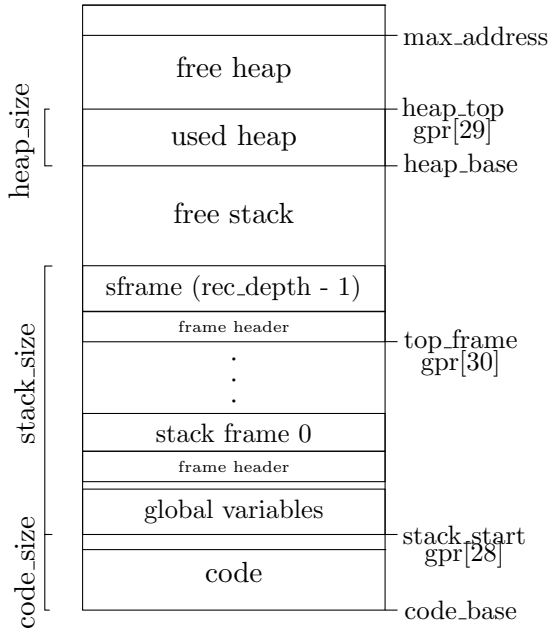

Figure 2.4: Memory layout of the Compiled Program

and,

- the topmost stack frame (i. e., toplm_base s_{asm} = to_nat32 (s_{asm}.gprs ! 30)).

All parameters of the last function call that created the topmost stack frame are stored relatively to address toplm_base. Leinenbach gives a more detailed description of the memory layout in his thesis [Lei08, Chapter 7.2].

Next, 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 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 table[3] (gm_st s_{C0}.mem), to the corresponding memory region in s_{asm} (*code consistency*), and

[3] Note 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.

- 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 s_{C0} s_{asm} ≡
  let *code* = codegen_program tt ft (gm_st s_{C0}.mem)
  in ∀i<|*code*|.
      decodable (s_{asm}.mm (program_base + i)) ∧
      to_instr (s_{asm}.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}:

is_initial_asm tt ft gst s_{asm} ≡
  is_valid_asm s_{asm} ∧ s_{asm}.dpc = 4 · program_base ∧ s_{asm}.pcp = s_{asm}.dpc + 4 ∧
  code_consistent tt ft (init_C0 ft gst) s_{asm} ∧
  (∀i∈gm_range tt ft gst. s_{asm}.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 statements[4], i. e., we require s_{asm}.pcp =

[4]Recall 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.

$s_{\mathsf{asm}}.\mathsf{dpc} + 4$. Finally, we assume a well-formed assembly state, i. e., $\mathsf{is\_valid\_asm}\ s_{\mathsf{asm}}$ (see page 18).

A few auxiliary predicates help to formally specify whether a compiled C0 program is successfully executed on the VAMP assembly level: The predicate 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, 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 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

$$
\begin{aligned}
&(\mathit{crange}, \mathit{arange}) \vdash_{\mathsf{asm}} s_{\mathsf{asm}} \rightarrow^{n} s'_{\mathsf{asm}} \equiv \\
&\ \delta^{n}_{\mathsf{asm}}\ s_{\mathsf{asm}} = s'_{\mathsf{asm}} \wedge \mathsf{is\_valid\_asm}\ s'_{\mathsf{asm}} \wedge s'_{\mathsf{asm}}.\mathsf{pcp} = s'_{\mathsf{asm}}.\mathsf{dpc} + 4 \wedge \\
&\ (\forall m{<}n.\ \neg\ \mathsf{mem\_write\_inside\_range}\ (\delta^{m}_{\mathsf{asm}}\ s_{\mathsf{asm}})\ \mathit{crange} \wedge \\
&\quad \mathsf{inside\_range}\ \mathit{crange}\ (\delta^{m}_{\mathsf{asm}}\ s_{\mathsf{asm}}).\mathsf{dpc} \wedge \\
&\quad \mathsf{mem\_access\_inside\_range}\ (\delta^{m}_{\mathsf{asm}}\ s_{\mathsf{asm}})\ \mathit{arange} \wedge \\
&\quad \neg\ \mathsf{is\_exception}\ (\delta^{m}_{\mathsf{asm}}\ s_{\mathsf{asm}}) \wedge \\
&\quad \mathsf{is\_legal\_instr}\ (\delta^{m}_{\mathsf{asm}}\ s_{\mathsf{asm}})\ (\mathsf{current\_instr}\ (\delta^{m}_{\mathsf{asm}}\ s_{\mathsf{asm}})))
\end{aligned}
$$

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 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 2.2). 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 (in such a way 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 $\mathsf{is\_compilable}$ *tt ft gst*.

Second, the required memory of C0 states s_{C0} change dynamically during the execution of a C0 program, e. g., with the recursion depth or by the allocation of heap variables. Predicate $\mathsf{sufficient\_memory}$ *max_address tt ft* s_{C0} 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 $\mathsf{is\_Asm}$ determines, whether a given C0 statement is an assembly statement. We denote the first statement of the remaining program in a C0 state s_{C0} by $\mathsf{fst\_stmt}\ s_{\mathsf{C0}}.\mathsf{prog}$.

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 $\bot$.

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* $\mathsf{alloc'}$*, and a final assembly state* $\mathsf{s'_{asm}}$ *in such a way that*

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

Formally:

$$
\begin{aligned}
&[\![\mathsf{is\_compilable}\ tt\ ft\ gst;\\
&\ \delta^{n}_{\mathsf{C0}}\ tt\ ft\ (\mathsf{init\_C0}\ ft\ gst) = \lfloor s'_{\mathsf{C0}} \rfloor;\\
&\ \forall i{\leq}n.\ \mathsf{sufficient\_memory}\ max\_address\ tt\ ft\ \lceil \delta^{i}_{\mathsf{C0}}\ tt\ ft\ (\mathsf{init\_C0}\ ft\ gst)\rceil;\\
&\ max\_address \leq 2^{32};\\
&\ \forall i{<}n.\ \neg\ \mathsf{is\_Asm}\ (\mathsf{fst\_stmt}\ \lceil \delta^{i}_{\mathsf{C0}}\ tt\ ft\ (\mathsf{init\_C0}\ ft\ gst)\rceil.\mathsf{prog});\\
&\ \mathsf{is\_initial\_asm}\ tt\ ft\ gst\ s_{\mathsf{asm}}]\!]\\
&\Longrightarrow \exists t\ alloc'\ s'_{\mathsf{asm}}.\\
&\qquad (\mathsf{code\_range}\ tt\ gst\ ft, \mathsf{address\_range}\ max\_address) \vdash_{\mathsf{asm}} s_{\mathsf{asm}} \rightarrow^{t} s'_{\mathsf{asm}} \wedge\\
&\qquad \mathsf{consistent}\ tt\ ft\ s'_{\mathsf{C0}}\ alloc'\ s'_{\mathsf{asm}} \wedge s'_{\mathsf{asm}}.\mathsf{sprs} = s_{\mathsf{asm}}.\mathsf{sprs}
\end{aligned}
$$

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 $\mathsf{init\_C0}$ *ft 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 [5]. □

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. For the induction step, the definition of valid C0 programs presented in [Lei08] does not suffice. Furthermore, the compiler requires some additional assumptions about the execution environment. Besides valid C0 states, the definition valid_C0 includes a relation between the number of returns in the program rest and the recursion depth of the C0 state. Moreover, translatable_programs formalizes *static* resource restrictions demanding that the generated code fits into the memory of the target machine. The extension of the validity predicate for C0 states that are compiled to VAMP assembly is proposed in [AHL+09]. The stronger definition of valid C0 states is given below:

$$
\begin{aligned}
&\mathsf{valid\_C0}\ tt\ ft\ s_{\mathsf{C0}} \equiv \\
&\quad s_{\mathsf{C0}} \in \mathsf{valid\_C0confs}\ tt\ ft \wedge \\
&\quad (\mathsf{nr\_toplevel\_returns}\ (\mathsf{s2l}\ s_{\mathsf{C0}}.\mathsf{prog}) + 1) = \mathsf{recursion\_depth}\ s_{\mathsf{C0}}.\mathsf{mem} \wedge \\
&\quad (tt,\ ft,\ \mathsf{gm\_st}\ s_{\mathsf{C0}}.\mathsf{mem}) \in \mathsf{translatable\_programs}
\end{aligned}
$$

Now, we present the stronger lemma of the induction step:

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 when 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 $\mathsf{alloc'}$, and an assembly state $\mathsf{s'}_{\mathsf{asm}}$ so that

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

Formally:

$$
\begin{aligned}
&[\![\mathsf{valid\_C0}\ tt\ ft\ s_{\mathsf{C0}};\ \mathsf{is\_valid\_asm}\ s_{\mathsf{asm}};\ s_{\mathsf{asm}}.\mathsf{pcp} = s_{\mathsf{asm}}.\mathsf{dpc} + 4; \\
&\ \mathsf{consistent}\ tt\ ft\ s_{\mathsf{C0}}\ alloc\ s_{\mathsf{asm}};\ \delta_{\mathsf{C0}}\ tt\ ft\ s_{\mathsf{C0}} = \lfloor s'_{\mathsf{C0}} \rfloor; \\
&\ \neg\ \mathsf{is\_Asm}\ (\mathsf{fst\_stmt}\ s_{\mathsf{C0}}.\mathsf{prog});\ \mathsf{sufficient\_memory}\ max\_address\ tt\ ft\ s_{\mathsf{C0}}; \\
&\ \mathsf{sufficient\_memory}\ max\_address\ tt\ ft\ s'_{\mathsf{C0}};\ max\_address \leq 2^{32}]\!] \\
&\Longrightarrow \exists t\ alloc'\ \mathsf{s'}_{\mathsf{asm}}. \\
&\qquad (\mathsf{code\_range}\ tt\ (\mathsf{gm\_st}\ s_{\mathsf{C0}}.\mathsf{mem})\ ft,
\end{aligned}
$$

[5] http://www.verisoft.de/VerisoftRepository.html

address_range $max\_address$) $\vdash_{\mathsf{asm}} s_{\mathsf{asm}} \rightarrow^t s'_{\mathsf{asm}} \wedge$
consistent tt ft s'_{C0} $alloc'$ $s'_{\mathsf{asm}} \wedge s'_{\mathsf{asm}}.\mathsf{sprs} = s_{\mathsf{asm}}.\mathsf{sprs}$

2.5 Switching the Layers - Inlined Assembly Code

In this section, we describe in detail how to deal with inline assembly code within a C0 program. The semantics of C0 with portions of inline assembly cannot be solely described with C0 states. In fact, we have to switch between two semantical layers. For this purpose, Gargano et al. [GHLP05] proposed an approach to deal with code using C0 statements and portions of inline assembly. This approach requires to maintain the compiler consistency relation after the execution of each single instruction in the inlined assembly portion. The method turned out to be inconvenient because it lead to excessive complex formal proofs. Therefore, the earlier approach was improved and used by Starostin & Tsyban [ST08]. Here, we present their technique and the definitions we will use later on.

Figure 2.5 depicts a szenario where a C0 program with inlined assembly code is executed. As long as no inline assembly code occurs, we apply C0 small-step semantics. Otherwise, the execution is switched to a consistent assembly state and continues directly there. In our example the assembly statement Asm *il sid* is encountered in the C0 state s^i_{C0}. Then, we switch to a consistent assembly state $s^{s(i)}_{\mathsf{asm}}$. The switching is justified by compiler correctness that relates states of both semantic layers by function consistent. Next, we execute the instruction list *il* according to the assembly semantics. After the list of assembly instructions has been executed, we switch back to the C0 level. Thus, function C0_asm_upd constructs a C0 state s^{i+1}_{C0} that is consistent to the final assembly state $s^{s(i)+t}_{\mathsf{asm}}$. Hereinafter, we describe how to construct a consistent new C0 state out of the new and old assembly states and the old C0 state.

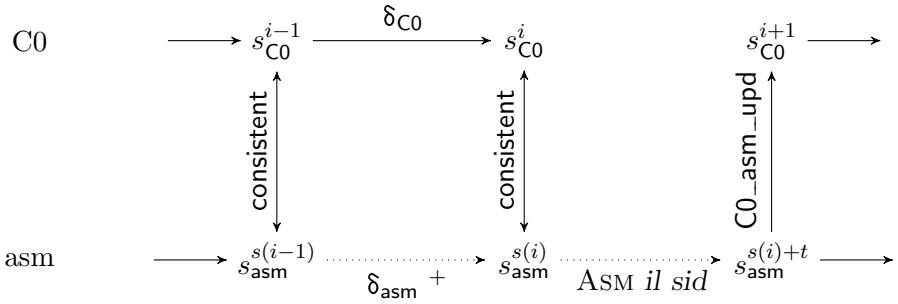

Figure 2.5: $\mathrm{C0}_A$ semantics with Inline Assembly Code

A new C0 state s'_{C0} might possibly be affected by the assembly instructions. Therefore, we define a function C0_asm_upd that computes a new C0 state with an updated memory out of an old C0 state s_{C0}, the old and new assembly states s_{asm} and s'_{asm} respectively. Furthermore, this function takes an allocation function *alloc* that relates

C0 and assembly states and a list of elementary g-variables *gl* comprising all possibly altered memory objects. We start to explain how the new C0 memory is constructed from the new assembly state s'_{asm}.

First, we show how to construct an elementary C0 memory cell from a given type t and an integer value z. The function elem_memcell_construction returns a typed C0 memory cell (e. g., memcellChar if the type is a character) when the given type is elementary and the integer value has a meaningful interpretation (e. g., value 2 is not meaningful if the type is Boolean). Otherwise, the function returns $\bot$. Formally:

elem_memcell_construction $t\ z \equiv$
  if $t =$ Boolean $\wedge\ z = 0$ **then** $\lfloor$memcellBool False$\rfloor$
  elsif $t =$ Boolean $\wedge\ z = 1$ **then** $\lfloor$memcellBool True$\rfloor$
  elsif $t =$ Integer **then** $\lfloor$memcellInt $z\rfloor$
  elsif $t =$ UnsgndT **then** $\lfloor$memcellUnsigned (to_nat32 z)$\rfloor$
  elsif $t =$ CharT $\wedge\ -2^7 \leq z \wedge z < 2^7$ **then** $\lfloor$memcellChar $z\rfloor$
  elsif $(\exists tn.\ t =$ Ptr $tn) \wedge z = 0$ **then** $\lfloor$memcellPtr NullPointer$\rfloor$ **else** $\bot$

Tsyban [Tsy09, Def. 8.3] defines a function C0_mem_asm_update_gvar that updates a C0 memory state s_{C0}.pstate.mem at the address of an elementary g-variable g. This definition is the centerpiece of the C0 state construction because here the link between the updated assembler memory and the corresponding affected C0 memory object is established. Therefore, the function takes the allocation function *alloc* together with the type table tt, a g-variable g and the modified assembly state s'_{asm}. First, the function extracts the type of the given g-variable g (recall function addrof_gvar in Section 2.3). Then, it computes the integer value that is stored in the memory of the assembly state af the allocated address of g (i. e., s'_{asm}.mm (fst (*alloc* g) div 4)). When a C0 memcell construction with these values fails, C0_mem_asm_update_gvar returns $\bot$. Otherwise, the function updates an elementary memory object denoted by the g-variable g with an initialized data slice. The value of the data slice is exactly the converted integer value that was stored in the memory of s'_{asm}. The definition of C0_mem_asm_update_gvar is given below:

C0_mem_asm_update_gvar $tt\ m\ s'_{\mathsf{asm}}$ *alloc* $g \equiv$
  let $tn =$ (addrof_gvar tt (extract_symbolconf m) g).ad_tn
  in let$_\bot$ $f =$ elem_memcell_construction tn (s'_{asm}.mm (fst (*alloc* g) div 4))
    in mem_update $tt\ m\ g$
      (|ds_lval $= g$, ds_type $= tn$, intermediate = False, ds_initialized = True,
       ds_data $= \lambda i.$ **if** $i = 0$ **then** f **else** default|)

Finally, function C0_mem_asm_upd is inductively defined over a list *gl* of all modified elementary g-variables. If this list is empty, the function simply returns an unchanged memory. Otherwise, it returns a new C0 memory state by updating all g-variables of the list *gl* sequentially with function C0_mem_asm_update_gvar.

For a successful C0-state construction, however, a successful memory construction does not suffice. Moreover, several restrictions have to be fulfilled to guarantee that execution

of inlined assembly instructions does not destroy the C0 state. We briefly present these requirements below:

- The new assembly machine s'_{asm} has finished the execution of the instruction list i. e.,
  $\mathsf{pcs\_exec\_value\_correct}\ s_{\mathsf{C0}}\ s_{\mathsf{asm}}\ s'_{\mathsf{asm}} =$
  $(s'_{\mathsf{asm}}.\mathsf{dpc} = s_{\mathsf{asm}}.\mathsf{dpc} + 4 \cdot |il| \wedge s'_{\mathsf{asm}}.\mathsf{pcp} = s'_{\mathsf{asm}}.\mathsf{dpc} + 4)$
- The code region of the compiled C0 program has not been modified:
  $\mathsf{code\_region\_const}\ tt\ pt\ s_{\mathsf{C0}}\ s_{\mathsf{asm}}\ s'_{\mathsf{asm}} =$
  $(\forall ad.\ \mathsf{program\_base} \leq ad \wedge$
  $ad < \mathsf{program\_base} + |\mathsf{codegen\_program}\ tt\ pt\ (\mathsf{gm\_st}\ s_{\mathsf{C0}}.\mathsf{mem})| \longrightarrow$
  $s'_{\mathsf{asm}}.\mathsf{mm}\ ad = s_{\mathsf{asm}}.\mathsf{mm}\ ad)$
- The registers pointing to the stack start, the topmost stack frame and the next heap element to allocate remain unchanged:
  $\mathsf{reg\_pointers\_const}\ s_{\mathsf{asm}}\ s'_{\mathsf{asm}} \equiv$
  $\mathsf{sbase}\ s'_{\mathsf{asm}} = \mathsf{sbase}\ s_{\mathsf{asm}} \wedge$
  $\mathsf{toplm\_base}\ s'_{\mathsf{asm}} = \mathsf{toplm\_base}\ s_{\mathsf{asm}} \wedge \mathsf{toph}\ s'_{\mathsf{asm}} = \mathsf{toph}\ s_{\mathsf{asm}}$
- All g-variables of list gl that have to be updated have to fulfill several requirements:
  - they are either global, heap or local variables of the top most stack frame:
    $\mathsf{given\_gvars\_not\_not\_top\_local}\ s_{\mathsf{C0}}.\mathsf{mem}\ gl \equiv$
    $\forall x \in \mathsf{set}\ gl.$
    $\mathsf{is\_global\_gvar}\ x \vee \mathsf{is\_top\_local\_gvar}\ s_{\mathsf{C0}}.\mathsf{mem}\ x \vee \mathsf{is\_heap\_gvar}\ x$
  - they are either root g-variables or initialized:
    $\mathsf{given\_gvars\_initialized\_or\_root}\ s_{\mathsf{C0}}.\mathsf{mem}\ gl \equiv$
    $\forall x \in \mathsf{set}\ gl.\ \mathsf{gvar\_initialized}\ s_{\mathsf{C0}}.\mathsf{mem}\ x \vee \mathsf{is\_root\_gvar}\ x$
  - the type of the referenced data is elementary:
    $\mathsf{given\_gvars\_elementary}\ tt\ sc\ gl \equiv \forall x \in \mathsf{set}\ gl.\ \mathsf{elementary\_gvar}\ tt\ sc\ x$
  - all g-variables are reachable:
    $\mathsf{given\_gvars\_reachable}\ tt\ s_{\mathsf{C0}}.\mathsf{mem}\ gl \equiv$
    $\forall x \in \mathsf{set}\ gl.\ x \in \mathsf{reachable\_gvars}\ tt\ s_{\mathsf{C0}}.\mathsf{mem}$
- Only variables given in gl are manipulated after the execution of the assembly code. In other words the stack and heap memory remains unchanged if the allocated address does not belong to a variable in list gl:
  $\mathsf{only\_given\_variables\_changed}\ tt\ pt\ s_{\mathsf{C0}}.\mathsf{mem}\ s_{\mathsf{asm}}\ s'_{\mathsf{asm}}\ alloc\ gl \equiv$
  $\forall ad.\ \mathsf{compute\_sbase}\ tt\ pt\ (\mathsf{gm\_st}\ s_{\mathsf{C0}}.\mathsf{mem}) \leq ad \wedge$
  $ad < \mathsf{heap\_base} + \mathsf{asize\_heap}\ s_{\mathsf{C0}}.\mathsf{mem.hm.st} \wedge$
  $ad\ \mathsf{div}\ 4 \notin \mathsf{set}\ (\mathsf{map}\ (\lambda x.\ \mathsf{fst}\ (alloc\ x))\ \mathsf{div}\ 4)\ gl) \longrightarrow$
  $s'_{\mathsf{asm}}.\mathsf{mm}\ (ad\ \mathsf{div}\ 4) = s_{\mathsf{asm}}.\mathsf{mm}\ (ad\ \mathsf{div}\ 4)$

Note that all these constraints have to hold in the final assembly state s'_{asm}, during the assembly execution they might not be fulfilled. We collect all requirements into predicate $\mathsf{preconds\_C0\_asm\_upd}$.

Finally, we present the C0-state-constuction function $\mathsf{C0\_asm\_upd}$ that we improved for convenience and adapted to our purposes. It takes the old C0 state s_{C0}, the assembly state before and after the execution of the inline assembly portion (i. e., s_{asm} and s'_{asm}), an allocation function *alloc* and a list of changed elementary g-variables gl. If predicate

preconds_C0_asm_upd does not hold or the construction of the C0 memory does not succeed, the construction of a corresponding C0 state fails and the function returns an empty state $\bot$. Otherwise, the memory is updated successfully and the first statement from the program rest is removed with function remove_fst_stmt. Note that the new states in both language layers are consistent by construction.

Definition 2.4 (Construction of a Consistent C0 State)**.** We construct a new C0 state from an old C0 state s_{C0}, an old and a new assembly state s_{asm} and s'_{asm}, an allocation function *alloc* and a list of changed g-variables *gl* by applying function C0_asm_upd. Formally:

C0_asm_upd s_{C0} s_{asm} s'_{asm} *alloc* *gl* ≡
  let$_\bot$ m' = C0_mem_asm_upd s_{C0}.ttab s_{C0}.pstate.mem s'_{asm} *alloc* *gl*
  in if preconds_C0_asm_upd s_{C0} s_{asm} s'_{asm} *alloc* *gl*
    then $\lfloor s_{C0}(\!|$pstate := $(\!|$mem = m', prog = remove_fst_stmt s_{C0}.pstate.prog$|\!)|\!)\rfloor$
    else $\bot$

The definition from above slightly differs from the original one proposed in [Tsy09, Def. 8.4]. In Section 6.1 we use this function for process states which are modelled as monolithic C0 states. Therefore, we adapted the construction function C0_asm_upd to monolithic C0 states. Second, the definitions differ in the manipulation of the program rest. The former definition substituted the first statement with a Skip which had to be removed by a following C0 small-step transition. This transition always succeeds and its successor state is still consistent to the new assembly state s'_{asm}. Hence, we simplified the treatment of the program rest by directly removing the first statement.

2.6 Interrupts

This section reports briefly about interrupts and their definition in Isabelle/HOL. Müller & Paul give a more detailed description about interrupts in [MP00, Chapter 5].

Program computations may be disturbed by incoming event signals called *interrupts*. The activation of an interrupt should result in a procedure called *exception handler* that takes care of the problem signalled by the occuring interrupt.

Interrupts may be classified according to the following criteria:

- internal or external interrupts, depending on the interrupt source
- maskability
- the type (abort, repeat or continue)

We distinguish two sources of interrupts: The currently executed program might *internally* generate an interrupt to signal some error conditions. These *exceptions* occuring at runtime are i. e., illegal instruction, misalignment or overflow. Interrupts can also be used to switch control from a user program to the kernel. Therefore, the assembly language supports a special instruction called *trap* which serves as a system call for user applications to request a service of the operating system kernel.

External interrupts are generated by hardware peripherals to trigger the activation of the corresponding device driver. In the automotive subproject, *external* interrupts are

j	name	description	maskable	external	type
0	*reset*	reset	no	yes	abort
1	*ill*	illegal instruction	no	no	abort
2	*mal*	misalignment	no	no	abort
3	*pff*	page fault on fetch	no	no	repeat
4	*pfls*	page fault on load/ store	no	no	repeat
5	*trap*	trap instruction	no	no	continue
6	*ovf*	integer overflow	yes	no	continue
13..31	*eev*[j]	external interrupts	yes	yes	continue

Table 2.3: Interrupts

i. e., a reset signal or timer events.

Interrupts are called maskable when they can be ignored under software control. Therefore, a special mask register stores a bitvector that decides whether an interrupt will be ignored or not.

The interrupt type specifies how to proceed after an interrupt has occured. Either the program execution is aborted, the same instruction is repeated when the program execution is resumed or the next instruction is executed after the interrupt handling.

The VAMP architecture provides interrupts numbered by indices $0 \leq j \leq 31$ which are depicted in Table 2.3. The indices also specify *priorities* of the interrupts. Small indices correspond to higher priorities which means that in case of simultaneously active interrupts the one with the higher priority is handled whereas all the others are ignored.

The information of occurred interrupts are stored in two registers: the *exception cause register* and the *exception data register*. The former one stores a bitvector which indicates the occurred interrupts, the latter one contains additional data which is written from some interrupts i. e., traps. The kernel gets this information by two variables encoding the register content to unsigend 32-bit integer.

Further on, we consider interrupts on the CVM level and introduce all necessary preliminaries. For a detailed explanation of CVM state components or device states we refer to Section 2.8, Section 2.7.

We assume that the actual user process running on the processor is given as an assembly program s_{asm}. Then, the following predicates hold when an *internal* interrupt occurs during the program execution: there might be illegal instructions $\mathsf{is\_illegal\_instr}\ s_{\mathsf{asm}}$, misalignments $\mathsf{is\_misaligned\_pc}\ s_{\mathsf{asm}} \vee \mathsf{is\_misaligned\_data}\ s_{\mathsf{asm}}$, page faults on fetch $\mathsf{is\_outranged\_pc}\ s_{\mathsf{asm}}$ or load/store $\mathsf{is\_outranged\_data}\ s_{\mathsf{asm}}$, traps $\mathsf{is\_trap}\ s_{\mathsf{asm}}$ and overflows $\mathsf{is\_overflow}\ s_{\mathsf{asm}}$.

We subsume all internal events that are generated from a user process s_{asm} into a bitvector, where the i-th bit is set when the corresponding interrupt occurred (see Table 2.3). The conversion from a predicate into a bit value is realized with function bool2bit. The bitvector storing *internal* interrupts, is formally defined as:

$\mathsf{iev}\ s_{\mathsf{asm}} \equiv$

[bool2bit (is_overflow s_{asm}), bool2bit (is_trap s_{asm}),
bool2bit (is_outranged_data s_{asm}), bool2bit (is_outranged_pc s_{asm}),
bool2bit (is_misaligned_pc $s_{\mathsf{asm}} \vee$ is_misaligned_data s_{asm}),
bool2bit (is_illegal_instr s_{asm})]

Furthermore, there are several devices that might generate interrupts. All devices are summarized into state s_{dev} which stores device identifiers together with their state. When the device with identifier j generates an interrupt, the j-th bit of the vector will be set. The function computing the bitvector for all *external* interrupts is given below:

eev s_{dev}

We mentioned before that interrupts can be ignored under software control. Therefore, a special purpose register called *status register* stores an interrupt mask. Each bit of this mask defines whether the corresponding event signal is disabled **0** or enabled **1**. Note that the bit values for unmaskable interrupts (i. e., illegal instruction, misalignment, page faults and the trap instruction) are always **1**. A masked bitvector is then defined by the bitwise conjunction of the mask and the interrupt vector. This operation with a given mask *mask* is encapsulated in function mca_iev s_{asm} *mask* for internal and in function mca_eev s_{dev} *mask* for external events.

For the computation of an overall masked cause bitvector, we neglected the circumstance that not only user processes but also the kernel can be executed on the processor. Therefore, this function uses an optional state s_{asm} which distinguishes whether a user process is currently running ($\lfloor proc \rfloor$) or the kernel computes ($\perp$). Then, the masked cause bitvector of all interrupts is a concatenation of several subvectors. We start with the external masked bitvector mca_eev which can be accessed by indices greater then 13. All interrupts with indices $7 \leq j \leq 12$ are masked out. The next six values of the bitvector encode the internal interrupts. When the kernel computes no internal interrupt can occur, therefore we simply append a list of six **0**. Otherwise, a user process executes and we use the masked bitvector of internal events. Finally, we do not expect a reset signal during normal execution, hence the rightmost value of the vector is disabled **0**.

We get the entire masked cause bitvector by applying function mca_bv on an optional current process state s_{asm}, the entire device state s_{dev} and the mask *mask*. Formally:

mca_bv s_{asm} s_{dev} *mask* $\equiv$
mca_eev s_{dev} *mask* $\circledcirc$
replicate 6 **0** $\circledcirc$
(**case** s_{asm} **of** $\perp \Rightarrow$ replicate 6 **0** | $\lfloor proc \rfloor \Rightarrow$ mca_iev *proc* *mask*) $\circledcirc$ [**0**]

When the kernel is entered, it gets information of the *exception cause register* and the *exception data register* in form of unsigend 32-bit integer values. We finally present both conversion functions.

The bitvector stored in the *exception cause register* is converted to an unsigend 32-bit integer by using function bv_to_nat. We encapsulate this conversion into function mca_nat:

mca_nat s_{asm} s_{dev} *mask* $\equiv$ bv_to_nat (mca_bv s_{asm} s_{dev} *mask*)

The second function edata_nat computes the content of an *exception data register* which is written additionally from some interrupts i. e., traps. It is only used, when the currently executed user process s_{asm} causes an internal interrupt. In case of a misalignment or page fault on fetch, the function returns the delayed pc of the user process s_{asm}. The predicate is_load_store distinguishes load and store instructions from other VAMP operations. When the current instruction is a legal load or store operation, the function returns the effective address. When the user process calls the kernel with the trap instruction, the kernel gets the immediate constant of the call i. e., the trap number (current_instr s_{asm} = TRAP $n \Longrightarrow$ imm_arg (current_instr s_{asm}) = n). Otherwise, function edata_nat returns value 0. Formally:

edata_nat $s_{\mathsf{asm}} \equiv$
if is_misaligned_pc $s_{\mathsf{asm}} \vee$ is_outranged_pc s_{asm} **then** s_{asm}.dpc
elsif $\neg$ is_illegal_instr $s_{\mathsf{asm}} \wedge$ is_load_store (current_instr s_{asm}) **then** ea s_{asm}
elsif is_trap s_{asm} **then** to_nat32 (imm_arg (current_instr s_{asm})) **else** 0

2.7 Devices

The Verisoft project aims at the pervasive formal verification of entire computer systems employing input/output devices for storage, communication and user interaction. At the hardware level, these devices are integrated as memory-mapped devices that may generate interrupts but do not use direct-memory access. The device drivers which control the devices are included in the operating system or in a user application. There is a device framework which provides a common device model in order to simplify the integration for individual devices into the model stack independent from the specification layer [AH08]. In this section, we briefly introduce the framework for device models and present devices that are used in the automotive subproject.

2.7.1 Device Automata

In the Verisoft project, a number of different devices are considered, such as: a hard disk hd, a network interface card nic, a serial interface uart, a simple timer timer and a bus interface called *automotive bus controller* abc. We denote any of these devices with x $\in$ {hd, abc, nic, uart, timer} and use a predicate is_conf_x that holds when a device state is of type x.

A device may interact with a processor on the one hand and an external environment on the other hand, as depicted in Figure 2.6 on the next page. The device model is pseudo-parallel in that sense, that it performs either an internal transition with a processor input or an external transition with an input from the external environment exclusively.

More precisely, an internal memory interface supports the interaction with the processor. The processor may request a read or write access to the device with an input *mifi*. The device performs an internal transition δ_{x}^{m} on the current device state s_{x} and returns a successor device state s'_{x} together with additional outputs to the processor and to the

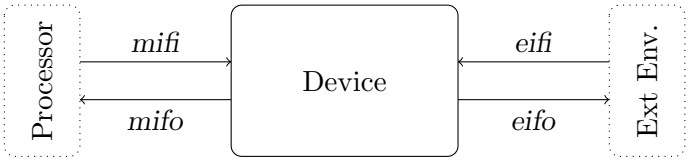

Figure 2.6: Devices

external environment.

A memory interface input *mifi* consists of four components:

1. a read flag mifi_rd
2. a write flag mifi_wr
3. a port address mifi_a of type portT
4. a word-size data input mifi_din

The read and write flag are assumed to be mutually exclusive, i. e., $\neg$ (*mifi*.mifi_rd $\wedge$ *mifi*.mifi_wr). A memory interface output *mifo* is a word-sized data output that is returned when the processor has a read access on the device. We define the idle memory interface input as $\mathsf{mifi}_\perp$. and the idle memory interface output $\mathsf{mifo}_\perp$, respectively. Additionally, the processor can access the device system to read out an interrupt vector of currently active interrupts.

Each device type internally specifies its own internal transition function. The internal transition function of a device of type x is given below:

$$\delta^m_{\mathsf{x}}\ \mathit{mifi}\ s_{\mathsf{x}} = (s'_{\mathsf{x}},\ \mathit{mifo},\ \mathit{eifo})$$

A device function that performs multiple steps on a device state of type x over a list of given processor inputs is defined recursively:

$$\begin{array}{l}\delta^{m*}_{\mathsf{x}}\ (\mathit{mifi} \odot \mathit{mifis})\ s_{\mathsf{x}} = \\ (\textbf{let}\ (s',\ \mathit{mifo},\ \mathit{eifo}) = \delta^m_{\mathsf{x}}\ \mathit{mifi}\ s_{\mathsf{x}};\\ \quad (s'',\ \mathit{mifos},\ \mathit{eifos}) = \delta^{m*}_{\mathsf{x}}\ \mathit{mifis}\ s'\\ \textbf{in}\ (s'',\ \mathit{mifo} \odot \mathit{mifos},\ \mathit{eifo} \odot \mathit{eifos}))\end{array}$$

The communication with the external environment, however, takes the current device state s_{x} and an external input *eifi*. The external device transition function returns the successor state s'_{x} and an output *eifo* to the external environment. The memory interface is shared by all device types in contrast to the type specific in- and outputs of the external interface (eifi,eifo). We denote an idle external interface input with $\mathsf{eifi\_x}_\perp$ and an idle external interface output with $\mathsf{eifo\_x}_\perp$. The external device transition is specific for each device type x:

$$\delta^e_{\mathsf{x}}\ \mathit{eifi}\ s_{\mathsf{x}} = (s'_{\mathsf{x}},\ \mathit{eifo})$$

In the following, we use $\delta^{m*}_{\mathsf{devs}}$ and δ^e_{devs} when we speak in general about devices. These functions operate as mediators over lists of inputs and call, depending on the state type, the corresponding underlying internal and external transition functions.

From the view of the operating system devices can be accessed and manipulated by communicating via its ports, whereas the internal data structure of the device remains hidden. The maximal number of word-sized ports is bounded by PORT_COUNT $\equiv 2^{10}$. All meaningful ports addresses are defined by the datatype portT $\equiv$ {0..<PORT_COUNT} and fulfill the validity predicate valid_port.

The upper bound of usable devices is limited by DEV_COUNT $\equiv$ 8. In order to distinguish between them, we use device identifiers within the datatype devnumT $\equiv$ {1..DEV_COUNT}. All meaningful device identifiers fulfill predicate valid_devid. In the implementation, the smallest device identifier starts from DEVICE_FIRST $\equiv$ 13. Hence, the interrupt of a device with identifier n can be derived from the n-th element of the entire bitvector. The device identifiers of the implementation are converted to abstract numbers with nat2dev. This function subtracts DEVICE_FIRST $-$ 1 in order to obtain a meaningful abstract device identifier in the correct range.

The upper-layer computational models integrate several of different devices and should be able to interact with a number of devices with varying types. Therefore, we use a device vector s_{dev} that keeps device numbers together with their state device states. Recall that we used this state already in section Section 2.6. Furthermore, the external state component used for XCalls (recall Section 2.1) includes this vector to embed a number of devices. From a given device identifier and a list of external interface outputs function create_eifo_list computes a list of pairs which tags a list of external interface outputs with the given device identifier.

In the automotive subproject, we consider two different devices: The first one is a hard disk with a boot region. It is only used during initialization to load an OS image from the boot region to the memory of an application. After this phase, we neglect the hard disk in all further considerations. Hence, we only present parts of the hard disk model that are necessary in the course of this work. A more detailed description of the entire specified hardware model can be found in Alkassar & Hillebrand [AH08].

The device number of the hard disk in the implementation is DEVICE_HD1 $\equiv$ 14 and after conversion it is denoted with SWAP_DID. The hard disk state consists of the disk size in sectors $\mathsf{s_{hd}}$.S, the disk content $\mathsf{s_{hd}}$.sm, a sector buffer $\mathsf{s_{hd}}$.buf, a sector buffer pointer $\mathsf{s_{hd}}$.bp, and further components that are not important in the course of this work. A predicate is_validconf_hd encapsulates all validity constraints for the hard disk i. e., the range of the disk size, the lengths of the disk content and the sector buffer and an upper bound for the buffer pointer. The disk content is called valid when all values are 32-bit-naturals:

$$\mathsf{is\_valid\_disk\_content}\ s_{\mathsf{hd}} \equiv \forall i{<}|s_{\mathsf{hd}}.\mathsf{hd\_sm}|.\ \mathsf{asm\_nat}\ (s_{\mathsf{hd}}.\mathsf{hd\_sm}\ !\ i)$$

The second device used in our distributed system is of more importance. The automotive bus controller *ABC* is responsible for the communication between different system components. In the implementation, its identifier is given as a natural number ABC_ID $\equiv$ 13. The abstract device identifier is defined with ABC_DID (certainly, nat2dev ABC_ID = ABC_DID holds). We present a more detailed description of the ABC-device model in Section 4.1.

2.8 CVM: A Programming Framework for Operating Systems

Our operating system (OS) is implemented on the verified RISC processor VAMP [BJK+06] using a programming framework called *communicating virtual machines* (CVM) [IT08]. This framework encapsulates the necessary hardware-specific low-level functionality for operating systems built on the VAMP. It provides basic mechanisms for address translation and processor virtualization as well as the communication with memory-mapped devices. Technically, CVM constitutes the central interrupt-service routine of the OS, which is executed whenever an interrupt occurs in the system. The interrupt-service routine saves the hardware-specific context. Then, it handles possibly occurred page faults and otherwise, passes control on to the higher layers of the OS. The higher software layers may control the low-level mechanisms by so-called *primitives*. This software architecture permits the implementation of an operating system almost independently from hardware in C0 without assembly.

CVM constitutes a computational model where several user machines are interacting with a kernel. It is parametrized with a so-called *abstract kernel* which allows to compute independently from a particular kernel representation. The Verisoft project uses two different abstract kernels: a general purpose microkernel VAMOS and our real-time operating system OLOS. Abstract kernels can be linked to the CVM framework on source code level in order to obtain a *concrete kernel*.

In the next subsections, we present the CVM state and transition and itemize CVM primitives that are used in OLOS.

2.8.1 CVM State

In this subsection, we briefly summarize the formal definition of CVM states.

States of the CVM model s_{cvm} comprise the following components:

- an abstract kernel state cvm_kernel,
- virtual user processes cvm_up, and
- a device component cvm_dev

The abstract kernel is modelled in CVM as an abstract monolithic C0 state with typed memory as we introduced in Section 2.3. The function table of the C0 machine requires entries of all used CVM primitives and a top-level function called `kdispatch` must be defined. Moreover, this function handles all interrupts which are passed on from CVM. In CVM, user processes are modelled as VAMP assembly machines (recall Section 2.2) with virtual memory. The number of processes running on CVM is fixed by the constant PID_MAX ≡ 128. Identifiers of the user processes are defined by the datatype procnumT ≡ {1..<PID_MAX}. Natural numbers are converted to process identifiers with function nat2pid and pid2nat converts vice versa. Accordingly, cvm_up has a component userprocesses providing a mapping between process identifiers *pid* and VAMP assembly states. Moreover, a component s_{cvm}.cvm_up.currentp distinguishes between kernel computation $\perp$ and $\lfloor pid \rfloor$ denoting the execution of the user process *pid*. In the implementation, however, process identifier 0 is reserved for kernel computation. Furthermore, cvm_up has a component statusreg that holds the interrupt mask in terms of a 32-bit natural

number. The mask determines which interrupts are enabled. In the following, we write s_{ups} when we only consider the state of component cvm_up.

Finally, the state has a device component that stores a device vector as described in Section 2.7.

After power-up init_cvmup defines the initial state of component cvm_up:

$$
\begin{aligned}
&\mathsf{init\_cvmup} \equiv \\
&\quad (\!|\mathsf{userprocesses} = \\
&\qquad \lambda \mathit{pid}.\ \textbf{if}\ \mathsf{pid2nat}\ \mathit{pid} = 0\ \textbf{then}\ \mathsf{default}\ \textbf{else}\ \mathsf{init\_process}\ (\mathsf{pid2nat}\ \mathit{pid}), \\
&\qquad \mathsf{currentp} = \bot,\ \mathsf{statusreg} = 8254 |\!)
\end{aligned}
$$

The user processes are initialized with function init_process, where the program counters, the memory, the general purpose registers and almost all special purpose registers are set to 0. Exceptional cases are the mode register, the status register, and the registers holding the page table origin and the page table length. Tsyban [Tsy09, Definition 5.5] describes the initialization of user processes in more detail. Component currentp is set to $\bot$ and statusreg stores the interrupt mask given with 8254. This value encodes the bitvector $[\mathbf{1}, \mathbf{0}, \mathbf{0}, \mathbf{0}, \mathbf{0}, \mathbf{0}, \mathbf{0}, \mathbf{0}, \mathbf{1}, \mathbf{1}, \mathbf{1}, \mathbf{1}, \mathbf{1}, \mathbf{0}]$ which masks out all interrupts except for the illegal, misalignment, page faults, trap, and external interrupt of the ABC device.

Valid User Processes. In this thesis, we rely on the fact that initial virtual machines fulfill validity requirements that are preserved after each transition step. Formally, predicate is_valid_cvmup encapsulates these conditions:

$$
\begin{aligned}
&\mathsf{is\_valid\_cvmup}\ s_{\mathsf{cvm}} \equiv \\
&\quad \forall i.\ \mathsf{is\_valid\_asm}\ (s_{\mathsf{cvm}}.\mathsf{userprocesses}\ i) \wedge \\
&\qquad -1 \leq (s_{\mathsf{cvm}}.\mathsf{userprocesses}\ i).\mathsf{sprs}\ !\ \mathsf{PTL} < \mathsf{to\_int32}\ \mathsf{TVM\_MAXPAGES} \wedge \\
&\qquad (s_{\mathsf{cvm}}.\mathsf{userprocesses}\ i).\mathsf{sprs}\ !\ \mathsf{MODE} \neq 0
\end{aligned}
$$

Each virtual machine is represented as valid assembler machine satisfying is_valid_asm and running in user mode. Additionally, each virtual machine has either no allocated memory i. e., its number of pages is -1 or the number of pages is less than a fixed upper bound TVM_MAXPAGES.

2.8.2 CVM Transitions

A CVM transition δ_{cvm} takes a CVM state s_{cvm} and an external device input *eifi* as parameters, computing the successor state s'_{cvm} or an empty state $\bot$ in case of an occurring runtime error. Additionally, it might return an optional output to the external environment.

Depending on the input parameters, we distinguish between three different possibilities:

1. $\mathit{eifi} \neq \bot$: the device performs a step
2. $\mathit{eifi} = \bot \wedge s_{\mathsf{cvm}}.\mathsf{cvm\_up.currentp} = \bot$: the kernel is computing
3. Otherwise, the current process is executed

The CVM model is semi-parallel since either devices, the kernel or one of the user processes make progress in their execution. When the external device input *eifi* is not empty the device performs an external step $\delta^{e}_{\mathsf{devs}}$ that only modifies the specified device state and generates an external output. If there is no external input (i. e., $eifi = \bot$) and the current process identifier is set to $\bot$, the kernel is computing. Summarized there are three cases, when the kernel performs a step. In the situation that there is no user process in the system to be resumed, the kernel simply waits for incoming interrupts to be restarted. The second case handles a switch from kernel execution to start the computation of the next scheduled user process or to wait for incoming interrupts (defined in the first case). Finally, the kernel step models transitions of the abstract kernel that is either a C0 step or an execution of a CVM primitive (primitives will be explained in more detail in Section 2.8.3). When a current process performs a step there might be three possibilities: User steps without an interrupt are simply executed by applying the VAMP assembly transition function δ_{asm}. When an interrupt occurs during a user process step, CVM invokes the abstract kernel in order to handle this problem. The way of proceeding depends on the kind of interrupt. In case of a runtime error i. e., the user execution is aborted, the virtual machine of the current process remains unchanged. In case of external interrupts, trap instructions or overflows, the current process is allowed to take a step according to δ_{asm}. As long as no runtime error occurs i. e., user_step_progress holds, the user process progresses in its execution. The updated currently scheduled user process after a user step ist determined by function userprocesses_step:

userprocesses_step s_{ups}.userprocesses $pid \equiv$
  if user_step_progress (s_{ups}.userprocesses pid)
    then s_{ups}.userprocesses($pid := \delta_{\mathsf{asm}}$ (s_{ups}.userprocesses pid))
  else s_{ups}.userprocesses

2.8.3 CVM Primitives

Besides ordinary C0 functions, the abstract kernel requires mechanisms to access and manipulate data structures that are not visible to the C0 variables of the kernel i. e., registers or the memory of user processes. Therefore, CVM provides special functions, so-called CVM *primitives* which can access and manipulate states of user processes and devices. Primitives are functions with portions of inline-assembly code realizing basic operations that constitute functionality of the kernel.

These operations cover e. g.,

- enabling and disabling interrupts
- exchange data between processes itself or processes and I/O-devices
- allocation and deallocation of memory

Figure 2.7 on the following page shows all CVM primitives used in the OLOS implementation. The kernel uses primitives above the double line to interact with a user process whereas the functions below access or manipulate a device. The kernel uses several primitives (denoted with *) to exchange messages. CVM provides several primitives that permit copy operations of arbitrarily typed C0 values between the kernel and user

processes or devices. C0 does not support a generic programming concept like C++ templates. Hence, we use C preprocessing macros [Ame99] to specify generic patterns for individually typed copy functions. In particular, the actual C0 program code and the implementation of the copy primitives 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 the generic primitive functionality. The C preprocessor then expands the macros from the pattern to a particular function implementation. In the OLOS implementation messages are of type `t_kmsg` thus, we define macros to use typed copy primitives [6].

The following macro e. g., expands to the definitions of the functions `cvm_physIOInRange_t_kmsg` and `cvm_physIOOutRange_t_kmsg` providing I/O operations on port ranges for kernel variables:

```
DEFINE_CVM_PHYS_IO_RANGE(t_kmsg)
```

The preprocessor expands the macros to typed functions that copy data between the kernel and a specified device. The function parameters are the device identifier `dev_id`, the port address `port` and a message given by a dereferenced pointer `*ptr`. The signature of function `cvm_physIOInRange_t_kmsg` e. g., is given by:

```
int cvm_physIOInRange_t_kmsg
    (unsigned int dev_id, unsigned int port,t_kmsg *ptr)
```

We refer the interested reader to the preprocessed C0 code of typed CVM primitives in A.1.7.

CVM primitive	description
`cvm_reset`	initializes process
`cvm_alloc`	allocates additional memory
`cvm_load_os`	loads an process image from the boot region
`cvm_get_gpr`	reads gpr of a process
`cvm_set_gpr`	writes gpr of a process
`cvm_v2pcopy`*	copies from virtual to physical memory
`cvm_p2vcopy`*	copies from physical to virtual memory
`cvm_physIOInRange`*	executes I/O operations from port ranges to OS variables
`cvm_physIOOutRange`*	executes I/O operations from OS variables to port ranges
`cvm_out_word`	writes data to port of device

Figure 2.7: CVM primitves

The specification of the CVM primitives in C0 small-step semantics is hierarchically structured. Functions on lower layers specify the effects of primitives on subcomponents

[6] Note that OLOS maintains the compiled code of C0 applications. Thus, the C0 message type of the applications differs from the message type of the kernel (see also Section 3.4.1).

of states in higher layers. Hence, the primitive functions on upper layers are defined by reusing the results of their corresponding function on a lower layer. Thus, this framework provides suitable functions describing primitive effects on states at different levels of abstractions.

Figure 2.8 gives an overview on the components that are affected by the CVM primitives. The operating system uses primitives listed above the double line to interact with a single virtual machine or primitives to interact with a device presented below the double line. The return values of functions on the lowest layer are depicted in the third column of this tabular. Functions that access a virtual machine either return the read data (d or dl) or the manipulated user process state. Primitives communicating with a device internally call device transition functions and return an updated device vector together with outputs to the memory ml and external interface el. However, we often regard the entire user-process component s_{ups} or return a special output type from the device. Functions exec_<primitivename> on the second layer embed the functions of the lowest layer and operate either on the user-process component or on the entire device vector. The fourth column presents the return values of these functions. Functions that interact with a user process either pass the read value or embed the result of the lowest layer into the user-process component. CVM primitives for device communication use the result value from the underlying functions to create a tagged output list el_{t} to the external environment that is returned together with the device vector. These functions can be related to extended states in the Hoare logic (we will use these functions in Section 5.1.1 to define Simpl functions for CVM primitives). For each primitive there are several conditions to guarantee a successful computation. These requirements are summarized into precondition predicates pre_<primitivename>. The validity predicates for process and device identifiers, ports and register numbers hold when the given number has a meaningful interpretation i. e., valid_pid $pid \equiv pid \in$ procnumT.

		cvm_<pn>	exec_cvm<pn>	execprim_cvm<pn>		
	pn	result	result	kernel	ext	out
s_{asm}	Reset	s_{asm}	s_{ups}	0	s_{cvm}.cvm_up	[]
	Alloc	s_{asm}	s_{ups}	0	s_{cvm}.cvm_up	[]
	LoadOS	s_{asm}	s_{ups}	0	s_{cvm}.cvm_up	[]
	GetGPR	d	d	d	x	[]
	SetGPR	s_{asm}	s_{ups}	0	s_{cvm}.cvm_up	[]
	V2Pcopy	dl	dl	$(dl, 0)$	x	[]
	P2Vcopy	s_{asm}	s_{ups}	0	s_{cvm}.cvm_up	[]
s_{dev}	OutWord	$(s_{\mathsf{dev}}, ml, el)$	$(s_{\mathsf{dev}}, el_{\mathsf{t}})$	0	s_{cvm}.cvm_dev	el_{t}
	PhysIOOutRange	$(s_{\mathsf{dev}}, ml, el)$	$(s_{\mathsf{dev}}, el_{\mathsf{t}})$	0	s_{cvm}.cvm_dev	el_{t}
	PhysIOInRange	$(s_{\mathsf{dev}}, ml, el)$	$(s_{\mathsf{dev}}, ml, el_{\mathsf{t}})$	$(ml, 0)$	s_{cvm}.cvm_dev	el_{t}

Figure 2.8: Affected components of CVM primitives

Finally, functions execprim_<primitivename> on the uppest layer specify the function-

ality of CVM primitive execution on CVM states. These definitions are directly used by the CVM transition function δ_{cvm} (cf. Section 2.8.2). These functions are partial i. e., they return $\perp$ if the corresponding preconditions of the called primitive do not hold. Otherwise, they return an updated successor state s'_{cvm} together with an output to the external environment. Effects of functions on the uppest layer are depicted in the last three columns of Figure 2.8 on the preceding page. Column *kernel* presents the values that are stored in the kernel. Reading primitives store the read data together with the result variable of the called primitive. A successful executed primitive always returns 0 to the kernel, the only exception is function execprim_cvmGetGPR that updates the return variable of the primitive with the read data d. Column *ext* shows which external components have to be updated in the new CVM state: This is either the user-process component, the device component or none of them. These updates take over results of the corresponding exec_<primitivename> functions. Finally, the last column depicts the output to the external environment. Primitives interacting with a user process return an empty output list whereas device communicating primitives return a list of tagged outputs el_{t}.

According to Figure 2.8 on the previous page we classify CVM primitives into functions that only read data, manipulate the external state or both. The only primitive that returns read data to the kernel and additionally may change the external state is execprim_cvmPhysIOInRange. This is the case because the transition functions are individual for each device type. So there could be read accesses that internally change the device state. The ABC state, however, remains unchanged when its buffers are read.

In the following paragraphs, we briefly introduce all precondition predicates and the effects of CVM primitives on the lowest layer that are used in OLOS. On the next layer, we only present one exec_<primitivename> function for the different cases exemplarily since the remaining functions resemble the examples. In the course of this thesis, we use definitions of the exec_<primitivename> functions. Thus, we only present one CVM primitive on the uppest layer exemplarily and refer to a more detailed definition in [Tsy09, Chapter 5.3].

Initialization of User Processes. OLOS uses three CVM primitives in order to initialize its user processes. The user process component is updated after their execution with a new value of a given virtual machine.

The primitive cvm_Reset initializes a single user process.
The sole precondition of predicate pre_cvmReset is the constraint of a valid process identifier:

pre_cvmReset $pid \equiv$ valid_pid pid

The effects of function cvm_Reset initializing a user process s_{asm} are the following:

- The program counters are set to their initial values,
- the special purpose register PTL holding the page table length is initialized with -1 i. e., the user process occupies no memory
- the special purpose register MODE storing the mode is set to 1 i. e., to user mode

- all other registers are set initially to 0

Formally:

$$
\begin{array}{l}
\mathsf{cvm\_Reset}\ s_{\mathsf{asm}} \equiv \\
\quad s_{\mathsf{asm}}(\!|\mathsf{dpc} := 0,\ \mathsf{pcp} := 4,\ \mathsf{gprs} := \mathsf{map}\ (\lambda x.\ 0)\ s_{\mathsf{asm}}.\mathsf{gprs}, \\
\qquad \mathsf{sprs} := \mathsf{map}\ (\lambda x.\ 0)\ s_{\mathsf{asm}}.\mathsf{sprs}[\mathsf{PTL} := -1,\ \mathsf{MODE} := 1]|\!)
\end{array}
$$

Function exec_cvmReset embeds the effects of primitive cvm_Reset into the user process component. The virtual machine specified by its process identifier is updated with the primitive result. Formally:

$$
\begin{array}{l}
\mathsf{exec\_cvmReset}\ s_{\mathsf{ups}}\ \mathit{pid} \equiv \\
\quad s_{\mathsf{ups}} \\
\quad (\!|\mathsf{userprocesses} := s_{\mathsf{ups}}.\mathsf{userprocesses} \\
\qquad (\mathsf{nat2pid}\ \mathit{pid} := \mathsf{cvm\_Reset}\ (s_{\mathsf{ups}}.\mathsf{userprocesses}\ (\mathsf{nat2pid}\ \mathit{pid})))|\!)
\end{array}
$$

We present function execprim_cvmReset on the uppest layer to give a notion how the kernel component of a CVM state s_{cvm} is modified after a primitive execution. If the precondition pre_cvmReset is satisfied the user processes component is modified by exec_cvmReset. Furthermore, function set_primres (recall Section 2.3) updates the result variable of the kernel with integer value Intg 0 to signal a successful primitive execution and rm_scall removes the first statement of the program rest. The output to the external environment is an empty list []. If the precondition of the primitive is not fulfilled the function returns $\bot$. The definition of function execprim_cvmReset is given below:

$$
\begin{array}{l}
\mathsf{execprim\_cvmReset}\ s_{\mathsf{cvm}}\ \mathit{pid} = \\
\textbf{if}\ \mathsf{pre\_cvmReset}\ \mathit{pid} \\
\quad \textbf{then}\ \lfloor (s_{\mathsf{cvm}} \\
\qquad (\!|\mathsf{cvm\_kernel} := s_{\mathsf{cvm}}.\mathsf{cvm\_kernel} \\
\qquad\qquad (\!|\mathsf{pstate} := (\!|\mathsf{mem} = \lceil \mathsf{set\_primres}\ (\mathsf{Intg}\ 0)\ s_{\mathsf{cvm}}.\mathsf{cvm\_kernel} \rceil, \\
\qquad\qquad\qquad \mathsf{prog} = \mathsf{rm\_scall}\ s_{\mathsf{cvm}}.\mathsf{cvm\_kernel}.\mathsf{pstate}.\mathsf{prog}|\!)|\!), \\
\qquad \mathsf{cvm\_up} := \mathsf{exec\_cvmReset}\ s_{\mathsf{cvm}}.\mathsf{cvm\_up}\ \mathit{pid}|\!), \\
\qquad [])\rfloor \\
\textbf{else}\ \bot
\end{array}
$$

Note that the C0 small-step specifications of the other primitives are quite similar and depend on the particular CVM primitive. They vary in the modified subcomponent, the output list to the external environment and an additional manipulation of kernel variables. We do not present the other functions here, since we do not use them in the course of this thesis.

Primitive cvm_Alloc extends the virtual memory of a user process by a given number of pages. The precondition predicate pre_cvmAlloc encapsulates the constraint that the process identifier has a meaningful interpretation. Moreover, the sum of all user-process sizes (already allocated number of pages) and the number of pages *pages* that is supposed to be allocated do not exceed the maximal number of available user pages TVM_MAXPAGES. Formally:

$$\begin{aligned}
&\mathsf{pre_cvmAlloc}\ s_{\mathsf{ups}}\ pid\ pages \equiv \\
&\quad \mathsf{valid_pid}\ pid \wedge \\
&\quad (\textstyle\sum i{\in}\mathsf{procnumT}.\ \mathsf{mm_size}\ (s_{\mathsf{ups}}.\mathsf{userprocesses}\ (\mathsf{nat2pid}\ i))) + pages \\
&\quad \leq \mathsf{TVM_MAXPAGES}
\end{aligned}$$

The effects of function cvm_Alloc are described below:

- The number of pages *pages* is added to the current value of the page table length which is stored in special purpose register PTL and,
- the given number of pages *pages* are allocated and set to 0 at the end of the memory of the virtual machine.

Formally:

$$\begin{aligned}
&\mathsf{cvm_Alloc}\ s_{\mathsf{asm}}\ pages \equiv \\
&\quad s_{\mathsf{asm}} \\
&\quad (\!|\mathsf{sprs} := s_{\mathsf{asm}}.\mathsf{sprs}[\mathsf{PTL} := s_{\mathsf{asm}}.\mathsf{sprs}\ !\ \mathsf{PTL} + \mathsf{to_int32}\ pages], \\
&\qquad \mathsf{mm} := \lambda i.\ \textbf{if}\ i < \mathsf{to_nat32}\ (s_{\mathsf{asm}}.\mathsf{sprs}\ !\ \mathsf{PTL} + 1) \cdot (\mathsf{PAGE_SIZE\ div}\ 4) \vee \\
&\qquad\qquad\qquad (\mathsf{to_nat32}\ (s_{\mathsf{asm}}.\mathsf{sprs}\ !\ \mathsf{PTL} + 1) + pages) \cdot \\
&\qquad\qquad\qquad (\mathsf{PAGE_SIZE\ div}\ 4) \\
&\qquad\qquad\qquad \leq i \\
&\qquad\qquad\qquad \textbf{then}\ s_{\mathsf{asm}}.\mathsf{mm}\ i \\
&\qquad\qquad \textbf{else}\ 0 |\!)
\end{aligned}$$

Finally, Olos uses primitive cvm_LoadOS. This function loads an image of the operating system from the boot region of the hard disk to the virtual memory of a user process s_{asm}.

For a successful execution we require the validity of the process identifier and that the total amount of pages *pages* fits into the virtual memory of the specified user process. The latter condition is encapsulated in predicate address_in_mem:

$$\begin{aligned}
&\mathsf{address_in_mem}\ s_{\mathsf{ups}}\ pid\ ad \equiv \\
&\quad ad < \mathsf{to_nat32}\ ((s_{\mathsf{ups}}.\mathsf{userprocesses}\ (\mathsf{nat2pid}\ pid)).\mathsf{sprs}\ !\ \mathsf{PTL} + 1) \cdot \mathsf{PAGE_SIZE}
\end{aligned}$$

Furthermore, the device identifier *did* should correlate to the swap disk, the disk content has to be valid and the entire image should be fit into the hard disks memory. All these conditions are encapsulated in predicate pre_cvmLoadOS:

$$\begin{aligned}
&\mathsf{pre_cvmLoadOS}\ s_{\mathsf{ups}}\ s_{\mathsf{dev}}\ pages\ pid\ did\ start_page \equiv \\
&\quad \mathsf{valid_pid}\ pid \wedge \\
&\quad \mathsf{address_in_mem}\ s_{\mathsf{ups}}\ pid\ (pages \cdot \mathsf{PAGE_SIZE} - 1) \wedge \\
&\quad \mathsf{nat2dev}\ did = \mathsf{SWAP_DID} \wedge \\
&\quad \mathsf{is_conf_HD}\ (s_{\mathsf{dev}}\ \mathsf{SWAP_DID}) \wedge \\
&\quad \mathsf{is_valid_disk_content}\ (s_{\mathsf{dev}}\ \mathsf{SWAP_DID}) \wedge \\
&\quad (start_page + pages) \cdot \mathsf{PAGE_SIZE} \leq |(s_{\mathsf{dev}}\ \mathsf{SWAP_DID}).\mathsf{hd_sm}|
\end{aligned}$$

The effects of the Cvm primitive cvm_LoadOS are the following: We get the image by extracting a number of pages *pages* starting from page *start_page*. This list is copied to the memory of the virtual machine s_{asm}. Formally:

$\mathsf{cvm\_LoadOS}\ s_{\mathsf{asm}}\ s_{\mathsf{dev}}\ pages\ did\ start\_page \equiv$
  let $img = \mathsf{extr}\ (start\_page \cdot \mathsf{PAGE\_SIZE})\ (pages \cdot \mathsf{PAGE\_SIZE})$
    $(s_{\mathsf{dev}}\ (\mathsf{nat2dev}\ did)).\mathsf{hd\_sm}$
  in $s_{\mathsf{asm}}(\!|\mathsf{mm} := \lambda i.\ \textbf{if}\ |img| \leq i\ \textbf{then}\ s_{\mathsf{asm}}.\mathsf{mm}\ i\ \textbf{else}\ \mathsf{to\_int32}\ (img\ !\ i)|\!)$

cvm_Alloc and cvm_LoadOS solely modify a user process, therefore their functions on the user-process component are similar to exec_cvmReset.

Accessing User Processes. CVM provides two primitives to read or write a register of a user process. The preconditions of the predicates for register access are equal. The requirements for writing to a register (pre_cvmSetGPR) and reading from a register (pre_cvmGetGPR) include the validity of the process and the register identifier:
$\mathsf{valid\_pid}\ pid \wedge \mathsf{valid\_regnum}\ r$.

Then, writing a value *val* into a specified register r is defined as

$\mathsf{cvm\_SetGPR}\ s_{\mathsf{asm}}\ r\ val \equiv s_{\mathsf{asm}}(\!|\mathsf{gprs} := s_{\mathsf{asm}}.\mathsf{gprs}[r := \mathsf{to\_int32}\ val]|\!)$

whereas, reading from a register r is denoted with:

$\mathsf{cvm\_GetGPR}\ s_{\mathsf{asm}}\ r \equiv \mathsf{to\_nat32}\ (s_{\mathsf{asm}}.\mathsf{gprs}\ !\ r)$

CVM primitive cvm_GetGPR only accesses the virtual machine and returns a value to the operating system. Hence, function exec_cvmGetGPR returns the value after calling cvm_GetGPR with the specified user process:

$\mathsf{exec\_cvmGetGPR}\ s_{\mathsf{ups}}\ pid\ r \equiv \mathsf{cvm\_GetGPR}\ (s_{\mathsf{ups}}.\mathsf{userprocesses}\ (\mathsf{nat2pid}\ pid))\ r$

Furthermore, CVM supports two primitives to provide copy operations between kernel variables located in the physical memory and the virtual memory of user processes. The requirements for this copy primitives are equal and demand for the validity of the user process identifier. Furthermore, the data should fit into the allocated memory region of the user process. This can be ensured when the last address $(sa + (len - 1))$ of data with length *len* lies within the allocated memory area. Then, due to monotonicity all smaller addresses lie within this range and the entire data fits into the memory. The predicates for reading from a user process pre_cvmV2Pcopy and writing to a user process pre_cvmP2Vcopy encapsulate these constraints.
Formally: $\mathsf{valid\_pid}\ pid \wedge \mathsf{address\_in\_mem}\ s_{\mathsf{ups}}\ pid\ (sa + (len - 1))$

Reading data with length *len* from the virtual memory of a user process s_{asm} starting at address *sa* is defined as:

$\mathsf{cvm\_V2Pcopy}\ s_{\mathsf{asm}}\ sa\ len \equiv \mathsf{map}\ s_{\mathsf{asm}}.\mathsf{mm}\ [sa\ \mathsf{div}\ 4..{<}sa\ \mathsf{div}\ 4 + len]$

Writing a datalist *dl* from the physical to the virtual memory of a specified user process s_{asm} starting at address *sa* is denoted as:

$\mathsf{cvm\_P2Vcopy}\ s_{\mathsf{asm}}\ sa\ dl \equiv$
  s_{asm}
  $(\!|\mathsf{mm} := \lambda i.\ \textbf{if}\ i < sa\ \mathsf{div}\ 4 \vee sa\ \mathsf{div}\ 4 + |dl| \leq i\ \textbf{then}\ s_{\mathsf{asm}}.\mathsf{mm}\ i$
    $\textbf{else}\ dl\ !\ (i - sa\ \mathsf{div}\ 4)|\!)$

Function exec_cvmV2Pcopy resembles exec_cvmGetGPR since it only accesses the user process, whereas exec_cvmP2Vcopy manipulates the user process and therefore is similar to cvm_Reset.

Device Communication. For the device communication Olos uses three primitives: one function cvm_OutWord to write a single word to a device port and two typed primitives to exchange data consisting of several words: cvm_PhysIOInRange and cvm_PhysIOOutRange.

The precondition pre_cvmOutWord that ensures to write a single word successfully contains the validity of the device identifier *did* and the port address *port*.

$\mathsf{pre\_cvmOutWord}\ did\ port \equiv \mathsf{valid\_devid}\ did \wedge \mathsf{valid\_port}\ port$

Function cvm_OutWord writes a single word *data* to port *port* of the device with identifier *did*. Internally, the function executes a device transition $\delta^{m*}_{\mathsf{devs}}$ on a device state. The memory interface input signals a write request to port *port* with data *data*. Finally, we embed the updated device state into the device vector and return the outputs additionally to the memory and the external interface. Formally:

$\mathsf{cvm\_OutWord}\ s_{\mathsf{dev}}\ did\ port\ data \equiv$
  $\textbf{let}\ (st, mifo, eifo) =$
    $\delta^{m*}_{\mathsf{devs}}\ [(\!|\mathsf{mifi\_rd} = \mathsf{False}, \mathsf{mifi\_wr} = \mathsf{True}, \mathsf{mifi\_a} = port, \mathsf{mifi\_din} = data|\!)]$
    $(s_{\mathsf{dev}}\ (\mathsf{nat2dev}\ did))$
  $\textbf{in}\ (s_{\mathsf{dev}}(\mathsf{nat2dev}\ did := st), mifo, eifo)$

The embedding function exec_cvmOutWord returns the device state after calling primitive cvm_OutWord on the one hand and a created list of external outputs on the other hand.

$\mathsf{exec\_cvmOutWord}\ s_{\mathsf{dev}}\ did\ port\ data \equiv$
  $\textbf{let}\ (devs', devout, eifos) = \mathsf{cvm\_OutWord}\ s_{\mathsf{dev}}\ did\ port\ data$
  $\textbf{in}\ (devs', \mathsf{create\_eifo\_list}\ (\mathsf{nat2dev}\ did)\ eifos)$

Primitives exchanging larger data with a device cvm_PhysIOInRange and cvm_PhysIOOutRange encapsulate word-wise reading and writing starting from port *port*. Besides the validity of the device identifier *did*, the predicate to exchange data of length *len* demands for validity of the last port address *port* + *len*. Then, due to monotonicity all former port addresses are valid as well. The data exchanging preconditions for devices are defined as:

$\mathsf{pre\_cvmPhysIORange}\ did\ port\ len \equiv \mathsf{valid\_devid}\ did \wedge \mathsf{valid\_port}\ (port + len)$

Both device communication primitives internally call function $\delta^{m*}_{\mathsf{devs}}$ which performs several transitions on a device, depending on the data length *len*. The transition function is called with a read or write request from the processor. Finally, the device vector is updated with the final state of the recursive function and the corresponding outputs to the memory and the external interface are returned.

The primitive cvm_PhysIOInRange reading from a device calls the transition function $\delta^{m*}_{\mathsf{devs}}$ with a memory interface input list that signals read requests for the next *len* ports starting from port address *port*. The read data is given back to the operating system as a list of memory interface outputs. Thus, reading data of length *len* from the device with identifier *did* starting at port *port* is formally given with:

```
cvm_PhysIOInRange s_dev did port len ≡
  let (st, mifos, eifos) =
        δ^m*_devs (map (λport. (|mifi_rd = True, mifi_wr = False, mifi_a = port,
                                 mifi_din = default|))
                       [port..<port + len])
                  (s_dev (nat2dev did))
  in (s_dev(nat2dev did := st), mifos, eifos)
```

Function exec_cvmPhysIOInRange returns the state and creates a tagged list of external interface outputs. Furthermore, the read data from the device is converted from a list of natural numbers to a list of integer values.

```
exec_cvmPhysIOInRange s_dev did port len ≡
  let (devs', mifos, eifos) = cvm_PhysIOInRange s_dev did port len
  in (devs', create_eifo_list (nat2dev did) eifos, map to_int32 mifos)
```

Similarily, writing a datalist *dl* from the kernel to a device specified by its identifier *did* starting at port *port* calls function $\delta^{m*}_{\mathsf{devs}}$. Here, the memory interface input list signals write requests and delivers the data in $|dl|$ words to the successing ports. CVM primitive cvm_PhysIOOutRange returns the updated device vector together with the output lists to the memory and the external environment. Formally:

```
cvm_PhysIOOutRange s_dev did port dl ≡
  let (st, mifos, eifos) =
        δ^m*_devs (map (λ(p, d). (|mifi_rd = False, mifi_wr = True, mifi_a = p, mifi_din = d|))
                       (zip [port..<port + |dl|] dl))
                  (s_dev (nat2dev did))
  in (s_dev(nat2dev did := st), mifos, eifos)
```

Function exec_cvmPhysIOOutRange resembles function exec_cvmOutWord with the difference that it calls primitive cvm_PhysIOOutRange.

So far, we have introduced all functions and preconditions that are necessary to manipulate the user process or the device component. Until now, we cannot define the effects of CVM primitives in Simpl because we do not know the program state of the OLOS implementation yet. Therefore, we postpone the specification of the CVM primitives in Simpl to Section 5.1.1.

2.9 The Verification Environment Isabelle/Simpl for C0

For the verification of C0, a fragment of C, we use a general program-verification framework for sequential imperative programming languages: Isabelle/Simpl [Sch05, Sch06]. It is built as a conservative extension on top of Isabelle/HOL. The key feature of Isabelle/Simpl we use is the notion of a total correctness Hoare-triple: $\Gamma \vdash_t P\ c\ Q$. This judgment claims that in procedure environment Γ, given an initial state for which the precondition P holds, execution of statement c terminates and for the final state the postcondition Q holds. The assertions P and Q are formalized as sets of states. A program state represents memory and heap content mapped to the variables at some point of the program execution. We can generate the state representation from the program source code with a tool called `c0_check`, which was developed in the Verisoft project. The state space of a program is represented as a record, where every variable is a field. For common language constructs a concrete syntax hides the state-record. Whenever a record field of the state is referred to, we set the acute prefix ´ in front of the variable name i. e., ´var refers to field var in the record.

C0 variable		Simpl variable	
`bool`	`Sendflag`	sendflag	:: bool
`unsigned int`	`csn`	csn	:: nat
`int`	`result`	result	:: int
`unsigned int`	`AST[NS]`	AST	:: nat list

Figure 2.9: Simpl representation of basic types

Figure 2.9 depicts some C0 variables of basic types and their representation in Simpl. Our program models use pointers that do not support address arithmetic. In Simpl they are not typed and presented as an abstract type ref. The null pointer is defined as Null. Complex types in C0 i. e., structures or arrays of fundamental types are represented as a number of Simpl variables where each of them has a basic type.

In order to manipulate pointers to objects the verification environment includes a model of a program heap. The heap model, that is actually used in the verification environment, is the adapted split heap approach that goes back to Burstall [Bur72] and was successfully applied by Bornat [Bor00], Metha and Nipkow [MN05]. The heap presentation then uses functions that map variables of type ref to objects located on the heap. The main advantage of this heap model is that it excludes overlapping between different fields of structures by construction.

As an example for pointers and memory objects, we show the definition of message buffers and messages in the Olos implementation. There we consider a message type that consists of a list of `MSGLENGTH` integer values. Then, message buffers are represented as a list of `MSGCOUNT` pointers referencing the addresses where the messages are located.

```
typedef int t_kmsg[MSGLENGTH];
typedef t_kmsg *p_kmsg;
```

```
p_kmsg MB[MSGCOUNT];
```

The generated Simpl representation of this piece of C0 code returns a heap function that refers to messages located on the heap and a list of pointers:
heap_msg:: ref ⇒ int list
MB :: ref list

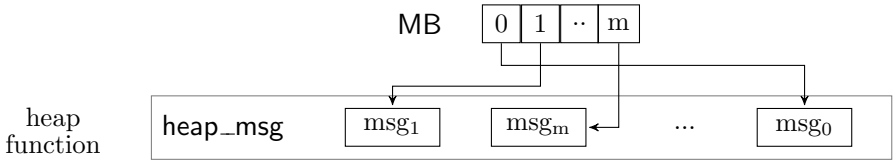

Figure 2.10: Messages on the heap

Accessing a memory object e. g., a message located in buffer ´MB ! ´n is denoted with ´MB ! ´n → ´heap_msg. Figure 2.10 depicts how messages are accessed via their message pointers on the heap.

Expressions in Simpl are HOL expressions. Statements, in contrast, are represented by a datatype, which we present in pseudo-code notation employing Isabelle's powerful syntax-translation machinery. In contrast to expressions, statements are represented by an abstract datatype. The statement syntax is highly abstract, e. g., Basic f represents a state update using function f. In order to present programs in conventional terms, we employ Isabelle's powerful syntax translation machinery and denote

- a program variable by ´var,
- an assignment by ´res_nat :=$_g$ 3,
- a conditional by **IF**$_g$ b **THEN** s_1 **ELSE** s_2 **FI**,
- a procedure call by ´var := **CALL**$_g$ square(7),
- a field initialization of new memory object by
  ´MB ! ´n :=$_g$ **NEW** [´heap_msg := replicate MSGLENGTH 0] etc.

The index $_g$ is used to instruct the parser to generate guards that protect against runtime faults like overflows.

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

procedures square(var | res_nat) =
´res_nat :=$_g$ ´var · ´var

It has one formal parameter called var and the result to return to the calling function is held in variable res_nat, i. e., the bar separates input and output parameters.

Calling the functionality of the procedure, we write ´res_nat :== PROC square(´var) as shorthand for the code of procedure `square`:

$\Gamma \vdash \{\!|\sigma.\ ^{\sigma}\mathsf{var} = x|\!\}$ ´res_nat :== PROC square(´var) $\{\!|\tau.\ ^{\tau}\mathsf{res\_nat} = x|\!\}$

states that procedure `square` assigns the value of its argument to the return variable. Procedure names of generated functions from a given C0 program always have the prefix fun_, in such a way that the generated Simpl procedure of e. g., C0 function `cvm_reset` is named fun_cvm_reset. The brackets {|...|} contain a HOL-expression and allow the ´ notation. Furthermore, {|σ. ...|} is a shorthand to fix the current state {s. σ = s ...}, ᵒvar abbreviates σ.var and refers to the value of variable var at fixed state σ.

2.9.1 Dealing with Inline Assembly in Isabelle/Simpl

In the context of C0 program verification we often have to deal with inline assembly code. Recall that these low-level computations can be encapsulated in XCalls (see Section 2.1) modifying an external state. It is not necessary to introduce a new Hoare rule into Simpl to handle XCalls. We rather express the semantics of an XCall by a single, guarded Basic command. A guard resembles possible preconditions to guarantee a successful execution of the following command. We write {| precondition |} ⟼ in front of a command to ensure that the command is only executed when the precondition (i. e., the boolean expression inside the brackets) is fulfilled. The definitions of some primitives require updates on several variables. A syntax extension allows to perform multiple assignments enclosed in BASIC and END, separated by commata. Finally, we can introduce bound variables in Simpl via customized let expressions. LET a IN b differs from the "regular" let-expressions by allowing to directly refer to program variables.

We use a small example to demonstrate the characteristics in the use of the described Simpl syntax: When we want to express the effects of the typed CVM primitive `cvm_v2p_copy_t_kmsg` we obtain the following signature of the corresponding Simpl function:
fun_cvm_v2pcopy_t_kmsg (pid, va, msg | msg, res_int)
The input parameters of the function are the process identifier pid, a start address *va* where the message is stored in the application and a variable msg containing a message value that is replaced by the new message. Note that this function has two output parameters: an updated message msg on the one hand and the result variable res_int on the other hand.

From Section 2.1 we know that a C0 state is extended with an additional component representing external states. In our case, the external state is a user process component ´up. Then, we can formalize the precondition pre_cvmV2Pcopy of the primitive and additionally require that the message length is fixed. The guard that ensures these requirements is given below:
{| pre_cvmV2Pcopy ´up ´pid (length ´msg) ´va ∧ length ´msg = MSGLENGTH |} ⟼

In order to describe the effects of the CVM primitive `cvm_v2p_copy_t_kmsg` in Simpl, we make use of the syntax described above. With the LET expression enclosed by BASIC and END we simulataneously assign the read message (i. e., the result of function exec_cvmV2Pcopy) and the return variable 0 to the C0 program. Formally:

BASIC
LET (*rval*, *msg*) = (0, exec_cvmV2Pcopy ´up ´pid ´va MSGLENGTH)
IN ´res_int :== *rval*, ´msg :== *msg*

END

If a program calls a procedure that updates more than one variable, we cannot use the syntax for procedure calls where the result is assigned to a left variable. Procedures with more than one modified variable are called with their input and output parameters. In our example, fun_cvm_v2pcopy_t_kmsg is called with variable ´ca that stores a process identifier, a variable ´msg_ptr storing the start address in the applications memory and the content of message buffer ´MB!(´msg_nr). Then, the function returns the new message value in buffer ´MB!(´msg_nr) and updates the result variable ´dummy. This is denoted with:

CALL$_{\mathsf{g}}$ fun_cvm_v2pcopy_t_kmsg(´ca,´msg_ptr,´MB ! ´msg_nr→´heap_msg,´MB ! ´msg_nr→´heap_msg,´dummy)

2.9.2 Code Verification in Isabelle/Simpl

In this subsection, we briefly describe the technique we use to prove implementation correctness of OLOS. Figure 2.11 depicts a comprehensive view on the verification target.

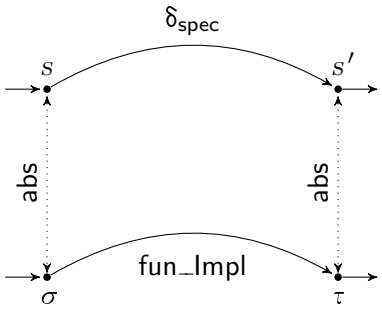

Figure 2.11: Simulation proof

Roughly speaking we relate an implementation state σ to a corresponding abstract state s via an abstraction function abs. On the concrete layer, the Simpl function fun_Impl encapsulates a number of implementation steps. Implementation correctness holds when all possible transitions of fun_Impl result into successor states τ that can be abstracted to corresponding states of the specification s' after a correlating transition δ_{spec} in the abstract model.

This oversimplified goal can be reformulated in Isabelle/Simpl as follows:

$\Gamma \vdash_{\mathsf{t}} \{\sigma.\ \mathsf{abs}\ \sigma = s\}\ \mathsf{PROC}\ \mathsf{fun\_Impl}()\ \{\tau.\ \mathsf{abs}\ \tau = \delta_{\mathsf{spec}}\ s\}$

The detailed correctness proof and its preliminaries are described more specifically in Chapter 5.

Verifying a program that uses nested procedure calls requires to consider the hierarchical program structure. We start with procedures on the lowest level that do not contain any procedure call. For each procedure we specify a so called *modifies clause* and a Hoare triple. The modifies clause only states all global state components (heap

functions and global variables) that are changed during the procedure execution without giving any information how they are updated. This circumstance is described in the procedure specification given as Hoare triple.

The verification condition generator searches for each procedure call that has to be verified whether a modifies clause can be found in the context. Its existence simplifies a procedure call by considering only changed variables. For nested procedure calls the VCG automatically uses specifications of already verified procedures.

3 OLOS Design and Implementation

> Vision without implementation is hallucination.
>
> *Benjamin Franklin*

Contents

This chapter reports on the design principles of a single ECU (Section 3.1), presents the underlying communication protocol (Section 3.2) and outlines the implementation of our real-time operating system Olos in Section 3.3. Initially, Steffen Knapp provided some code portions to illustrate his overall idea of the operating system. My contribution, however, has been the first working version of OLOS as well as the later reimplementation, which forms the current code base. To Mark Hillebrand, I am indebted for a reliable build- and testing environment as well as for many suggestions for code improvement.

The applications running on an ECU are written as C0 programs. The communication between them and our operating system, however, requires portions of inline assembly code. Hiding these code fragments into functions has several advantages: On the one hand, every application simply calls these primitives in order to request services from Olos, on the other hand, all of them can now be written in plain C0. In Section 3.4, we introduce all necessary functions for the kernel communication that form the system call library of Olos.

Finally, all ECUs communicate by exchanging messages. On the upper layers these messages are written in C0, whereas the operating system and the devices work with the compiled code. We dedicate a subsection Section 3.4.1 to the special message format and introduce some useful definitions based on this type.

3.1 ECU Structure

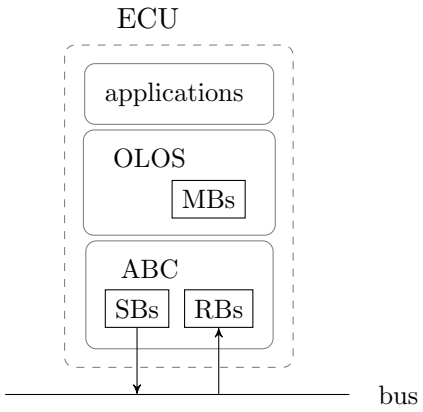

Figure 3.1: ECU structure

The distributed system considered in the automotive subproject of Verisoft comprises a number of components that are connected via a communication bus. These so-called *electronic control units* (ECUs) (depicted in Figure 3.1) consist of a general-purpose RISC processor and an *automotive bus controller* (ABC). The latter takes care of the timely transmission and reception of messages. This device is responsible for clock synchronization, thus decoupling the processor from the communication bus [KP07b]. In its mediator role, the ABC buffers the messages from both sides, using pairs of send (SB) and receive buffers (RB). We use double buffers because the parallel access from bus and processor to a single buffer would involve the risk of reading partially transmitted messages. The concept of buffer pairs ensures that the processor reads only completely transmitted messages.

Our operating system Olos runs on the processor, providing a virtual processor abstraction to the applications that share the same physical processor. Olos supports its own message buffers (MB) for the communication between applications (on the same as well as on different physical processors). Moreover, it includes functions in order to initialize the system and realize the scheduling policy. The operating system separates the applications from the bus and is responsible for the communication between these layers. More precisely, it organizes the correct flow within a slot by controlling the bus communication and scheduling the applications.

3.2 Partitioning Time

In our distributed system several ECUs share the same bus. A communication protocol formally describes rules in order to guarantee a fair and failure-free message exchange. The schedule of the transmission times on the bus is statically fixed and repeated perpetually. A period, or *round*, is subdivided into equal time slices, the so-called *slots*.

For each slot, we designate which component has the permission to send on the bus and determine a message buffer that reads a message from the device or sends its content to it. Furthermore, we schedule one application per slot to be executed. We keep this time organizing information within several scheduling tables. Each ECU features its own set of these tables:

Send-permission Table SPT The send-permission table SPT specifies for each slot whether the ECU is allowed to send on the bus.

Application Scheduling Table AST The application scheduling table AST determines the computing application of the ECU for each slot.

Buffer-index Table BT The buffer-index table BT identifies in each slot which message buffer of OLOS is exchanged with the device. An entry of BT in one slot is always related to the message that is broadcast on the bus in the same slot.

Table 3.1 shows an example of scheduling tables for two communicating ECUs. In this example, each round consists of four slots. Every ECU features its own local scheduling tables, hence we use the indices 1 and 2 to distinguish between the tables of both ECUs. If we regard SPT_1, we see that ECU 1 is permitted to send a message to the bus in slot 2. As BT_1 determines, this message is stored in message buffer 3. Note that there is a delay of one slot, i. e., if an application wants to send a message in slot 2, it must write it at least one slot earlier into the according message buffer. For instance, application 1 is scheduled in slot 0 and 1 (see AST_1). It may write a message into buffer 3 to be sent in slot 2. This value will be sent regardless of whether application 2 overwrites the buffer in slot 2. In slot 3, all ECUs store the transmitted message into the buffer determined by the entry of slot 2 in the corresponding BT. In particular, ECU 1 stores the message in buffer 3 and ECU 2 in buffer 2.

ECU 1					ECU 2				
slot	0	1	2	3	slot	0	1	2	3
SPT_1	no	no	yes	no	SPT_2	yes	yes	no	yes
BT_1	1	2	3	0	BT_2	0	1	2	1
AST_1	1	1	2	0	AST_2	0	1	2	3

Table 3.1: Example scheduling tables

We divide each slot into four phases: device communication, receive, compute, and send. Figure 3.2 on the next page depicts them for slot 2 according to the scheduling tables given in Table 3.1. In each phase the involved ECU components are highlighted. Note that in each phase either the ABC device (illustrated with light gray) or the currently scheduled application (highlighted with dark gray) are involved.

Device Communication. In this phase, the ABC device is communicating with the other ECUs via the bus. In our example, ECU 1 has the sending permission for this

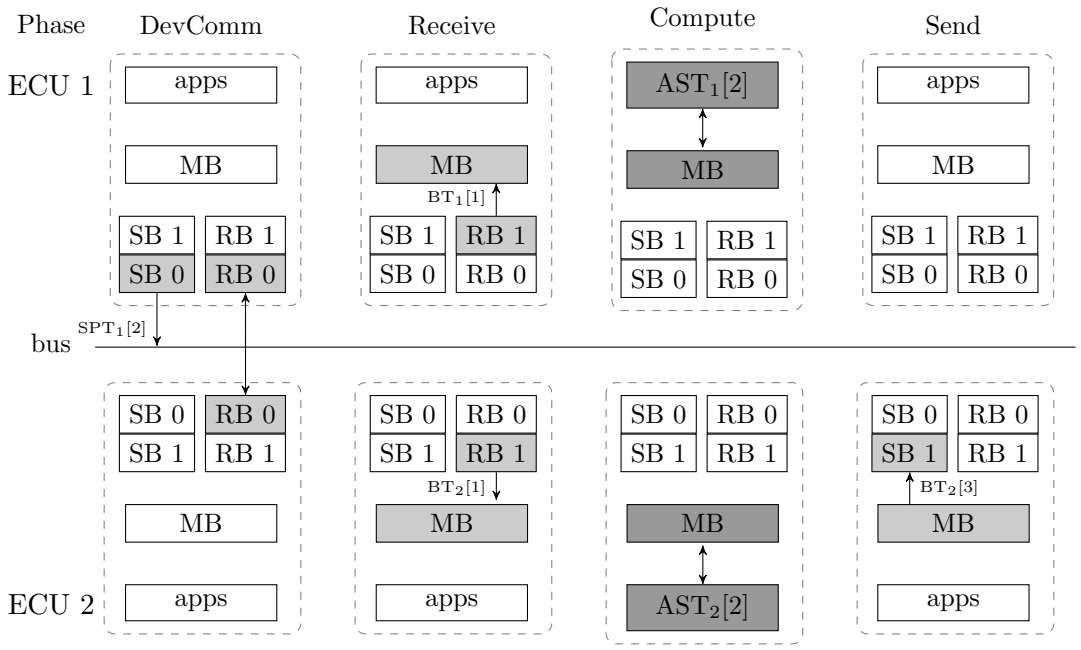

Figure 3.2: Transition phases during slot 2

slot. It broadcasts the message from the send buffer SB 0. All ECUs receive this message from the bus. Note that this phase does not involve OLOS.

Receive. The operating system reads the receive buffer RB 1 from the ABC device into its own message buffers. Figure 3.2 shows how the operating system reads the message that the device has received in the previous slot into the message buffer indicated by the BT table from the previous slot.

Compute. The AST specifies the application that is executed during this phase of the current slot. This application may compute locally or exchange messages with OLOS.

Send. This phase is only present if the corresponding ECU is permitted to send in the next slot – in our example, ECU 2. The operating system writes the message to be sent in the next slot into the ABC's send buffer SB 1.

Note that OLOS either exchanges messages with an application or with the ABC device. Recall that the ABC device has double buffers. When exchanging messages with the operating system, implicitly the buffers facing the processor are concerned. For the device communication, on the contrary, only the bus-facing buffers are involved. Note that the operating system is not aware of the device-communication phase because the bus-facing buffers are invisible for OLOS.

The receive and send buffers are swapped after each slot in such a way that processor

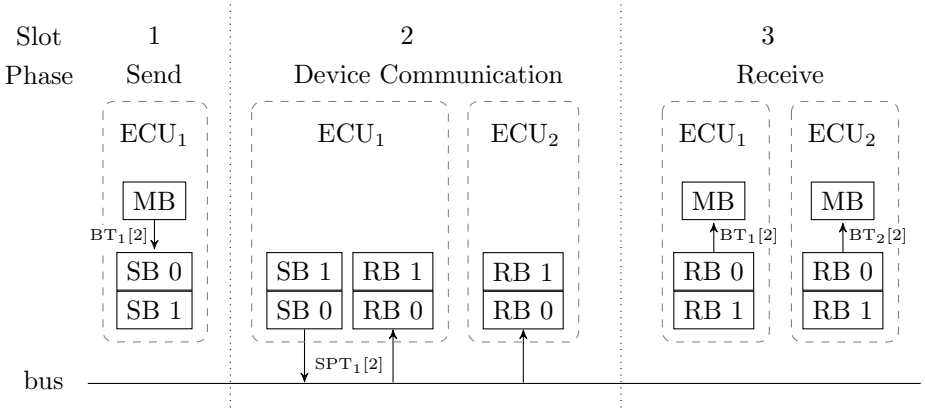

Figure 3.3: Message Transmission Delay

and bus access them alternately. In the implementation, the buffer access works with a parity bit. When the current slot number is even, the send and receive buffer with index 0 are visible for the processor. Otherwise, the buffers with index 1 are accessed from the processor. The buffer swapping is invisible for the operating system. Due to this mechanism, the transmission of a message from one ECU's message buffer to a second ECU takes two slots. Figure 3.3 briefly sketches this situation neglecting all unused components of the ECUs. In this sketch, we see that the path of a message from one ECU's message buffer into another one is determined by the sending-permission-table entry $SPT_1[2]$ and the local BT entries $BT_1[2]$ and $BT_2[2]$. Due to the transmission delay, the operating system computes the previous or next slot number to obtain the appropriate BT entry.

3.3 OLOS Implementation

3.3.1 OLOS Variables and Data Structures

In this section, we introduce the global variables and data structures that are used in the OLOS implementation. The foundation of our correctness proof is the Simpl state and the functions that are generated from the C0 implementation code. Therefore, we directly describe the implementation in Simpl and refer the interested reader to the C0 source code which is shown in Chapter A. Additionally, Table A.1 on page 198 presents the C0 variables of the implementation side by side with the generated Simpl variables.

All applications running under OLOS are known initially and their number is bounded by PROCCOUNT. An additional array PAGEC of length PROCCOUNT + 1 keeps the required memory size in pages for each application. The nth-array element corresponds to the user process with the same process identifier and the first field is set to 0 since process identifier 0 is reserved for the kernel. OLOS initializes PAGEC with the application sizes that are kept in PAGEC_CONF.

Applications never communicate directly with each other or with the ABC device. All messages are sent and received via the kernel that controls the message transfer. Therefore, OLOS provides a number of MSGCOUNT message buffers in order to store messages coming either from an application or the ABC device. The buffers are modelled as a pointer array MB to the kernel message type with a fixed length MSGLENGTH. (recall, Section 2.9 for the Simpl representation of the message buffers).

The number of slots is fixed by SLOTCOUNT. OLOS provides three tables of length SLOTCOUNT that are responsible for the scheduling of a round: the application scheduling table AST, the buffer index table BT and the sending permission table SPT. The initial table configurations are stored in the constants AST_CONF, BT_CONF and SPT_CONF.

Furthermore, there are several internal variables implementing the functionality of the communication protocol: First, variable csn keeps track of the current slotnumber. Its predecessor and successor value is computed by subtraction and addition modulo SLOTCOUNT i. e., $(\sigma.\mathsf{csn} + \mathsf{SLOTCOUNT} - 1) \bmod \mathsf{SLOTCOUNT}$ and $(\sigma.\mathsf{csn} + 1) \bmod \mathsf{SLOTCOUNT}$. We abbreviate the computation of the predecessor and successor slot number with two functions prev and next.

The main function of OLOS is entered with two external variables: Variable eca stores the masked exception cause as a natural number and the second variable edata keeps additional exception data (recall Section 2.6). The value of eca signals external interrupts i. e., $(\sigma.\mathsf{eca} \wedge_\mathsf{u} 32) \neq 0$ in case of a trap or $(\sigma.\mathsf{eca} \wedge_\mathsf{u} 8192) \neq 0$ when a pending interrupt line of the ABC device was detected. Additionally to the variables eca and edata, we require information how the operating system should react to an incoming timer interrupt $(\sigma.\mathsf{eca} \wedge_\mathsf{u} 8192) \neq 0$. Therefore, OLOS features a Boolean variable sendflag in order to distinguish a slot boundary (receive phase, $\sigma.\mathsf{sendflag} = \mathsf{False}$) from the start of a send phase i. e., $\sigma.\mathsf{sendflag} = \mathsf{True}$.

The OLOS main function returns a variable ca. Either it stores the identifier of the application that will be resumed after returning from the kernel. It can be derived from the current entry of the application scheduling table AST. Otherwise, ca holds the special value IDLE which forces the system to wait for the next incoming timer interrupt.

		entering OLOS			return from OLOS	
Variables		eca	edata	sendflag	sendflag	ca
Phase	Init	reset	*	*	False	IDLE
	Receive	timer	*	False	SPT [next csn]	AST [csn]
	Send	timer	*	True	False	IDLE
	Compute	trap	> 0	SPT [next csn]	SPT [next csn]	AST [csn]
			0			IDLE

reset $= ((\mathsf{eca} \wedge_\mathsf{u} 1) \neq 0)$ // bit 1 of interrupt vector is set
timer $= \neg$ reset $\wedge ((\mathsf{eca} \wedge_\mathsf{u} 8192) \neq 0)$ // bit 13 of interrupt vector is set
trap $= \neg$ timer $\wedge ((\mathsf{eca} \wedge_\mathsf{u} 32) \neq 0)$ // bit 5 of interrupt vector is set

Figure 3.4: OLOS phases and variables

The right columns of Table 3.4 on the facing page depicts the variable values of eca, edata and sendflag when the main function of OLOS is entered. These values are sufficient to determine the actual phase. The left columns present the values of the variables sendflag and ca after OLOS returns from its computation. Depending on the phase these values are set to certain values.

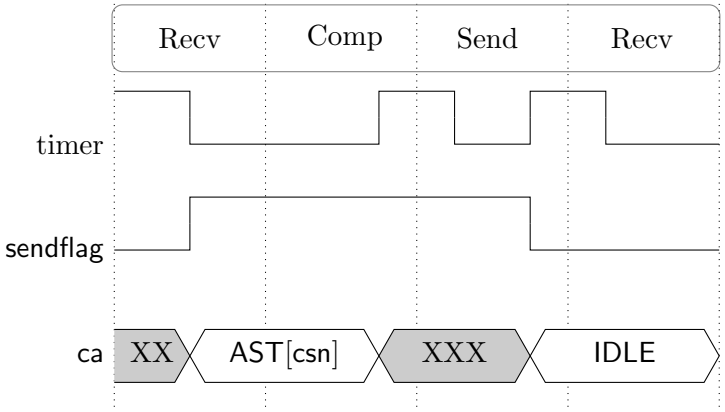

Figure 3.5: Timing diagram of OLOS phases and variables

The timing diagram Figure 3.5 depicts the values of the timer interrupt bit together with the variables sendflag and ca. In the szenario we regard the OLOS variables within an entire slot where the ECU is allowed to send. The receive phase starts with a set timer interrupt bit and an unset sendflag variable. The value of ca is irrelevant. During this phase the interrupt is cleared, variable sendflag is set (because the ECU has the sending permission) and variable ca holds the identifier of the currently scheduled application. OLOS enters the send phase when the timer is raised again regardless of whether the application terminated or not. In the send phase OLOS sets variable sendflag to False, variable ca to IDLE and clears the interrupt. Then, the next timer interrupt signals to start a new slot. Note that timing diagrams only show a snapshot of the variables behaviour in a certain situation.

Port	Address	Description
SEND_PORT	0	Send buffer start
RECV_PORT	256	Receive buffer start
CONFR_PORT	512	Configuration register start
COMR_PORT	767	Command register port
Command		**Description**
SETRD_COM		Signals termination of the initialization
CLEARINT_COM		Clears ABC interrupt

Table 3.2: ABC I/O ports and commands

Finally, the operating system communicates with the ABC device which is accessed

by using its identifier ABC_ID. Besides the send and receive buffers, the device features several configuration registers. After power-up the kernel writes their intial values that are stored in CB_CONF into the registers. A detailed description of their content is given in Table 4.1 on page 77. OLOS communicates with the device using ports. These word-sized I/O ports are mapped to the ABC's send and receive buffers, the configuration registers and a special command register. The latter is used from the processor to manipulate the device state by sending special words to the command port. All ports and the command codes used by OLOS are summarized in Table 3.2 on the previous page.

3.3.2 The Top-Level Function

Whenever an interrupt arrives the CVM framework saves the old processor context and passes control to the top-level function of OLOS. This function called `kdispatch` additionally gets two arguments: the *interrupt cause*, a natural number encoding the bit vector of occured interrupts, and the *exception data*, which is the provided immediate constant when a trap instruction is executed.

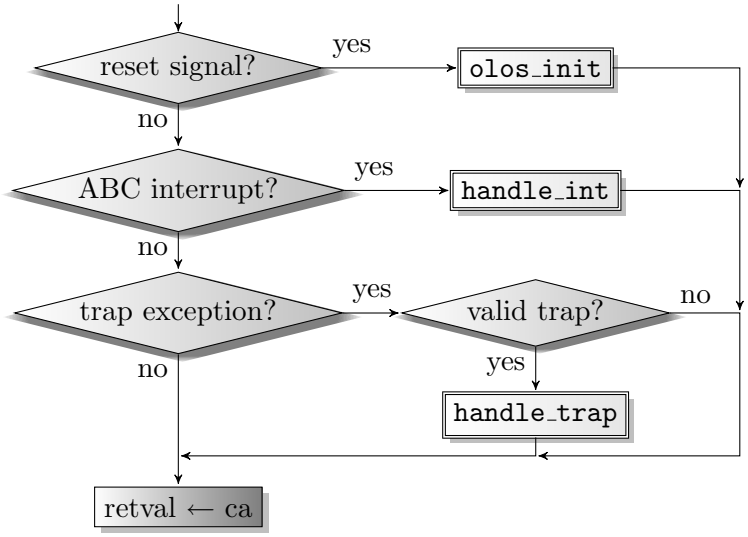

Figure 3.6: The program-flow diagram of function `kdispatch`

Figure 3.6 shows the control-flow graph of this function after entering OLOS from the CVM framework. The corresponding C0 implementation code is depicted in A.1.6. From the interrupt cause, the function distinguishes three cases:

Initialization. After power-up, the processor generates a *reset* interrupt and the CVM framework sets up its internal data structures. Afterwards, it passes the interrupt on to the `kdispatch` function. `kdispatch` directly calls the function `olos_init` with a set reset-interrupt bit and ignores the remaining interrupt vector. `olos_init` initializes

internal data structures of Olos and configures the ABC devices as described in the next subsection.

ABC Interrupt. The ABC device raises its interrupt line when expecting data exchange with the operating system. From Section 3.2 we know that Olos either receives a message from the device (in the receive phase) or sends a message to a device buffer (in the send phase). The communication with the ABC devices is encapsulated in the function `handle_int`.

Trap Exception. The current application may communicate with the operating system during the compute phase via trap exceptions. This hardware mechanism is used to request the transfer of messages between Olos and the applications. Furthermore, an application signals its termination (for the current slot) by a trap exception. The function `handle_trap` implements the handling of the operating system in case of an incoming trap.

Potential other interrupts are ignored. The current application might cause a number of exceptions during its execution by, e. g., misaligned addresses or attempts to access memory outside its dedicated address range. Such exceptions certainly should not occur in a fully verified system but despite that, even a malicious application can only harm those applications it exchanges messages with.

The function `kdispatch` returns the value of the global variable ca to Cvm. It either holds the value of the AST table in the current slot determining the currently running application or the special value IDLE. Upon return, the Cvm framework transfers control to the corresponding application or waits for interrupts, respectively.

3.3.3 System Initialization

The initialization of Olos is encapsulated in function `olos_init` and includes (a) the initialization of internal data structures, (b) the initialization of all applications, (c) and the configuration of the ABC device. This fact is demonstrated in the generated Simpl function fun_olos_init of the implementation code Figure 3.7 on the next page (see A.1.5 for the C0 code of the function). fun_olos_init employs generated Simpl functions (cf. Section 2.9) of all used Cvm primitives. Although, we have not defined these functions in Simpl, we know the effects of their C0 function counterparts from Section 2.8.3.

Initialization of OLOS Data Structures. In this part, the scheduling tables and PAGEC are initialized with the values of their corresponding configuration tables. The global variables ca, csn and sendflag are set so that the first incoming timer interrupt starts the receive phase of slot 0. The first loop creates new message buffers with zero-initialized message values.

Setting-up Applications. A second loop prepares and loads all applications sequentially. fun_cvm_reset configures the PCs and registers of each application. Then fun_

```
procedures fun_olos_init( | res_int) =
´AST :=g AST_CONF;
´BT :=g BT_CONF;
´SPT :=g SPT_CONF;
´PAGEC :=g PAGEC_CONF;
´csn :=g SLOTCOUNT − 1;
´ca :=g IDLE;
´sendflag :=g False;

(∗ Part I: Message−Buffer Initialization ∗)
´n :=g 0;
WHILEg ´n < MSGCOUNT
DO ´MB ! ´n :=g NEW [´heap_msg := replicate MSGLENGTH 0];
    ´n :=g ´n + 1
OD;

(∗ Part II: Application Initialization ∗)
´n :=g 1;
´next :=g 0;
WHILEg ´n ≤ PROCCOUNT
DO ´dummy := CALLg fun_cvm_reset(´n);
    ´dummy := CALLg fun_cvm_alloc(´n,´PAGEC ! ´n);
    ´dummy := CALLg fun_cvm_load_os(´PAGEC ! ´n,´n,´next);
    ´next :=g ´next + ´PAGEC ! ´n;
    ´n :=g ´n + 1
OD;

(∗ Part III: Device Initialization ∗)
´conf_buff :=g CB_CONF;
´n :=g 0;
WHILEg ´n < SLOTCOUNT
DO ´dummy := CALLg fun_cvm_out_word(ABC_ID,CONFR_PORT + ´n,´conf_buff ! ´n);
    ´n :=g ´n + 1
OD;
´dummy := CALLg fun_cvm_out_word(ABC_ID,COMR_PORT,SETRD_COM);
´res_int :=g 0
```

Figure 3.7: Implementation of the initialization function

cvm_alloc allocates the required application size in memory. Finally fun_cvm_load_os loads the application to the first address directly behind the already used memory.

ABC Configuration. The last part of fun_olos_init initializes the ABC device. All configuration data stored in array conf_buff is written consecutively to the ABC's configuration registers starting at address CONFR_PORT. This data include system-specific information like the slot length in hardware cycles, the number of slots per round, the size of a message in bytes or the sending permission table.

Finally, the set-ready signal SETRD_COM is sent to the command port COMR_PORT of the ABC device ABC_ID using CVM primitive fun_cvm_out_word. Thereby, we indicate the termination of the initialization phase.

3.3.4 Communicating with the Device

The communication between the ABC device and the operating system takes place during the receive or the send phase. The ABC signals the start of these phases by a raised interrupt line. The handling of an incoming interrupt from the device is encapsulated in the OLOS function `handle_int`. Figure 3.8 shows the corresponding Simpl function fun_handle_int (the C0 code is depicted in A.1.3). The same as in the initialization function, we use generated Simpl functions to express the effects of CVM primitives. Although, we have not specified the ABC device transition and the CVM primitives in Simpl yet, we know from Section 2.8.3 that the corresponding C0 functions `cvm_physIOInRange_t_kmsg` and `cvm_physIOOutRange_t_kmsg` are used to exchange messages with a specified device.

```
procedures fun_handle_int( | res_int) =
IF_g ´sendflag
THEN ´uresult :=_g ´BT ! ((´csn + 1) mod SLOTCOUNT);
  ´dummy := CALL_g fun_cvm_physIOOutRange_t_kmsg(´MB ! ´uresult→´heap_msg,
                ABC_ID,SEND_PORT);
  ´ca :=_g IDLE;
  ´sendflag :=_g False
ELSE ´csn :=_g (´csn + 1) mod SLOTCOUNT;
  ´sendflag :=_g ´SPT ! ((´csn + 1) mod SLOTCOUNT);
  ´uresult :=_g ´BT ! ((´csn + SLOTCOUNT − 1) mod SLOTCOUNT);
  CALL_g fun_cvm_physIOInRange_t_kmsg(ABC_ID,RECV_PORT,
    ´MB ! ´uresult→´heap_msg,´MB ! ´uresult→´heap_msg,´dummy);
  ´ca :=_g ´AST ! ´csn
FI;
´dummy := CALL_g fun_cvm_out_word(ABC_ID,COMR_PORT,
              CLEARINT_COM);
´res_int :=_g 0
```

Figure 3.8: Implementation of the device-interrupt handling

The function distinguishes both communication phases by the current value of the variable sendflag.

In case that it is True, the ECU is permitted to send on the bus in the next slot. Then, our operating system starts the send phase and transfers the designated message to the send buffer of the device using the CVM primitive fun_cvm_physIOOutRange_t_kmsg (i. e., the function sends the message to port address SEND_PORT of the ABC device). Afterwards, OLOS sets variable σ.ca = IDLE and lowers the sendflag so that the system simply awaits the next incoming interrupt indicating the slot boundary. The send phase is omitted when the ECU is not allowed to send in the next slot.

When OLOS receives an incoming interrupt from the device and the send flag was not set (False) it starts immediately the receive phase of a new slot. Therefore, it increases the current slot number csn and sets the send flag sendflag to the new value according to the entry of the sending permission table SPT. The operating system writes the message from the ABC's receive buffer into the designated message buffer using function fun_cvm_physIOInRange_t_kmsg. Finally, it schedules the current application (according to the AST entry) before the compute phase starts.

At the end of both device communication phases, OLOS instructs the device with primitive fun_cvm_out_word to lower its interrupt flag before returning to CVM.

3.3.5 Implementing System Calls - the Trap Handler

As sketched in Section 2.8, the `trap` instruction is the designated hardware mechanism to request services from the operating system. We call these requests *system calls*. The trap instruction features an immediate constant, which allows us to distinguish different kinds of system calls. Furthermore, parameters of a system call are delivered via specified registers and a designated result register 22 stores the return value of the system call.

OLOS implements three system calls (shown in Table 3.3): The calls *olosSendMsg* and *olosRecvMsg* are used for the message exchange between OLOS and the applications. Moreover, an application should signal with the call *olosExFinished* that it has reached a synchronization point, thus finishing the computation intended for the current slot. The application is resumed for further computation in a future slot. In a perfect system, all applications finish on time and call *olosExFinished*. Our operating system, however, works correctly even if this requirement is violated.

OLOS system call	`trap`	description
`olosExFinished()`	0	termination signal for current slot
`olosSendMsg (msgp, mn)`	1	send message to OLOS MB
`olosRecvMsg (msgp, mn)`	2	receive message from OLOS MB

Table 3.3: OLOS system calls

If an application executes a `trap` instruction and the provided immediate constant corresponds to one of the three system calls, the dispatcher calls the function `handle_trap`. The C0 code of the trap handler is presented in A.1.4. Figure 3.9 on the facing page shows the control-flow diagram of this function.

The implementation of the *olosExFinished* call is simple: OLOS sets its global variable

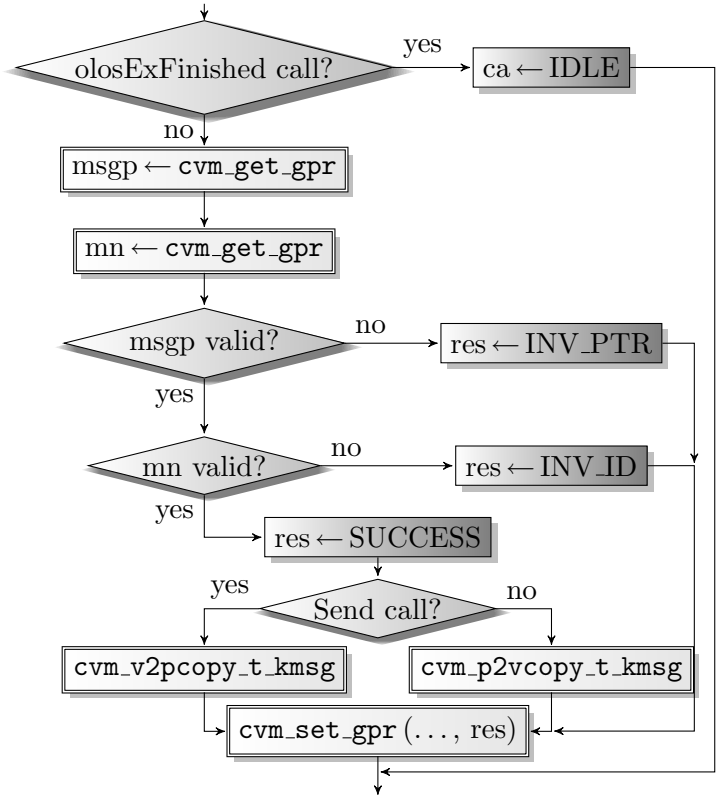

Figure 3.9: The program-flow diagram of function `handle_trap`

`ca` to the value IDLE and returns, which causes the processor to idle until an ABC interrupt occurs. This interrupt marks the end of the computation phase.

The remaining two system calls take two parameters:

- a pointer *msgp* into the memory of the application indicating either the message value (for the *olosSendMsg* call) or the designated memory that should accommodate the message (for the *olosRecvMsg* call), and
- a message number *mn* identifying the message buffer in OLOS as source or destination of the message, respectively.

These parameters are held in registers. The implementation reads them using the CVM primitive `cvm_get_gpr` and checks their validity. This means in particular that *msgp* contains a valid memory address and the complete message fits into the applications memory or *mn* is a valid buffer identifier. If these checks fail, OLOS simply stores an error value in the result register 22 of the current application (using the CVM primitive `cvm_set_gpr`).

In the successful case, the message value is transferred either from OLOS's message buffer into the memory of the application using the CVM primitive `cvm_p2vcopy_t_kmsg`

(for *Send*), or from the application into OLOS using `cvm_v2pcopy_t_kmsg` (for *Receive*). Finally, the designated result register is set to the value SUCCESS.

3.4 The System Call Library

As we have learned in the last section, there are several applications running on each ECU. Applications are usually written in C0 (recall Section 2.3). In order to request a service from the operating system it uses the mechanism of system calls. Unfortunately, this mechanism requires the support of inline assembly portions within C0 programs. We encapsulate the invocation of the system calls with the `trap` instruction in a library of C0 functions and hide thereby the portion of inline-assembly code. Possible parameters of the function are passed via specified registers (`r11` and `r12`) and register `r22` stores the result value after the execution of the `trap` instruction. Besides the fact that the application code can now be written in plain C0, the functions can be reused from each application. We call this library maintaining these functions *system call library*.

The implementation of the function `olosExFinished` is presented in Table 3.4. An application uses this call to signal the termination of its execution. Hence, we neither require function parameters nor a return value. Thus, we immediately return 0 after we called OLOS with the assembly instruction `trap 0`.

```
int olosExFinished()
{
  asm { trap(0); };
  return 0;
}
```

Table 3.4: C0-Code of `olosExFinished`

```
int olosSendMsg (p_msg msgp, unsigned int mn)
{
  int result;
  asm { lw(r11, r30, asm_offset(msgp));
        lw(r12, r30, asm_offset (mn));
        trap(1);
        sw(r22, r30, asm_offset(result));
    };
  return result;
}
```

Table 3.5: C0-Code of `olosSendMsg`

The implementations of the message transferring system calls (i. e., `olosSendMsg` and `olosRecvMsg`) are very similar. Their code fragments only differ in the function name and the trap number (shown in Table 3.3 on page 68). Hence, we restrict ourselves to the presentation (Table 3.5) and description of function `olosSendMsg`.

The function parameters of `olosSendMsg` are the message pointer `msgp` and the message number `mn`. The trap handler of OLOS (recall Section 3.3.5) takes care of the meaningful interpretation of these variables. The message transmission only succeeds when both parameters are valid.

Both C0 variables have to be related to the compiled code. Whenever a function is called, the stack frame is extended by a new local frame. There, the function parameters

and the local variables are stored. Recall that the address of the top most stack frame is kept in register `r30` (Figure 2.4 on page 27). The offset `asm_offset` specifies the displacement of both parameters with respect to the base address of the top most stack.

For further VAMP-assembly computations, the memory operation `lw` loads the parameter values into the registers `r11` and `r22` of the calling application. After invoking the `trap` instruction, the operating system reads these registers and handles the system call (A.1.4 presents the C0 source code of the trap handler) and writes the result value into the designated result register `r22`. Afterwards, the store word instruction `sw` copies the content of result register `r22` back into the local result variable of the function. Finally, the function passes its local result variable back to the calling application.

3.4.1 The Message Structure

The last subsection is dedicated to the special message format that was prespecified by our project partners from the technical university of Munich [BBG+08]. They modelled applications running on OLOS on the top most layer as automata (AutoFocus Task Model AFTM). Therefore, they used their CASE tool AutoFocus which was developed for modelling distributed systems.

Initially, our operating system was designed to deal with different message types. Later on, it turned out that there was only one message format used by their generated AutoFocus-tasks. In the following, we do not worry about the meaning of the message variables. Nevertheless, our verification proofs rely on the specific message format and we briefly introduce the necessary information.

Our proofs rely on the result of the `c0_check` tool. Hence, we present the C0 type declaration together with the automatically generated C0 small-step types.

First, there are several type definitions that are based on integer values. They are converted internally to the elementary type Integer.

```
typedef int TYPE_CMV;
typedef int TYPE_CCMV;
typedef int TYPE_CSMV;
typedef int TYPE_Signal;
```

Moreover, a structural type `TYPE_Coordinate` defines coordinates with two integer coordinate values called `xCoord` and `yCoord`. The juxtaposition of the C0 declaration and the generated C0 small-step type is depicted in Figure 3.12 on the next page. The C0 small-step coordinate type TYPE_Coordinate′ty consists of a two field structure with the two variables. Each one of them is given by its name and type.

The entire message format `TYPE_Message_Struct` contains `MSGLENGTH = 40` variables of elementary type. The first variable is defined as an `unsigned int`, whereas the remaining 39 values are integers represented as three coordinates, 22 different signal values and a dummy array collecting the residual unused variables. `c0_check` automatically reduces all elementary type definitions to their underlying basic types e. g., `TYPE_CMV` simply becomes an Integer in the C0 small-step type definition. Then, the type definition of the C0 small-step message type TYPE_Message_Struct′ty includes only variables

```
struct TYPE_Coordinate{
  int xCoord;
  int yCoord;          };
```

Figure 3.10: C0 Type declaration

```
TYPE_Coordinate'ty ≡ Struct
    [("xCoord", Integer),
     ("yCoord", Integer)]
```

Figure 3.11: C0 small-step types

Figure 3.12: Declaration of a Coordinate Type

of the aggregate coordinate type TYPE_Coordinate'ty, an integer array, and the basic types UnsgndT and Integer. The original C0 definition and the generated one are placed side by side in Figure 3.15 on the facing page.

Applications are written in C0 and use the structured message type that is stored in their typed memory. The operating system, however, maintains the compiled code of the applications where the message format is converted into plain integer lists and kept in the untyped assembly memory. Hence, OLOS operates directly on those integer lists. The C0 compiler takes care of the conversion between untyped assembly memory and the typed C0 memory. For the specification, we have to define two conversion functions that turn the message type into a list of integer values and vice versa.

Variable values in the C0 memory model are stored in memory cells. A single memory cell contains values of an elementary type, whereas values of aggregate types are stored as a consecutive sequence of memory cells. For each elementary type, there are functions to read a value of a memory cell (e. g., mem2int) and functions to store a value into a typed cell (e. g., int2mem). The memory of assembly processes, the kernel message buffers and the device buffers in contrast only store integer values.

A message stored in the C0 memory can be converted into a plain integer list of length MSGLENGTH with function struct2intlist. The first component of the message is read from the memory cell with function mem2unsigned and is converted to an integer value with to_int32. The remaining variables simply have to be read from their memory cells using function mem2int. Afterwards, function map glues all elementary values to an integer list of length MSGLENGTH. Formally:

```
struct2intlist data ≡
  map (λx. if x < MSGLENGTH
             then if x = 0 then to_int32 (mem2unsigned (data x))
                  else mem2int (data x)
           else default)
    [0..<MSGLENGTH]
```

From a given integer list of length MSGLENGTH, we reconstruct an updated C0 memory with function intlist2struct. This function converts the first element into a natural number with to_nat32 and stores it into a memory cell with function unsigned2mem. The remaining message values are consecutively written to the C0 memory cells using int2mem. The other cells are set to default. Function intlist2struct is defined as:

```
intlist2struct data ≡
```

```
struct TYPE_Message_Struct{
  unsigned int Field;
  TYPE_CMV crash19;
  TYPE_CSMV connection_status;
  TYPE_Signal c2eCall;
  struct TYPE_Coordinate coord;
  TYPE_Signal started23;
  struct TYPE_Coordinate outCoord;
  TYPE_CCMV connection_control;
  TYPE_Signal finished26;
  TYPE_CCMV connection_control27;
  TYPE_Signal
       Taskmodel_connection_failed_channel;
  struct TYPE_Coordinate coord29;
  TYPE_Signal c2MP;
  TYPE_Signal started31;
  TYPE_Signal finished32;
  TYPE_Signal c2GPS;
  int x;
  int y;
  TYPE_Signal finished36;
  TYPE_Signal started37;
  TYPE_Signal end_eCall2c;
  TYPE_Signal start_eCall2c;
  TYPE_Signal start_mP2c;
  TYPE_Signal start_GPS2c;
  TYPE_Signal end_GPS2c;
  TYPE_Signal end_mP2c;
  int dummy[11];          };
```

Figure 3.13: C0 type declaration

```
TYPE_Message_Struct'ty ≡ Struct [
                          ("Field", UnsgndT),
                          ("crash19", Integer),
                          ("connection_status", Integer),
                          ("c2eCall", Integer),
                          ("coord", TYPE_Coordinate'ty),
                          ("started23", Integer),
                          ("outCoord", TYPE_Coordinate'ty),
                          ("connection_control", Integer),
                          ("finished26", Integer),
                          ("connection_control27", Integer),
("Taskmodel_connection_failed_channel",
                                          Integer),
                          ("coord29", TYPE_Coordinate'ty),
                          ("c2MP", Integer),
                          ("started31", Integer),
                          ("finished32", Integer),
                          ("c2GPS", Integer),
                          ("x", Integer),
                          ("y", Integer),
                          ("finished36", Integer),
                          ("started37", Integer),
                          ("end_eCall2c", Integer),
                          ("start_eCall2c", Integer),
                          ("start_mP2c", Integer),
                          ("start_GPS2c", Integer),
                          ("end_GPS2c", Integer),
                          ("end_mP2c", Integer),
                          ("dummy", Arr 11 (Integer))]
```

Figure 3.14: C0 small-step types

Figure 3.15: Declaration of the Message Type

λx. **if** $x <$ MSGLENGTH
then if $x = 0$ **then** unsigned2mem (to_nat32 ($data$! x))
else int2mem ($data$! x)
else default

Whenever an application requests to receive a message from the operating system, it calls function `olosRecvMsg`. Section 2.5 reported how to deal with inline assembly code within a C0 program and how to construct a C0 state after an executed portion of inline assembly code. In order to update the memory, function C0_asm_upd takes a list of elementary g-variables. We introduce a function msg_gvars that takes the root

g-variable *gvar* of a message and returns all of its elementary g-variables as a list. Thus, we can hide the entire list of g-variables into a function that extracts them after its application:

```
msg_gvars gvar ≡
  [gvar_str gvar "Field", gvar_str gvar "crash19",
   gvar_str gvar "connection_status", gvar_str gvar "c2eCall",
   gvar_str (gvar_str gvar "coord") "xCoord",
   gvar_str (gvar_str gvar "coord") "yCoord", gvar_str gvar "started23",
   gvar_str (gvar_str gvar "outCoord") "xCoord",
   gvar_str (gvar_str gvar "outCoord") "yCoord",
   gvar_str gvar "connection_control", gvar_str gvar "finished26",
   gvar_str gvar "connection_control27",
   gvar_str gvar "Taskmodel_connection_failed_channel",
   gvar_str (gvar_str gvar "coord29") "xCoord",
   gvar_str (gvar_str gvar "coord29") "yCoord", gvar_str gvar "c2MP",
   gvar_str gvar "started31", gvar_str gvar "finished32",
   gvar_str gvar "c2GPS", gvar_str gvar "x", gvar_str gvar "y",
   gvar_str gvar "finished36", gvar_str gvar "started37",
   gvar_str gvar "end_eCall2c", gvar_str gvar "start_eCall2c",
   gvar_str gvar "start_mP2c", gvar_str gvar "start_GPS2c",
   gvar_str gvar "end_GPS2c", gvar_str gvar "end_mP2c",
   gvar_arr (gvar_str gvar "dummy") 0, gvar_arr (gvar_str gvar "dummy") 1,
   gvar_arr (gvar_str gvar "dummy") 2, gvar_arr (gvar_str gvar "dummy") 3,
   gvar_arr (gvar_str gvar "dummy") 4, gvar_arr (gvar_str gvar "dummy") 5,
   gvar_arr (gvar_str gvar "dummy") 6, gvar_arr (gvar_str gvar "dummy") 7,
   gvar_arr (gvar_str gvar "dummy") 8, gvar_arr (gvar_str gvar "dummy") 9,
   gvar_arr (gvar_str gvar "dummy") 10]
```

In the following chapter that reports on the formal implementation correctness proof we consider applications and messages from the operating system's view. Thus, applications modelled as VAMP assembly machines and messages are plain integer lists. Chapter 6, in contrast, presents an approach to pervasively verify applications running on top of OLOS. Hence, we also consider C0 applications exchanging messages with the presented structured type.

4 Formal Specification of an ECU

> An abstraction is one thing that represents several real things equally well.
>
> *Edsger Dijkstra, quoted by David Lorge Parnas: "Use the Simplest Model, But Not Too Simple".*

Contents

In this chapter, we introduce a mathematical description of an entire ECUs behaviour. Note that OLOS relies on a specific protocol with the ABC, i. e., assuming a certain device state, the operating system only expects a restricted set of device inputs. Consequently, we verify the correctness of the operating system with respect to a device model which specifies the behaviour of the ABC. The ABC automaton is described in Section 4.1.

Section 4.2 describes a formal specification of the communication interface between applications and the operating system which was inspired to large extent by [DDWS08, DDB08]. Moreover, we instantiate our formal model for assembly and C0 processes. This technique is crucial to prove application simulation aside from the functional correctness theorem.

The presentation of an entire ECU model consisting of the operating system and the ABC device is shown in Section 4.3.

Finally, we refer to the thesis of Knapp [Kna08] that deals with a distributed ECU model. We point out some slight differences between both specifications and present a simulation theorem between our ECU model and Knapp's model of a single ECU within a single slot in Section 4.3.1.

4.1 ABC Automaton

This section presents the state and the transition function of our ABC device and specifies some predicates concerning the ABC state. Previously, the behaviour of the ABC

device was only partially specified to the extent needed for the interaction with the operating system. More specifically, the device featured only the processor-visible buffers and the device-communication phase had been neglected. We have extended the specification by adding support for modelling the double buffers from the implementation, on the one hand, and the communication of the device with the external environment, on the other hand.

ABC States. As sketched in Section 3.2, the ABC device is responsible for the timely transmission and reception of messages. For this purpose, the ABC device features a double buffer for outgoing messages (*send buffers*) and a second one for incoming messages (*receive buffers*). In addition, the device requires a considerable amount of system-specific information. Namely, this information comprises the slot length in hardware cycles, the sending permission table SPT, and custom-tailored parameters like the message size or the number of slots per round. Furthermore, the device features a timer that signals the start of the send phase to Olos, which requires information about the length of the receive and compute phases in hardware cycles. All this information is held in the so-called *configuration registers*, which are set up by Olos in an initialization phase right after power up. This phase is identified by an *initialization flag*, which is initially raised and remains enabled until Olos signals the completion of the initialization by writing the set-ready command. Hence, the ABC device switches from initialization to running mode. Finally, the ABC state includes a *slot counter* keeping track of the current slot number, a *send flag* distinguishing send- and receive phase, and an *interrupt flag*, representing the ABC's interrupt line. Summarizing, the state of the ABC automaton is represented in Isabelle/HOL by a record with the following components:

- the receive buffers $s_{\mathsf{abc}}.\mathsf{RB}$ and send buffers $s_{\mathsf{abc}}.\mathsf{SB}$, modelled as lists of messages,
- the configuration registers $s_{\mathsf{abc}}.\mathsf{CR}$,
- three flags classifying ABC protocol states, i. e., an interrupt flag $s_{\mathsf{abc}}.\mathsf{INT}$, an initialization flag $s_{\mathsf{abc}}.\mathsf{IP}$, and a send flag $s_{\mathsf{abc}}.\mathsf{SF}$ and finally,
- the current slot number $s_{\mathsf{abc}}.\mathsf{CSN}$

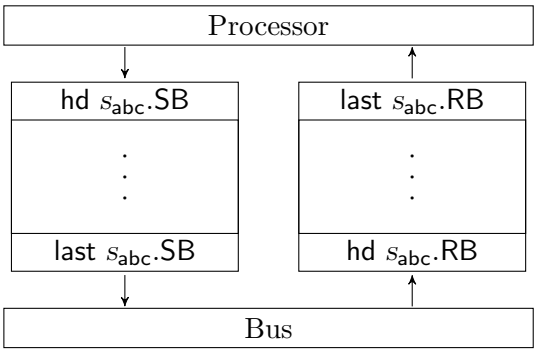

Figure 4.1: Send and Receive Buffer Model

The send and receive buffers are represented as lists of arbitrary length. Hence, we may use the model with an arbitrary number of receive and send buffers. Most notably, we can model the ABC with single as well as double buffers. Note that in contrast to the implementation, the specification does not have a parity bit. Instead, we model the swapping of the buffers by rotating their contents. Figure 4.1 on the preceding page depicts the send and receive buffer model. The processor-facing buffers are always addressed by hd s_{abc}.SB and last s_{abc}.RB. The bus, on the contrary, accesses the buffers with last s_{abc}.SB and hd s_{abc}.RB.

In the single buffer model, both expressions are obviously equal, i. e., hd s_{abc}.SB = last s_{abc}.SB and hd s_{abc}.RB = last s_{abc}.RB. Each buffer entry stores a message, which is itself represented as an integer list.

Configuration register	Description
s_{abc}.CR ! 0	number of slots per round (SLOTCOUNT)
s_{abc}.CR ! 1	size of messages in bytes (4 · MSGLENGTH)
s_{abc}.CR ! 2	an offset value the sender waits before sending
s_{abc}.CR ! 3	the slot length counted in hardware cycles
s_{abc}.CR ! 4	a wake-up value (used to generate interrupts within a slot)
s_{abc}.CR ! 5	an initial waiting value
s_{abc}.CR ! 6, s_{abc}.CR ! 7	lower and higher part of the SPT

Table 4.1: Content of ABC configuration registers

There are 8 configuration registers. Table 4.1 describes their content. We are particularly often referring to the sending permission SPT. Hence, we encapsulate the access of the i-th SPT-entry by function read_sendl s_{abc} i and define:

read_sendl s_{abc} $i \equiv$ SPT ! i

ABC Transitions. Before we formally define the transitions of the ABC device, we first give an abstract overview of the protocol that the device uses for the communication with the operating system and the external environment. Figure 4.2 on the next page illustrates the transitions within this protocol. Note that the diagram shows symbolic names though technically, the protocol states are encoded by three flags. Table 4.2 on the following page translates the symbolic names into the corresponding flag combination and vice versa. The *init* state resembles the initialization phase and is indicated by a raised initialization flag. Otherwise, we distinguish states from their send and interrupt flag as follows: In the *read* state the interrupt line is raised and the send flag disabled, whereas in the *write* state interrupt and send flag are enabled. In the idle states, the interrupt flag is not set and we distinguish from the $idle_{w}$ where the send flag is raised and $idle_{rd}$.

Recall that our device interacts with the processor, on the one hand, and the external environment, on the other hand (Section 2.7). Hence, we annotate flowing data with e for external environment and m for the processor. Inputs are denoted by $\downarrow$, and outputs by $\uparrow$ (i. e., $e\downarrow$ is shorthand for *eifi*, whereas $m\uparrow$ abbreviates *mifo*).

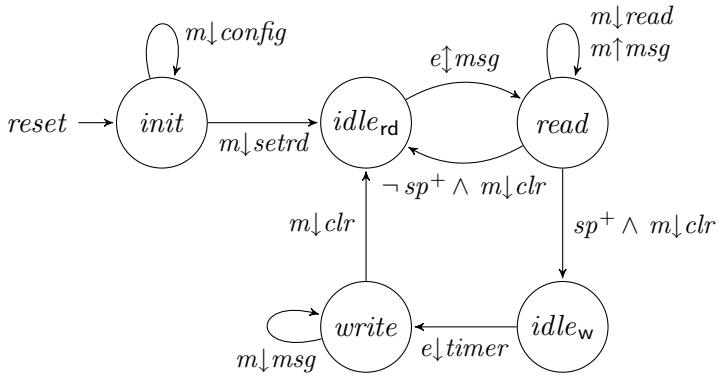

Figure 4.2: State diagram for the ABC

As described, we start in the *init* state, where OLOS consecutively writes the configuration (*config*) into the corresponding registers and finally issues the set-ready command (*setrd*).

When the ABC sees the set-ready command, the device switches into *running mode* and enters $idle_{\mathsf{rd}}$ and awaits an input from the bus ($e{\updownarrow}msg$). More specifically, we check whether the ECU has the send permission (*sp*) in the current slot. If so, the device is waiting for a timer event, then outputs the message from its bus-facing send buffer to the bus and writes the same message to its bus-facing receive buffer [1]. Otherwise, it awaits a message from the bus and stores it into its bus-facing receive buffer. We abbreviate "if *sp* then ($e{\downarrow}timer$, $e{\uparrow}msg$) else $e{\downarrow}msg$" by $e{\updownarrow}msg$. This abbreviation essentially encapsulates two possible transitions depending on the send permission.

When the receive buffer has been written, the device enters the *read* state. In this state the device expects a request from the processor ($m{\downarrow}read$) to read a message from the ABC's processor-facing receive buffer ($m{\uparrow}msg$). In analogy to the set-ready command, the processor acknowledges the successful reception by the command *clear interrupt* (*clr*). Depending on the send permission *in the next slot* (denoted by sp^+), the device either enters the read- or the write-idle state ($idle_{\mathsf{rd}}$ or $idle_{\mathsf{w}}$, respectively). The former

[1] Here we assume an ideal bus transmission where entire messages are sent and received without data losses

Symbolic names	s_{abc}.IP	s_{abc}.INT	s_{abc}.SF
init	True	-	-
read	False	True	False
write	False	True	True
$idle_{\mathsf{rd}}$	False	False	False
$idle_{\mathsf{w}}$	False	False	True

Table 4.2: Automaton states related to their flag combination

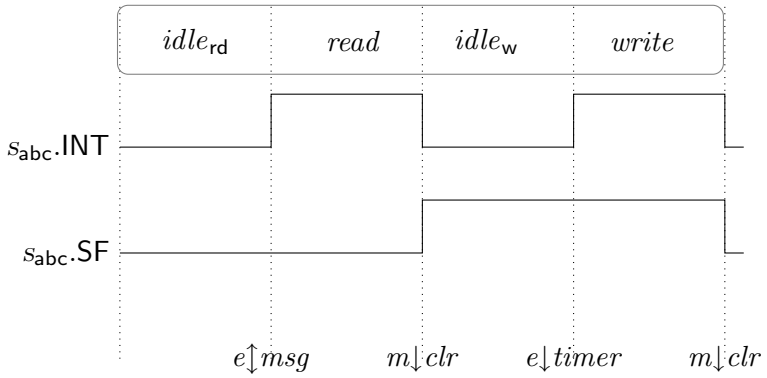

Figure 4.3: Timing diagram of ABC flags

has been discussed before and the latter analogously awaits the beginning of the send phase. This phase is deterministically entered after a configurably fixed time. We model the elapsed time as an external event $e\downarrow timer$.

With the external timer event, the device enters the *write* state in order to allow the processor writing a message ($m\downarrow msg$) into the processor-facing send buffer. When the transmission is finished, the device transfers back to the $idle_{rd}$ after getting a clear-interrupt command from the processor (like in the receive case). We can relate the *read* state to the receive phase of the ECU and the *write* state to its send phase, respectively. Depending on the sending permission of an ECU the ABC automaton is situated in one of the idle states during the Compute Phase. Finally, the device-communication phase is encapsulated in the transition $e\updownarrow msg$.

Figure 4.3 shows the interrupt flag INT and the send flag SF of an ABC state s_{abc} during one slot in a timing diagram. In this szenario we assume that the ABC has the send permission during that slot. This diagram impressively depicts the characteristic flag combination of each state in a certain chronological sequence.

After we have given an overview of the abstract ABC automaton, we will present the formal definitions of the ABC transition functions.

Several transitions in the automaton rely on inputs from the processor (denoted with $m\downarrow$). $m\downarrow setrd$ and $m\downarrow clrint$ are commands written from the processor to the command register of the device. These inputs force the device to change its state internally. The following predicates indicate that an input *mifi* from the processor is a *clearint* or a *setrd* command:

$$\begin{aligned}&\mathsf{is\_clearint}\ mifi \equiv\\ &\quad mifi.\mathsf{mifi\_din} = \mathsf{CLEARINT\_COM} \wedge mifi.\mathsf{mifi\_a} = \mathsf{COMR\_PORT} \wedge mifi.\mathsf{mifi\_wr}\end{aligned}$$

$$\begin{aligned}&\mathsf{is\_setrd}\ mifi \equiv\\ &\quad mifi.\mathsf{mifi\_din} = \mathsf{SETRD\_COM} \wedge mifi.\mathsf{mifi\_a} = \mathsf{COMR\_PORT} \wedge mifi.\mathsf{mifi\_wr}\end{aligned}$$

Both predicates hold when the write flag $mifi.\mathsf{mifi\_wr}$ is enabled, the port equals

the command port *mifi*.mifi_a = COMR_PORT and the incoming data resembles the individual command code (CLEARINT_COM respectively SETRD_COM).

Except the read access from the processor ($m{\downarrow}read$), all the other transitions with processor input manipulate the ABC state. These transitions are encapsulated in function mifi_write to compute a successor state from the current device state and a processor input. We distinguish between four cases:

- $m{\downarrow}msg$: **Processor writes a message into send buffer**
  The processor updates the visible ABC send buffer (i. e., hd s_{abc}.SB = s_{abc}.SB ! 0) with the given data when the port address refers to the send buffer.

- $m{\downarrow}clrint$: **Processor sends command** CLEARINT_COM
  This transition either occurs in the *write* or the *read* state. Both states differ in the value of the send flag s_{abc}.SF. In the *write* state (s_{abc}.SF), we simply disable the interrupt flag and the send flag to enter $idle_{rd}$. In the *read* state ($\neg$ s_{abc}.SF), the device clears the interrupt flag and sets the send flag to a new value corresponding to the entry in the SPT of the next slot. Afterwards, the state proceeds to one of the idle states.

- $m{\downarrow}setrd$: **Processor sends command** SETRD_COM
  Function mifi_write clears the init flag to indicate the termination of the initialization phase. Moreover, the send flag is reset to change to $idle_{rd}$. Finally, the current slot number is initialized with SLOTCOUNT − 1 such that the device starts with an incoming external event in slot 0 (the slot number is increased directly then).

- $m{\downarrow}config$: **Processor writes the configuration registers**
  The last branch of the function defines the initialization of the configuration registers. We require that the init flag is set and the port address refers to one of the registers.

Function mifi_write includes all the transitions mentioned above:

```
mifi_write mifi s_abc ≡
 let new_data = to_int32 mifi.mifi_din; addr = mifi.mifi_a
 in if SEND_PORT ≤ addr < RECV_PORT
      then s_abc
           (|SB := s_abc.SB
             [0 := (s_abc.SB ! 0)[addr − SEND_PORT := new_data]]|)
    elsif is_clearint mifi
      then if s_abc.SF then s_abc(|INT := False, SF := False|)
           else s_abc
                (|INT := False,
                 SF := read_sendl s_abc
                         ((s_abc.CSN + 1) mod s_abc.CR ! 0)|)
    elsif is_setrd mifi
      then s_abc(|IP := False, SF := False, CSN := SLOTCOUNT − 1|)
```

elsif s_{abc}.IP $\wedge$ CONFR_PORT $\leq$ *addr* $<$ COMR_PORT
then s_{abc}(|CR := s_{abc}.CR[*addr* − CONFR_PORT := *mifi*.mifi_din]|)
else s_{abc}

Finally, we present the internal transition function δ^m_{abc} of the ABC device that includes all processor communicating functions. The function takes the current ABC state s_{abc} and an input from the processor *mifi*. It returns a triple consisting of the successor state s'_{abc} and outputs to the processor *mifo* and the external environment *eifo*. We distinguish between read and write requests that are mutually exclusive. When the processor wants to read i. e., *mifi*.mifi_rd, the ABC state remains unchanged and the output to the processor is the data of the processor visible receive buffer (i. e., last s_{abc}.RB) at a certain port address. A raised write flag *mifi*.mifi_wr, on the contrary, changes the successor ABC state s'_{abc} by applying function mifi_write and always returns 0 to the processor. In both cases, we return an empty output to the external environment i. e., $\mathsf{eifo\_abc}_\perp$. The internal transition function δ^m_{abc} is defined as follows:

δ^m_{abc} *mifi* $s_{\mathsf{abc}} \equiv$
(**if** *mifi*.mifi_wr **then** mifi_write *mifi* s_{abc} **else** s_{abc},
if RECV_PORT $\leq$ *mifi*.mifi_a $<$ CONFR_PORT $\wedge$ *mifi*.mifi_rd
then to_nat32 (last s_{abc}.RB ! (*mifi*.mifi_a − RECV_PORT))
else 0,
$\mathsf{eifo\_abc}_\perp$)

Apart from the processor communication, the ABC device interacts with the external environment i. e., the bus. The device starts after the initialization phase in state $idle_{\mathsf{rd}}$ and expects an incoming external event. The incoming external event immediately initiates the start of a new slot. The entire device communication is hidden in transition $e\updownarrow msg$. We encapsulate all necessary computations into function abc_incoming_msg. It takes the current device state s_{abc} and a given message *msg* to return an updated ABC state. After the external event has triggered a new slot, the ABC device raises its interrupt flag and increases the actual slot number. The value of SLOTCOUNT is kept in a configuration register (Table 4.1 on page 77). Note that Olos increases its implementation variable of the current slot number in the receive phase i. e., after it detects the ABC's raised interrupt flag. This causes a delay between both current slot number representations. At the beginning of a new slot, we have to shift the buffer contents. For the send buffer, we use function rotate1 to model the buffer swapping. It simply shifts all elements left and appends the first element at the end of the list. The receive buffer, on the contrary, updates the first element with the message *msg* and appends the old buffer contents except the last element by using function butlast (we can imagine that all buffer contents are shifted right and the last one is dropped). abc_incoming_msg is formally defined as:

abc_incoming_msg *msg* $s_{\mathsf{abc}} \equiv$
s_{abc}
(|INT := True, CSN := (s_{abc}.CSN + 1) mod s_{abc}.CR ! 0,
RB := *msg* ⊙ butlast s_{abc}.RB, SB := rotate1 s_{abc}.SB|)

Now, we formalize the external transition function δ^e_{abc} of the ABC device. It takes the current ABC state s_{abc} and an input *eifi* from the external environment. The function returns a pair consisting of the successor state s'_{abc} and an output to the external environment *eifo*. In a certain device state, the automaton expects a restricted set of device inputs. Whenever the ABC gets an undefined input or an empty input $\mathsf{eifi\_abc}_\perp$, the transition function returns an unchanged state and an empty output to the bus: $(s_{\mathsf{abc}}, \mathsf{eifo\_abc}_\perp)$. Otherwise, we distinguish between incoming timer events or messages. We expect an incoming timer event in one of the idle states where the interrupt flags are disabled. The timer event in state $idle_{\mathsf{w}}$ signals the start of the send phase. Here, the transition function raises the interrupt flag and returns empty output value. The ABC device with the sending permission awaits a timer event in $idle_{\mathsf{rd}}$ signalling a slot boundary. Hence, we obtain a new ABC state by using function abc_incoming_msg and write the message of the bus-facing send buffer to the bus. Note that the bus-visible receive buffer of the sending ABC is updated directly with the message. Otherwise, all devices without sending permission in state $idle_{\mathsf{rd}}$ await an incoming message from the external environment. Moreover, the transferred message indicates the start of a new slot. In this case, the transition function applies function abc_incoming_msg to obtain the successor state s'_{abc} and returns an empty output to the bus. The external transition function of the ABC device δ^e_{abc} in Isabelle/HOL is defined as:

δ^e_{abc} *eifi* $s_{\mathsf{abc}} \equiv$
  case *eifi* **of** $\mathsf{eifi\_abc}_\perp \Rightarrow (s_{\mathsf{abc}}, \mathsf{eifo\_abc}_\perp)$
  | eifi_abc_timer ⇒
    if s_{abc}.INT **then** $(s_{\mathsf{abc}}, \mathsf{eifo\_abc}_\perp)$
    elsif s_{abc}.SF **then** $(s_{\mathsf{abc}}(\!|\mathsf{INT} := \mathsf{True}|\!), \mathsf{eifo\_abc}_\perp)$
    else (abc_incoming_msg (last s_{abc}.SB) s_{abc}, $\lfloor$last s_{abc}.SB$\rfloor$)
  | eifi_abc_msg *msg* ⇒
    if s_{abc}.INT ∨ s_{abc}.SF **then** $(s_{\mathsf{abc}}, \mathsf{eifo\_abc}_\perp)$
    else (abc_incoming_msg *msg* s_{abc}, $\mathsf{eifo\_abc}_\perp$)

Predicates on ABC States. We regard states of the ABC device at different points in time. The constraints the device should fulfill in the *init* state, directly after the configuration from the operating system and after all device transitions are specified in the following predicates.

The validity of messages given as integer lists is encapsulated in a predicate called is_valid_intlistMsg. This predicate takes a message as argument and holds when the length equals MSGLENGTH and all message values are signed 32-bit integers. Formally:

is_valid_intlistMsg $msg \equiv$
  $|msg| = \mathsf{MSGLENGTH} \wedge (\forall x{<}\mathsf{MSGLENGTH}.\ \mathsf{asm\_int}\ (msg\ !\ x))$

Predicate valid_ABC_buffers holds if an ABC state s_{abc} has valid ABC buffers. It includes requirements of the buffer lengths i. e., the ABC device should provide at least one send and receive buffer, whereas the number of configuration registers is fixed with

8. Moreover, the send and receive buffers contain valid data with fixed length. For a given ABC state predicate valid_ABC_buffers holds

valid_ABC_buffers s_{abc} ≡
  $1 \leq |s_{abc}.SB| \wedge$
  $1 \leq |s_{abc}.RB| \wedge$
  $|s_{abc}.CR| = 8 \wedge$
  $(\forall i{<}|s_{abc}.SB|.$ is_valid_intlistMsg $(s_{abc}.SB\ !\ i)) \wedge$
  $(\forall i{<}|s_{abc}.RB|.$ is_valid_intlistMsg $(s_{abc}.RB\ !\ i))$

After reset, the ABC is in the *init* state where it awaits the processor to configure its registers. At that point of time, we require that the ABC device has valid buffers, the init flag s_{abc}.IP signals that the device is in the initialization phase and the interrupt flag s_{abc}.INT is initially disabled. All constraints that should hold directly after power-up are collected in predicate abc_before_olosinit:

abc_before_olosinit s_{abc} ≡
  valid_ABC_buffers $s_{abc} \wedge \neg\ s_{abc}.INT \wedge s_{abc}.IP$

After the initialization phase we demand for validity of the ABC device. This includes, besides the validity of ABC buffers that the configuration registers have been set to the correct values. More specifically, the designated registers should hold the correct number of slots per round, the message length and the sending permission table. Furthermore, the device is in the running mode that is indicated by a disabled initialization flag s_{abc}.IP. We summarize all these requirements in predicate is_valid_ABC. Formally:

is_valid_ABC s_{abc} ≡
  $s_{abc}.CR\ !\ 0 = SLOTCOUNT \wedge$
  $s_{abc}.CR\ !\ 1\ div\ 4 = MSGLENGTH \wedge$
  map (read_sendl s_{abc}) [0..<SLOTCOUNT] = SPT $\wedge$
  valid_ABC_buffers $s_{abc} \wedge \neg\ s_{abc}.IP$

Directly after the operating system configured the ABC device, we assume that a device state s_{abc} is initial. An initial state has to fulfill the validity constraints. Furthermore, the ABC device should start with slot number 0. Directly after the initialization, the device awaits an incoming event from the external environment to start the first slot. The corresponding transition function increases the current slot number, hence the initial slot number after power-up is set to SLOTCOUNT − 1. Finally, the flags have to be set up correctly after initialization. More specifically, all flags are disabled ($\neg\ s_{abc}.IP \wedge \neg\ s_{abc}.SF \wedge \neg\ s_{abc}.INT$) to start from state $idle_{rd}$. All constraints are encapsulated into predicate is_initial_ABC that is satisfied when an ABC state s_{abc} is initial. Formally:

is_initial_ABC s_{abc} ≡
  is_valid_ABC $s_{abc} \wedge$
  $s_{abc}.CSN = SLOTCOUNT - 1 \wedge \neg\ s_{abc}.INT \wedge \neg\ s_{abc}.SF$

We require that validity holds after the initialization phase and is preserved under internal and external ABC-transitions.

Lemma 4.1 (δ^m_{abc} preserves State Validity)**.** *From a given valid ABC device state* is_valid_ABC s_{abc} *that receives a wellformed input from the processor in case of a write request (i. e.,* $mifi$.mifi_wr $\longrightarrow$ asm_nat $mifi$.mifi_din*), we can conclude that an internal ABC transition preserves validity. Formally*

is_valid_ABC (fst (δ^m_{abc} *mifi* s_{abc}))

Proof. The internal ABC-transition distinguishes between read or write requests from the processor. In the first case the ABC state is only accessed but not modified. Hence, validity holds obviously. Otherwise, function mifi_write manipulates the ABC state. We distinguish between the four possible cases:

1. Writing data into the send buffer:
   In this case the configuration registers, the initialization flag and the receive buffers remain unchanged. Hence, we have to prove that the send buffers are valid. The buffer length does not change after a write access. From the assumption we know that the written data is an unsigned 32-bit integer. Function *natwd_to_intwd* converts this value to a signed 32-bit integer. Hence, the buffer is valid after writing data into the send buffer.
2. Clearing the interrupt flag:
   The *clearint* command modifies the values of the interrupt and the send flag. Thus, all constraints on buffers and the initialization flag are still satisfied.
3. Change into running mode:
   In case of a *setrd* command, the current slot number the interrupt and the initialization flag are set. Hence, buffer validity is preserved and the configuration registers remain unchanged. The value of the initialization flag is overwritten with False, thus all validity conditions hold.
4. Initialize the configuration registers:
   We can drop this case, since the initialization of the configuration registers is only possible in the initialization mode. A valid ABC state s_{abc} is always in running mode (i. e., the initialization flag is cleared $\neg\ s_{\mathsf{abc}}$.IP).

□

Lemma 4.2 (δ^e_{abc} preserves State Validity)**.** *We assume that messages coming from the external environment are always valid (i. e.,* $\forall msg.\ eifi =$ eifi_abc_msg $msg \longrightarrow$ is_valid_intlistMsg msg*) Then, for a valid ABC device state* s_{abc} *validity is preserved under an external ABC transition. Formally:*

is_valid_ABC (fst (δ^e_{abc} *eifi* s_{abc}))

Proof. We prove this lemma by case distinction over the external input *eifi*:

1. $eifi =$ eifi_abc$_\perp$:
   The state is not modified and obviously validity still holds.

2. *eifi* = eifi_abc_timer:
   In the case that the interrupt flag is already set the state is unchanged. Otherwise, when the send flag is set, only the interrupt flag is raised. Hence, all validity constraints are still satisfied. When neither the interrupt flag nor the send flag where set, the state is updated with function abc_incoming_msg where the message is the content of the bus-facing send buffer. The conditions on the configuration registers and the initialization flag obviously hold. Validity of the buffers is preserved since the buffer lengths are not changed and rotating or shifting valid values within a buffer still satisfy the requirements.
3. *eifi* = eifi_abc_msg *msg*:
   In the case that either the interrupt or the send flag are set the state is not manipulated. Otherwise, the receive buffer is updated by calling function abc_incoming_msg with the incoming message value *msg*. Setting the interrupt flag and the current slot number and rotating valid buffers do not violate the requirements of state validity. From the assumption that all incoming messages are valid, we conclude that the receive buffers are valid after receiving a message.

□

4.2 Application Abstraction

Before we specify the behaviour of the whole ECU, we draw the readers attention to the interaction between applications and the operating system Olos. We have learned from Section 3.3.5 that applications use system calls in order to request services from the operating system. Their effect is not only the manipulation of application states but they may also modify the state of the operating system.

As we have seen in Section 3.4, system calls are implemented with portions of inline assembly code. This code is hidden in a library of C0 functions. Accordingly, we would like to abstract the specification of system calls as C0 machines. Daum et al. [DDWS08, DDB08] specified a clean interface between their operating system Vamos and their processes. Therefore, they encapsulated processes as self-contained automata and presented an abstract process model. This definition separates the concepts of the interface between an operating system and its processes from the actual specification of a system call.

We adapt this idea and slightly change it for our purposes. Hence, we present an abstract application model and instantiate this model for assembly and C0 processes. Finally, we prove the validity of the C0 application specification and the Vamp assembly counterpart.

Thus, we are able to regard C0 and assembler applications independently and prove a simulation theorem that relates both models separated from the operating system.

4.2.1 Application Model

In this subsection, we introduce an abstract application model that allows a clear separation between the operating system and the applications.

Outputs Ω_{app}	Inputs Σ_{app}
ε_Ω (no call to a primitive)	ε_Σ (internal step)
SendMsg *msgval msgnum*	SendSuccess InvalidMsgNr
RecvMsg *msgnum*	RecvSuccess *msgval* InvalidMsgNr
ExFinish	FinishSuccess
InvPtrErr	InvPtrResponse
StuckErr	—
Undefined_Trap	Continue

Table 4.3: Interface between Olos and its applications

Definition 4.1 (Application Model)**.** The process model is an input-output automaton $\mathcal{A}_{\mathsf{app}}$ given by a tuple
$(\mathcal{S}_{\mathsf{app}}, \Omega_{\mathsf{app}}, \Sigma_{\mathsf{app}}, \omega_{\mathsf{app}}, \delta_{\mathsf{app}}, \mathsf{is\_valid}_{\mathsf{app}}, \mathsf{is\_init}_{\mathsf{app}}, \mathsf{size}_{\mathsf{app}})$ where

- $\mathcal{S}_{\mathsf{app}}$ is the state,
- Ω_{app} is the output alphabet,
- Σ_{app} is the input alphabet,
- ω_{app} is an output function,
- δ_{app} is the transition function,
- $\mathsf{is\_valid}_{\mathsf{app}}$ is a predicate determining wheter an applications state is valid,
- $\mathsf{is\_init}_{\mathsf{app}}$ is a predicate indicating that an application state is initial and,
- $\mathsf{size}_{\mathsf{app}}$ is a function computing the size of an application

The state $\mathcal{S}_{\mathsf{app}}$ depends on the individual programming language that is in our case, C0 or Vamp assembly. The interface between the applications and the operating system is defined by Σ_{app} and Ω_{app} and is shared by all application models. Table 4.3 presents both alphabets, the output alphabet (an application request) together with the corresponding responses from Olos (inputs).

Output Alphabet. The output alphabet Ω_{app} comprises all possible system calls, the inquiry of an 'ordinary' local step and some error conditions. The output ε_Ω denotes the intention to perform a local computation in the next step. SendMsg *msgval msgnum*, RecvMsg *msgnum* and ExFinish are outputs in order to call the operating system (system calls). The message transferring system calls take two arguments which are stored in registers. However, these values may not have meaningful interpretations. We have e. g., only a bounded number of message buffers so that the message number may be out of range. The operating system takes care of the individual handling of an incoming *msgnum*. Before the application inquires communication with the operating system it checks whether conditions of a successful transfer are given (i. e., the start address is valid and there is enough memory space to store a whole message). If this is not the case, the application indicates this with output InvPtrErr. Otherwise, it

requests a system call service with the corresponding output. Whenever an application specifies a system call number that is not associated with any of our calls, this is signalled with the output UNDEFINED_TRAP. At least, an application could generate exceptions like illegal page faults or overflows. These exceptions throw the output STUCKERR. VAMP assembly applications are quite robust since they can handle invalid pointers or meaningless system call numbers. C0 programs, in contrast, immediately return an error state when the execution of a transition fails. This state cannot be left any more. We summarize all undesired outputs into predicate is_runtime_error:

$$\mathsf{is\_runtime\_error}\ i \equiv i = \textsc{StuckErr} \vee i = \textsc{Undefined\_Trap} \vee i = \textsc{InvPtrErr}$$

All outputs are summarized in the left row in Table 4.3 on the facing page.

Input Alphabet. The input alphabet Σ_{app} reflects all possible responses of the operating system reacting to the given outputs. Therefore, OLOS passes different inputs to the transition function δ_{app}. In order to perform a local computation, the transition function δ_{app} gets the input ε_{Σ}. Note that local computations depend on the individual state. The inputs SENDSUCCESS, RECVSUCCESS *msgval* and FINISHSUCCESS are the corresponding responses of the operating system to the kernel calls in case of success. The application outputs SENDMSG and RECVMSG use parameter *msgnum* which may be a value out of range. When applications request to exchange messages with an undefined message buffer , OLOS signals this error with input INVALIDMSGNR. Similarly, the operating system replies to an output INVPTRERR with the corresponding input INVPTRRESPONSE. The output STUCKERR has no corresponding input since the operating system does not change a stuck application state. Finally, we pass CONTINUE to the transition function δ_{app} whenever the application continues with the next instruction.

Valid OLOS Reponses. In the last paragraph we noticed that our operating system OLOS responds to an incoming inquiry only with one of the specified inputs. Table 4.3 on the preceding page emphasizes this strong correlation between outputs and inputs. Formally, we collect the matching output-input pairs $(\omega_{\mathsf{app}}\ s_{\mathsf{app}}, i)$ in a set olos_responses:

(SENDMSG *msgval msgnumb*, SENDSUCCESS) ∈ olos_responses
(SENDMSG *msgval msgnumb*, INVALIDMSGNR) ∈ olos_responses
is_valid_intlistMsg *msgval* ⟹
(RECVMSG *msgnumb*, RECVSUCCESS *msgval*) ∈ olos_responses
(RECVMSG *msgnumb*, INVALIDMSGNR) ∈ olos_responses
(EXFINISH, FINISHSUCCESS) ∈ olos_responses
(INVPTRERR, INVPTRRESPONSE) ∈ olos_responses
(UNDEFINED_TRAP, CONTINUE) ∈ olos_responses
$(\varepsilon_{\Omega}, \varepsilon_{\Sigma})$ ∈ olos_responses

Result Values. The responses of the operating system to the calling application additionally determine the result value that is written to the result register or variable of

the application. When the input indicates a successful execution of the system call, the result value is 0. If one of the system call parameters is invalid, this value is set to a corresponding error code.

result_value_int $\sigma \equiv$
  case σ **of** InvPtrResponse $\Rightarrow$ INVALID_POINTER
  | InvalidMsgNr $\Rightarrow$ INVALID_MSGNUM
  | _ $\Rightarrow$ 0

Predicates. The validity predicate $\mathsf{is\_valid}_{\mathsf{app}}$ includes constraints for the wellformedness of the particular state depending on $\mathcal{S}_{\mathsf{app}}$,. When a C0 program is started, it is compiled by the verfied C0 compiler and saved in a binary executable file which stores the initial memory content of code and data of the corresponding assembly program. The operating system initializes the registers of the application and sets its memory to the specified memory image. While the image uniquely describes the initial assembly state, the original C0 program cannot be reconstructed since the variable names are not preserved in the compiled code. Thus, the generic definition of application models uses an initialization predicate $\mathsf{is\_init}_{\mathsf{app}}$ that relates initial application states and memory images. Furthermore, function $\mathsf{size}_{\mathsf{app}}$ computes the amount of virtual memory that is used by an application.

Valid Applications. So far, we have described the intuition of application automata which interact with the operating system. We restrict these models to a special class of automata by defining a set of rules. These rules form the specification of a valid application and have to be discharged within each individual instantiation of an abstract model. In the following paragraph, we present all necessary rules that specify valid applications.

The first rule (valid_success) assumes a valid application state together with an output requesting to send a message. We require that the successor state of the application is valid after a successful execution of the corresponding system call.

$$\frac{\mathsf{is\_valid}_{\mathsf{app}}\ s_{\mathsf{app}} \qquad \omega_{\mathsf{app}}\ s_{\mathsf{app}} = \textsc{SendMsg}\ msgval\ msgnum}{\mathsf{is\_valid}_{\mathsf{app}}\ (\delta_{\mathsf{app}}\ \textsc{SendSuccess}\ s_{\mathsf{app}})} \quad \text{(valid\_success)}$$

Additionally, the same assumptions should ensure that the message sent to the operating system is valid. The wellformedness constraints of a message include a fixed message length and the condition that all message values are signed 32-bit integers. (valid_success_msg) encapsulates this requirement:

$$\frac{\mathsf{is\_valid}_{\mathsf{app}}\ s_{\mathsf{app}} \qquad \omega_{\mathsf{app}}\ s_{\mathsf{app}} = \textsc{SendMsg}\ msgval\ msgnum}{\mathsf{is\_valid\_intlistMsg}\ msgval} \quad \text{(valid\_success\_msg)}$$

When a valid application requests to receive a message from the operating system, we demand additionally that the message is well-formed. Then, the successful execution of the corresponding system call leads to a valid successor state:

$$\frac{\mathsf{is\_valid}_{\mathsf{app}}\ s_{\mathsf{app}} \qquad \omega_{\mathsf{app}}\ s_{\mathsf{app}} = \textsc{RecvMsg}\ \mathit{msgnum} \qquad \mathsf{is\_valid\_intlistMsg}\ \mathit{msgval}}{\mathsf{is\_valid}_{\mathsf{app}}\ (\delta_{\mathsf{app}}\ (\textsc{RecvSuccess}\ \mathit{msgval})\ s_{\mathsf{app}})} \quad \text{(valid\_succ\_recv)}$$

Rule (valid_succ_finish) postulates that validity of an application is preserved after the application signals its termination ($\omega_{\mathsf{app}}\ s_{\mathsf{app}} = \textsc{ExFinish}$) and the kernel reacts with the corresponding transition δ_{app} FinishSuccess s_{app}.

$$\frac{\mathsf{is\_valid}_{\mathsf{app}}\ s_{\mathsf{app}} \qquad \omega_{\mathsf{app}}\ s_{\mathsf{app}} = \textsc{ExFinish}}{\mathsf{is\_valid}_{\mathsf{app}}\ (\delta_{\mathsf{app}}\ \textsc{FinishSuccess}\ s_{\mathsf{app}})} \quad \text{(valid\_succ\_finish)}$$

An inquiry of an application for a message exchange with the operating system always includes paramenter *msgnum*. Whenever the application tries to access an unspecified message buffer, the operating system calls the transition function with the input InvalidMsgNr. Assuming a valid application state and a request for kernel communication, we can conclude that the successor state remains valid.

$$\frac{\mathsf{is\_valid}_{\mathsf{app}}\ s \qquad \omega_{\mathsf{app}}\ s = \textsc{SendMsg}\ \mathit{msgval}\ \mathit{msgnum} \vee \omega_{\mathsf{app}}\ s = \textsc{RecvMsg}\ \mathit{msgnum}}{\mathsf{is\_valid}_{\mathsf{app}}\ (\delta_{\mathsf{app}}\ \textsc{InvalidMsgNr}\ s)} \quad \text{(valid\_inv\_msgnum)}$$

Furthermore, validity is preserved after the operating system executes the transition function δ_{app} InvPtrResponse. We define this constraint in rule (valid_inv_pointer):

$$\frac{\mathsf{is\_valid}_{\mathsf{app}}\ s}{\mathsf{is\_valid}_{\mathsf{app}}\ (\delta_{\mathsf{app}}\ \textsc{InvPtrResponse}\ s)} \quad \text{(valid\_inv\_pointer)}$$

Even though the operating system ignores a request from a valid application and continues with the next instruction (Continue) the successor state remains well-formed.

$$\frac{\mathsf{is\_valid}_{\mathsf{app}}\ s}{\mathsf{is\_valid}_{\mathsf{app}}\ (\delta_{\mathsf{app}}\ \textsc{Continue}\ s)} \quad \text{(valid\_continue)}$$

A valid application state that intends to perform a local step (signalled with output ε_Ω) remains valid after executing the transition function of the underlying individual model. This constraint is formulated below:

$$\frac{\mathsf{is\_valid}_{\mathsf{app}}\ s \qquad \omega_{\mathsf{app}}\ s = \varepsilon_\Omega}{\mathsf{is\_valid}_{\mathsf{app}}\ (\delta_{\mathsf{app}}\ \varepsilon_\Sigma\ s)} \quad \text{(valid\_step)}$$

We assume to have an initial application state and moreover, all values of the image *img* are signed 32-bit integers and the number of allocated pages is bounded by TVM_MAXPAGES. Note that we forbid applications with no allocated memory. Then, the initial application state fulfills the validity constraints.

$$\frac{\mathsf{is\_init}_{\mathsf{app}}\ img\ pages\ s \qquad \forall ad{<}|img|.\ \mathsf{asm\_int}\ (img\ !\ ad) \qquad 0 < pages \qquad pages \leq \mathsf{TVM\_MAXPAGES}}{\mathsf{is\_valid}_{\mathsf{app}}\ s} \tag{valid_initial}$$

From an initial application state we can conclude that the amount of allocated virtual memory equals the number of pages *pages*. This is formalized in (init_app_size):

$$\frac{\mathsf{is\_init}_{\mathsf{app}}\ img\ pages\ s}{\mathsf{size}_{\mathsf{app}}\ s = pages} \tag{init_app_size}$$

Finally, we assume a valid application state that does not signal a runtime error. Moreover, the successor state after the transition function with the corresponding input also does not notify a runtime error. Then, the program size remains unchanged after all valid transitions.

$$\frac{\mathsf{is\_valid}_{\mathsf{app}}\ s \qquad \neg\ \mathsf{is\_runtime\_error}\ (\omega_{\mathsf{app}}\ s) \qquad \neg\ \mathsf{is\_runtime\_error}\ (\omega_{\mathsf{app}}\ (\delta_{\mathsf{app}}\ \sigma\ s))}{\mathsf{size}_{\mathsf{app}}\ (\delta_{\mathsf{app}}\ \sigma\ s) = \mathsf{size}_{\mathsf{app}}\ s} \tag{delta_app_size}$$

This concept allows to clearly separate the tasks of the operating system and applications and we may use either assembly or C0 machines. Note that there exist error cases where C0 and assembly applications react in different manners. In case of an invalid message pointer, assembly applications simply return an error code to the application whereas C0 programs lead to a runtime error. The same can be said for the call of undefined functions. Whereas, assembly applications would continue with the next instruction, C0 programs give up processing. Therefore, we consider lateron only computations that do not lead to runtime errors in C0. Then, we can guarantee that all instructions of a C0 program can be simulated by a sequence of assembly executions.

4.2.2 Assembly Applications

Instantiating the abstract process model with assembly processes extends the semantics of the sequential VAMP assembly language. Whereas in the sequential fragment of this language the TRAP instruction is simply illegal, we use this instruction for the communication with the operating system.

The state of assembly processes is identical to the formal semantics of VAMP assembly presented in Section 2.2. In this subsection, we define the output and transition functions as well as the predicates for assembly states.

Output Function. Before we present the output function ω_{app}, we first introduce some helpful definitions. In case that the application requests message exchange with the operating system, it has to check whether the start address is divisible by 4 and whether

the entire message fits into the memory that is bounded by mm_size d · PAGE_SIZE. This information is stored in the predicate valid_sa:

valid_sa *arg* $s_{\mathsf{asm}} \equiv 4$ dvd *arg* $\wedge$ *arg* $+ 4 \cdot$ MSGLENGTH $<$ mm_size $s_{\mathsf{asm}} \cdot$ PAGE_SIZE

In order to communicate with the operating system the application uses the TRAP instruction with an immediate constant i to distinguish between different requests. For a given TRAP instruction the output function ωtrap computes the corresponding output. *olosExFinished* is called with trap number 0, hence we return EXFINISH. The system calls exchanging messages with OLOS have two parameters: a message pointer to the start address in the application memory and the index of the message buffer that is involved in the message transfer. For convention these parameters are stored in the assembly machine in the general-purpose registers 11 and 12, whereas general-purpose register 22 always keeps the return value. The message transferring calls *olosSendMsg* and *olosRecvMsg* rely on the correctness of the start address that is stored in the register 11: If this value is not valid the function returns INVPTRERR. Note that the validity of the message number is handled by the kernel. Otherwise, we distinguish between a successful send request ($i = 1$) and a successful receive request ($i = 2$) from the application. The first one returns SENDMSG together with the message value (extracted from the application memory with a given start address and the message length) and the message number. The output of the latter request is RECVMSG with the message number. Every other immediate constant specifies a value that is not associated with any call of our operating system. Then, the function returns output UNDEFINED_TRAP. Formally:

```
current_instr s_asm = TRAP i ⟹
ωtrap s_asm ≡
 if i = 0 then ExFinish
 elsif i = 1
      then if ¬ valid_sa (to_nat32 (s_asm.gprs ! 11)) s_asm
             then InvPtrErr
           else let msg =
                      map s_asm.mm
                       [to_nat32 (s_asm.gprs ! 11) div 4..<
                        to_nat32 (s_asm.gprs ! 11) div 4 + MSGLENGTH]
                in SendMsg msg (to_nat32 (s_asm.gprs ! 12))
 elsif i = 2
      then if ¬ valid_sa (to_nat32 (s_asm.gprs ! 11)) s_asm
             then InvPtrErr
           else RecvMsg (to_nat32 (s_asm.gprs ! 12))
 else Undefined_Trap
```

The overall output function for assembly applications ω_{app} takes an assembly state s_{asm} to return the corresponding output. An instruction may generate exceptions like illegal instruction is_illegal_instr, misalignments (is_misaligned_data,is_misaligned_pc) or page

faults (is_outranged_pc, is_outranged_data). If such an exception occurs the function returns output StuckErr. Otherwise, we distinguish the current instruction between TRAP or other assembly instructions. The former uses function ωtrap to determine the output, the latter returns ε_Ω. With the formalizations from above we define the overall output function for assembly applications ω_{app} as follows:

$\omega_{app}\ s_{asm} \equiv$
 if is_illegal_instr s_{asm} $\vee$
  is_misaligned_pc s_{asm} $\vee$
  is_misaligned_data s_{asm} $\vee$
  is_outranged_pc s_{asm} $\vee$ is_outranged_data s_{asm}
  then StuckErr
 else case current_instr s_{asm} **of**
  TRAP $i \Rightarrow$ ωtrap s_{asm} | _ $\Rightarrow$ ε_Ω

Transition Function. The transition function for assembly applications δ_{app} returns a successor application state depending on the current application state s_{asm} and the kernel response. The effects of TRAP instructions depend on their immediate constants: there are three possible distinct modifications on application states. Hence, we formalize three seperate functions to encapsulate these independent state updates:

1. **Writing a message into the application memory.**
   Function write_msg specifies the changes of an application that receives a message from the kernel. It takes the current application state s_{asm} and the message value *msg* to update the memory. Register s_{asm}.gprs ! 11 keeps the start address of the memory region where the message is stored. Formally:

   write_msg s_{asm} *msg* $\equiv$
    let *sa* = to_nat32 (s_{asm}.gprs ! 11) div 4
    in s_{asm}(|mm := λi. **if** $i < sa \vee sa + |msg| \leq i$ **then** s_{asm}.mm i **else** *msg* ! $(i - sa)$|)

2. **Updating the result register.**
   Whenever the application inquires a message exchange with the operating system, the result register 22 will be updated with a result value. Recall that result_value_int σ depends on the input σ. Function write_result computes from a given application state and an input σ a state with an updated result register:

   write_result σ s_{asm} $\equiv$ s_{asm}(|gprs := s_{asm}.gprs[22 := result_value_int σ]|)

3. **Increasing the program counters.**
   Finally, a TRAP instruction increases the program counters of an application in such a way that the computation proceeds with the next instruction. Function inc_pcs encapsulates the modification of the program counters and is formally defined as follows:

$$\mathsf{inc\_pcs}\ s_{\mathsf{asm}} \equiv s_{\mathsf{asm}}(\!|\mathsf{dpc} := s_{\mathsf{asm}}.\mathsf{pcp},\ \mathsf{pcp} := (s_{\mathsf{asm}}.\mathsf{pcp} + 4)\ \mathsf{mod}\ 2^{32}|\!)$$

The transition function δ_{app} is specified by a case distinction over all possible application inputs σ. There are two cases where simply the program counters of the application are increased. Then, the application simply continues with the next instruction. This happens after the system call *olosExFinished* ($\sigma =$ FinishSuccess) or in the case of an undefined trap number ($\sigma =$ Undefined_Trap). The successor state of an application has an updated result register and increased program counters in case of a successful send request SendSuccess or when a message transfer system call had invalid arguments ($\sigma =$ InvPtrResponse $\vee$ $\sigma =$ InvalidMsgNr). When the application successfully received a message from Olos, its memory is changed additionally to the result register and the program counters. Finally, a local step in the sequential assembly semantics is performed in case of an empty input ε_Σ of the kernel.

```
δapp σ sasm ≡
 case σ of
 SendSuccess ⇒
  inc_pcs (write_result SendSuccess sasm)
 | RecvSuccess val ⇒
    inc_pcs
     (write_result (RecvSuccess val)
       (write_msg sasm val))
 | InvPtrResponse ⇒
    inc_pcs (write_result InvPtrResponse sasm)
 | InvalidMsgNr ⇒
    inc_pcs (write_result InvalidMsgNr sasm)
 | εΣ ⇒ δasm sasm | _ ⇒ inc_pcs sasm
```

Predicates. The validity predicate $\mathsf{is\_valid}_{\mathsf{app}}$ constrains a state-space type of an application model to the well-formed states. For the assembly semantics, these constraints are defined in the predicate is_valid_asm. Furthermore, we require two more properties for assembly applications. The system is in user mode i. e., $\neg$ is_system_mode s_{asm}. Moreover, the application has either no allocated memory (i. e., its page table length equals -1) or the number of allocated pages is less than TVM_MAXPAGES. Formally, we define:

$$\mathsf{is\_valid}_{\mathsf{app}}\ s_{\mathsf{asm}} \equiv \mathsf{is\_valid\_asm}\ s_{\mathsf{asm}} \wedge \neg\ \mathsf{is\_system\_mode}\ s_{\mathsf{asm}} \wedge$$
$$-1 \leq s_{\mathsf{asm}}.\mathsf{sprs}\ !\ \mathsf{PTL} < \mathsf{int}\ \mathsf{TVM\_MAXPAGES}$$

After power-up, the initial state of an assembly process is uniquely determined by the memory image *img* on the one hand and the application size in pages on the other hand (*pages*). The definition of a function that computes the initial state with these parameters is given below:

$$\begin{aligned}&\mathsf{init\_asm}\ \mathit{img}\ \mathit{pages} \equiv\\ &\quad (\!|\mathsf{dpc} = 0,\ \mathsf{pcp} = 4,\ \mathsf{gprs} = \mathsf{replicate}\ 32\ 0,\\ &\quad\ \mathsf{sprs} = \mathsf{replicate}\ 32\ 0[\mathsf{PTL} := \mathsf{to\_int32}\ \mathit{pages} - 1,\ \mathsf{MODE} := 1],\\ &\quad\ \mathsf{mm} = \lambda \mathit{ad}.\ \textbf{if}\ \mathit{ad} < |\mathit{img}|\ \textbf{then}\ \mathit{img}\ !\ \mathit{ad}\ \textbf{else}\ 0\ |\!)\end{aligned}$$

The function init_asm sets the program counters to the start addresses in the memory. The mode register MODE is set to 1 indicating that the application runs in user mode. The register PTL holds the intended size *pages* of the application, so that $\mathsf{size}_{\mathsf{app}}$ (init_asm *img pgs*) = *pgs*. All other registers are set to 0. The image is written at the beginning of the main memory, the remaining memory cells are filled with 0.

Now, we formalize the initialization predicate for assembly applications by demanding that the current state has to equal the initial one:

$$\mathsf{is\_init}_{\mathsf{app}}\ \mathit{img}\ \mathit{pgs}\ s_{\mathsf{asm}} \equiv s_{\mathsf{asm}} = \mathsf{init\_asm}\ \mathit{img}\ \mathit{pgs}$$

Application Size. The application size $\mathsf{size}_{\mathsf{app}}$ is simply specified as:

$$\mathsf{size}_{\mathsf{app}}\ s_{\mathsf{asm}} \equiv \mathsf{mm\_size}\ s_{\mathsf{asm}}$$

Validity. All the definitions presented previously specify a model for assembly applications. We now prove that this specification is valid, indeed.

Theorem 4.3 (Validity of the Assembly Application Model)**.** *The presented assembly application model is valid.*

Proof. We formally show that the assembly application model fulfills all 11 validity rules presented in Section 4.2.1. The successor states of transitions with the input σ = FinishSuccess or σ = Continue simply differ in their program counters. According to the definition of valid assembly states these have to be 32-bit naturals. After increasing the program counters, they remain 32-bit naturals. Hence, axiom (valid_succ_finish) and axiom (valid_continue) hold. Transition functions that increase the program counters and additionally update the result register of an application have the inputs σ = SendSuccess, σ = InvalidMsgNr or σ = InvPtrResponse. We have learned that the increasing of the program counters does not harm the validity predicate of assembly applications. The remaining requirements concern the general purpose registers: We know that an update does not change the length of the register file but its content. All result values are 32-bit integers so that an update does not violate validity. Thus, we showed the rules (valid_success), (valid_inv_msgnum) and (valid_inv_pointer). Axiom (valid_success_msg) assumes that the application requests to send a message to the operating system. According to the output definition this message is stored in the main memory of the application. The argument holding the message extracted from the main memory has the required length and all message values are 32-bit integers. Receiving a message which fulfills validity constraints preserves the wellformedness of the successor state according to the considerations we made before (axiom (valid_succ_recv)). Axiom (valid_step) holds by applying a lemma of Alexandra Tsyban stating that validity is preserved under all assembly transitions. Due to the assumptions on the parameters *pages*

and *img*, we can conclude (valid_initial) directly by examining the definition of *asm_init*. Finally, axiom (init_app_size) follows directly from the definition of function $\mathsf{size}_{\mathsf{app}}$ and axiom (delta_app_size) holds naturally since neither our operating system nor the application itself changes the register PTL that stores the memory size. □

4.2.3 C0 Applications

C0 applications extend the sequential C0 language. Recall that we introduced a special library including functions to permit communication between an application and the operating system. In this subsection, we describe in more detail how the model of a C0 application is defined.

C0 States. In contrast to assembly applications, we cannot reuse the state of the sequential C0 semantics. The sequential transition function is partial i. e., the transition function δ_{C0} returns $\bot$ in case of a programming error like null-pointer dereferencing. Note that this is also the case for δ_{C0}. In order to keep track of such errors, we extend the state of the sequential semantics by $\bot$. In the remainder, we use S_{C0} for a C0 application state that is either empty or consists of a non-empty C0 monolithic state $\lfloor s_{\mathsf{C0}} \rfloor$. As we defined in Section 2.3, the sequential program state is kept in a record component $s_{\mathsf{C0}}.\mathsf{pstate}$ that includes the current memory state $s_{\mathsf{C0}}.\mathsf{pstate.mem}$ on the one hand, and the program rest $s_{\mathsf{C0}}.\mathsf{pstate.prog}$ on the other hand. Moreover, we keep the static program definition i. e., the function table $s_{\mathsf{C0}}.\mathsf{ftab}$ and the type table $s_{\mathsf{C0}}.\mathsf{ttab}$ within the application state.

Finally, we extend the state by a component $\mathsf{c0\_pages}\ s_{\mathsf{C0}}$ that stores the memory size of the application. We use this information to justify the memory sufficiency which is an implicit assumption of the sequential C0 semantics. Summarized, our C0 application state comprises the following components:

- the program state $s_{\mathsf{C0}}.\mathsf{pstate}$,
- the function table $s_{\mathsf{C0}}.\mathsf{ftab}$,
- the type table $s_{\mathsf{C0}}.\mathsf{ttab}$ and,
- the application memory size $\mathsf{c0\_pages}\ s_{\mathsf{C0}}$

Output Function. Before we present the output function ω_{app}, we define an auxiliary function get_output. We use this function to distinguish between our system calls and 'ordinary' function calls and to generate the corresponding output. get_output takes three parameters: the function name *fn*, the parameter list *es* and a non-empty monolithic C0 state s_{C0}. Formally, we define:

```
get_output fn es s_C0 ≡
  if fn = "olosExFinish" then ExFinish
  elsif fn = "olosSendMsg" ∨ fn = "olosRecvMsg"
        then let msgval = c0_eval s_C0 (Deref (es ! 0));
                 msgnum = c0_eval s_C0 (es ! 1)
             in if msgval = ⊥ ∨ msgnum = ⊥ then StuckErr
```

```
        elsif fn = "olosSendMsg"
              then SendMsg (struct2intlist ⌈msgval⌉)
                   ⌈msgnum⌉
              else RecvMsg ⌈msgnum⌉
  else ε_Ω
```

The function distinguishes between the different system call function names and returns the corresponding output. When the application wants to exchange messages with the operating system, the parameter list consists of two elements: Its first value (*es* ! 0) holds the message pointer and the second one (*es* ! 1) stores the message number. The evaluation functions c0_eval evaluate both parameters and return an optional data value. When the evaluation fails or the returned data slice is not initialized the function returns $\perp$. Otherwise, it returns the value of the evaluated data slice. Note that the evaluation function c0_eval of the dereferenced message pointer returns directly the value of the message. When one of these evaluations fails, the return value is given by $\perp$ and the output is set to StuckErr. Otherwise, the outputs to the operating system include additional arguments: In case of a send request, the parameters are the message (converted from its special C0 format to a plain integer list with fixed length) and the message number. The receive request, on the contrary, simply returns the evaluated message number to the operating system. All function names that differ from the names of the system call functions are treated as ordinary function calls. Hence, function get_output returns ε_Ω.

Now, we embed get_output into the overall output function ω_{app}. This function takes an application state and returns an output value. Recall that an empty state $S_{C0} = \perp$ indicates that a runtime error has occurred in the computations before. When the function gets an empty state $\perp$, the application has insufficient memory i. e., predicate sufficient_memory does not hold, or the current statement is an inline assembly statement the function returns output StuckErr. Recall, however, if statements with inlined assembly are executed in the C0 semantics the compiler-correctness theorem does not hold. Although there are different techniques to cope with this restriction [GHLP05, ST08], we exclude inlined assembly in C0 processes. The Olos library has especially been implemented to circumvent the use of additional assembly code within a C0 program. Otherwise, we distinguish over all possible current statements. In the case that the current statement is a function call, we check whether the memory space is also sufficient after extending the local stack extended_stack_in_mem. ω_{app} returns output StuckErr if the new created stack frame of the called function *fn* does not fit into the memory. Otherwise, we apply function get_output in order to compute the corresponding output to the function call. All remaining statements are 'ordinary' C0 statements and return output ε_Ω in order to perform a local step. Formally, the overall output function of C0 applications are defined as follows:

```
ωC0 S_C0 ≡
  case S_C0 of ⊥ ⇒ StuckErr
  | ⌊c⌋ ⇒
```

```
if sufficient_memory (c0_pages c · PAGE_SIZE) c.ttab c.ftab
    c.pstate ∧
   ¬ is_Asm (fst_stmt c.pstate.prog)
   then case fst_stmt c.pstate.prog of
         SCall lv fn ps sid ⇒
           if extended_stack_in_mem c fn then get_output fn ps c
           else StuckErr
         | _ ⇒ εΩ
else StuckErr
```

Note that all failures lead to one unique output - StuckErr. C0 programs neither return an error code nor continue with the next instruction when it fails in computing. Whenever an error occurs, the program gets stuck.

Transition Function. Whenever an application uses a system call in order to communicate with the operating system, the current statement is a function call. The effects of a system call function depend on the particular function name. Similar to the assembly model, there are three possible distinct modifications on an application state. Below, we define three seperate functions to specify the effects on C0 application states:

1. **Writing a message into the applications memory.**
   When an application receives a message, the changes can be encapsulated in a function receiving. This function takes the current application state and the message value *msg* and returns an optional successor state. If this function gets an empty state or the current statement is not a function call, it returns an empty state. Otherwise, it updates the memory with a data slice (recall Section 2.3), that has an aritrary left value, is initialized, and contains a message value with the corresponding type. The message is stored on the address that is given by the left value of the evaluated dereferenced pointer variable. This may be either in the global or the heap memory. Formally:

```
receiving S_C0 msg ≡
  let⊥ s_C0 = S_C0
  in case fst_stmt s_C0.pstate.prog of
     SCall lv fn es sid ⇒
       let⊥ m = mem_update s_C0.ttab s_C0.pstate.mem
                  ⌈eval_m s_C0 (Deref (es ! 0))⌉.ds_lval
                  (|ds_lval = default, ds_type = TYPE_Message_Struct'ty,
                    intermediate = True, ds_initialized = True,
                    ds_data = msg|)
       in ⌊s_C0(|pstate := s_C0.pstate(|mem := m|)|)⌋
     | _ ⇒ ⊥
```

2. **Setting the result variable**
   At the end of a function call, the result value is stored in the left variable. Function

set_result takes an input σ and the current application state and computes the successor state with an updated left variable value. An empty application state will be returned unchanged. Otherwise the function uses set_primres to update the left variable with the integer result value that is determined by the input σ. The address of the left variable of the called function is either located in the global memory or the former top most local memory frame (i. e., before the function has extended the stack). If this update fails, function set_result returns $\bot$ otherwise it updates the memory of the application. We define function set_result as follows:

```
set_result σ S_C0 ≡
  let⊥ s_C0 = S_C0; m = set_primres (Intg (result_value_int σ)) s_C0
  in ⌊s_C0(|pstate := s_C0.pstate(|mem := m|)|)⌋
```

3. **Deleting the system call**
   After the execution of a function, the first statement of the program has to be deleted in order to compute the next statement. Therefore, we define a function del_syscall that simply replaces the system call in the list of remaining statements by a SKIP instruction. An empty application state is returned unchanged. Formally:

```
del_syscall S_C0 ≡
  let⊥ s_C0 = S_C0
  in ⌊s_C0(|pstate := s_C0.pstate(|prog := rm_scall s_C0.pstate.prog|)|)⌋
```

The transition function of C0 applications is defined as a case distinction over the input σ. When the application receives a message from the operating system, it consecutively calls the functions receiving, set_result and del_syscall. Note that a message coming from the operating system as a list of integer values has to be converted back into the message format of C0 applications. A successful message transfer to the operating system ($\sigma =$ SendSuccess) and after the call *olosExFinished* ($\sigma =$ FinishSuccess), we update the left variable of the function call with 0, in case of an invalid message number with -1. All C0 statements except the system call functions (indicated by an empty input: $\sigma = \varepsilon_\Sigma$) use the monolithic transition function. All remaining inputs indicate runtime errors and return the empty state $\bot$. The transition function is formally given by:

```
δ_app σ S_C0 ≡
  case σ of SendSuccess ⇒ del_syscall (set_result SendSuccess S_C0)
  | RecvSuccess val ⇒
      del_syscall
       (set_result (RecvSuccess val)
         (receiving S_C0 (intlist2struct val)))
  | FinishSuccess ⇒ del_syscall (set_result FinishSuccess S_C0)
  | InvalidMsgNr ⇒ del_syscall (set_result InvalidMsgNr S_C0)
  | ε_Σ ⇒ δ_C0m ⌈S_C0⌉
  | _ ⇒ ⊥
```

Predicates. Before we introduce the predicates $\mathsf{is\_valid}_{\mathsf{app}}$ and $\mathsf{is\_init}_{\mathsf{app}}$ of the C0 application model, we first specify a condition that has to hold in both of them. In order to figure out whether an assembly memory image can be abstracted to a C0 application, we need to know exactly how the library functions of our system calls are implemented. We specify a predicate for each system call function that holds, when the corresponding function and its precise function definition belong to the function table.

Recalling the implementation of the system call functions Section 3.4, we know that function `ExFinish` e. g., has neither function parameters nor local variables. The return type of all our system call primitives is a signed 32-bit integer. The function body consists of a composition of two statements - a single inline assembly instruction and a return statement. The former is simply the TRAP instruction calling the kernel. The return statement directly passes integer value 0 to the calling application, where Prim (Intg 0) is a primitive value. ExFinish_in_ft holds when there are two statement identifiers so that function "olosExFinish" and its precise function definition belong to the function table. Formally:

```
ExFinish_in_ft ft ≡
 ∃sid sid'.
  ("olosExFinish",
   (|body =
      Comp (Asm [trap 0] sid) (Return (Lit (Prim (Intg 0))) sid'),
    params = [], rettype = Integer, stack_vars = []|))
  ∈ set ft
```

From the calling interface of function `olosRecvMsg`, we already know that it has two parameters, a pointer named "pmsg" to the message of type "TYPE_Message_Struct" and a message number with name "mn" which is an unsigned 32-bit integer. Furthermore, we know that the return type of the function is a signed 32-bit integer. However, this does not suffice.

We present a predicate RecvMsg_in_ft which holds when function "olosRecvMsg" and its precise function definition belong to the function table.

```
RecvMsg_in_ft ft ≡
 ∃sid sid'.
  ("olosRecvMsg",
   (|body =
      Comp
       (Asm [llw 11 30 16, llw 12 30 20, trap 2, lsw 22 30 24] sid)
       (Return (VarAcc "result") sid'),
    params = [("pmsg", Ptr "TYPE_Message_Struct"), ("mn", UnsgndT)],
    rettype = Integer, stack_vars = [("result", Integer)]|))
  ∈ set ft
```

Additionally, to the function parameters and the return type, we learn that this function obtains exactly one local variable that is called "result" and is a signed 32-bit integer.

Furthermore, the function body consists of a composition of two statements - a portion of inlined assembly code and a return statement. Except the annotation and the statement identifiers, the statements are accurately defined. Predicate SendMsg_in_ft is almost identical to RecvMsg_in_ft, it only differs in the trap number.

Finally, predicate libolos_included summarizes all declarations of the library functions together with the requirement that the message type belongs to the type table. Formally:

libolos_included ft $tt \equiv$
  ("TYPE_Message_Struct", TYPE_Message_Struct$'$ty) $\in$ set tt $\wedge$
  ExFinish_in_ft ft $\wedge$ SendMsg_in_ft ft $\wedge$ RecvMsg_in_ft ft

The validity predicate $\text{is\_valid}_{\text{app}}$ holds either for empty states or holds when states of the application model fulfill well-formedness constraints. We have taken the validity predicate valid_C0 directly from the sequential semantics. Moreover, we require that the system call functions should be declared in the system call library. In contrast to In der Rieden &Tsyban [IT08, IdR09] we do not regard the system-call library and the kernel code as two distinct programs that can be linked under certain preconditions. Instead, we already regard the linked C0 program and specify the class of C0 programs that are compatible with our library. More precisely, this means that the names of all functions have to be disjoint. Finally, the mainfunction has to be part of the function table.

$\text{is\_valid}_{\text{app}}$ $S_{C0} \equiv$
  case S_{C0} **of** $\bot \Rightarrow$ True
  | $\lfloor s_{C0} \rfloor \Rightarrow$
    size_{app} $S_{C0} \leq$ TVM_MAXPAGES $\wedge$
    valid_C0 s_{C0}.ttab s_{C0}.ftab s_{C0}.pstate $\wedge$
    libolos_included s_{C0}.ftab s_{C0}.ttab $\wedge$
    fst (hd s_{C0}.ftab) = mainfunction

The initialization predicate $\text{is\_init}_{\text{app}}$ subsumes all constraints a state has to fulfill after power-up. It determines whether the C0 state S_{C0} is a valid initial state that can be compiled into the memory image *img*. The application should have the correct size which is determined by the number of pages *pages*. Furthermore, the C0 program has to be compilable and compatible with the system call library. The state of the C0 program must be initial and the predicate is_compilation ensures that the code of the compiled program is stored in *img*. Moreover, the address range for the global variables is initialized with zeros. We describe the predicate formally with:

$\text{is\_init}_{\text{app}}$ *img pages* $S_{C0} \equiv$
  $\exists s_{C0}$.
    $S_{C0} = \lfloor s_{C0} \rfloor \wedge$
    size_{app} S_{C0} = *pages* $\wedge$
    libolos_included s_{C0}.ftab s_{C0}.ttab $\wedge$
    ($\exists$*gms*.
      is_compilable s_{C0}.ttab s_{C0}.ftab *gms* $\wedge$

$\lfloor s_{C0}.\mathsf{pstate} \rfloor = \mathsf{initial\_conf}\ s_{C0}.\mathsf{ftab}\ gms \wedge$
$\mathsf{is\_compilation}\ s_{C0}.\mathsf{ttab}\ s_{C0}.\mathsf{ftab}\ gms\ img)$

Application Size. The application size $\mathsf{size}_{\mathsf{app}}$ returns 0 for empty states and component $\mathsf{c0\_pages}\ s_{C0}$ otherwise. Formally:

$\mathsf{size}_{\mathsf{app}}\ S_{C0} \equiv \textbf{case}\ S_{C0}\ \textbf{of}\ \bot \Rightarrow 0 \mid \lfloor s_{C0} \rfloor \Rightarrow \mathsf{c0\_pages}\ s_{C0}$

Validity. So far, the definitions above specify a model for C0 applications. Validity of this model is given, when all rules presented in Section 4.2.1 can be discharged.

Theorem 4.4 (Validity of the C0 Application Model)**.** *The presented C0 application model is valid.*

Proof. The proofs of all 11 axioms are in principle similar to the proofs of assembly applications. Axioms (valid_inv_pointer) and (valid_continue) are very simple. According to their input values the transition function sets the successor state of the C0 application to the error state $\bot$. This state is valid due to the definition of $\mathsf{is\_valid}_{\mathsf{app}}$. The next proofs benefit from the fact that neither the type and function table nor the application size change under the used functions. The axioms concerning the application size ((init_app_size), (delta_app_size)) directly follow from the definition $\mathsf{is\_init}_{\mathsf{app}}$ and this fact. Axiom (valid_initial) more or less requires to unfold the definitions of the predicates. The preservation of validity after an 'ordinary' C0 small-step transition is given by theorem 5.17 of [Lei08]. The remaining axioms concerning the system calls demand in priciple that none of the state updating functions leads to the error state. Moreover, the current statement is not modified during these executions. Summarizing, we can say that valid memory updates and the replacing of the first statement of the program list into a SKIP statement do not harm the validity of a C0 state. In a nutshell, the correctness of the message format results from the type correctness of the function parameters and the definition of the convertion functions. □

4.3 Abstract ECU Automaton

In this section, we consider the abstract automaton describing the behaviour of the entire ECU. Basically, the ECU consists of two different components - a processor with a running operating system and several applications on the one hand, and the ABC device on the other hand. Furthermore, we introduce predicates in order to constrain ECU states and relate our ECU model to the distributed model of Steffen Knapp.

ECU State and Transition Function. First, we present the abstract ECU automaton together with its transition function. A state of the ECU model is defined by the following components:

- an application mapping $s_{\mathsf{ecu}}.\mathsf{AM}$,
- the message buffers $s_{\mathsf{ecu}}.\mathsf{MB}_{\mathsf{a}}$,

- an idle flag $s_{\mathsf{ecu}}.\mathsf{idleflag}$ and,
- the ABC device $s_{\mathsf{ecu}}.\mathsf{abc\_dev}$

The application mapping $s_{\mathsf{ecu}}.\mathsf{AM}$ is a mapping from process identifiers to an abstract application type. Hence, we can instantiate the ECU model with one of the specified valid application models (either C0 or assembly applications). The message buffers $s_{\mathsf{ecu}}.\mathsf{MB_a}$ are modelled as a list of integer lists, and an *idle flag* ($s_{\mathsf{ecu}}.\mathsf{idleflag}$) abstracting whether an application or the kernel is running. Furthermore, we literally embed the state of the ABC automaton Section 4.1 into our state ($s_{\mathsf{ecu}}.\mathsf{abc\_dev}$). We modelled the abstract ECU state as slim as possible by eliminating redundant information. Therefore, a lot of informations can be extracted from the ABC state such as the current slot number, the sending permission table, the send flag or the number of slots per round.

In Figure 4.4, the state diagram of the ECU automaton, it is possible to recognize the transition phases of a slot as sketched in Figure 3.2 on page 60. The states *recv* and *send* in the diagram directly correspond to the beginning of the receive phase and the send phase. During the compute phase within one slot, the automaton is in one of the states *comp*, $idle_{\mathsf{r}}$, and $idle_{\mathsf{s}}$. Finally, the device communication is depicted by $e\updownarrow msg$.

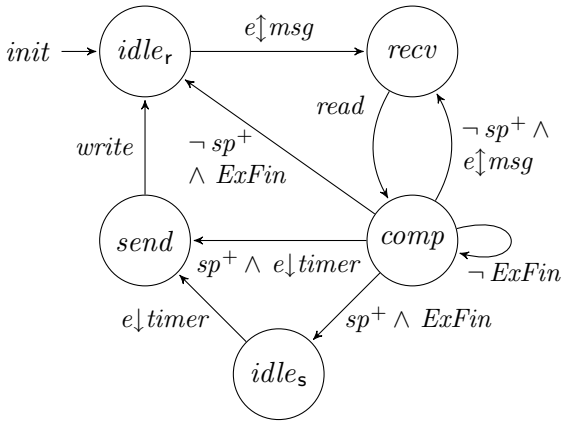

Figure 4.4: State diagram for the ECU

In the ECU specification, we abstract from the initialization phase after power-up. The initial state is $idle_{\mathsf{r}}$, where the ECU awaits an input from the environment. The corresponding transition resembles the device-communication phase.

Symbolic names	$s_{\mathsf{ecu}}.\mathsf{abc\_dev.INT}$	$s_{\mathsf{ecu}}.\mathsf{abc\_dev.SF}$	$s_{\mathsf{ecu}}.\mathsf{idleflag}$
$idle_{\mathsf{r}}$	False	False	True
recv	True	False	*
comp	False	*	False
$idle_{\mathsf{s}}$	False	True	True
send	True	True	*

Figure 4.5: Automaton states related to their flag combination

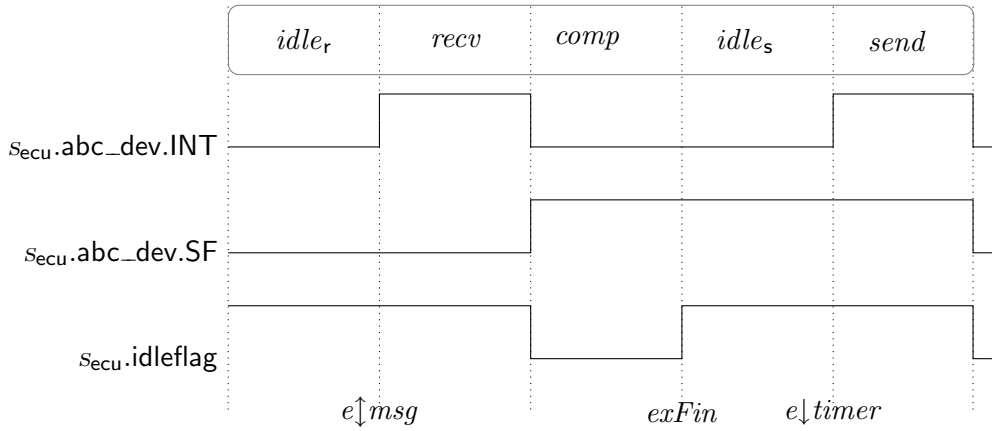

Figure 4.6: Timing diagram of ECU components

The device communication $e\updownarrow msg$ triggers a new slot and the ECU changes to the receive state (*recv*). Recall that the current slot number which is part of the ABC device is directly increased during this transition Section 4.1. In the implementation, however, OLOS sets its corresponding current slot variable not until the receive phase. In this state, the ABC's interrupt is raised and its send flag is not set. The transition labelled *read* represents the actual receive phase, where OLOS reads the ABC's receive buffer into its message buffer and instructs the ABC to clear its interrupt. Furthermore, the send flag is set to the value of sp^+ (the ECU's send permission in the next slot).

Immediately afterwards, the ECU is in the compute state *comp*. The idle flag as well as the ABC's interrupt flag are cleared. There are several transitions possible from this state. All transitions involve a computation of the application scheduled by the AST table. If there is no external event and the application does not issue an *olosExFinished* call, the ECU remains in the *comp* state. In case of an *olosExFinished* call, the idle flag is set and the ECU descends into an idle state, $idle_s$ or $idle_r$, depending on the send permission in the next slot, i. e., the value of the previously set send flag. If an external event (inputs $e\updownarrow msg$ or $e\downarrow timer$) occurs, the interrupt line is raised. An external event during the *comp* state means that the application has exceeded its worst-case execution time. The ECU reacts just as if it had been in an idle state: If the ECU has the send permission in the next slot, it proceeds to the send state, otherwise to the receive state.

Finally, if the ECU is in the *send* state, the transition labeled *write* describes the send phase, where OLOS writes the content of a message buffer to the ABC's send buffer, resets the send flag, sets the idle flag, and finally requests the ABC to clear its interrupt.

Figure 4.6 presents the interrupt flag s_{ecu}.abc_dev.INT, the send flag s_{ecu}.abc_dev.SF and the idle flag s_{ecu}.idleflag of an ECU state s_{ecu} during one slot. In this szenario we assume that the ECU has the sending permission and the current scheduled application terminates before the timer interrupt $e\downarrow timer$ occurs. The timing diagram depicts for a certain case how the states are related to the flag combination (cf. Table 4.5 on the preceding page).

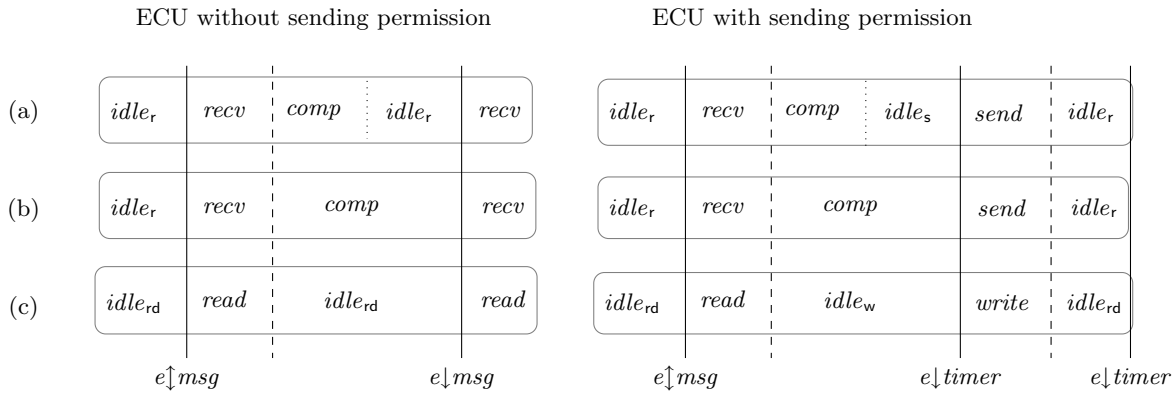

Figure 4.7: Relation of ECU and ABC model

Note that the ECU automaton refines the ABC automaton since its device component is an ABC state. We can relate every state of the ECU model to a corresponding one of the ABC model. Figure 4.7 depicts the state relation between the ECU and the ABC automaton. The columns distinguish between the sending permission and the rows illustrate (a) an ECU slot execution with application termination, (b) an ECU slot execution with deadline violation and, (c) a slot execution from the view of the ABC model. The dotted lines represent the transition after system call *olosExFinished* has been invoked, the dashed lines show the transitions where the processor clears the interrupt line of the ABC device. Finally, external events trigger either the start of a new slot or the beginning of the send phase.

Before we formalize the transition function of the abstract ECU automaton, we first present some functions defining individual transitions. In the following definitions, we use the functions next and prev in order to compute the next and previous current slotnumber (modulo SLOTCOUNT).

We specify function Recv_Phase encapsulating the behaviour of the ECU in the receive phase. The function gets a pair of scheduling tables (AST, BT) and the current ECU state s_{ecu} and returns the successor state s'_{ecu}.

Recv_Phase (AST, BT) $s_{\mathsf{ecu}} \equiv$
  let *abc* $= s_{\mathsf{ecu}}$.abc_dev; *csn* $=$ *abc*.CSN; *msgnum* $= BT$! prev *csn*
  in s_{ecu}
    (|MB$_a$:= s_{ecu}.MB$_a$[*msgnum* := last *abc*.RB], idleflag := False,
     abc_dev := *abc*(|INT := False, SF := read_sendl *abc* (next *csn*)|)|)

First, we extract the current slot number from the ABC and compute the entry of the buffer index table of the previous slot. This number determines the message buffer that stores the message broadcast on the bus in the previous slot. Now, we can copy the content of the processor visible receive buffer last *abc*.RB into the message buffer and disable the idle flag. Finally, we update the ABC state by clearing the interrupt flag

and setting the send flag to the value of the next slot according to the send-permission table (stored in an ABC configuration register).

In the ECU automaton transition *write* describes the send phase. We express the effects of this transition in function Send_Phase. Again, this function takes the application scheduling table and the buffer index table as a pair (AST, BT) additionally to the current state. The successor state s'_{ecu} after the execution of the send phase Send_Phase is formalized in Isabelle/HOL as follows:

```
Send_Phase (AST, BT) s_ecu ≡
  let abc = s_ecu.abc_dev; msgnum = BT ! next abc.CSN
  in s_ecu
     (|idleflag := True,
      abc_dev := abc(|SB := abc.SB[0 := s_ecu.MB_a ! msgnum], INT := False,
                     SF := False|)|)
```

Similar to the receive phase, we first compute the number of the message buffer. This time, we increase the current slot number since we want to read the buffer storing the message that will be broadcast in the next slot. Then, we enable the idle flag. In the device component, the function disables the interrupt and the send flag and copies the entire message from the selected message buffer to the processor visible ABC send buffer (recall that $\mathsf{hd}\ s_{\mathsf{abc}}.\mathsf{SB} = s_{\mathsf{abc}}.\mathsf{SB}\ !\ 0$ holds). Note that we manipulated the device component directly instead of using the ABC device transition. Hence, this allows to read the functions more easily and presents their effects readily comprehensible.

When the ABC device raises its interrupt flag before the idle flag is set, the current executed application has exceeded its worst-case execution time. The ECU handles this deadline violation with function Comp_DL_Violation which is executed before the receive or send phase are started as usual. The successor state s'_{ecu} will be computed from the current state s_{ecu} and the pair of tables (AST, BT).

```
Comp_DL_Violation (AST, BT) s_ecu ≡
  let procnum =
        if s_ecu.abc_dev.SF then nat2pid (AST ! s_ecu.abc_dev.CSN)
        else nat2pid (AST ! prev s_ecu.abc_dev.CSN);
      app = s_ecu.AM procnum
  in case ω_app app of StuckErr ⇒ s_ecu
     | ε_Ω ⇒ s_ecu(|AM := s_ecu.AM(procnum := δ_app ε_Σ app)|)
     | _ ⇒ s_ecu(|AM := s_ecu.AM(procnum := δ_app Continue app)|)
```

This function first determines which application has been executed until the timer interrupt occured. Therefore, it examines the entries of the application scheduling table AST. When the send flag is set, the interrupt signals the start of a send phase and we use the AST entry for the current slot number. Otherwise, the application had been interrupted when a new slot started and the slot number in the ABC device had already been increased. Hence, the interrupted application corresponds to the AST entry in the previous slot. Then, we examine the output function of the interrupted application

ω_{app} *app*. The ECU state remains unchanged when the output is StuckErr. An empty output ε_Ω permits the application to perform a last local step. Otherwise, the application executes a transition with input Continue.

As long as no external event occurs (i. e., $e{\downarrow}timer$ or $e{\updownarrow}msg$) in the *comp* state, we are in the compute phase. During this phase, the currently scheduled application is executed and may exchange messages with the operating system. When the application signals its termination with *ExFin*, we can add the idle states to the compute phase, too. In this phase, only the currently scheduled application and the message buffers are involved whereas the other ECU components remain unchanged. Hence, we formulate a transition δ_{cc} that simply takes a single application state on the one hand and the message buffers on the other hand. The function recognizes the output of the application and computes a corresponding input:

$\delta_{cc}\ (s_{app}, mb) \equiv$
case $\omega_{app}\ s_{app}$ **of**
SendMsg *msgval msgnum* $\Rightarrow$
if *msgnum* $<$ MSGCOUNT
then (δ_{app} SendSuccess s_{app}, mb[*msgnum* := *msgval*])
else (δ_{app} InvalidMsgNr s_{app}, mb)
| RecvMsg *msgnum* $\Rightarrow$
if *msgnum* $<$ MSGCOUNT
then (δ_{app} (RecvSuccess (mb ! *msgnum*)) s_{app}, mb)
else (δ_{app} InvalidMsgNr s_{app}, mb)
| ExFinish $\Rightarrow$ (δ_{app} FinishSuccess s_{app}, mb)
| InvPtrErr $\Rightarrow$ (δ_{app} InvPtrResponse s_{app}, mb)
| StuckErr $\Rightarrow$ (s_{app}, mb)
| Undefined_Trap $\Rightarrow$ (δ_{app} Continue s_{app}, mb)
| $\varepsilon_\Omega \Rightarrow$ ($\delta_{app}\ \varepsilon_\Sigma\ s_{app}$, mb)

In the case that the output of the current application denotes a system call, the operating system reacts with the corresponding response. The application may request to exchange messages with the operating system $\omega_{app}\ s_{app}$ = SendMsg *msgval msgnum* $\vee$ $\omega_{app}\ s_{app}$ = RecvMsg *msgnum*. The kernel checks whether the message number indicates an existing message buffer and either handles the system call or returns an error. When the application sends a message, it is written in the corresponding message buffer. The application may also signal the termination of its execution $\omega_{app}\ s_{app}$ = ExFinish. Olos handles this call with δ_{app} FinishSuccess s_{app}. This transition function cannot change the idle flag so that we postpone this component manipulation and change it into an extra function that embeds the extracted state into an entire ECU state. When the output of the application is empty ε_Ω, it performs a local step. The other cases are error cases: Output StuckErr returns the former state, and continue error Undefined_Trap and invalid pointer InvPtrErr are handled from the application depending on the used model (C0 program or Vamp assembly).

Finally, we embed the extracted successor state of function δ_{cc} into a function Compute_Phase. This function takes an entire ECU state s_{ecu} and the pair of tables (*AST*,

BT) to return the successor state s'_{ecu}. We use function current_pid to select a process identifier from the application scheduling table in the current slot:

current_pid (AST, BT) $s_{\mathsf{ecu}} \equiv$ nat2pid $(AST$! s_{ecu}.abc_dev.CSN$)$

Moreover, function current_proc encapsulates the state of the currently scheduled application:

current_proc (AST, BT) $s_{\mathsf{ecu}} \equiv s_{\mathsf{ecu}}$.AM (current_pid (AST, BT) s_{ecu})

Function Compute_Phase applies function δ_{cc} on the selected current application and the message buffers. Then, it updates the involved components with the result of function δ_{cc}. At this place, we raise the idle flag of the operating system, when an application signals its termination (i. e., "olosExFinish" is called). The successor state s'_{ecu} of the compute phase can be obtained by executing function Compute_Phase with a pair of tables (AST, BT) and the current ECU state s_{ecu}:

Compute_Phase (AST, BT) $s_{\mathsf{ecu}} \equiv$
 let *procnum* = current_pid (AST, BT) s_{ecu};
  app = current_proc (AST, BT) s_{ecu}
 in s_{ecu}
  (|AM := s_{ecu}.AM(*procnum* := fst (δ_{cc} (*app*, s_{ecu}.$\mathsf{MB_a}$))),
   $\mathsf{MB_a}$:= snd (δ_{cc} (*app*, s_{ecu}.$\mathsf{MB_a}$)),
   idleflag := **if** ω_{app} *app* = ExFinish **then** True **else** s_{ecu}.idleflag|)

All presented functions formalizing the receive, compute and send phase involve the operating system Olos. We collect these internal changes into one transition called δ_{olos}. The transition takes the current ECU state s_{ecu} and the pair of scheduling tables (AST, BT) to determine the successor ECU state s'_{ecu}. In the following formulae, we abbreviate the scheduling table pair (AST, BT) with Θ on higher abstraction layers (where their access is not visible any more). In case of an enabled interrupt flag of the ABC device the function decides on the ABC's send flag whether the ECU enters the receive or the send phase. When it detects a deadline violation (i. e., $\neg$ s_{ecu}.idleflag), function Comp_DL_Violation is applied before handling this case. With a disabled ABC interrupt it depends on the ECU's idle flag whether the ECU is in the compute phase (i. e., $\neg$ s_{ecu}.idleflag) or simply waits (i. e., s_{ecu}.idleflag). Formally, we define:

δ_{olos} Θ $s_{\mathsf{ecu}} \equiv$
 if s_{ecu}.abc_dev.INT
  then let *ecu'* =
    if s_{ecu}.idleflag **then** s_{ecu}
    else Comp_DL_Violation Θ s_{ecu}
   in if *ecu'*.abc_dev.SF **then** Send_Phase Θ *ecu'*
    else Recv_Phase Θ *ecu'*
 elsif $\neg$ s_{ecu}.idleflag **then** Compute_Phase Θ s_{ecu} **else** s_{ecu}

So far, we have neglected the transitions relying on the external input that only change the device component of the ECU state. The entire transition function of our abstract ECU automaton is formalized in function δ_{ECU}. This transition function takes the pair of scheduling tables denoted with Θ, an optional external input *eifi* and the current ECU state s_{ecu}. We distinguish processor computation in case of an empty input ($eifi = \bot$) from external devices steps ($eifi = \lfloor i \rfloor$). Recall that the transition function of the ABC δ^e_{abc} has been introduced in Section 4.1. When the external input is empty, we apply function δ_{olos}, otherwise we update the ECU's device component with the successor state of the extrnal ABC transition. Thus, the definition of the ECU transition function in Isabelle/HOL is given as follows:

$\delta_{\mathsf{ECU}}\ \Theta\ eifi\ s_{\mathsf{ecu}} \equiv$
  case *eifi* **of** $\bot \Rightarrow \delta_{\mathsf{olos}}\ \Theta\ s_{\mathsf{ecu}}$
  $\mid \lfloor i \rfloor \Rightarrow s_{\mathsf{ecu}}(\!|\mathsf{abc\_dev} := \mathsf{fst}\ (\delta^e_{\mathsf{abc}}\ i\ s_{\mathsf{ecu}}.\mathsf{abc\_dev})|\!)$

Predicates on ECU States. We restrict the scheduling tables and our ECU states by several constraints that should hold directly after initialization or after all transitions of the automaton. We introduce some predicates that encapsulate these requirements.

The application scheduling table AST and the buffer index table BT are called valid, when predicate valid_tables holds. Both tables should have SLOTCOUNT entries with meaningful interpretations i. e., the application scheduling table AST only contains defined application identifiers whereas the buffer index table BT includes indices to existing message buffers. Formally:

$\mathsf{valid\_tables}\ (AST, BT) \equiv$
  $|AST| = \mathsf{SLOTCOUNT} \wedge$
  $|BT| = \mathsf{SLOTCOUNT} \wedge$
  $(\forall i{<}|AST|.\ 0 < AST\,!\,i \wedge AST\,!\,i \leq \mathsf{PROCCOUNT}) \wedge (\forall i{<}|BT|.\ BT\,!\,i < \mathsf{MSGCOUNT})$

A valid message buffer consists of MSGCOUNT buffers which store messages in a well-defined format i. e., they have length MSGLENGTH and each value is a signed 32-bit integer. We encapsulate these requirements in the predicate is_valid_MB:

$\mathsf{is\_valid\_MB}\ mb \equiv$
  $|mb| = \mathsf{MSGCOUNT} \wedge (\forall mn{<}\mathsf{MSGCOUNT}.\ \mathsf{is\_valid\_intlistMsg}\ (mb\,!\,mn))$

The definition of a valid abstract ECU state is given in predicate $\mathsf{is\_validState_a}$. We require that all applications, message buffers and the ABC device are valid. Moreover, the current slot number has to be less than SLOTCOUNT. The validity predicate for an abstract ECU state is given below:

$\mathsf{is\_validState_a}\ s_{\mathsf{ecu}} \equiv$
  $(\forall pid.\ \mathsf{is\_valid_{app}}\ (s_{\mathsf{ecu}}.\mathsf{AM}\ pid)) \wedge$
  $\mathsf{is\_valid\_MB}\ s_{\mathsf{ecu}}.\mathsf{MB_a} \wedge$
  $s_{\mathsf{ecu}}.\mathsf{abc\_dev.CSN} < \mathsf{SLOTCOUNT} \wedge \mathsf{is\_valid\_ABC}\ s_{\mathsf{ecu}}.\mathsf{abc\_dev}$

Right after initialization, the ECU state has to be valid, the ABC device is initial and the idle flag is enabled. An ECU state satisfying these constraints, is called initial and the following predicate $\mathsf{is\_initState_a}$ holds:

$\mathsf{is\_initState_a}\ s_{\mathsf{ecu}} \equiv$
$\quad \mathsf{is\_validState_a}\ s_{\mathsf{ecu}} \wedge \mathsf{is\_initial\_ABC}\ s_{\mathsf{ecu}}.\mathsf{abc\_dev} \wedge s_{\mathsf{ecu}}.\mathsf{idleflag}$

Equality of Device Manipulations. In the transition functions dealing with device communication i. e., Recv_Phase and Send_Phase we directly manipulated the device component instead of using the transition function δ_{abc}. This was simply done for a better readability of the functions. Here, we want to justify that the presented device manipulations of the ECU model are equal to the effects of multiple executions of the ABC internal transition function.

The first lemma states that the content of the updated message buffer after the receive phase equals the output list to the operating system that is computed by multiple executions of the ABC internal transition. More specifically, the multiple internal ABC transitions include a read request followed by a *clearint* command of the processor.

Lemma 4.5 (Equal Messages after Receive Phase and Multiple Internal ABC Transitions)**.** *We assume that the current ECU state is valid i. e.,* $\mathsf{is\_validState_a}\ s_{\mathsf{ecu}}$*, the send-flag is not set* $\neg\ s_{\mathsf{ecu}}.\mathsf{abc\_dev.SF}$ *and the tables are valid* $\mathsf{valid\_tables}\ (AST, BT)$ *Then, the updated message buffer after the receive phase equals the content of the processor-facing ABC receive buffer after a read request followed by a clear-interrupt command of the processor.*

$(\mathsf{Recv\_Phase}\ (AST, BT)\ s_{\mathsf{ecu}}).\mathsf{MB_a}\ !\ (BT\ !\ \mathsf{prev}\ s_{\mathsf{ecu}}.\mathsf{abc\_dev.CSN}) =$
$\mathsf{map\ to\_int32}$
$\quad (\mathsf{fst}\ (\mathsf{snd}\ (\delta^{m*}_{\mathsf{abc}}\ (\mathsf{map}\ (\lambda \mathit{port}.\ (\!|\mathsf{mifi\_rd} = \mathsf{True}, \mathsf{mifi\_wr} = \mathsf{False}, \mathsf{mifi\_a} = \mathit{port},$
$\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{mifi\_din} = \mathsf{default}|\!))$
$\qquad\qquad\qquad\qquad [\mathsf{RECV\_PORT}..<\mathsf{RECV\_PORT} + \mathsf{MSGLENGTH}])$
$\qquad\qquad s_{\mathsf{ecu}}.\mathsf{abc\_dev})))$

Proof. After the receive phase the content of the updated message buffer equals the message stored in the processor-facing receive buffer. Hence, we prove by induction that the multiple read request of the operating system returns exactly this message. The internal ABC transition function $\delta^{m}_{\mathsf{abc}}$ returns in case of a read request on a valid port the message value stored in the processor-facing receive buffer at exactly this address. Hence, reading sequentially all port addresses up to the length of the message returns exactly the required message. □

Furthermore, the effects of the receive phase and the internal ABC transition on the ABC component are equal.

Lemma 4.6 (ABC Equality after Receive Phase Execution and Multiple Internal ABC Transitions)**.** *We assume that the current ECU state is valid i. e.,* $\mathsf{is\_validState_a}\ s_{\mathsf{ecu}}$ *and*

the sendflag is not set $\neg$ s_{ecu}.abc_dev.SF *Then, the ABC state after the execution of the receive phase equals the ABC state after multiple internal ABC transitions where* Olos *requests to read a message and writes the clear interrupt command afterwards.*

$$
\begin{aligned}
&(\mathsf{Recv\_Phase}\ \Theta\ s_{\mathsf{ecu}}).\mathsf{abc\_dev} = \\
&\mathsf{fst}\ (\delta^{m*}_{\mathsf{abc}}\ (\mathsf{map}\ (\lambda \mathit{port}.\ (\!|\mathsf{mifi\_rd} = \mathsf{True},\ \mathsf{mifi\_wr} = \mathsf{False},\ \mathsf{mifi\_a} = \mathit{port},\\
&\qquad\qquad\qquad\qquad \mathsf{mifi\_din} = \mathsf{default}|\!))\\
&\qquad [\mathsf{RECV\_PORT}..{<}\mathsf{RECV\_PORT} + \mathsf{MSGLENGTH}] \circledcirc\\
&\qquad [(\!|\mathsf{mifi\_rd} = \mathsf{False},\ \mathsf{mifi\_wr} = \mathsf{True},\ \mathsf{mifi\_a} = \mathsf{COMR\_PORT},\\
&\qquad\ \ \mathsf{mifi\_din} = \mathsf{CLEARINT\_COM}|\!)])\\
&\quad s_{\mathsf{ecu}}.\mathsf{abc\_dev})
\end{aligned}
$$

Proof. The read request of the internal ABC transition does not change the ABC state, whereas the transition performs a write access to clear the interrupt line of the ABC device. The lowered send flag indicates that we have to raise the interrupt and set the send flag to the sending permission table SPT entry of the next slot. By unfolding the definitions of the receive specification and the internal ABC transition function $\delta^{m}_{\mathsf{abc}}$ we obtain exactly that. □

Finally, the effects of the send phase and multiple internal ABC transitions on the ABC component are equal. The multiple internal ABC transitions encapsulate the write request of the processor on the one hand and the sending of the *clearint* command on the other hand.

Lemma 4.7 (ABC Equality after Send Phase Execution and Multiple Internal ABC Transitions)**.** *Assuming that we have well-formed scheduling tables i. e.,* valid_tables (AST, BT)*, the current state is valid i. e.,* s_{ecu}.abc_dev.SF *and its send flag is raised i. e.,* $\mathsf{is\_validState_a}\ s_{\mathsf{ecu}}$*. Then, the ABC state after the execution of the send phase equals the ABC state after multiple internal ABC transitions where* Olos *requests to write a message and sends the clear interrupt command afterwards.*

$$
\begin{aligned}
&(\mathsf{Send\_Phase}\ (AST, BT)\ s_{\mathsf{ecu}}).\mathsf{abc\_dev} = \\
&\mathsf{fst}\ (\delta^{m*}_{\mathsf{abc}}\ (\mathsf{map}\ (\lambda(p, d).\ (\!|\mathsf{mifi\_rd} = \mathsf{False},\ \mathsf{mifi\_wr} = \mathsf{True},\ \mathsf{mifi\_a} = p,\ \mathsf{mifi\_din} = d|\!))\\
&\qquad (\mathsf{zip}\ [\mathsf{SEND\_PORT}..{<}\mathsf{SEND\_PORT} + \mathsf{MSGLENGTH}]\\
&\qquad\quad (\mathsf{map}\ \mathsf{to\_nat32}\ (s_{\mathsf{ecu}}.\mathsf{MB_a}\ !\ (BT\ !\ \mathsf{next}\ s_{\mathsf{ecu}}.\mathsf{abc\_dev.CSN})))) \circledcirc\\
&\qquad [(\!|\mathsf{mifi\_rd} = \mathsf{False},\ \mathsf{mifi\_wr} = \mathsf{True},\ \mathsf{mifi\_a} = \mathsf{COMR\_PORT},\\
&\qquad\ \ \mathsf{mifi\_din} = \mathsf{CLEARINT\_COM}|\!)])\\
&\quad s_{\mathsf{ecu}}.\mathsf{abc\_dev})
\end{aligned}
$$

Proof. The write request of the processor only changes the send buffer of the ABC device. We prove that the word-wise update of the send buffer equals the direct update by induction. The clear interrupt command only manipulates the interrupt flag and the send flag of the device. Both values are set to False since the send flag of the ECU was raised. Hence, the manipulations of the ABC device after a send phase are equal to the effects after we have applied $\delta^{m*}_{\mathsf{abc}}$ with the corresponding processor inputs. □

4.3.1 Excursion: Towards a Distributed OLOS Model

In our thesis, we focus on the behaviour of a single ECU whereas Steffen Knapp considers a system of ECUs connected by an automotive bus in his thesis [Kna08]. For various reasons, his formalization of a single ECU slightly differs from ours. This section illustrates the differences and presents a simulation proof that formally links both models.

First, the partition of transition phases is a little different. More specifically, Knapp considers global and local transition phases. Local transitions encapsulate all ECU-local computations i. e., reading and writing the ABC buffer and executing the scheduled application. The local receive, compute and send phase of Steffen Knapp's thesis can be related to our receive, compute and send phase. Note that the local computations are individual for every ECU. Knapp abstracts all local transitions that are sequentially executed within one slot into one big transition which is called global compute. Similarly to the local structure, he defines a global receive and send transition phase, where the automotive bus is involved. In principle, he splits our device-communication phase into two phases where the ECU's read or write accesses on the bus are strictly separated. Table 4.4 relates both phase distributions.

Knapp		Schmidt
global send		DevComm
global recv		
global comp	local recv	Recv
	local comp	Comp
	local send	Send

Table 4.4: Model phases

Knapp	Schmidt
RB	last s_{abc}.RB
SB	hd s_{abc}.SB
oldm	hd s_{abc}.RB
newm	last s_{abc}.SB

Table 4.5: Buffer Relation

Second, Knapp abstracts the buffer pairs from the implementation (see Section 3.1) to one buffer in the ECU-local ABC component and two global message buffers oldm and newm. For the duration of the global compute phase, we relate the buffer components as shown in Table 4.5: The processor-facing buffers in our model relate to Knapp's receive RB and send buffer SB component. The old global message oldm is stored in the bus-facing receive buffers of our model, and the new global message newm can be found in the bus-facing send buffer of the sending ECU. For the distributed model, Knapp's approach avoids redundancies because in the implementation, each ECU contains its own copy of the broadcast messages, which are identical if we assume an ideal bus. Nevertheless, we need pairs of buffers in our single ECU model in order to reason about computations over slot boundaries. Otherwise, we could not model the delay in the message transmission (see Figure 3.3 on page 61).

Furthermore, Knapp's model has stronger assumptions about the implementation. Most notably, his model cannot handle deadline violations and non-terminating processes. For our implementation of the OLOS kernel we have considered all events that may occur although they do not conform to desired constraints. Hence, the specification

is a complete functional description.

Figure 4.8 on the facing page illustrates both models in comparison. The sketch once more relies on the example with the given scheduling tables Table 3.1 on page 59 in Section 3.2. At the top, we depicted the transmission delay of a message in our model using the buffer swapping of the implementation. Below we depict Knapp's model with its different phase distribution and the global message buffers. Note that a message transmission delay in both models takes two slots. We distinguish between different messages: The one that is broadcast on the bus in slot 2, is denoted with $*$ whereas the message broadcast before (i. e., slot 1) is labeled with $\circ$. The message that will be sent in the next slot (i. e., slot 3) is denoted with $\bullet$. Note that the message broadcast in the previous slot $\circ$ does not appear in our model. In slot 2, however, message $\circ$ is stored in all processor-facing receive buffers of our model.

It is clearly recognizable that our send and receive phases equal the local send and receive phase in the lower model. For the bus communication, we regard both models in several steps: In the global send phase of slot 1, ECU 1 of the lower model copies the message into the global buffer newm. The corresponding action in the model above is the buffer rotation in the device-communication phase in slot 2. In the same phase, the bus-facing receive buffers are updated with the content of the bus-facing send buffer of the sending ECU. In Knapp's model, we relate the bus communication with the message transfer during slot 2 where the value of the new global message buffer newm is copied to the old global message buffer oldm. Finally, our model rotates the buffer in slot 3 during the device-communication phase. This corresponds to the receive buffer update in slot 3 during the global receive phase. Obviously, both models satisfy the transmission protocol although they deal with different phases and state components. For further details of the distributed OLOS model, we refer the interested reader to the thesis of Steffen Knapp [Kna08].

Both models focus on different aspects. Knapp concentrates on the device communication whereas our work considers finite computation traces of a single ECU. Nevertheless, there is a formal need to relate both models: Knapp [Kna08, chapter 23] reasoned about the global send and receive phase in his model and left the argumentation about the global compute phase open for future work. This thesis is concerned with exactly this gap. Knapp's modelling of the global compute phase, however, is more abstract than our formalization. Below, we develop a simulation theorem bridging both formalisms.

If we assume the timely termination of the applications in Knapp's model, we are able to relate both models and show simulation for an entire local phase. In other words, the currently scheduled application $\mathsf{current\_proc}\ \Theta\ s_{\mathsf{ecu}}$ executes a certain number n of local transitions δ_{cc} before it signals its termination by calling *olosExFinish*. Then, function ω_{app} returns the corresponding output EXFINISH. Assuming that the involved message buffers are valid, Knapp formalizes timely termination of applications as follows:

$$\forall mb.\ \mathsf{is\_valid\_MB}\ mb \longrightarrow (\exists n.\ \omega_{\mathsf{app}}\ (\mathsf{fst}\ ((\delta_{\mathsf{cc}}{}^{n})\ (\mathsf{current\_proc}\ \Theta\ s_{\mathsf{ecu}},\ mb))) = \textsc{ExFinish})$$

Note that for the duration of the local transitions, only the processor-facing buffers are involved. Thus, the buffers facing the bus can be neglected. The abstraction from

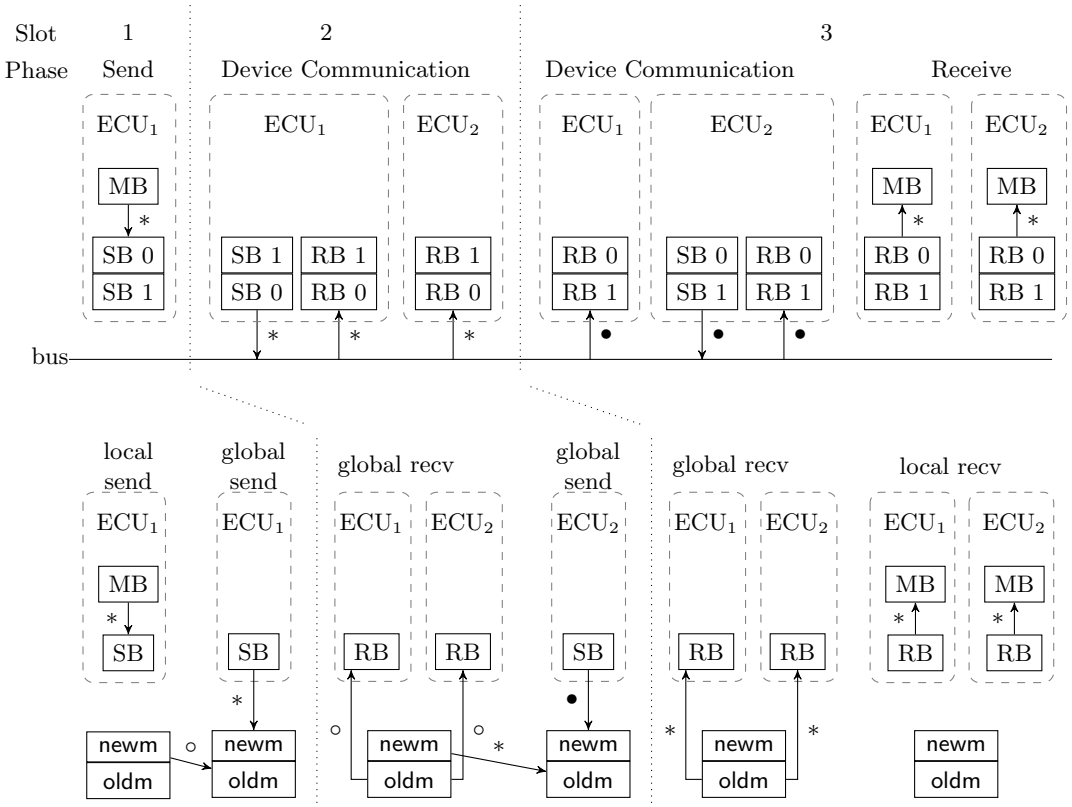

Figure 4.8: Comparison of both models

double to single buffers is straightforward: we simply map the processor-facing side of the double buffers to a single one. The abstraction function for the ABC models is given below:

$$\mathsf{abs\_abc}\ s_{\mathsf{abc}} \equiv s_{\mathsf{abc}}(\!|\mathsf{SB} := [\mathsf{hd}\ s_{\mathsf{abc}}.\mathsf{SB}],\ \mathsf{RB} := [\mathsf{last}\ s_{\mathsf{abc}}.\mathsf{RB}]|\!)$$

For convenience, we furthermore wrap the abstraction of an entire ECU state s_{ecu} to a state of Knapp's Olos model into the function $\mathsf{abs}_{\mathsf{ECU}}$.

We state a relation between both models during a global compute phase. A global compute transition δ_{lcomp} of Knapp's Olos model simulates a number of local computation steps during the one slot of our ECU model. More specifically:

Theorem 4.8 (Global Compute Phase Simulation)**.** *We assume*

- *a valid state* s_{ecu}*, i. e.,* $\mathsf{is\_validState}_{\mathsf{a}}\ s_{\mathsf{ecu}}$*,*
- *valid scheduling tables* Θ*, i. e.,* $\mathsf{valid\_tables}\ \Theta$*,*
- *the state* s_{ecu} *is a receive state, i. e., the interrupt flag is raised and the send flag is lowered (*$s_{\mathsf{ecu}}.\mathsf{abc\_dev.INT} \wedge \neg\ s_{\mathsf{ecu}}.\mathsf{abc\_dev.SF}$*),*
- *there has not recently been a deadline violation, i. e., the idle flag is set* $s_{\mathsf{ecu}}.\mathsf{idleflag}$*, and*
- *the currently scheduled application* $\mathsf{current\_proc}\ \Theta\ s_{\mathsf{ecu}}$ *will eventually terminate:*
  $\forall mb.\ \mathsf{is\_valid\_MB}\ mb \longrightarrow$
  $(\exists n.\ \omega_{\mathsf{app}}\ (\mathsf{fst}\ ((\delta_{\mathsf{cc}}{}^{n})\ (\mathsf{current\_proc}\ \Theta\ s_{\mathsf{ecu}},\ mb))) = \textsc{ExFinish})$

Under these assumptions, there exists an input sequence is *in such a way that a global compute transition* δ_{lcomp} *of the abstracted* s_{ecu} *state is equal to the state abstraction after a computation with* is*.*

Formally:

$$\exists is.\ \delta_{\mathsf{lcomp}}\ (\mathsf{abs}_{\mathsf{ECU}}\ s_{\mathsf{ecu}}) = \mathsf{abs}_{\mathsf{ECU}}\ (\mathsf{fold}\ (\delta_{\mathsf{ECU}}\ \Theta)\ is\ s_{\mathsf{ecu}})$$

Proof. We compare both models starting at a slot boundary. The first δ_{ECU} transition handles the receive phase and is called with an empty input $\perp$. After this phase, the send flag is set to the entry of the sending permission table in the next slot. Then, both ECUs enter the compute phase were they do several steps until the scheduled application finishes its execution. Hence, a fixed number of empty inputs $\perp$ until the application terminates is given. Now, we distinguish on the ECU's send flag: when the ECU is not permitted to send, the global compute phase is done. Otherwise, we expect a timer input $\lfloor\mathsf{eifi\_abc\_timer}\rfloor$ of the external environment i. e., the bus. This signal indicates the start of the send phase. A final δ_{ECU} transition with an empty input $\perp$ executes the send phase in order to complete the global compute phase for the sending ECU. □

5 Implementation Correctness

> On peut avoir trois principaux objets dans l'étude de la vérité:
> l'un, de la découvrir quand on la cherche;
> l'autre, de la démontrer quand on la possède;
> le dernier, de la discerner d'avec le faux quand on l'examine.[1]
>
> *Blaise Pascal, quoted in: "Pensées", Article II: Réflexions sur la géométrie en général*

Contents

This chapter reports on formal implementation correctness proof using Hoare logic as an effective method of C0 program verification. In the last chapter, we presented an abstract model of the entire ECU including applications, the ABC device and OLOS. However, we certainly have to ensure that the ECU model really abstracts the OLOS implementation. For this reason, we aim at a formal simulation proof between implementation and specification by induction.

Recall that OLOS is implemented to run as a CVM kernel machine. Therefore, we represent this framework in our programming model and define the effects of CVM integrating OLOS as a Simpl function. Combining this kernel execution with the external device transition $\delta^{e}_{\mathsf{abc}}$, we obtain an overall implementation step $\delta^{\mathcal{I}}_{\mathsf{ECU}}$. Note that in case

[1] In the study of truth, we may pursue three main objectives:
first, to discover it when we search for it;
secondly, to prove it when we possess it;
thirdly, to discern it from falsehood when we examine it.

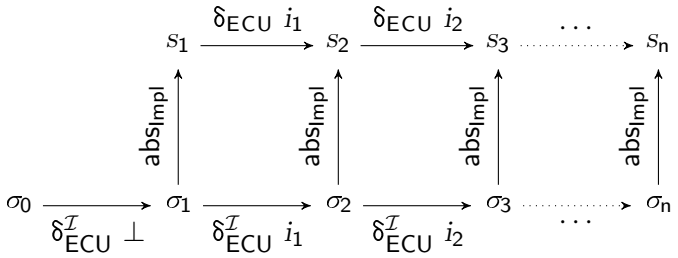

Figure 5.1: Simulation between implementation and model

of a kernel execution the implementation transition $\delta^{\mathcal{I}}_{ECU}$ internally might perform several C0 small-steps.

Figure 5.1 depicts the proof sketch of the implementation correctness for finite traces: The induction start is based on the fact that the successor state σ_1 after the first step at power-up can be abstracted via an abstraction function abs_{Impl} to an initial state s_1 of the ECU model. The induction step formalizes that the semantic effects of an implementation step $\delta^{\mathcal{I}}_{ECU}$ can be expressed by the abstract transition function δ_{ECU} and the abstraction function is preserved. Additionally, we collect all properties that have to hold during the kernel executions into a so-called implementation invariant.

In the remainder of this chapter, we establish the required framework for the implementation layer (Section 5.1). In Section 5.2 we relate the implementation states to corresponding abstract ECU states (as introduced in Section 4.3) by formalizing an abstraction function.

Based on all these prerequisites, we finally formulate our correctness theorem in Section 5.3. We show by induction, that the abstraction function and the implementation invariant hold after each kernel execution. Furthermore, we claim that the invariant is preserved for each transition of the ABC device. Putting all the results together, we are able to show the implementation correctness for finite traces for the overall transition function $\delta^{\mathcal{I}}_{ECU}$. This result has been presented in [DSS09].

5.1 Implementation Layer

In order to prove that our Olos implementation satisfies the abstract specification we first present our implementation in the Hoare logic environment provided by Isabelle/Simpl. Thus, a syntax translator automatically generates Simpl code from the source code given as a C0 program. In the next subsection we define Cvm primitives used by Olos in Simpl. Then, we present validity predicates on the Olos implementation state. These are used to formalize pre- and postconditions of Hoare triples to prove functional correctness of all Olos functions.

Afterwards, we define a Simpl function for a kernel execution that embeds the Olos implementation into a Cvm step. Finally, we show the implementation invariant of the entire implementation state.

5.1.1 Defining CVM Primitives in Simpl

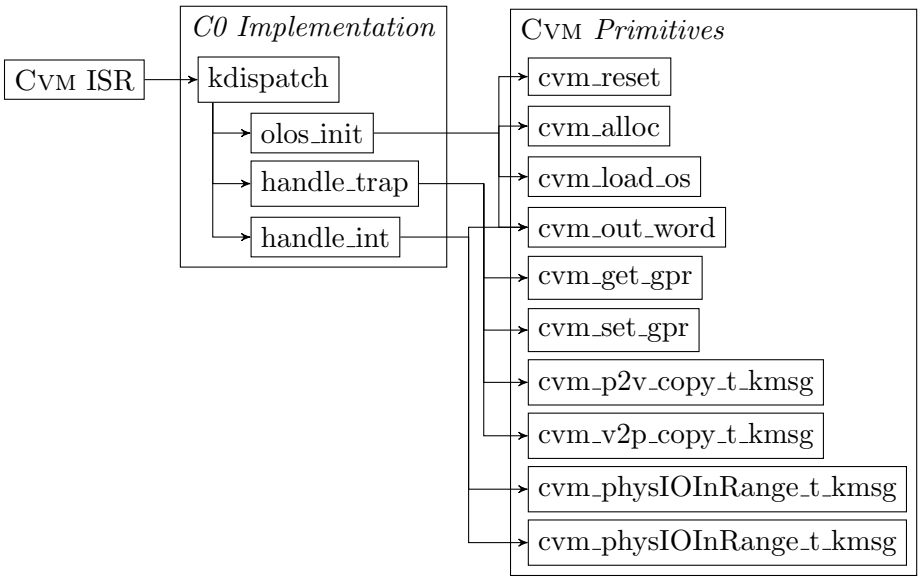

Figure 5.2: OLOS Call Graph

In contrast to a normal application program, an operating system is not entirely written in C. We have hidden all the hardware-specific assembly code within CVM primitives. OLOS functions use these primitives in order to exchange data with a certain device or user process. Figure 5.2 depicts a call graph of the CVM's interrupt service routine which gives an overview of the dependencies between all OLOS functions and the used CVM primitives. The code of CVM primitives has been verified on the ISA level of the processor [ST08]. Note that these code fragments cannot be proved in Isabelle/Simpl. Nevertheless in Simpl, we can express the effects of those low-level computations that are visible to the C0 programmer by their effect on external variables (recall XCalls in Section 2.1). First, we augment the state of the overall program with an additional program variable cvmX representing the external, hardware-specific state. cvmX consists of two components:

- fst σ.cvmX stores the information of CVM userprocesses which equals subcomponent cvm_up of a CVM state as introduced in Section 2.8.1 and,
- snd σ.cvmX is the device vector as described in (Section 2.7).

We define two functions in order to avoid long names in the definitions of the remaining chapter. We use cvm_apps σ.cvmX $\equiv$ (fst σ.cvmX).userprocesses
to consider only the user process vector and abbreviate the selection of the ABC device from the entire device vector by
cvm_abc σ.cvmX $\equiv$ snd σ.cvmX (nat2dev ABC_ID).

In Section 2.8.3 we introduced functions that specified the effects of CVM primitives in C0 small-step semantics. In the remaining chapter we reason about Simpl programs

using CVM primitives in the Hoare logic. Hence, we also require a specification of the CVM primitive semantics in Simpl as well. We have defined all primitives used by OLOS as transition functions with multiple assignments enclosed in BASIC and END. Afterwards these functions are embedded into procedure bodies. Then, the Vcg later on automatically unpacks the procedure bodies and hence, we can directly use the inlined function specification.

In Section 2.8.3 we have learned that CVM primitives can be classified according to their effects. They either access a component of the external state in order to read a value, manipulate the component or does both. Furthermore, we distinguish between typed and untyped primitives. We exemplarily introduce Simpl functions for different primitive cases.

First, we regard an untyped primitive that solely modifies the state of a user process. Therefore, we show the Simpl definition of the CVM primitive that initializes a user process (the underlying definitions are given in paragraph 2.8.3). Function fun_cvm_reset takes a process identifier pid as input parameter and result variable is given with res_int. The precondition pre_cvmReset ´pid must be fulfilled in order to guarantee successful execution of the primitive. Therefore, we guard the following command against a violation of this requirement. The updated user-process component is computed with function exec_cvmReset. Then, the new user process component and the functions result value 0 are assigned simultaneously to the first component of the external state cvmX and the return variable of the OLOS (by using the BASIC and END syntax). Formally:

```
procedures fun_cvm_reset (pid| res_int) =
{| pre_cvmReset ´pid |} ⟼
BASIC
LET (ext, rval) = (exec_cvmReset (fst ´cvmX) ´pid, 0)
IN ´cvmX :== (ext, snd ´cvmX), ´res_int :== rval
END
```

Note that all untyped primitive functions that solely manipulate the user process component can be defined in the same manner. More precisely, these functions are fun_cvm_alloc, fun_cvm_load_os and fun_cvm_set_gpr. Only their function signature, the guard and the manipulation function depend on the particular primitive.

Untyped CVM primitives that solely manipulate a device state update the second component of the external state cvmX instead. The external output to the is simply neglected. In particular, this solely concerns function fun_cvm_out_word.

Next, we show primitive fun_cvm_get_gpr that does not modify the external state but returns read data to the return variable res_nat of the kernel. We require that the precondition pre_cvmGetGPR of the primitive holds. The Simpl function directly returns the value of function exec_cvmGetGPR (cf. definitions in paragraph 2.8.3).

```
procedures fun_cvm_get_gpr (pid, reg | res_nat) =
{| pre_cvmGetGPR ´pid ´reg |} ⟼
´res_nat :=g exec_cvmGetGPR (fst ´cvmX) ´pid ´reg
```

This function is the only primitive that directly returns the read data to the result variable of the C0 program.

Finally, we show the Simpl function fun_cvm_physIOInRange_t_kmsg that modifies the external state and returns a message and the result value to the operating system. This CVM primitive is specific for OLOS and therefore typed. More specifically, the guards of typed primitives do not only contain the precondition of the CVM primitives (defined in paragraph 2.8.3) but additionally fix the length of the copied data.

fun_cvm_physIOInRange_t_kmsg updates simultaneously the modified component of the external state, the return value und the message buffer using the BASIC and END syntax. The return value of function exec_cvmPhysIOInRange is split and assigned internally to a changed device state *ext*, the list of outputs to the external environments *eifos* and the list of outputs to the processor which forms the message *msg*. Note that only the potentially updated state (the ABC state is not changed when its buffers are read) and the message are relevant in this function. Hence, reading data from the processor-facing buffers of a device with identifier dev_id starting at address port is defined as:

```
procedures fun_cvm_physIOInRange_t_kmsg (dev_id, port, msg | msg, res_int) =
{| pre_cvmPhysIORange ´dev_id ´port (length ´msg) ∧ length ´msg = MSGLENGTH |} ⟼
BASIC
LET ((ext, eifos, msg), rval) =
      (exec_cvmPhysIOInRange (snd ´cvmX) ´dev_id ´port MSGLENGTH, 0)
IN ´cvmX :== (fst ´cvmX, ext), ´res_int :== rval, ´msg :== msg
END
```

All typed CVM primitives additionally contain the requirement length ´msg = MSGLENGTH in the guard definition. fun_cvm_v2pcopy_t_kmsg resembles fun_cvm_get_gpr but returns a message value on the one hand and a result variable on the other hand to the kernel (cf. example in Section 2.9.1). The remaining typed primitives are fun_cvm_p2vcopy_t_kmsg and fun_cvm_physIOOutRange_t_kmsg. The former function manipulates the user process component whereas the latter one modifies the device component of the external state. While their guard differs from the guard of untyped primitives, the Simpl definition of their function bodies resembles the one of fun_cvm_reset.

In this subsection, we have seen that the definition of a CVM primitive in Simpl depends on certain aspects: the number of modified kernel variables, the class of component in the modified external variable and the distinction between typed and untyped CVM primitives. Note that the correctness of the CVM primitive specification has not be shown on Simpl level (see also Section 7.4).

5.1.2 Predicates of the OLOS Implementation State

In this subsection we present an initialization and a validity predicate for OLOS implementation states σ. These predicates are based on several constraints concerning e. g., the scheduling tables, message buffers or the devices. First we describe the requirements

of particular variables before we define the initialization and the validity predicate for the overall OLOS implementation state.

Tables and Constants. OLOS uses three scheduling tables that have to fulfill several constraints. Predicate valid_Schedule takes all tables as arguments and holds when all requirements on the tables are fulfilled. All table lengths are given by the number SLOTCOUNT of slots per round. Furthermore, the tables should only contain specified entries. Therefore, the application scheduling table AST must not exceed the maximal number of applications PROCCOUNT whereas entires of the buffer index table BT are bounded by MSGCOUNT, the number of specified message buffers. Formally:

valid_Schedule SPT AST BT $\equiv$
$|SPT| = \mathsf{SLOTCOUNT} \wedge$
$|AST| = \mathsf{SLOTCOUNT} \wedge$
$|BT| = \mathsf{SLOTCOUNT} \wedge$
$(\forall i{<}|AST|.\ 0 < AST\ !\ i \leq \mathsf{PROCCOUNT}) \wedge (\forall i{<}|BT|.\ BT\ !\ i < \mathsf{MSGCOUNT})$

During the initialization OLOS uses an array storing the number of pages per application in order to allocate sufficient memory. Predicate valid_AppMemAlloc formalizes several restrictions on the entries of that array. Each entry of $PAGEC$ stores the number of pages of the application with the corresponding identifier. Since 0 is reserved for the kernel, the first component is set to 0 and the array length is given by PROCCOUNT + 1. For the other entries we require that the number of pages is larger than 0 (i. e., all applications have allocated memory) and less than the maximum total virtual memory size given by TVM_MAXPAGES pages. Finally, the total sum of all the table entries should not exceed this upper bound. The definition of predicate valid_AppMemAlloc is given below:

valid_AppMemAlloc $PAGEC$ $\equiv$
$|PAGEC| = (\mathsf{PROCCOUNT} + 1) \wedge$
$(\forall i{\in}\{0{<}..{<}|PAGEC|\}.\ 0 < PAGEC\ !\ i \wedge PAGEC\ !\ i < \mathsf{TVM\_MAXPAGES}) \wedge$
list_sum $PAGEC \leq$ TVM_MAXPAGES

Some implementation variables remain unchanged after the initialization. The scheduling tables are initialized with values from corresponding configuration tables. The memory size of each application has been stored in one of its special purpose registers and the status register contains the interrupt mask. These constants are set after power-up and remain unchanged in all further executions. We summarize all constant values of an implementation state σ in predicate olos_consts:

olos_consts SPT BT AST $PAGEC$ s_{ups} $\equiv$
$SPT = \mathsf{SPT\_CONF} \wedge$
$BT = \mathsf{BT\_CONF} \wedge$
$AST = \mathsf{AST\_CONF} \wedge$
$(\forall i{\in}\{0{<}..{<}|PAGEC|\}.\ \mathsf{mm\_size}\ (s_{\mathsf{ups}}.\mathsf{userprocesses}\ (\mathsf{nat2pid}\ i)) = PAGEC\ !\ i) \wedge$
$s_{\mathsf{ups}}.\mathsf{statusreg} = 8254$

Message Buffers. The message buffers of an implementation state are realized as an array of message pointers *ps* pointing to messages stored in the heap. Recall that we use a heap function *heap* to refer directly to a message on the heap. Validity of message buffers demands for several restrictions. The number of OLOS message buffers is given by MSGCOUNT. Furthermore, we require these buffers to be distinct and different from Null. Each buffer contains only valid messages (i. e., predicate is_valid_intlistMsg holds). Finally, we summarize all constraints mentioned above in a validity predicate for message buffers valid_MB:

valid_MB *ps heap* ≡
  $|ps|$ = MSGCOUNT ∧
  distinct *ps* ∧
  Null ∉ set *ps* ∧ (∀*i*<MSGCOUNT. is_valid_intlistMsg (*heap* (*ps* ! *i*)))

After the initialization the message buffers should fulfill predicate init_MB. All message buffers should be valid and contain initial values (i. e., messages are initialized with an array of MSGLENGTH zeros). Formally:

init_MB *ps heap* ≡
  valid_MB *ps heap* ∧
  (∀*i*<MSGCOUNT. *heap* (*ps* ! *i*) = replicate MSGLENGTH 0)

Devices. In the OLOS implementation we use two kinds of devices - the hard disk and the ABC device. After power-up the boot region of the former device is accessed to load an OS image to the memory of an application and the latter is responsible for the bus communication. The device identifiers of both devices should correspond directly to a hard disk state and an ABC state respectively. This constraint is hidden in predicate olos_devices:

olos_devices s_{dev} ≡
  is_conf_ABC (s_{dev} (nat2dev ABC_ID)) ∧
  is_conf_HD (s_{dev} (nat2dev DEVICE_HD1))

In Section 4.1 we have specified several conditions for the ABC state. Now, we consider all used devices and their state requirements at three different points in time:

1. **After power-up before the OLOS initialization**
   The predicate devices_before_olosinit encapsulates preconditions of all devices that should hold before OLOS starts the initialization:

   devices_before_olosinit s_{dev} ≡
     olos_devices s_{dev} ∧
     abc_before_olosinit (s_{dev} (nat2dev ABC_ID)) ∧
     is_validconf_hd (s_{dev} (nat2dev DEVICE_HD1)) ∧
     is_valid_disk_content (s_{dev} (nat2dev DEVICE_HD1)) ∧
     list_sum PAGEC_CONF · PAGE_SIZE
     ≤ |(s_{dev} (nat2dev DEVICE_HD1)).hd_sm|

We assume that OLOS 'knows' both devices and that the ABC device fulfills the requirements mentioned above. Furthermore, the state and the content of the hard disk should be valid and the sum of all application pages with size PAGE_SIZE has to fit into the memory of the hard disk.

2. **After OLOS initialization**
   After initialization, predicate initial_devices should hold. It ensures olos_devices and states that OLOS initialized the configuration registers of the ABC device with values from CB_CONF. Moreover, the ABC device must be in an initial state. Formally:

   $$\begin{aligned}&\mathsf{initial\_devices}\ s_{\mathsf{dev}} \equiv\\ &\quad \mathsf{olos\_devices}\ s_{\mathsf{dev}} \wedge\\ &\quad (s_{\mathsf{dev}}\ (\mathsf{nat2dev\ ABC\_ID})).\mathsf{CR} = \mathsf{CB\_CONF} \wedge\\ &\quad \mathsf{is\_initial\_ABC}\ (s_{\mathsf{dev}}\ (\mathsf{nat2dev\ ABC\_ID}))\end{aligned}$$

3. **After all transitions** A valid device component requires that predicate olos_devices has to be fulfilled. Furthermore, the configuration registers of the ABC device remain unchanged (i. e., they store the initial values of CB_CONF) and the ABC device is valid. When all conditions above are satisfied the validity predicate of the device component valid_devices holds:

   $$\begin{aligned}&\mathsf{valid\_devices}\ s_{\mathsf{dev}} \equiv\\ &\quad \mathsf{olos\_devices}\ s_{\mathsf{dev}} \wedge\\ &\quad (s_{\mathsf{dev}}\ (\mathsf{nat2dev\ ABC\_ID})).\mathsf{CR} = \mathsf{CB\_CONF} \wedge\\ &\quad \mathsf{is\_valid\_ABC}\ (s_{\mathsf{dev}}\ (\mathsf{nat2dev\ ABC\_ID}))\end{aligned}$$

Relating Redundant Components. Recall that the implementation state of OLOS includes an own send flag and a current slot number. Therefore, we specify how these variables are related to their counterparts in the ABC state. We compare the components when we enter and leave the OLOS main function. Certainly, the redundant information must not be contradictory. We introduce a predicate olos_abc_consis that takes the send flag and the current slot number of our implementation state and the ABC state. We require that the send flags have to be equal. The current slot numbers, on the contrary, are not equal in every situation. More specifically, they are shifted when OLOS starts a new slot (note that the send flag is not set $\neg\ sf$). The ABC device receives a message or a timer signal from the bus and immediately increases its slot number and raises the interrupt flag $s_{\mathsf{abc}}.\mathsf{INT}$. Afterwards, the operating system reads the raised interrupt flag and starts with the receive phase. Hence, the implementation variable storing the slot number has a delay to its ABC counterpart. We formalize this gap by increasing the implementation variable artificially when the variables encode a receive phase. Otherwise, the current slot numbers are equal. Formally:

$$\mathsf{olos\_abc\_consis}\ sf\ csn\ s_{\mathsf{abc}} \equiv$$

$sf = s_{\mathsf{abc}}.\mathsf{SF} \wedge$
$s_{\mathsf{abc}}.\mathsf{CSN} = (csn + \textbf{if } s_{\mathsf{abc}}.\mathsf{INT} \wedge \neg\, sf \textbf{ then } 1 \textbf{ else } 0) \bmod \mathsf{SLOTCOUNT}$

Initial OLOS Implementation States. With all prerequisites in place we summarize all the conditions of an implementation state after power-up in predicate $\mathsf{is\_initState_i}$. For an initial implementation state, we require that the constant variables are initialized according to olos_consts, the table storing the number of pages for each application equals PAGEC_CONF, the slot number is SLOTCOUNT − 1, variable σ.ca is set to IDLE and the send flag is disabled. Furthermore, the message buffers and the devices should be in their initial states, all applications are valid (recall Section 2.8.1) and redundant components are consistent (i. e., olos_abc_consis holds). Formally:

$\mathsf{is\_initState_i}\ \sigma \equiv$
$\mathsf{olos\_consts}\ \sigma.\mathsf{SPT}\ \sigma.\mathsf{BT}\ \sigma.\mathsf{AST}\ \sigma.\mathsf{PAGEC}\ (\mathsf{fst}\ \sigma.\mathsf{cvmX}) \wedge$
$\sigma.\mathsf{PAGEC} = \mathsf{PAGEC\_CONF} \wedge$
$\sigma.\mathsf{csn} = \mathsf{SLOTCOUNT} - 1 \wedge$
$\sigma.\mathsf{ca} = \mathsf{IDLE} \wedge$
$\neg\ \sigma.\mathsf{sendflag} \wedge$
$\mathsf{init\_MB}\ \sigma.\mathsf{MB}\ \sigma.\mathsf{heap\_msg} \wedge$
$\mathsf{is\_valid\_cvmup}\ (\mathsf{fst}\ \sigma.\mathsf{cvmX}) \wedge$
$\mathsf{initial\_devices}\ (\mathsf{snd}\ \sigma.\mathsf{cvmX}) \wedge$
$\mathsf{olos\_abc\_consis}\ \sigma.\mathsf{sendflag}\ \sigma.\mathsf{csn}\ (\mathsf{cvm\_abc}\ \sigma.\mathsf{cvmX})$

Valid OLOS Implementation States. There are several restrictions on implementation states that should hold for all further transitions. These requirements are encapsulated in $\mathsf{is\_validState_i}$. A valid implementation state includes that constant variables remain unchanged after their initialization. Furthermore, we require that variable σ.ca either equals IDLE or stores the identifier of the currently scheduled application. The current slot number is valid i. e., the value of variable σ.csn is less than the fixed number of slots SLOTCOUNT per round. Moreover, the table σ.PAGEC, the message buffers, the applications and the devices must be valid and olos_abc_consis holds additionally. The definition of the validity predicate for OLOS implementation states $\mathsf{is\_validState_i}$ is given below:

$\mathsf{is\_validState_i}\ \sigma \equiv$
$\mathsf{olos\_consts}\ \sigma.\mathsf{SPT}\ \sigma.\mathsf{BT}\ \sigma.\mathsf{AST}\ \sigma.\mathsf{PAGEC}\ (\mathsf{fst}\ \sigma.\mathsf{cvmX}) \wedge$
$(\sigma.\mathsf{ca} = \sigma.\mathsf{AST}\ !\ \sigma.\mathsf{csn} \vee \sigma.\mathsf{ca} = \mathsf{IDLE}) \wedge$
$\sigma.\mathsf{csn} < \mathsf{SLOTCOUNT} \wedge$
$\mathsf{valid\_AppMemAlloc}\ \sigma.\mathsf{PAGEC} \wedge$
$\mathsf{valid\_MB}\ \sigma.\mathsf{MB}\ \sigma.\mathsf{heap\_msg} \wedge$
$\mathsf{is\_valid\_cvmup}\ (\mathsf{fst}\ \sigma.\mathsf{cvmX}) \wedge$
$\mathsf{valid\_devices}\ (\mathsf{snd}\ \sigma.\mathsf{cvmX}) \wedge$
$\mathsf{olos\_abc\_consis}\ \sigma.\mathsf{sendflag}\ \sigma.\mathsf{csn}\ (\mathsf{cvm\_abc}\ \sigma.\mathsf{cvmX})$

5.1.3 Functional Correctness of OLOS Functions

Now we draw the readers attention back to the generated Simpl code of our OLOS implementation. In Section 2.9, we have defined the notation of a Hoare triple in order to prove total correctness. For each OLOS function c in procedure environment Γ where the precondition P holds for the current state we have to discharge postcondition Q for the successor state.

In the following, we try to restrict the preconditions on the smallest set of assumptions and to use as few variables as possible. In contrast, the postcondition should include the largest set of properties that is possible to conclude. Furthermore, when a variable is not modified in a special case of the function, we explicitly note this in the predicate. Then, properties of unchanged variables are excluded a priori. In a large proof, this permits to concentrate only on proof goals concerning changed variables.

A modifies clause equals a record update specification and enumerates all globals that are changed during the execution of a function. If the verification generator works on a procedure call it checks whether it can find a modifies clause in the context. If one is present the procedure call is internally simplified before the Hoare rules are applied.

In the following, we present the pre- and postconditions for each OLOS function and the corresponding Hoare triples.

Function fun_olos_init**.** The precondition of function fun_olos_init restricts the number of used message buffers to MSGCOUNT. No application has allocated virtual memory yet i. e., the sum of used pages equals 0. The external state variable cvmX has been initialized: CVM set up the user process component, whereas the device vector fulfills some initial assumptions. Formally, the precondition $\text{olos\_init}_{\text{pre}}$ summarizes:

$$
\begin{aligned}
&\text{olos\_init}_{\text{pre}}\ \sigma \equiv \\
&\quad |\sigma.\text{MB}| = \text{MSGCOUNT} \wedge \\
&\quad (\textstyle\sum i{\in}\text{procnumT}.\ \text{mm\_size}\ ((\text{fst}\ \sigma.\text{cvmX}).\text{userprocesses}\ (\text{nat2pid}\ i))) = 0 \wedge \\
&\quad \text{fst}\ \sigma.\text{cvmX} = \text{init\_cvmup} \wedge \text{devices\_before\_olosinit}\ (\text{snd}\ \sigma.\text{cvmX})
\end{aligned}
$$

After the execution of the initialization function fun_olos_init the successor state τ equals the initial state $\text{is\_initState}_{\text{i}}$ and all table contents are well-formed i. e., the predicates valid_Schedule and valid_AppMemAlloc hold. The postcondition of fun_olos_init is denoted with:

$$
\begin{aligned}
&\text{olos\_init}_{\text{post}}\ \tau \equiv \\
&\quad \text{is\_initState}_{\text{i}}\ \tau \wedge \text{valid\_Schedule}\ \tau.\text{SPT}\ \tau.\text{AST}\ \tau.\text{BT} \wedge \text{valid\_AppMemAlloc}\ \tau.\text{PAGEC}
\end{aligned}
$$

Now, we can formalize the Hoare triple for the OLOS initialization function.

Theorem 5.1 (Functional Correctness of fun_olos_init)**.** *For an implementation state σ that fulfills the precondition* $\text{olos\_init}_{\text{pre}}$ *we can conclude that the successor state τ satisfies the postcondition predicate* $\text{olos\_init}_{\text{post}}$ *after the execution of* fun_olos_init*. Furthermore, the result variable of the function is set to* 0*. Formally:*

$\Gamma \vdash_t \{\!|\sigma.\ \mathsf{olos\_init}_{\mathsf{pre}}\ \sigma|\!\}\ \acute{}\mathsf{res\_int} :== \mathsf{PROC\ fun\_olos\_init}()$
$\{\tau.\ \mathsf{olos\_init}_{\mathsf{post}}\ \tau \wedge \tau.\mathsf{res\_int} = 0\}$

Proof. In Figure 3.7 on page 66 we divided the implementation code into three logical parts each containing one loop. We keep this division for our correctness proof in order to keep neat proof goals. Thus, we obtain several subgoals in order to establish invariants for the loops from the premises and conditions we gained so far, to discharge the postconditions of a loop and to collect all conclusions to the main postcondition. We split this proof into three subgoals each containing one loop:

1. Initialization of OLOS data structures:
   From the precondition we can directly conclude that $\mathsf{init\_cvmup.statusreg} = 8254$ holds by unfolding init_cvmup. Moreover, the scheduling tables and the OLOS variables PAGEC, ca, csn and sendflag are set to the desired values. valid_Schedule and valid_AppMemAlloc hold after the initialization of the scheduling tables (we obtain this result by unfolding the initial table configurations). Note that all results derived so far still hold in the postcondition unless the involved components are not changed during the execution of the remaing function. The invariant of the loop initializing the message buffers now includes the following constraints:
   a) the preconditions $\mathsf{olos\_init}_{\mathsf{pre}}$
   b) the already derived results
   c) for each loop cycle $n < \mathsf{MSGCOUNT}$ we require that all buffers initialized so far are different from Null, their value is initial and that they are distinct i. e.,
   $\forall i{<}n.\ \sigma.\mathsf{MB}\ !\ i \neq \mathsf{Null} \wedge$
   $\sigma.\mathsf{heap\_msg}\ (\sigma.\mathsf{MB}\ !\ i) = \mathsf{replicate\ MSGLENGTH}\ 0 \wedge$
   $(\forall j{<}i.\ \sigma.\mathsf{MB}\ !\ i \neq \sigma.\mathsf{MB}\ !\ j)$
   The generated NEW command always creates internally new pointers that are disjoint from the already existing ones and the null pointer Null. When these constraints are fullfilled for each loop cycle obviously init_MB holds after the loop execution. We abbreviate all results gained so far with postcond_loop1.
2. Application initialization:
   The second loop initializes the applications and claims in its invariant that the requirements below hold:
   a) the preconditions $\mathsf{olos\_init}_{\mathsf{pre}}$
   b) the already derived results postcond_loop1
   c) in all loop cycles $n \leq \mathsf{PROCCOUNT}$ starting from 1 additionally holds:
      i. variable *next* containts the total number of pages allocated so far:
      $next = \mathsf{list\_sum}\ (\mathsf{take}\ n\ \mathsf{PAGEC\_CONF})$,
      ii. all applications initialized so far have the correct size:
      $\forall i.\ 0 < i \wedge i < n \longrightarrow \mathsf{mm\_size}\ ((\mathsf{fst}\ \sigma.\mathsf{cvmX}).\mathsf{userprocesses}\ (\mathsf{nat2pid}\ i)) = \sigma.\mathsf{PAGEC}\ !\ i$,
      iii. all page sizes of PAGEC_CONF are less or equal than the maximal number of pages TVM_MAXPAGES:
      $\mathsf{list\_sum\ PAGEC\_CONF} \leq \mathsf{TVM\_MAXPAGES}$, and
      iv. the sum of the already allocated pages and the pages that are not yet

allocated is less or equal TVM_MAXPAGES:
$(\sum i{\in}\mathsf{procnumT}.\ \mathsf{mm\_size}\ ((\mathsf{fst}\ \sigma.\mathsf{cvmX}).\mathsf{userprocesses}\ (\mathsf{nat2pid}\ i))) +$
$\mathsf{list\_sum}\ (\mathsf{drop}\ n\ \mathsf{PAGEC\_CONF})$
$\leq \mathsf{TVM\_MAXPAGES}$

When the invariant holds for each loop cycle finally olos_consts holds. Note that the user-process component is modified in each loop cycle by applying several CVM primitives. Therefore, we have to discharge loop invariant requirements and the corresponding primitive preconditions on the one hand and to prove that validity of the user-process component is_valid_cvmup is preserved after each application update on the other hand. All results derived until now are abbreviated in postcond_loop2.

3. ABC configuration:
   Finally, the last loop writes the configuration registers of the ABC device. The loop invariant includes the following conditions:
   a) the precondition devices_before_olosinit holds,
   b) the postcondition postcond_loop2 is satisfied, and
   c) in each loop cycle $n < \mathsf{SLOTCOUNT}$ the registers configured so far store the corresponding value of initial table configuration CB_CONF:
   $\forall i.\ 0 \leq i \wedge i < n \longrightarrow (\mathsf{cvm\_abc}\ \sigma.\mathsf{cvmX}).\mathsf{CR}\ !\ i = \mathsf{CB\_CONF}\ !\ i$

   olos_devices can be derived directly from precondition devices_before_olosinit. The loop only modifies the device component of the external state variable so that the other conditions still hold after the loop execution. OLOS uses the CVM primitive fun_cvm_out_word to write the configuration registers of the ABC. Hence, we have to discharge the precondition pre_cvmOutWord for each cycle. Due to the satisfied loop invariant $(\mathsf{cvm\_abc}\ \sigma.\mathsf{cvmX}).\mathsf{CR} = \mathsf{CB\_CONF}$ holds after the loop execution. Finally, OLOS sends the set-ready command to the ABC device using fun_cvm_out_word. Internally the ABC state is updated with the internal device function δ^m_{abc} (recall Section 4.1). It remains to discharge predicate is_initial_ABC after all manipulations of the ABC device: All modifications so far left the interrupt flag and the send and receive buffers unchanged so that we conclude
   $\neg\ (\mathsf{cvm\_abc}\ \sigma.\mathsf{cvmX}).\mathsf{INT} \wedge \mathsf{valid\_ABC\_buffers}\ (\mathsf{cvm\_abc}\ \sigma.\mathsf{cvmX})$ from predicate abc_before_olosinit. In case of an set-ready command δ^m_{abc} sets the initialization flag, the send flag and the current slot number of the ABC device to its initial values (i. e., $\neg\ (\mathsf{cvm\_abc}\ \sigma.\mathsf{cvmX}).\mathsf{IP} \wedge \neg\ (\mathsf{cvm\_abc}\ \sigma.\mathsf{cvmX}).\mathsf{SF} \wedge (\mathsf{cvm\_abc}\ \sigma.\mathsf{cvmX}).\mathsf{CSN} = \mathsf{SLOTCOUNT} - 1$). Putting all partial results together we have shown that the successor implementation state τ is actually initial.

□

Function fun_handle_int**.** The precondition of function fun_handle_int demands for the validity of message buffers, the scheduling tables and the current slot number. The variable ca should not be greater than PROCCOUNT. Formally:

$\mathsf{handle\_int}_{\mathsf{pre}}\ \sigma \equiv$
$\mathsf{valid\_MB}\ \sigma.\mathsf{MB}\ \sigma.\mathsf{heap\_msg} \wedge$

$$\mathsf{valid\_Schedule}\ \sigma.\mathsf{SPT}\ \sigma.\mathsf{AST}\ \sigma.\mathsf{BT} \wedge \sigma.\mathsf{csn} < \mathsf{SLOTCOUNT} \wedge \sigma.\mathsf{ca} \leq \mathsf{PROCCOUNT}$$

We formulate a postcondition predicate $\mathsf{handle\_int}_{\mathsf{post}}$ to check whether the successor state after the execution of fun_handle_int fulfills the desired properties. Therefore, it takes the old and the new implementation state (i. e., state σ before and τ after the execution) and compares the components of the new state with the values we expect after the execution of fun_handle_int. Variables that may be modified by fun_handle_int are: ca, the current slotnumber csn, the send flag sendflag, the message values heap_msg, and the external state cvmX, more specifically the device component.

Recall that this function is responsible for the execution of the send and receive phase (Section 3.3.4). Thus, the effects on the new implementation state depend on the old value of the send flag σ.sendflag.

1. σ.sendflag:
   A raised send flag indicates the start of the send phase, where OLOS sends data from its message buffer to the ABC device. In this case, we expect that the current slot number and the message values of the new state τ are not changed. Moreover, after the send phase the new send flag τ.sendflag should be set to False and variable τ.ca equals IDLE. We require that the device component of the new implementation state has an updated send buffer and a cleared interrupt flag. We use function exec_cvmPhysIOOutRange to express how the old device component is affected when OLOS sends a message to the ABC device (recall Section 2.8.3). Then, function exec_cvmOutWord returns the expected device component after the updated device component $\mathit{devs}_{\mathsf{s}}$ has cleared its interrupt flag. This result should be equal to the device component of the new state τ.

2. $\neg$ σ.sendflag:
   In the case the send flag is not set we are in the receive phase. Here, we require that new implementation state τ has an increased current slot number, its send flag equals the entry of the sending permission table SPT in the next slot and variable τ.ca is set to the entry of the application scheduling table AST in the current slot. We express the effects of OLOS reading the receive buffer of the ABC with function exec_cvmPhysIOInRange and assign the results to a new state $\mathit{devs}_{\mathsf{r}}$, the output list to the external environment *eifos* and the read message *msg*. Then, we expect that only the data of the designated message buffer is updated with the message *msg* from the ABC's receive buffer (recall Section 3.3.4 that the buffer index is determined from the entry of the buffer index table BT in the previous slot). The other messages remain unchanged. Finally, we require the interrupt flag of the ABC device has been cleared. Thus, we model this effect by applying function exec_cvmOutWord on the device component $\mathit{devs}_{\mathsf{r}}$. After all manipulations the variables ca and csn should be valid.

If state τ fulfills all desired modifications it satisfies the postcondition $\mathsf{handle\_int}_{\mathsf{post}}$. Formally:

```
handle_int_post σ τ ≡
 if σ.sendflag
   then let devs_s =
              fst (exec_cvmPhysIOOutRange (snd σ.cvmX) ABC_ID SEND_PORT
                   (σ.heap_msg
                    (σ.MB ! (σ.BT ! ((σ.csn + 1) mod SLOTCOUNT)))))
        in τ.csn = σ.csn ∧
           τ.sendflag = False ∧
           τ.ca = IDLE ∧
           τ.heap_msg = σ.heap_msg ∧
           τ.cvmX =
           (fst σ.cvmX,
            fst (exec_cvmOutWord devs_s ABC_ID COMR_PORT CLEARINT_COM))
 else let (devs_r, eifos, msg) =
            exec_cvmPhysIOInRange (snd σ.cvmX) ABC_ID RECV_PORT MSGLENGTH
      in τ.csn = (σ.csn + 1) mod SLOTCOUNT ∧
         τ.sendflag = σ.SPT ! ((τ.csn + 1) mod SLOTCOUNT) ∧
         τ.ca = σ.AST ! τ.csn ∧
         (∀i. τ.heap_msg i =
              if i = σ.MB ! (σ.BT ! ((SLOTCOUNT + τ.csn − 1) mod SLOTCOUNT))
                then msg
              else σ.heap_msg i) ∧
         τ.cvmX =
         (fst σ.cvmX,
          fst (exec_cvmOutWord devs_r ABC_ID COMR_PORT CLEARINT_COM)) ∧
 τ.csn < SLOTCOUNT ∧ τ.ca ≤ PROCCOUNT
```

Now, we can formalize a theorem stating the functional correctness of the OLOS function fun_handle_int.

Theorem 5.2 (Functional Correctness of fun_handle_int)**.** *Assuming that an implementation state σ fulfills predicate* handle_int$_{pre}$*, we can conclude that after the execution of function* fun_handle_int *the successor state τ satisfies the postcondition predicate* handle_int$_{post}$ *and the function result variable is set to* 0*. Formally:*

$$\Gamma \vdash_t \{\!|\sigma.\ \text{handle\_int}_{pre}\ \sigma|\!\}\ \text{´res\_int} :== \text{PROC fun\_handle\_int}()\ \{\tau.\ \text{handle\_int}_{post}\ \sigma\ \tau \wedge \tau.\text{res\_int} = 0\}$$

Proof. For a successful CVM primitive execution we have to discharge the preconditions of all used functions. This includes the validity of the device identifiers and the ports. The identifier of our ABC device is less than the maximal device number, hence it is valid Section 2.7. With the fixed message length of MSGLENGTH ≡ 40 words the messages fit into the send and the receive buffer. From the discharged preconditions we know that the CVM primitives can be executed successfully. The Simpl definitions of the used primitives manipulate the external state variable with the underlying exec_

<primitivename> functions. Hence, we conclude that the updated device component of the successor state fulfills the desired constraints. Examining the executed code Figure 3.8 on page 67 on the one hand and the postcondition $\mathsf{handle\_int}_{\mathsf{post}}$ on the other hand, we see that the OLOS variables indeed are set to the correct values. The computations on the current slot number are modulo operations, hence the number is always less than SLOTCOUNT. Variable ca also can not be larger than PROCCOUNT, since it is either IDLE $\equiv 0$ or an entry from the application scheduling table. From the table validity, we can conclude that all values in AST are less or equal to PROCCOUNT. □

Function fun_handle_trap**.** The precondition of function fun_handle_trap that handles system calls takes the global variables of the implementation state σ and additionally the parameter n of the function. We require that the function is only invoked with specified system calls while an application is scheduled (i. e., σ.ca $\neq$ IDLE). Furthermore, the current slot number, the message buffers and all tables should be valid. Finally, the entries of PAGEC should be less or equal to the number of allocated pages in the memory of the corresponding user process. These requirements are summarized in $\mathsf{handle\_trap}_{\mathsf{pre}}$:

$$
\begin{aligned}
&\mathsf{handle\_trap}_{\mathsf{pre}}\ \sigma\ n \equiv \\
&\quad n \in \{\mathsf{CALL\_SEND}, \mathsf{CALL\_RECV}, \mathsf{CALL\_EXFINISHED}\} \wedge \\
&\quad (0 < \sigma.\mathsf{ca} \wedge \sigma.\mathsf{ca} \leq \mathsf{PROCCOUNT}) \wedge \\
&\quad \sigma.\mathsf{csn} < \mathsf{SLOTCOUNT} \wedge \\
&\quad \mathsf{valid\_MB}\ \sigma.\mathsf{MB}\ \sigma.\mathsf{heap\_msg} \wedge \\
&\quad \mathsf{valid\_Schedule}\ \sigma.\mathsf{SPT}\ \sigma.\mathsf{AST}\ \sigma.\mathsf{BT} \wedge \\
&\quad \mathsf{valid\_AppMemAlloc}\ \sigma.\mathsf{PAGEC} \wedge \\
&\quad (\forall i{\in}\{0{<}..{<}|\sigma.\mathsf{PAGEC}|\}. \\
&\qquad \mathsf{address\_in\_mem}\ (\mathsf{fst}\ \sigma.\mathsf{cvmX})\ i\ (\sigma.\mathsf{PAGEC}\ !\ i \cdot \mathsf{PAGE\_SIZE} - 1))
\end{aligned}
$$

The postcondition of function fun_handle_trap includes the effects on the changed global state variables after the function execution. The function only modifies the user-process component of the external variable cvmX, variable ca and a message value heap_msg. Similar to the postcondition predicate in the paragraph before, we describe the desired effects on the old implementation state σ and compare it with the successor state τ we obtain after calling function fun_handle_trap. When the postcondition $\mathsf{handle\_trap}_{\mathsf{post}}$ is satisfied the trap handler works in the expected way.

We require that the variables ca and csn fulfill their wellformedness constraints after the execution of function fun_handle_trap. The effects on the other components depend on the particular call that is determined by the trap number n. We distinguish between the different numbers:

1. $n =$ CALL_EXFINISHED:
   In this case the application signals its termination for the current slot. Here, we simply expect that ca is set to IDLE whereas the other components remain unchanged.
2. $n \neq$ CALL_EXFINISHED:
   Otherwise, ca will not be changed. The other system calls are used for message

transmission between OLOS and the calling application. Recall Section 3.3.5, that these calls use two registers (register 11 and 12) storing their parameter values and one register (register 22) that keeps the result value of the call. We load both parameter values with function exec_cvmGetGPR from the general-purpose registers into the variables *sa* and *mn*. Moreover, updating a user-process component by setting the result register to a value is provided by function exec_cvmSetGPR. We abbreviate this function in *set_res*. Now, we check the parameter values of the system calls:

- invalid system call parameters:
  When either the start address is not aligned (i. e., $(sa \wedge_u 3) \neq 0$), the entire message does not fit into allocated memory of the application (i. e., σ.PAGEC ! σ.ca $\cdot$ PAGE_SIZE $\leq sa + 4 \cdot$ MSGLENGTH) or the message number tries to access a non-existing message buffer (i. e., MSGCOUNT $\leq mn$) all messages remain unchanged. Depending on the invalid parameter, OLOS writes the corresponding error value in the result register of the application.
- valid system call parameters:
  Then, we require that OLOS writes 0 into the result register of the application to indicate a successful message transmission. Furthermore, when the application sent a write request (n = CALL_SEND) to OLOS, we expect that the data in the designated message buffer is updated with a message from the application. Function exec_cvmV2Pcopy computes the data that the application writes to the kernel. Otherwise, the application receives a message from the kernel. We model the user-process component after OLOS wrote a message into the application's memory with exec_cvmP2Vcopy.

If the successor state τ fulfills the desired expectations after function fun_handle_trap the postcondition handle_trap$_{post}$ holds. Formally, all requirements for global variables of an implementation state σ, the successor state τ and the function parameter n are encapsulated in predicate handle_trap$_{post}$:

```
handle_trap_post σ τ n ≡
 (let sa = exec_cvmGetGPR (fst σ.cvmX) σ.ca 11;
      mn = exec_cvmGetGPR (fst σ.cvmX) σ.ca 12;
      set_res = λres ups. exec_cvmSetGPR ups σ.ca 22 (to_nat32 res)
  in if n = CALL_EXFINISHED
       then τ.cvmX = σ.cvmX ∧ τ.ca = IDLE ∧ τ.heap_msg = σ.heap_msg
     else τ.ca = σ.ca ∧
          if σ.PAGEC ! σ.ca · PAGE_SIZE ≤ sa + 4 · MSGLENGTH ∨ (sa ∧_u 3) ≠ 0
            then τ.cvmX = (set_res INVALID_POINTER (fst σ.cvmX), snd σ.cvmX) ∧
                 τ.heap_msg = σ.heap_msg
          elsif MSGCOUNT ≤ mn
                then τ.cvmX = (set_res INVALID_MSGNUM (fst σ.cvmX), snd σ.cvmX) ∧
                     τ.heap_msg = σ.heap_msg
          elsif n = CALL_SEND
                then τ.cvmX = (set_res 0 (fst σ.cvmX), snd σ.cvmX) ∧
```

$$
\begin{array}{l}
\qquad\qquad\qquad\quad (\forall i.\ \tau.\mathsf{heap\_msg}\ i = \\
\qquad\qquad\qquad\qquad \textbf{if}\ i = \sigma.\mathsf{MB}\ !\ mn \\
\qquad\qquad\qquad\qquad\quad \textbf{then}\ \mathsf{exec\_cvmV2Pcopy}\ (\mathsf{fst}\ \sigma.\mathsf{cvmX})\ \sigma.\mathsf{ca}\ sa\ \mathsf{MSGLENGTH} \\
\qquad\qquad\qquad\qquad \textbf{else}\ \sigma.\mathsf{heap\_msg}\ i) \\
\qquad\quad \textbf{else}\ \tau.\mathsf{cvmX} = \\
\qquad\qquad (set\_res\ 0 \\
\qquad\qquad\quad (\mathsf{exec\_cvmP2Vcopy}\ (\mathsf{fst}\ \sigma.\mathsf{cvmX})\ \sigma.\mathsf{ca}\ sa\ (\sigma.\mathsf{heap\_msg}\ (\sigma.\mathsf{MB}\ !\ mn))), \\
\qquad\qquad\ \mathsf{snd}\ \sigma.\mathsf{cvmX}) \wedge \\
\qquad\qquad \tau.\mathsf{heap\_msg} = \sigma.\mathsf{heap\_msg}) \wedge \\
\tau.\mathsf{csn} < \mathsf{SLOTCOUNT} \wedge \tau.\mathsf{ca} \leq \mathsf{PROCCOUNT}
\end{array}
$$

The functional correctness of the OLOS function `handle_trap` is now formalized with the pre- and postconditions we defined above.

Theorem 5.3 (Functional Correctness of fun_handle_trap)**.** *We assume for an implementation state σ that the preconditions* $\mathsf{handle\_trap}_{\mathsf{pre}}$ *hold. After the execution of* fun_handle_trap *we obtain a state τ that fulfills the postconditions* $\mathsf{handle\_trap}_{\mathsf{post}}$ *and the result variable is set to* 0*. Formally:*

$$
\begin{array}{l}
\Gamma \vdash_t \{\!|\sigma.\ \mathsf{handle\_trap}_{\mathsf{pre}}\ \sigma\ \acute{}\mathsf{n}|\!\}\ \acute{}\mathsf{res\_int} :== \mathsf{PROC}\ \mathsf{fun\_handle\_trap}(\acute{}\mathsf{n}) \\
\qquad \{\tau.\ \mathsf{handle\_trap}_{\mathsf{post}}\ \sigma\ \tau\ \sigma.\mathsf{n} \wedge \tau.\mathsf{res\_int} = 0\}
\end{array}
$$

Proof. First, we want to show that the preconditions of all used CVM primitives hold. The used registers 11, 12 and 22 are all in range. Furthermore, the process identifier should be less then PID_MAX which is due to transitivity obviously the case (PROCCOUNT < PID_MAX). The message copying primitives are only executed when the entire message fits into the allocated memory. The Simpl definitions of the used primitives modify the external state variable with the underlying exec_<primitivename> functions. Hence, we derive that the manipulated user-process component of successor state τ fulfills the expected behaviour. The current slotnumber has not been changed, hence it is less than SLOTCOUNT. The same considerations as in the proof Theorem 5.2 on page 128 lead to the fact that $\tau.\mathsf{ca} \leq \mathsf{PROCCOUNT}$. □

Function fun_kdispatch**.** The precondition predicate $\mathsf{kdispatch}_{\mathsf{pre}}$ of OLOS main function combines some preconditions on state σ. Additionally, it takes variable *eca* storing the exception cause as a parameter. In the case that a reset occurs (i. e., $(eca \wedge_{\mathsf{u}} 1) \neq 0$), we assume that the precondition of the initialization function $\mathsf{olos\_init}_{\mathsf{pre}}$ holds. Otherwise, we require that the state is valid and that a pending device interrupt in variable *eca* can be related to a raised interrupt flag in the ABC device. Furthermore, when a trap occurs variable ca should not be IDLE. Formally:

$$
\begin{array}{l}
\mathsf{kdispatch}_{\mathsf{pre}}\ \sigma\ eca \equiv \\
\quad \textbf{if}\ (eca \wedge_{\mathsf{u}} 1) \neq 0\ \textbf{then}\ \mathsf{olos\_init}_{\mathsf{pre}}\ \sigma \\
\quad \textbf{else}\ \mathsf{is\_validState}_{\mathsf{i}}\ \sigma \wedge \\
\qquad ((eca \wedge_{\mathsf{u}} 8192) \neq 0) = (\mathsf{cvm\_abc}\ \sigma.\mathsf{cvmX}).\mathsf{INT} \wedge \\
\qquad ((eca \wedge_{\mathsf{u}} 32) \neq 0 \longrightarrow \sigma.\mathsf{ca} \neq \mathsf{IDLE})
\end{array}
$$

The postcondition $\mathsf{kdispatch}_{\mathsf{post}}$ summarizes all properties that hold when we return from function `kdispatch`. It depends on the implementation state before (σ) and after (τ) its execution as well as on both function parameters *eca* and *edata*. The parameters encode the exception cause and additional data. We require the validity of the successor state τ after the execution of `kdispatch`. In the reset case (i. e., $(eca \wedge_{\mathsf{u}} 1) \neq 0$), state τ must even be an initial state. When an interrupt of the ABC device occurs (i. e., $(eca \wedge_{\mathsf{u}} 8192) \neq 0$), the postconditions of function `handle_int` should be satisfied. Finally, we expect that postcondition $\mathsf{handle\_trap}_{\mathsf{post}}$ holds after executing a valid system call from a user process (i. e., $(eca \wedge_{\mathsf{u}} 32) \neq 0$). This is denoted as:

$\mathsf{kdispatch}_{\mathsf{post}}\ \sigma\ \tau\ eca\ edata \equiv$
$\quad \mathsf{is\_validState}_{\mathsf{i}}\ \tau \wedge$
$\quad \textbf{if}\ (eca \wedge_{\mathsf{u}} 1) \neq 0\ \textbf{then}\ \mathsf{is\_initState}_{\mathsf{i}}\ \tau$
$\quad \textbf{elsif}\ (eca \wedge_{\mathsf{u}} 8192) \neq 0\ \textbf{then}\ \mathsf{handle\_int}_{\mathsf{post}}\ \sigma\ \tau$
$\quad \textbf{else}\ (eca \wedge_{\mathsf{u}} 32) \neq 0 \wedge edata \leq 2 \longrightarrow \mathsf{handle\_trap}_{\mathsf{post}}\ \sigma\ \tau\ edata$

The execution of the Olos main function may be seen as a single Olos transition from entering until leaving the kernel dispatcher. The Hoare Triple of this function describes the behaviour of our real-time operating system under certain assumptions:

Theorem 5.4 (Functional Correctness of fun_kdispatch)**.** *We assume that for an implementation state σ the precondition predicate $\mathsf{kdispatch}_{\mathsf{pre}}$ holds. Then, the state τ after returning from the operating systems main function the postcondition predicate $\mathsf{handle\_int}_{\mathsf{post}}$ is satisfied and the return value from the kernel is the value of variable* ca*. Formally:*

$$\forall \sigma\ \tau.\ \Gamma \vdash_{\mathsf{t}} \{\!|\sigma.\ \mathsf{kdispatch}_{\mathsf{pre}}\ \sigma\ \acute{}\mathsf{eca}|\!\}\ \acute{}\mathsf{res\_nat} :== \mathsf{PROC\ fun\_kdispatch}(\acute{}\mathsf{eca},\acute{}\mathsf{edata})\ \{\tau.\ \mathsf{kdispatch}_{\mathsf{post}}\ \sigma\ \tau\ \sigma.\mathsf{eca}\ \sigma.\mathsf{edata} \wedge \tau.\mathsf{res\_nat} = \tau.\mathsf{ca}\}$$

Proof. This proof includes mainly three subgoals.

1. In case of a reset, we can directly apply the results of Theorem 5.1 on page 124. Since an initial state is valid we are done.

After the initialization the tables are not changed any more, so that predicate olos_consts holds in all remaining cases. The validity of the message buffers valid_MB, the user process component is_valid_cvmup, the devices valid_devices and the consistency between redundant variables in Olos and the ABC device olos_abc_consis cannot be simply derived by applying the Hoare-Triples.

2. When an ABC interrupt occurs, we reuse the postconditions of proof Theorem 5.2 on page 128. Since the user process component is not involved is_valid_cvmup still holds. The hard disk is not affected any more and the device identifiers always belong to the same device type. Hence, proving device validity collapses to predicate is_valid_ABC. The configuration registers and the buffer lengths remain also unchanged after the initialization phase. We now distinguish between the different phases depending on the send flag:

(a) Send phase ($s_{\mathsf{abc}}.\mathsf{INT} \wedge s_{\mathsf{abc}}.\mathsf{SF}$):
valid_MB is satisfied because this component is not modified in this phase. The copy primitive exec_cvmPhysIOOutRange only updates the send buffer of the ABC device with data from the message buffer. The values of the old message buffers are valid, hence the updated send buffer gets a valid message as well. Sending the clear interrupt command with CVM primitive exec_cvmOutWord does not harm device validity. The effect on the ABC state is the following: the send flag and the interrupt flag are cleared. In the implementation this is done in the send phase, so that olos_abc_consis is fulfilled.

(b) Receive phase ($s_{\mathsf{abc}}.\mathsf{INT} \wedge \neg\, s_{\mathsf{abc}}.\mathsf{SF}$):
From the assumptions we know that all ABC buffers contain valid messages. Thus, we can conclude that the updated message buffer after CVM primitive exec_cvmPhysIOInRange fulfills predicate is_valid_MB. The read request does not change the ABC state so that we only consider the effects of exec_cvmOutWord. The validity of the ABC is not violated by its effects, hence is_valid_ABC holds. exec_cvmOutWord clears the interrupt and sets the send flag to the SPT entry of the next slot. In the send phase of the implementation, OLOS sets the send flag to the same value, hence they are consistent. When the implementation reacts to the raised interrupt flag of the device, the ABC's current slot number $s_{\mathsf{abc}}.\mathsf{CSN}$ has already been increased. In other words, at the time we enter function `kdispatch` csn and its ABC counterpart diverge for one. This conforms exactly the condition, we formulated in predicate olos_abc_consis.

3. In case of an incoming system call, we know that the postconditions of proof Theorem 5.3 on page 131 holds. The device component of the external state is not involved, hence valid_devices and olos_abc_consis hold. When a user process signals its termination (CALL_EXFINISHED) neither the message buffers nor the user process is changed. Thus, is_valid_MB and is_valid_cvmup obviously hold. In the remaining cases, the status register and the special purpose register are not modified. The result values that are written by function exec_cvmSetGPR are 32-bit integers, in such a way that a valid user process remains valid after updating the result register. In the receive case, the user process gets a valid message from the message buffer of the operating system, hence it remains valid. We can conclude that for all user process modifications predicate is_valid_cvmup is fulfilled. The message buffers are only affected when a user process sends a message to OLOS. We know that this message has a fixed length and consists of 32-bit integers, consequently valid_MB holds. Note that the message transfer from a user process memory to a message buffer and vice versa preserves the type of the exchanged data. When we have fixed lengths validity always holds.

All remaining cases, do not change the OLOS implementation state and therefore, validity of the successor state τ is given. □

5.1.4 Expressing a CVM Transition in Simpl

In this subsection, we define the Simpl function fun_cvmstep representing a CVM step where the concrete OLOS implementation is integrated. This function allows us to consider manipulations on the external component cvmX before invoking and after leaving the the OLOS main function fun_kdispatch.

The specification of a Simpl function combining CVM with a concrete kernel has been presented in [DDW09]. We extended this function in order to deal with initialization. The extended function fun_cvmstep was applied successfully in the context of the VAMOS microkernel in the thesis of Jan D.

Therefore, we model the reset interrupt as a global variable reset and assume that its initial value is True. The initial value corresponds to the processor state right after power-up. Additionally, we use two local variables: cp stores the return value of the OLOS kernel and proc holds the state of the currently scheduled application.

After power-up, the processor generates a reset signal so that variable reset is set to True. In this case fun_cvmstep changes the variable to False, initializes the applications, and invokes the OLOS main function fun_kdispatch with an interrupt vector of 1, i. e., exactly the reset interrupt is raised.

In further computations, a kernel execution consists of the following parts:

If there exists a current application, the application executes a single step. Then, CVM computes the masked exception cause and converts the value of the exception data register (recall Section 2.6) to invoke the OLOS main function fun_kdispatch if an enabled interrupt has been detected. The return value of the dispatcher function determines what CVM does next: if the return value is a valid process identifier CVM starts the currently scheduled application or otherwise idles until the next device interrupt occurs.

The formalization of the kernel execution function fun_cvmstep described above is given with:

```
procedures fun_cvmstep () =
IF_g ´reset THEN ´reset :=_g False;
                 ´eca :=_g 1;
                 ´edata :=_g 0;
                 ´cvmX :=_g (init_cvmup, snd ´cvmX)
ELSE IF_g (fst ´cvmX).currentp = ⊥
       THEN ´eca :=_g mca_nat ⊥ (snd ´cvmX) (fst ´cvmX).statusreg;
            ´edata :=_g 0
       ELSE ´proc :=_g cvm_apps ´cvmX ⌈(fst ´cvmX).currentp⌉;
            ´eca :=_g mca_nat ⌊´proc⌋ (snd ´cvmX) (fst ´cvmX).statusreg;
            ´edata :=_g edata_nat ´proc;
            ´cvmX :=_g
            ((fst ´cvmX)
             (|userprocesses := userprocesses_step (cvm_apps ´cvmX) ⌈(fst ´cvmX).currentp⌉|),
             snd ´cvmX)
       FI
FI;
```

```
IF_g 0 < ´eca
THEN ´cp := CALL_g fun_kdispatch(´eca,´edata);
  ´cvmX :=_g ((fst ´cvmX)(|currentp := if ´cp ∈ procnumT then ⌊nat2pid ´cp⌋ else ⊥|), snd ´cvmX)
FI
```

5.1.5 Defining the Implementation Invariant

The generated OLOS implementation state has been augmented by the external variable cvmX and the reset variable reset. Each kernel execution is constrained by certain properties and requirements. They are encapsulated in the implementation invariant $\mathsf{inv}_{\mathsf{Impl}}$ which is established at the initialization and preserved by the kernel executions, user steps and external device transitions.

Our implementation invariant comprises the following assumptions:

- validity of the OLOS implementation state $\mathsf{is\_validState}_i\ \sigma$
- consistency of the external state component (fst σ.cvmX).currentp and the OLOS variable ca
- disabled reset interrupt $\neg\ \sigma$.reset

Formally:

```
inv_Impl σ ≡
  is_validState_i σ ∧
  (fst σ.cvmX).currentp =
  if σ.ca ∈ procnumT then ⌊nat2pid σ.ca⌋ else ⊥ ∧
  ¬ σ.reset
```

5.2 Relating Implementation and Abstract States

After we have formalized implementation states and an overall transition function, we need a simulation relation between implementation states and states of the abstract model. More precisely, we use a function $\mathsf{abs}_{\mathsf{Impl}}$ that maps implementation states to model states:

```
abs_Impl σ ≡
  (|AM = cvm_apps σ.cvmX,
   MB_a = map (λn. σ.heap_msg (σ.MB ! n))
              [0..<MSGCOUNT],
   idleflag = (σ.ca = IDLE),
   abc_dev = cvm_abc σ.cvmX|)
```

This function constructs a state of the ECU automaton as described in Section 4.3. From the external CVM state, we extract the applications and the device state. They are stored in the components AM and abc_dev, respectively. Furthermore, the variable of the current application ca is abstracted to the idle flag, i. e., the flag is raised iff σ.ca

= IDLE. Finally, the message buffers are gathered from the different memory objects in the implementation, which are scattered over the heap. The abstraction function is depicted in Figure 5.3.

Furthermore, we prove two lemmata that relate properties of the implementation state to the corresponding ones of the abstracted specification state.

Lemma 5.5 (Validity after Abstraction)**.** *A valid implementation state can be related via* $\mathsf{abs}_{\mathsf{Impl}}$ *to a valid specification state. Formally:*

$$\mathsf{is\_validState}_{\mathsf{i}}\ \sigma \Longrightarrow \mathsf{is\_validState}_{\mathsf{a}}\ (\mathsf{abs}_{\mathsf{Impl}}\ \sigma)$$

Proof. Since the implementation state and the abstract state share the same application and ABC model their validity can be derived by simply unfolding some definitions. From the correct implementation state we know that messages are well-formed i. e., they have a defined length MSGLENGTH and all values are signed 32-bit integers. These requirements are also encapsulated in predicate is_valid_intlistMsg in the abstract model. Hence, valid message buffers in the implementation layer imply validity of abstract message buffers. The remaining properties can be derived directly by applying the definition of the abstraction function $\mathsf{abs}_{\mathsf{Impl}}$. □

Lemma 5.6 (Initial after Abstraction)**.** *An initial implementation state can be related via* $\mathsf{abs}_{\mathsf{Impl}}$ *to an initial specification state. Formally:*

$$\mathsf{is\_initState}_{\mathsf{i}}\ \sigma \Longrightarrow \mathsf{is\_initState}_{\mathsf{a}}\ (\mathsf{abs}_{\mathsf{Impl}}\ \sigma)$$

Proof. Unfolding the definition of initial abstract states and applying Lemma 5.5 we only have to show that the abstract ABC state is initial and the idle flag is raised. Both states share the same device model so that we can conclude an initial ABC state from an initial implementation state automatically. After power-up variable ca is set to IDLE. According to the definition of the abstraction function $\mathsf{abs}_{\mathsf{Impl}}$ the idle flag is raised. □

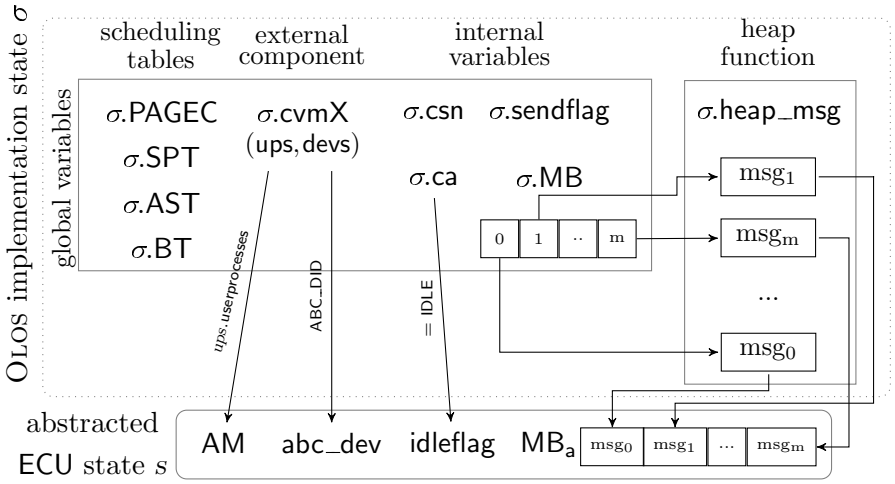

Figure 5.3: Abstracting the implementation state

5.3 Proving Correctness

In this section we present the formal simulation proof between the implementation and the specification (recall Figure 5.1 on page 116). First, we prove by induction that implementation invariant $\mathsf{inv}_{\mathsf{Impl}}$ holds after the induction start and is preserved after all kernel execution transitions fun_cvmstep. Moreover, the implementation state σ can be abstracted to an abstract ECU state after the initial step. We prove in the induction step that the semantic effects of the kernel step fun_cvmstep can be expressed by the abstract ECU transition δ_{ECU} and the abstraction function $\mathsf{abs}_{\mathsf{Impl}}$ is preserved. Finally, we define an overall implementation transition $\delta^{\mathcal{I}}_{\mathsf{ECU}}$ and prove the implementation correctness for finite traces.

5.3.1 Induction Start

The induction start formalizes the correct bootstrap at power up. As already mentioned, we assume that the reset variable initially has the value True. Additionally, we require well-formedness, e. g., the correct length of the list representing the array of message buffers. Finally, we presume that the peripheral device system is in an initial state. These constraints comprise well-formedness of the device state, on the one hand, and the correct set-up of flags (cf. definition in abc_before_olosinit Section 4.1), on the other hand. We combine all these assumptions in the constant at_power_up σ:

$$\mathsf{at\_power\_up}\ \sigma \equiv \sigma.\mathsf{reset} \wedge |\sigma.\mathsf{MB}| = \mathsf{MSGCOUNT} \wedge \mathsf{devices\_before\_olosinit}\ (\mathsf{snd}\ \sigma.\mathsf{cvmX})$$

After the initial kernel execution fun_cvmstep the invariant $\mathsf{inv}_{\mathsf{Impl}}$ holds for the successor state τ, and the abstracted state $\mathsf{abs}_{\mathsf{Impl}}\ \tau$ is an initial specification state:

Theorem 5.7 (Induction Start)**.** *After power-up, the initial kernel execution* fun_cvmstep *yields in a successor state* τ *that fulfills the implementation invariant* $\mathsf{inv}_{\mathsf{Impl}}$ *and can be related via* $\mathsf{abs}_{\mathsf{Impl}}$ *to an initial specification state. Formally:*

$$\begin{aligned}&\Gamma \vdash_{\mathsf{t}} \{\!|\sigma.\ \mathsf{at\_power\_up}\ \sigma|\!\}\ \mathsf{PROC}\ \mathsf{fun\_cvmstep}()\\ &\quad \{\tau.\ \mathsf{inv}_{\mathsf{Impl}}\ \tau \wedge \mathsf{is\_initState}_{\mathsf{a}}\ (\mathsf{abs}_{\mathsf{Impl}}\ \tau)\}\end{aligned}$$

Proof. Function fun_cvmstep disables the reset flag directly and sets the external state in such a way that all preconditions are fulfilled to reuse the result of Theorem 5.1 on page 124 (after power-up function fun_kdispatch invokes directly function fun_olos_init). An initial state implies that a state is valid. Since the return value of fun_kdispatch is IDLE component currentp is set to $\bot$. Hence, $\mathsf{inv}_{\mathsf{Impl}}\ \tau$ holds. Using Lemma 5.6 on the facing page we may also conclude that the abstracted state is initial as well. □

5.3.2 Induction Step

In the induction step, we assume that the invariant initially holds, i. e., $\mathsf{inv}_{\mathsf{Impl}}\ \sigma$. Additionally, we require that the implementation state can be abstracted via $\mathsf{abs}_{\mathsf{Impl}}$ to a specification state. Note that the postcondition of the induction start establishes

this assumption. After each execution of fun_cvmstep, the invariant should hold for the successor state. We show that the invariant and the simulation are preserved by the execution of fun_cvmstep on the implementation layer and a transition δ_{ECU} on the specification layer.

Theorem 5.8 (Induction Step)**.** *The simulation relation and the invariant are preserved under a transition* fun_cvmstep*. Formally:*

$$\Gamma \vdash_t \{\!|\sigma.\ \mathsf{inv}_{\mathsf{Impl}}\ \sigma \wedge s_{\mathsf{ecu}} = \mathsf{abs}_{\mathsf{Impl}}\ \sigma|\!\}\ \mathsf{PROC}\ \mathsf{fun\_cvmstep}()$$
$$\{\tau.\ \mathsf{inv}_{\mathsf{Impl}}\ \tau \wedge \mathsf{abs}_{\mathsf{Impl}}\ \tau = \delta_{\mathsf{ECU}}\ \bot\ s_{\mathsf{ecu}}\}$$

Proof. This prove is based implementation correctness of all OLOS functions applied in the scope of a CVM step. We benefit from the proved Hoare triples in two respects:

(a) the state validity preserved under the OLOS main function fun_kdispatch is used to discharge the implementation invariant after an kernel execution, and

(b) the successor state τ satisfying several postconditions serves to prove validity of the abstraction function $\mathsf{abs}_{\mathsf{Impl}}$. We abstract this state τ and compare it with the result of δ_{ECU} applied on state $\mathsf{abs}_{\mathsf{Impl}}\ \sigma$. (Note that we can directly use δ_{olos} due to the definition of the ECU transition function). If both states are equal, the abstraction function holds.

In this prove we seperate both targets: the preservation of the implementation on the one hand and the validity of the abstraction function function on the other hand.

1. Implementation invariant is preserved:
   After the initialization the reset variable reset remains unchanged after each kernel execution fun_cvmstep. Either CVM idles or the currently scheduled application executes an assembly step before the OLOS kernel is possibly invoked. Both actions do not violate the validity of the OLOS implementation state. If no interrupt occurs afterwards, nothing happens and the implementation invariant holds due to the assumption. Otherwise, the OLOS kernel is entered and we know from the functional correctness theorem (Theorem 5.4 on page 132) of the OLOS main function fun_kdispatch that the successor state is also valid $\mathsf{is\_validState}_{\mathsf{i}}$ and the return value is the value of the OLOS variable ca. Thus, variable currentp is related to the correct value and the invariant holds indeed.
2. The abstraction function holds after all transitions:
   We examine the postconditions of the called implementation functions and abstract them via $\mathsf{abs}_{\mathsf{Impl}}$ to an abstract state. This state has to be equal to the state we get after applying the abstract function δ_{olos} to the abstracted state $\mathsf{abs}_{\mathsf{Impl}}\ \sigma$. We distinguish between several cases:
   a) CVM idles (i. e., (fst σ.cvmX).currentp $= \bot$):
      This implies that the OLOS variable ca is abstracted to a set idle flag idleflag. In this situation only an ABC interrupt might occur since the others are disabled. If the interrupt is not generated, nothing happens and the successor states can be related via function $\mathsf{abs}_{\mathsf{Impl}}$. Otherwise, we can directly examine the postcondition of fun_handle_int since this function is applied when an interrupt occurs. We only have to distinguish between the values of the send

flag in the device component that is used in the implementation state as well as in the abstracted one:

- ¬ (cvm_abc σ.cvmX).SF:
  We have to consider every changed variable in the postcondition of the interrupt handler fun_handle_int and compute a successor state with function $\mathsf{abs}_{\mathsf{Impl}}$. Then compare this state with the result of an abstract Recv_Phase function. Variable τ.ca has been set to the current application schedule table entry, hence we conclude ¬ ($\mathsf{abs}_{\mathsf{Impl}}$ τ).idleflag. The message buffer and the ABC device have been modified by CVM primitives that rely on the multiple execution of the internal ABC transition. The message buffer and device equality is obtained by unfolding these primitives and applying the lemmata Lemma 4.5 on page 109 and Lemma 4.6 on page 109.
- (cvm_abc σ.cvmX).SF:
  In this case, we have to compare the abstracted postcondition of function fun_handle_int with the result of an abstract Send_Phase function. Variable τ.ca has been set to IDLE, thus we conclude ¬ ($\mathsf{abs}_{\mathsf{Impl}}$ τ).idleflag. The manipulation of the ABC device with CVM primitives equals the direct modification within function Send_Phase. This can be shown be unfolding the primitive definitions and apply lemma Lemma 4.7 on page 110. equal devices.

b) an application executes (i. e., (fst σ.cvmX).currentp = $\lfloor pid \rfloor$):
This implies that the OLOS variable ca is not zero and hence the abstracion is ¬ ($\mathsf{abs}_{\mathsf{Impl}}$ σ).idleflag. Function fun_cvmstep performs an assembly user step userprocesses_step. Then, we distinguish between several cases:

i. no ABC interrupt occurs:
   With this flag combination we apply function Compute_Phase on the abstract layer that computes an output ω_{app} from the user-process component and performs transition δ_{cc} according to the corresponding input.
   - runtime error occurs:
     on the implementation layer function userprocesses_step gets stuck (recall Section 2.8.2) and the OLOS kernel does not change this state. The output ω_{app} returns STUCKERR in this case and the application state also remains unchanged. The other components are not affected.
   - application requests service from the kernel:
     Then the current instruction is a TRAP instruction. In this case, function userprocesses_step increases the program counters and enters OLOS. Function fun_handle_trap handles the incoming interrupt and the successor state fulfills postcondition $\mathsf{handle\_trap}_{\mathsf{post}}$. When the trap number is invalid OLOS does nothing. On the abstract layer the output function computes UNDEFINED_TRAP and δ_{cc} returns a state with increased program counters.
     Otherwise, we distinguish on the particular trap number. In each case, we unfold the used CVM primitive functions of postcondition

$\mathsf{handle\_trap}_{\mathsf{post}}$ and compare it to the effects of function δ_{cc}. Unfolding the used functions manipulating the application on the abstract layer, we obtain the same results.

ii. an ABC interrupt occurs:
An occuring ABC interrupt in this situation implies a deadline violation and the OLOS function fun_handle_int is called. On the abstract layer, function Comp_DL_Violation adjusts the user-process component and starts the Send_Phase or the Recv_Phase respectively. After function Comp_DL_Violation the user-process component equals the user-process component after one userprocesses_step transition. The effects of the send or receive phase on the abstract layer are equal to the abstracted state fulfilling $\mathsf{handle\_int}_{\mathsf{post}}$ (this case resembles the one when CVM idles and an ABC interrupt occurs, the only difference is the manipulation of the user-process component before).

□

5.3.3 Simulation

In the last subsection we expand the correctness theorem to a simulation theorem over all finite traces. Until now we have proved that the ECU model abstracts kernel executions fun_cvmstep. However, an ECU may either perform kernel steps or device transitions. Thus, we define an overall implementation function $\delta^{\mathcal{I}}_{\mathsf{ECU}}$ that combines external device transitions and kernel execution steps including the OLOS implementation.

From the two previous theorems we know that function fun_cvmstep terminates. Moreover, C0 is deterministic. Thus, employing the soundness theorem of the Hoare logic [Sch06] allows us to interpret the proved Hoare triples on the operational semantics which results in our case in a transition function. In other words there exists a function that computes from a given pre-state σ exactly one resulting successor state τ. We call this function $\delta^{\mathcal{I}}_{\mathsf{CVM}}$. Now, we can formalize the overall implementation transition $\delta^{\mathcal{I}}_{\mathsf{ECU}}$ that combines the kernel computation and ABC transitions analogous to δ_{ECU} in Section 4.3. This function takes an implementation state σ and an optional input *eifi* from the external environment. Depending on this input the overall implementation function either performs a kernel step $\delta^{\mathcal{I}}_{\mathsf{CVM}}$ (in case that $eifi = \bot$) or an external ABC transition $\delta^{e}_{\mathsf{abc}}$. Formally:

$$
\begin{aligned}
&\delta^{\mathcal{I}}_{\mathsf{ECU}}\ eifi\ \sigma \equiv \\
&\ \textbf{case}\ eifi\ \textbf{of}\ \bot \Rightarrow \delta^{\mathcal{I}}_{\mathsf{CVM}}\ \sigma \\
&\ |\ \lfloor e \rfloor \Rightarrow \\
&\quad \sigma(\!|\mathsf{cvmX} := (\mathsf{fst}\ \sigma.\mathsf{cvmX}, (\mathsf{snd}\ \sigma.\mathsf{cvmX}) \\
&\qquad\qquad (\mathsf{nat2dev}\ \mathsf{ABC\_ID} := \mathsf{fst}\ (\delta^{e}_{\mathsf{abc}}\ e\ (\mathsf{cvm\_abc}\ \sigma.\mathsf{cvmX}))))|\!)
\end{aligned}
$$

Certainly, we require that the invariant is preserved after an external ABC device transition $\delta^{e}_{\mathsf{abc}}$. Therefore, we define a predicate valid_inputs that ensures validity of all incoming messages from the external environment:

$$\begin{aligned}&\mathsf{valid\_inputs}\ \mathit{eifis} \equiv\\ &\quad \forall i{\in}\mathsf{set}\ \mathit{eifis}.\ \forall \mathit{msg}.\ i = \lfloor \mathsf{eifi\_abc\_msg}\ \mathit{msg}\rfloor \longrightarrow \mathsf{is\_valid\_intlistMsg}\ \mathit{msg}\end{aligned}$$

Now, we can prove the theorem of the invariant preservation after the execution of an external ABC transition.

Theorem 5.9 (External ABC Transition preserves Invariant)**.** *We assume a current state σ where the invariant holds (i. e.,* $\mathsf{inv_{Impl}}\ \sigma$*) and a non-empty valid external input* valid_inputs *(eifi $\odot$ eifis) Then, the invariant is preserved under the external device transition* δ^e_{abc}*. Formally:*

$$\begin{aligned}&\mathsf{inv_{Impl}}\\ &(\sigma(\!|\mathsf{cvmX} := (\mathsf{fst}\ \sigma.\mathsf{cvmX}, (\mathsf{snd}\ \sigma.\mathsf{cvmX})\\ &\qquad\qquad (\mathsf{nat2dev\ ABC\_ID} := \mathsf{fst}\ (\delta^e_{\mathsf{abc}}\ \lceil \mathit{eifi}\rceil\ (\mathsf{cvm\_abc}\ \sigma.\mathsf{cvmX}))))|\!))\end{aligned}$$

Proof. Note that this transition only changes the device component of our state so that all predicates over other variables remain unchanged. Thus, we have to show that valid_devices and olos_abc_consis are satisfied. In the former predicate we can exclude all requirements that concern the hard disk. Moreover, we know that the external transition function never changes the send flag and is_valid_ABC is preserved under δ^e_{abc} (Lemma 4.2 on page 84). The remaining goal is to show the relation between the current slot numbers. In case that an incoming timer interrupt (j = eifi_abc_timer) arrives when the send flag is set, we simply raise the interrupt flag. When a message or a timer signal occur with a disabled send flag, the ABC device increases the current slotnumber and enables the interrupt flag. These conditions fulfill the if-branch of predicate olos_abc_consis so that $(\sigma.\mathsf{csn} + 1)\ \mathsf{mod}\ \mathsf{SLOTCOUNT}$ removes the delay to the implementation variable. □

Finally, we can formalize our simulation theorem, where we iterate transitions over a list of optional external inputs *is* using the function fold. Note that the implementation and the abstract transition function get the same input list. Thus, with an empty input the abstract function δ_{olos} simulates the overall implementation function fun_cvmstep. Otherwise, we use function δ^e_{abc} on both layers.

Theorem 5.10 (Simulation)**.** *For an initial implementation state σ_0, where* at_power_up σ_0 *holds, we obtain simulation on external inputs is fulfilling* valid_inputs *is between the implementation* $\delta^{\mathcal{I}}_{\mathsf{ECU}}$ *and its specification* δ_{ECU} *after the initial step* $\sigma_1 = \delta^{\mathcal{I}}_{\mathsf{ECU}}\ \bot\ \sigma_0$*.*
Formally:

$$\mathsf{abs_{Impl}}\ (\mathsf{fold}\ \delta^{\mathcal{I}}_{\mathsf{ECU}}\ \mathit{is}\ \sigma_1) = \mathsf{fold}\ \delta_{\mathsf{ECU}}\ \mathit{is}\ (\mathsf{abs_{Impl}}\ \sigma_1)$$

Proof. Theorem 5.7 on page 137 states that starting in the state σ_0 at power-up, the initialization step $\sigma_1 = \delta^{\mathcal{I}}_{\mathsf{ECU}}\ \bot\ \sigma_0$ establishes the implementation invariant as well as the simulation $\mathsf{abs_{Impl}}$ between implementation and specification states. This simulation is preserved under transitions of both models because of Theorem 5.8 on page 138 and Theorem 5.9 stating that external ABC transitions δ^e_{abc} preserve the invariant. □

6 Towards Pervasively Verified Applications

> Per ogni problema complesso esiste una soluzione semplice,
> ed è quella sbagliata.[1]
>
> *Umberto Eco, quoted in: "Il pendolo di Foucault"*

Contents

In this chapter, we describe an approach to pervasively verify applications [DSS10] running on top of our operating system Olos. The correct operation of applications inherently relies on the correctness of the operating system. An important challenge here is the interaction of the application programs with our operating system Olos that has the task to retrieve input data from applications and peripheral devices and transfers output data to these components. We report on the necessary theorems that allow to transfer verification results down to the operating system level and thus, establish a formal link between proofs on the application layer and those on the operating system layer [DDWS08]. More specifically, we extend the previously existing language stack from Section 2.1 to application programs that may communicate with Olos.

The overall proof plan is depicted in Figure 6.1 on the next page: The three rows depict the different semantic layers consisting of the languages Simpl, C0 and Vamp assembly. The left most column reflects the language stack as described in Section 2.1. There are transfer theorems [Sch06] stating that properties proved for Simpl hold on the C0 layer. Furthermore, the correct C0 compiler of Leinenbach & Petrova [LP08] translates sequential C0 programs to Vamp assembly (Theorem 2.1 on page 30). The second column depicts the extension of sequential C0 and assembly programs to application processes (denoted with $^{\mathsf{app}}$) where outputs and inputs model the comunication with the operating system. We introduced the extended languages in Section 4.2. In Section 6.1 we will present the extended compiler theorem Theorem 6.14 on page 166 that relates the C0 and the assembly process model.

[1] For every complex problem, there's a simple solution, and it's wrong.

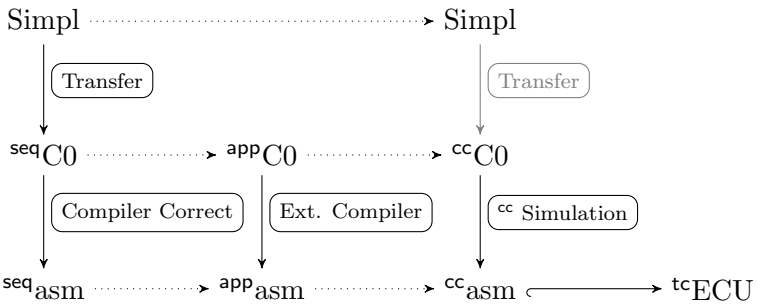

Figure 6.1: Extending the Language Stack towards Concurrency

The third column illustrates the extension of the language stack to *cooperative concurrent* applications (marked with ^{cc}). The term *cooperative concurrency* refers to the sequential execution of applications with system calls until a final call of the synchronization primitive *olosExFinished*. With our approach we prepare the basis to specify application programs interacting with Olos in Simpl. In the Hoare logic we can reason efficiently about sequential, type- safe, and assembly-free C0 programs. In contrast to the lower semantical layers, the specification of the system calls in Simpl does not require a language extension. Then, we can use transfer theorems to translate the results derived on the Simpl layer to C0. In Section 6.2 we present a simulation theorem (Theorem 6.15 on page 167) stating that all results on the C0 layer still hold at the assembly level. The lowest layer of this stack is embedded into a *true concurrent* ECU model (^{tc}ECU) with an interleaved application execution (last column). In Section 6.3, we provide a theorem (Theorem 6.16 on page 168) that allows us to infer properties about the entire ECU behavior by combining verification results from the applications running on the ECU. In the last section (Section 6.4) of this chapter, we give an example how this verification approach can be used in practice.

6.1 Process Simulation

Section 2.4 reported that Leinenbach & Petrova proved compiler correctness in form of a correctness theorem Theorem 2.1 on page 30 between sequential C0 and Vamp assembly. In this section, we extend their theorem to processes we introduced in Section 4.2. Most notably, we show that all implemented C0 functions in the system call library (Section 3.4) can be compiled into Vamp assembly code that is executed in a certain number of steps on the target machine.

We consider the interaction between an application and the operating system from the point of view of the application. According to Table 4.3 on page 86 an application sends an output to the operating system, which in turn responds with a corresponding input (except the output is StuckErr). The set olos_responses collects all these matching output-input pairs.

We relate C0 and assembly application states by the simulation relation $\mathsf{consis}_{\mathsf{app}}$.

Predicate $\mathsf{consis}_{\mathsf{app}}$ takes a C0 and an assembly state together with an allocation function *alloc*. Application consistency requires that previous executions did not get stuck (i. e., $S_{\mathsf{C0}} = \lfloor s_{\mathsf{C0}} \rfloor$), that the C0 simulation relation of Definition 2.1 on page 27 holds and the application sizes in both layers are equal. Formally:

$$\begin{aligned}&\mathsf{consis}_{\mathsf{app}}\ S_{\mathsf{C0}}\ \mathit{alloc}\ s_{\mathsf{asm}} \equiv\\ &\quad (\exists s_{\mathsf{C0}}.\ S_{\mathsf{C0}} = \lfloor s_{\mathsf{C0}} \rfloor \wedge \mathsf{consistent}\ s_{\mathsf{C0}}.\mathsf{ttab}\ s_{\mathsf{C0}}.\mathsf{ftab}\ s_{\mathsf{C0}}.\mathsf{pstate}\ \mathit{alloc}\ s_{\mathsf{asm}}) \wedge\\ &\quad \mathsf{size}_{\mathsf{app}}\ S_{\mathsf{C0}} = \mathsf{size}_{\mathsf{app}}\ s_{\mathsf{asm}}\end{aligned}$$

Figure 6.2 depicts the scenario of application simulation: We assume that previous executions did not get stuck i. e., the application state $S^j_{\mathsf{C0}} \neq \bot$ and the output ω is not StuckErr. On the C0 layer, the application state S^j_{C0} computes an output ω that is sent to the kernel. The corresponding response of the operating system to this output is the abstract transition $\delta_{\mathsf{app}}\ i$ leading to a successor state S^{j+1}_{C0}. We relate the C0 application state S^j_{C0} to a consistent state s^m_{asm} on the Vamp assembly layer. After a certain number of assembly transitions we reach a final state s^n_{asm} that can be related via $\mathsf{consis}_{\mathsf{app}}$ to the successor state S^{j+1}_{C0}. We prove that there exists a state s^t_{asm} on the assembly layer that generates the same output ω as the C0 state S^j_{C0}. Moreover, $\delta_{\mathsf{app}}\ i$ reflects the kernel response on the assembly layer (according to Section 4.2.1 we stated that the transition function δ_{app} can be used on both layers). All remaining Vamp assembly states have empty outputs and therefore lead to local sequential assembly steps δ_{asm}. Note that δ_{asm} equals $\delta_{\mathsf{app}}\ \varepsilon_{\Sigma}$.

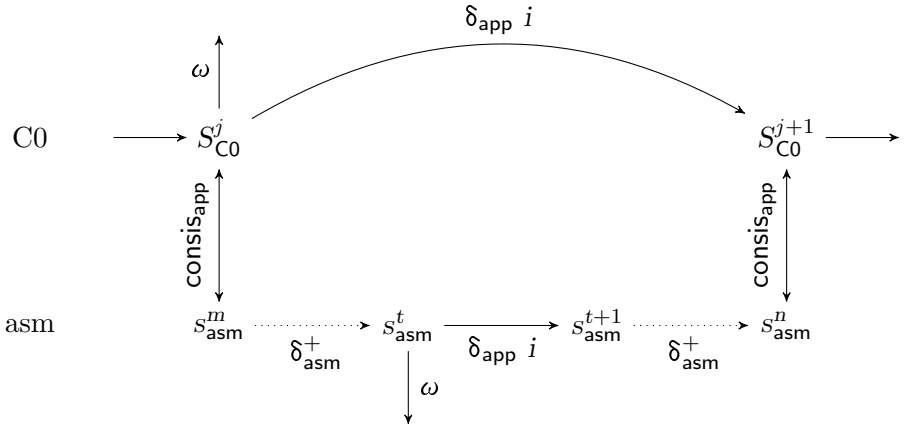

Figure 6.2: Application Simulation

A C0 transition simulates a certain number of assembly steps. Hence a single output-input pair $[oi]$ on the C0 layer simulates a corresponding output-input sequence on the assembly layer ois'. Certainly, the application models on C0 and the assembly layer should invoke the same primitives in such a way that output-input pairs are equal except for internal steps $(\varepsilon_{\Omega}, \varepsilon_{\Sigma})$. To express this fact, we define a normalization function $\gg ois' \ll$ over an output-input sequence ois' that deletes all internal steps. Formally: $\gg ois \ll\ \equiv [oi \in ois\ .\ oi \neq (\varepsilon_{\Omega}, \varepsilon_{\Sigma})]$

Successful execution of an application is characterized by the absence of runtime errors. We define the successful execution of a C0 application as follows:

Definition 6.1 (Successful Execution of a C0 Application)**.** The computation of a C0 application is called *successful execution* iff

- all output-input pairs are valid with respect to the current application state (i. e., $\forall oi \in ois.\ oi \in \mathsf{olos\_responses}$) and
- no runtime error occurs during the execution (i. e., $\neg\ \mathsf{is\_runtime\_error}$)

Formally, successful C0 application execution is defined inductively over output-input sequences:

$$\vdash^{\mathsf{proc}}_{\mathsf{C0}} S_{\mathsf{C0}} \xrightarrow{[]} S'_{\mathsf{C0}} = (S_{\mathsf{C0}} = S'_{\mathsf{C0}} \wedge \neg\ \mathsf{is\_runtime\_error}\ (\omega\mathsf{C0}\ S_{\mathsf{C0}}))$$

$$\vdash^{\mathsf{proc}}_{\mathsf{C0}} S_{\mathsf{C0}} \xrightarrow{(oi\ \odot\ ois)} S'_{\mathsf{C0}} =$$
$$(oi \in \mathsf{olos\_responses} \wedge$$
$$\vdash^{\mathsf{proc}}_{\mathsf{C0}} \delta\mathsf{C0}\ (\mathsf{snd}\ oi)\ S_{\mathsf{C0}} \xrightarrow{ois} S'_{\mathsf{C0}} \wedge \neg\ \mathsf{is\_runtime\_error}\ (\omega\mathsf{C0}\ S_{\mathsf{C0}}) \wedge \omega_{\mathsf{app}}\ S_{\mathsf{C0}} = \mathsf{fst}\ oi)$$

The successful execution of assembly applications is defined similarly. However, the sequenctial compiler correctness guarantees that the compiled code of a C0 program does not modify itself. More specifically, for each successor state s'_{asm} after an assembly instruction the code range of the memory should neither been written *crange* (i. e., $\mathsf{no\_mem\_write\_inside\_range}\ s_{\mathsf{asm}}\ s'_{\mathsf{asm}}\ crange$), nor should the program counters point outside this memory range (i. e., $\mathsf{inside\_range}\ crange\ s_{\mathsf{asm}}.\mathsf{dpc}$). Recall that we augmented the sequential VAMP assembly language with the TRAP instruction. Hence, the negation of predicate mem_write_inside_range is not sufficient to express that the code range of the memory has not been written by an assembly instruction. More precisely, we require that a current TRAP instruction does not change this memory as well. Then, we encapsulate the requirements that neither an assembly instruction of the sequential semantics nor a TRAP instruction alter the code range of the memory into predicate no_mem_write_inside_range.

Below, we define the successful execution of assembly applications:

Definition 6.2 (Successful Execution of a VAMP Assembly Application)**.** The computation of a VAMP assembly application is called *successful execution* with respcet to the code range *crange* iff

- all output-input pairs are valid with respect to the current application state,
- no runtime error occurs during the execution (i. e., $\neg\ \mathsf{is\_runtime\_error}$),
- the memory of *crange* is not written, and
- all fetched assembly instructions are located in *crange*

We specify successful VAMP assembly application executions inductively over output-input sequences:

$$crange \vdash^{\mathsf{proc}}_{\mathsf{asm}} s_{\mathsf{asm}} \xrightarrow{[]} s'_{\mathsf{asm}} = (s_{\mathsf{asm}} = s'_{\mathsf{asm}})$$

$$crange \vdash^{\mathsf{proc}}_{\mathsf{asm}} s_{\mathsf{asm}} \xrightarrow{(oi\ \odot\ is)} s'_{\mathsf{asm}} =$$

$(oi \in \mathsf{olos\_responses} \wedge$
$crange \vdash^{\mathsf{proc}}_{\mathsf{asm}} \delta_{\mathsf{asm}}\ (\mathsf{snd}\ oi)\ s_{\mathsf{asm}} \xrightarrow{is} s'_{\mathsf{asm}} \wedge$
$\neg\ \mathsf{is\_runtime\_error}\ (\omega\mathsf{asm}\ s_{\mathsf{asm}}) \wedge$
$\omega_{\mathsf{app}}\ s_{\mathsf{asm}} = \mathsf{fst}\ oi \wedge$
$\mathsf{no\_mem\_write\_inside\_range}\ s_{\mathsf{asm}}\ (\delta_{\mathsf{asm}}\ (\mathsf{snd}\ oi)\ s_{\mathsf{asm}})\ crange \wedge$
$\mathsf{inside\_range}\ crange\ s_{\mathsf{asm}}.\mathsf{dpc})$

Daum [DDWS08] formulated process simulation first for processes running on top of the microkernel Vamos. We adapted his approach to prove simulation for applications running on top of Olos. In the remaining section we aim to prove the fact that one transition of a C0 application simulates a transition sequence of the corresponding assembly application.

We assume a valid C0 application state S_{C0} that performs one successful execution step. Moreover, we have a consistent assembly state s_{asm} that is not in a delay slot (i. e., $s_{\mathsf{asm}}.\mathsf{pcp} = s_{\mathsf{asm}}.\mathsf{dpc} + 4$). Then, we aim to prove that there exists an output-input sequence ois', an allocation function $alloc'$ and a final assembly state s'_{asm} so that

- both application models invoke the same primitives (i. e., $\gg[oi]\ll\ = \gg ois'\ll$)
- the successor states of both layers are valid and consistent
- the final assembly state s'_{asm} is not in a delay slot (i. e., $s'_{\mathsf{asm}}.\mathsf{pcp} = s'_{\mathsf{asm}}.\mathsf{dpc} + 4$)

Formally:[2].

$[\![\mathsf{is\_valid}_{\mathsf{app}}\ S_{\mathsf{C0}};\ \mathsf{is\_valid}_{\mathsf{app}}\ s_{\mathsf{asm}};\ s_{\mathsf{asm}}.\mathsf{pcp} = s_{\mathsf{asm}}.\mathsf{dpc} + 4;$
$\mathsf{consis}_{\mathsf{app}}\ S_{\mathsf{C0}}\ alloc\ s_{\mathsf{asm}};\ \vdash^{\mathsf{proc}}_{\mathsf{C0}} S_{\mathsf{C0}} \xrightarrow{[oi]} S'_{\mathsf{C0}}]\!]$
$\Longrightarrow \exists ois'\ alloc'\ s'_{\mathsf{asm}}.$
$\gg[oi]\ll\ = \gg ois'\ll\ \wedge$
$\mathsf{code\_range}\ S_{\mathsf{C0}} \vdash^{\mathsf{proc}}_{\mathsf{asm}} s_{\mathsf{asm}} \xrightarrow{ois'} s'_{\mathsf{asm}} \wedge$
$\mathsf{is\_valid}_{\mathsf{app}}\ S'_{\mathsf{C0}} \wedge$
$\mathsf{is\_valid}_{\mathsf{app}}\ s'_{\mathsf{asm}} \wedge$
$s'_{\mathsf{asm}}.\mathsf{pcp} = s'_{\mathsf{asm}}.\mathsf{dpc} + 4 \wedge \mathsf{consis}_{\mathsf{app}}\ S'_{\mathsf{C0}}\ alloc'\ s'_{\mathsf{asm}}$

Basically, this statement can be shown by a case distinction on the output-input pair oi. The assumption that the C0 transition does not get stuck already excludes some cases. For the empty pair $(\varepsilon_\Omega, \varepsilon_\Sigma)$, we employ Lemma 2.2 on page 31 because internal steps can be broken down to transitions of the sequential language. The remaining cases are the three Olos system calls. For these cases, we consider the implementation of the system call functions. Then, we show application simulation in two steps:

1. each function of the system call library is simulated by the abstract C0 transition δ_{app}
2. all system call primitives implemented in C0 can by compiled to Vamp assembly code that is executed in a certain number of steps on the target machine

[2]For convenience we overload the definition of code_range_word for C0 applications:
$\mathsf{code\_range}\ S_{\mathsf{C0}} \equiv \mathsf{code\_range\_word}\ \lceil S_{\mathsf{C0}} \rceil.\mathsf{ttab}\ (\mathsf{gm\_st}\ \lceil S_{\mathsf{C0}} \rceil.\mathsf{pstate.mem})\ \lceil S_{\mathsf{C0}} \rceil.\mathsf{ftab}$

6.1.1 System Call Simulation

In this subsection, we prove a simulation theorem stating that the abstract transition of a system call on the C0 level equals the execution of the corresponding C0 function body. We know from Section 3.4 that each function of the system call library includes an inline assembly portion and a return statement in its body. Thus, their semantics cannot be solely described in C0 but we additionally have to consider the VAMP assembly layer.

Figure 6.3 on the next page exemplarily depicts an overview over the verification approach for the system-call function `olosRecvMsg`: All states and edges that are considered in this subsection are highlighted in black whereas the parts drawn in grey are proved in theorem Theorem 6.13 on page 165. On the C0 level, only a non-empty C0 application state $S_{\mathsf{C0}} = \lfloor \mathsf{s}^{j}_{\mathsf{C0}} \rfloor$ can request a service from the operating system. Otherwise, previous executions got stuck and the output ω would be given with STUCKERR. In our scenario the application requests with its output to receive a message from the operating system (i. e., $\omega_{\mathsf{app}}\ S_{\mathsf{C0}} = \text{RECVMSG}\ \mathit{msgnum}$). Recall Section 4.2.3, that in this case the application has sufficient memory, the current statement is a function call and the evalutation of the corresponding parameters does not fail. The effects of the response from the operating system are specified in transition $\delta_{\mathsf{app}}\ i$ that computes a non-empty successor state $S'_{\mathsf{C0}} = \lfloor \mathsf{s}^{j+1}_{\mathsf{C0}} \rfloor$. We prove that this state is equal to a state we obtain after the execution of the complete code implementing a system call function. This execution can be split into three parts due to the implementation of system calls:

- the execution of the function call statement SCALL,
- followed by the execution of the function body that contains
  - an inlined assembly portion followed by
  - a RETURN statement.

Thus, we compute the effects of a system call implementation execution on two layers - the C0 small-step semantics on the one hand and the VAMP assembly layer on the other hand. In Section 2.5 we described a method how to deal with inline assembly within C0 code. We integrate the results obtained on the VAMP assembly layer by using this technique. After the execution of the function call statement SCALL (i. e., in state s'_{C0}) we encounter the first statement of the function body. In our case it is a list of assembly instructions. Hence, we continue the execution on the VAMP assembly layer from a state s^{1}_{asm} that is consistent to s'_{C0}. From Section 3.4 we know that the portion of inline assembly contains two load instructions followed by a TRAP and a store instruction. Thus, we reach an assembly state s^{3}_{asm} after two transitions that generates the same output ω as state $\mathsf{s}^{j}_{\mathsf{C0}}$. The last step is equal to the first two steps a sequential VAMP assembly transition in such a way that all other outputs (except the one computed in state s^{3}_{asm}) are ε_{Ω}. When the list of assembly instructions has been executed successfully, we construct a C0 state s''_{C0} that is consistent to the final assembly state s^{5}_{asm}. Finally, the C0 state after a C0 small-step execution of the RETURN statement should be $\mathsf{s}^{j+1}_{\mathsf{C0}}$. Figure 6.3 on the next page

OLOS system calls can be classified into three different cases according to the modifications on the application state in the VAMP assembly layer:

- **registers and memory remain unchanged:**

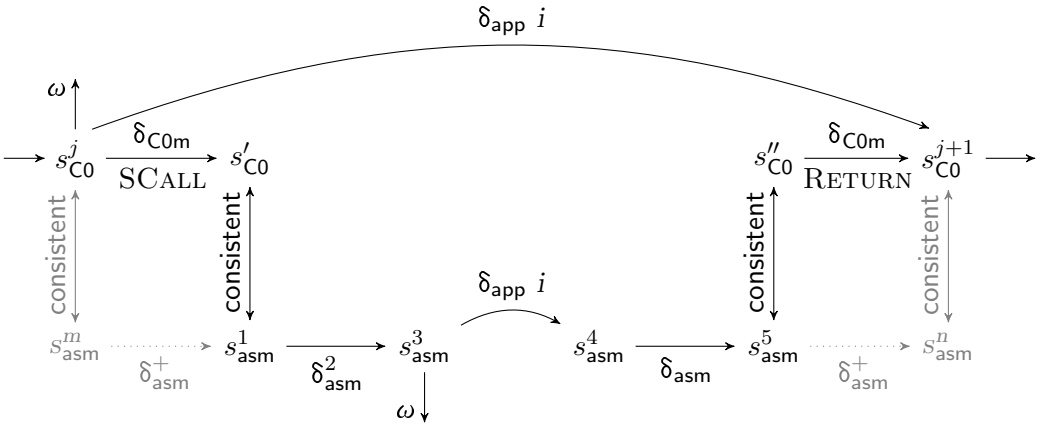

Figure 6.3: Verification plan for the C0 implementation of the `olosRecvMsg` primitive

System call *olosExFinished* neither modifies the memory of the application nor writes the result register on the Vamp assembly layer

- **the result register is modified:**
  This is either the case when the application successfully sent a message to the operating system (i. e., *olosSendMsg*) or the operating system returns an error value to the application (note that this might happen for the system calls *olosSendMsg* and *olosRecvMsg*)
- **the memory and result register are modified:**
  when the application requests to receive a message *olosRecvMsg* without causing an error the operating system stores the message into the memory and updates the result register of the application

In the following subsections we use the most complex case to exemplarily show the effects of the *olosRecvMsg* system call that causes no error. We explain stepwise the effects of all transition functions on C0 or Vamp assembly states and briefly describe how to treat the other cases. Finally we prove the simulation theorems stating that each function of the system call library is simulated by the abstract C0 transition δ_{app}.

Execution of the SCall Statement

We start in a C0 state $S_{\mathsf{C0}} = \lfloor s^j_{\mathsf{C0}} \rfloor$ where the output signals a request to receive a message from the operating system (i. e., $\omega_{\mathsf{app}}\ S_{\mathsf{C0}} =$ RecvMsg *msgnum*). This output implies that the memory is sufficient after extending the local stack and first statement is a function call of "olosRecvMsg" with two valid parameters. More specifically, the first statement of the program rest is given with:

$\mathsf{fst\_stmt}\ s^j_{\mathsf{C0}}.\mathsf{pstate.prog} =$ SCall *lv* "olosRecvMsg" *es sid*

Recall Section 4.2.3, that the parameter list *es* consists of two elements: the first one (*es* ! 0) keeps the message pointer and the second one (*es* ! 1) stores the message number.

The execution of a function call in the small-step semantics has the following effects:

- a new stack frame is created,
- the function parameters are evaluated and copied into this frame, and
- the function-call statement in the program rest is substituted by the body of the called function.

Recall that the transition function is partial (Section 2.3). Then, the transition function δ_{C0m} is only defined iff:

1. the function name *fn* has a corresponding definition in the function table,
2. the return variable *lv* is either a global variable or a variable of the topmost local memory frame, and
3. the evaluation of all parameters are valid.

For all OLOS system calls, we can show that the small-step semantics of the function call succeeds:

Lemma 6.1 (Successful Execution of the SCALL Statement)**.** *We assume a non-empty valid C0 state* S_{C0} *i. e.,* $\mathsf{is\_valid}_{\mathsf{app}}\ S_{\mathsf{C0}} \wedge S_{\mathsf{C0}} = \lfloor \mathsf{s}^{j}_{\mathsf{C0}} \rfloor$ *holds. Moreover, the application neither intends to perform a local step nor does the output function* ω_{app} *signal a runtime error i. e.,* $\omega_{\mathsf{app}}\ S_{\mathsf{C0}} \neq \varepsilon_{\Omega} \wedge \neg\ \mathsf{is\_runtime\_error}\ (\omega_{\mathsf{app}}\ S_{\mathsf{C0}})$ *is fullfilled. Thus, the current statement is a system call. Then we can conclude that the execution of the* SCALL *statement is not empty. Formally:*

$$\delta_{\mathsf{C0m}}\ \mathsf{s}^{j}_{\mathsf{C0}} \neq \bot$$

Proof. From the C0 state validity $\mathsf{is\_valid}_{\mathsf{app}}$ we know that the program rest is valid, hence the first statement a fortiori. A valid SCALL statement implies that the calling function is part of the function table and the variable access of the left variable is valid. The latter in turn means that the variable is either defined as a global or local variable on the topmost stack frame. For the system call *olosExFinished* we are done. The other system calls additionally require that the evaluation of the parameters does not fail. This fact can be derived by unfolding the output functions ω_{app} and $\mathsf{get\_output}$. □

After the successful execution of the SCALL statement in $\mathsf{s}^{j}_{\mathsf{C0}}$, we obtain a new C0 state s'_{C0}. Figure 6.4 on the facing page depicts the different C0 states before (a) and after (b) the execution of the SCALL. The modified memories are illustrated below the corresponding first statements of the program rest. C0 small-step executions of SCALL statements solely extend the local memory by creating a new stack frame for the function call and do not affect the global or the heap memory of the successor state. From Section 2.3 we know that each stack frame consists of a single memory frame stored together with the return destination of the calling function. Each memory frame again consists of a content storing elementary values in memory cells, a symbol table holding variables together with their types and a set of already initialized variables. After the stack extension the new stack-frame content includes the evaluated values of all function call parameters listed in *es*. Moreover, the symbol table of the topmost stack frame stores the parameters and the local variables of the function. From the precise definition of function *olosRecvMsg* in Section 4.2.3, function $\mathsf{toplm\_symbols}$ returns the following result:

toplm_symbols s'_{C0}.pstate.mem =
[("pmsg", Ptr "TYPE_Message_Struct"), ("mn", UnsgndT), ("result", Integer)]

Furthermore, the set of initialized variables consists of the variables "pmsg" and "mn". The left variable *lv* is either defined in the global memory or the second topmost stack frame of the local memory (i. e., the topmost stack frame *before* the new stack frame has been created). Thus, the return destination keeps the corresponding g-variable value (computed with function vlookup) of the left variable *lv*. Formally:

toprd s'_{C0}.pstate.mem = $\lceil$vlookup s^{j}_{C0}.pstate.mem *lv*$\rceil$

The first statement of the new program rest is the function body of the system call. We know from the precise definition of function *olosRecvMsg* Section 4.2.3 that the program rest of state s'_{C0} is given as follows:

fst_stmt s'_{C0}.pstate.prog =
COMP (ASM [llw 11 30 16, llw 12 30 20, TRAP 2, lsw 22 30 24] *sid*)
(RETURN (VARACC "result") *sid'*)

We proceed with the execution of the first statement.

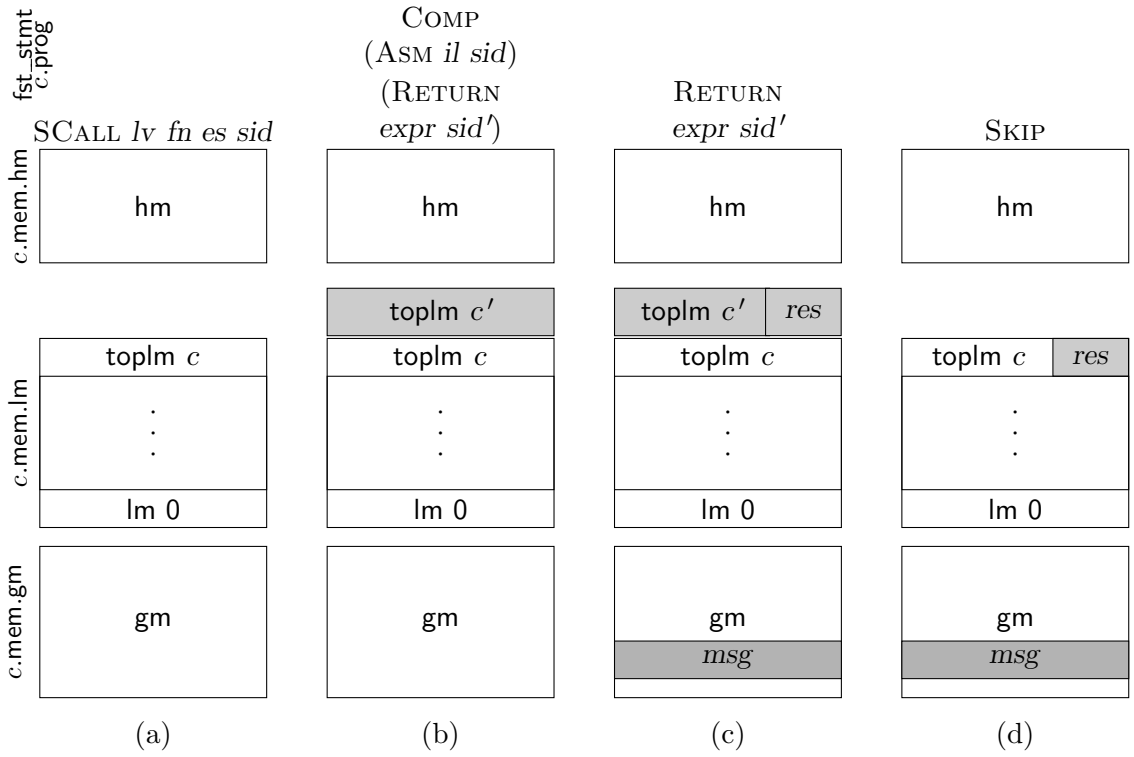

Figure 6.4: Program State during the System Call Execution

Execution of the Inline Assembly Code ASM *il sid*

When an assembly statement ASM *il sid* is encountered, we switch to the VAMP assembly layer and start arguing about an arbitrary, fixed assembly state s^1_{asm} that is consistent to s'_{C0}. From the C0 simulation relation Section 2.4 we know that control consistency holds i. e., the program counters point to the start of the list pf assembly instructions *il*. Then, we can process the assembly portion according to the VAMP assembly semantics. The memory layout of a compiled program has already been presented in Figure 2.4 on page 27. Recall that each local memory frame starts with a stack frame header that stores the return address and the return destination of a function, and the previous stack pointer. The return address specifies the address where to jump after the completion of the function. In the return destination we store the address where the result of the function will be written. Finally, each stack frame consists of the content of its C0 counterpart.

In the following lemmata and theorems we often use the same preconditions. Therefore, we define a predicate common_preconds that encapsulates all reusable requirements.

Definition 6.3 (Common Preconditions)**.** Predicate common_preconds encapsulates the following preconditions:

- we start in a non-empty C0 application state $S_{\mathsf{C0}} = \lfloor s^j_{\mathsf{C0}} \rfloor$,
- that is valid (i. e., $\mathsf{is\_valid}_{\mathsf{app}}\ S_{\mathsf{C0}}$),
- the application requests to receive a message from the operating system i. e., $\omega_{\mathsf{app}}\ S_{\mathsf{C0}} = \text{RECVMSG}\ mn$
- the execution of the SCALL statement returns the successor state s'_{C0} i. e., $\delta_{\mathsf{C0m}}\ s^j_{\mathsf{C0}} = \lfloor s'_{\mathsf{C0}} \rfloor$
- there exists an assembly state s^1_{asm} that is consistent to state s'_{C0} i. e., $\mathsf{consistent}\ s^j_{\mathsf{C0}}.\mathsf{ttab}\ s^j_{\mathsf{C0}}.\mathsf{ftab}\ s'_{\mathsf{C0}}.\mathsf{pstate}\ \mathit{alloc}\ s^1_{\mathsf{asm}}$
- Moreover, state s^1_{asm} is valid i. e., $\mathsf{is\_valid}_{\mathsf{app}}\ s^1_{\mathsf{asm}}$
- the application sizes of S_{C0} and s^1_{asm} are equal i. e., $\mathsf{size}_{\mathsf{app}}\ S_{\mathsf{C0}} = \mathsf{size}_{\mathsf{app}}\ s^1_{\mathsf{asm}}$
- Finally, the message *msgval* received from the application is valid i. e., is_valid_intlistMsg *msgval*

In the following lemma we aim at three goals:

- we require that state s^3_{asm} produces the same output as s^j_{C0} whereas all the other outputs are ε_Ω (i. e., the inline assembly portion includes exactly one TRAP instruction with the number that corresponds to the C0 system call function *olosRecvMsg* whereas the other instructions are local transitions of the sequential VAMP assembly language),
- the final state s^5_{asm} stores the function parameters in the registers 11 and 12 and the result value in the designated register 22. Moreover, the program counter points to the first address after the executed code and the memory stores the message value on the one hand and the result value on the other hand.
- in all transitions the program counter always points to addresses within the code range and the code range is not modified

Lemma 6.2 (Properties of Inline Assembly Code)**.** *We require that predicate* common_preconds *holds. With all these assumptions, we conclude three properties:*

1. *Outputs:*

$$\omega_{app}\ s^1_{asm} = \varepsilon_\Omega \wedge$$
$$\omega_{app}\ (\delta_{app}\ \varepsilon_\Sigma\ s^1_{asm}) = \varepsilon_\Omega \wedge$$
$$\omega_{app}\ (\delta_{app}\ \varepsilon_\Sigma\ (\delta_{app}\ \varepsilon_\Sigma\ s^1_{asm})) = \omega_{app}\ S_{C0} \wedge$$
$$\omega_{app}\ (\delta_{app}\ (\textsc{RecvSuccess}\ msgval)\ (\delta_{app}\ \varepsilon_\Sigma\ (\delta_{app}\ \varepsilon_\Sigma\ s^1_{asm}))) = \varepsilon_\Omega$$

2. *Final state:*

$$\delta_{app}\ \varepsilon_\Sigma\ (\delta_{app}\ (\textsc{RecvSuccess}\ msgval)\ (\delta_{app}\ \varepsilon_\Sigma\ (\delta_{app}\ \varepsilon_\Sigma\ s^1_{asm}))) =$$
$$(\textbf{let}\ startaddr = \text{to\_nat32}\ (s^1_{asm}.\text{mm}\ (\text{to\_nat32}\ (s^1_{asm}.\text{gprs}\ !\ 30)\ \text{div}\ 4 + 4))$$
$$\textbf{in}\ s^1_{asm}(\!|\text{gprs} := s^1_{asm}.\text{gprs}$$
$$[11 := s^1_{asm}.\text{mm}\ (\text{to\_nat32}\ (s^1_{asm}.\text{gprs}\ !\ 30)\ \text{div}\ 4 + 4),$$
$$12 := s^1_{asm}.\text{mm}\ (\text{to\_nat32}\ (s^1_{asm}.\text{gprs}\ !\ 30)\ \text{div}\ 4 + 5),\ 22 := 0],$$
$$\text{dpc} := 16 + s^1_{asm}.\text{dpc},\ \text{pcp} := 20 + s^1_{asm}.\text{dpc},$$
$$\text{mm} := (\lambda i.\ \textbf{if}\ i < startaddr\ \text{div}\ 4 \vee startaddr\ \text{div}\ 4 + |msgval| \leq i$$
$$\textbf{then}\ s^1_{asm}.\text{mm}\ i$$
$$\textbf{else}\ msgval\ !\ (i - startaddr\ \text{div}\ 4))$$
$$(\text{to\_nat32}\ (s^1_{asm}.\text{gprs}\ !\ 30)\ \text{div}\ 4 + 6 := 0)|\!)$$

3. *No code modifications:*

$$\text{inside\_range}\ (\text{code\_range}\ S_{C0})\ s^1_{asm}.\text{dpc} \wedge$$
$$\text{inside\_range}\ (\text{code\_range}\ S_{C0})\ (\delta_{app}\ \varepsilon_\Sigma\ s^1_{asm}).\text{dpc} \wedge$$
$$\text{inside\_range}\ (\text{code\_range}\ S_{C0})\ (\delta_{app}\ \varepsilon_\Sigma\ (\delta_{app}\ \varepsilon_\Sigma\ s^1_{asm})).\text{dpc} \wedge$$
$$\text{inside\_range}\ (\text{code\_range}\ S_{C0})$$
$$(\delta_{app}\ (\textsc{RecvSuccess}\ msgval)\ (\delta_{app}\ \varepsilon_\Sigma\ (\delta_{app}\ \varepsilon_\Sigma\ s^1_{asm}))).\text{dpc} \wedge$$
$$\text{inside\_range}\ (\text{code\_range}\ S_{C0})$$
$$(\delta_{app}\ \varepsilon_\Sigma\ (\delta_{app}\ (\textsc{RecvSuccess}\ msgval)\ (\delta_{app}\ \varepsilon_\Sigma\ (\delta_{app}\ \varepsilon_\Sigma\ s^1_{asm})))).\text{dpc} \wedge$$
$$\text{no\_mem\_write\_inside\_range}\ s^1_{asm}\ (\delta_{app}\ \varepsilon_\Sigma\ s^1_{asm})\ (\text{code\_range}\ S_{C0}) \wedge$$
$$\text{no\_mem\_write\_inside\_range}\ (\delta_{app}\ \varepsilon_\Sigma\ s^1_{asm})\ (\delta_{app}\ \varepsilon_\Sigma\ (\delta_{app}\ \varepsilon_\Sigma\ s^1_{asm}))\ (\text{code\_range}\ S_{C0})$$
$$\wedge$$
$$\text{no\_mem\_write\_inside\_range}\ (\delta_{app}\ \varepsilon_\Sigma\ (\delta_{app}\ \varepsilon_\Sigma\ s^1_{asm}))$$
$$(\delta_{app}\ (\textsc{RecvSuccess}\ msgval)\ (\delta_{app}\ \varepsilon_\Sigma\ (\delta_{app}\ \varepsilon_\Sigma\ s^1_{asm})))\ (\text{code\_range}\ S_{C0}) \wedge$$
$$\text{no\_mem\_write\_inside\_range}\ (\delta_{app}\ (\textsc{RecvSuccess}\ msgval)\ (\delta_{app}\ \varepsilon_\Sigma\ (\delta_{app}\ \varepsilon_\Sigma\ s^1_{asm})))$$
$$(\delta_{app}\ \varepsilon_\Sigma\ (\delta_{app}\ (\textsc{RecvSuccess}\ msgval)\ (\delta_{app}\ \varepsilon_\Sigma\ (\delta_{app}\ \varepsilon_\Sigma\ s^1_{asm}))))\ (\text{code\_range}\ S_{C0})$$

Proof. This proof executes all the instructions of the inline assembly code stepwise and examines the effects carefully: The entire list of assembly instructions is stored in the code range so that the program counters pointing to their addresses do not leave the code range. From the consistency, we know that the first instruction to be executed is the load-word instruction lw 11 30 16. This instruction is decodable, the effective address is smaller than the heap base, the program counters point to an address within the code range and are divisible by 4. Hence, no runtime error occurs. We described the effects of a load-word instruction in Section 2.2. The general-purpose register 11 holds the value of the first parameter of the function call and the program counters are increased. Thus, the next instruction is given by lw 12 30 20. Again no runtime

error occurs because this instruction is decodable, the increased program counters are divisible by 4 and the effective address is smaller than the heap base. After the execution of the second load-store instruction, the general-purpose register 12 stores the value of the second parameter of the system call and the program counters point to the next instruction. In our case the next instruction is TRAP 2 so that the output function returns RECVMSG *mn*. This output equals the one on the C0 layer. The effects of the corresponding transition function δ_{app} are defined in Section 4.2.2. The TRAP instruction writes the message into the global or heap memory range and the result value 0 into the result register 22. Hence, the code range remains unchanged. Furthermore, the program counters are increased and point to the next instruction. This instruction is the store-word instruction lsw 22 30 24. It is decodable, the program counters are divisible by 4 and lie within the code range. Thus, no runtime error occurs. The store-word instruction stores the return value 0 kept in the result register 22 into the main memory at the address of the local result variable of the system call. It does not overwrite the message since local variables are stored on the stack. This area does not overlap with the memory region for global or heap variables, hence the load-store target is addressed below the heap base. Moreover, the program counters are increased such that they point to the first address after the assembly code. During these step-by-step executions of the assembly code we derived all the results stated above. No runtime error occurred so that all states except one compute an empty output ε_Ω. Then, a local step of the sequential VAMP assembly language is performed. The exception occurs when the TRAP instruction is met. Here, the output function returns RECVMSG *mn* which is equal to the output computed in the starting state S_{C0} on the C0 layer (subgoal 1 of the proof is satisfied). All effects of the executions are reflected in the final assembly state (subgoal 2 holds). Finally, the code range remained unchanged and the monotonously advancing program counters never left this memory region (this satisfies subgoal 3). □

Furthermore, we can derive some properties of the state s^5_{asm} that is reached after the execution of all inline assembly instructions.

Lemma 6.3 (Properties of the Final Assembly State)**.** *We require that* common_preconds S_{C0} s^j_{C0} s'_{C0} s^1_{asm} *alloc mn msgval holds. Furthermore, state* s^5_{asm} *is the final state after the execution of the inline assembly code i. e.,* $s^5_{\mathsf{asm}} = \delta_{\mathsf{app}}\ \varepsilon_\Sigma\ (\delta_{\mathsf{app}}$ (RECVSUCCESS *msgval*) $(\delta_{\mathsf{app}}\ \varepsilon_\Sigma\ (\delta_{\mathsf{app}}\ \varepsilon_\Sigma\ s^1_{\mathsf{asm}})))$ *holds.*
Then, we conclude that the final state s^5_{asm} *fulfills well-formedness* is_valid_asm s^5_{asm}*, the special purpose registers have not been changed i. e.,* $s^5_{\mathsf{asm}}.\mathsf{sprs} = s^1_{\mathsf{asm}}.\mathsf{sprs}$ *is satisfied and the state is not in a delay slot in such a way that* $s^5_{\mathsf{asm}}.\mathsf{pcp} = s^5_{\mathsf{asm}}.\mathsf{dpc} + 4$ *is fulfilled. Formally:*

$$\mathsf{is\_valid\_asm}\ s^5_{\mathsf{asm}} \wedge s^5_{\mathsf{asm}}.\mathsf{sprs} = s^1_{\mathsf{asm}}.\mathsf{sprs} \wedge s^5_{\mathsf{asm}}.\mathsf{pcp} = s^5_{\mathsf{asm}}.\mathsf{dpc} + 4$$

Proof. These facts can be derived easily from Lemma 6.2 on page 152 examining the final state. □

Constructing a Consistent C0 State. After the successful execution of the entire assembly instruction list, we obtain the final assembly state s^5_{asm} (more specifically $s^5_{\mathsf{asm}} =$

δ_{app} (RecvSuccess *msg*) (δ_{app} ε_Σ (δ_{app} ε_Σ s^1_{asm}))). According to the proposed method in Section 2.5 we use the final state s^5_{asm} to construct a corresponding consistent C0 state s''_{C0}. We use the construction function C0_asm_upd that takes the old C0 state s'_{C0}, the old and new assembly states (i. e., s^1_{asm} and s^5_{asm} respectively), an allocation function *alloc* and a list of modified elementary g-variables. In the receive case, the message and the result value have been written into the memory of state s^5_{asm}. Recall Section 3.4.1 that the message type TYPE_Message_Struct'ty is not elementary. Instead of listing all elementary g-variables of the message, we introduced a function msg_gvars that takes the root g-variable of the message and computes this list. Thus, we obtain the root g-variable of the message by taking the left value of the evaluated dereferenced message pointer. Thus, the first part of the list containing all modified elementary g-variables is given by:

msg_gvars $\lceil$eval$_m$ s^j_{C0} (Deref (*es* ! 0))$\rceil$.ds_lval

For the updated result variable, we append the g-variable value of the local "result" variable of the topmost stack frame to the list. Hence, the entire list keeping the modified elementary g-variables is given below:

msg_gvars $\lceil$eval$_m$ s^j_{C0} (Deref (*es* ! 0))$\rceil$.ds_lval $\circledcirc$
[gvar_lm (recursion_depth s^j_{C0}.pstate.mem) "result"]

We first show that the preconditions preconds_C0_asm_upd hold, so that we can construct a consistent C0 state.

Lemma 6.4 (C0 Construction Preconditions)**.** *We assume that the common preconditions* common_preconds S_{C0} s^j_{C0} s'_{C0} s^1_{asm} *alloc mn msgval hold. Moreover, the first statement of the program rest in state* s^j_{C0} *is a call of function* "olosRecvMsg" *with the left variable lv and the parameter list es (i. e.,* fst_stmt s^j_{C0}.pstate.prog = SCall *lv* "olosRecvMsg" *es sid). Furthermore, state* s^5_{asm} *equals the final state after the execution of all inlined assembly instructions (i. e.,* s^5_{asm} = δ_{app} ε_Σ (δ_{app} (RecvSuccess *msgval*) (δ_{app} ε_Σ (δ_{app} ε_Σ *d*)))*). Then, we conclude that predicate* preconds_C0_asm_upd *is satisfied. Formally:*

preconds_C0_asm_upd s'_{C0} s^1_{asm} s^5_{asm} *alloc*
(msg_gvars $\lceil$eval$_m$ s^j_{C0} (Deref (*es* ! 0))$\rceil$.ds_lval $\circledcirc$
[gvar_lm (recursion_depth s^j_{C0}.pstate.mem) "result"])

Proof. After unfolding the definition of preconds_C0_asm_upd we encounter all subgoals defined in Section 2.5. Thus, we prove that each subgoal is fulfilled:

- Due to the final state derived in Lemma 6.2 on page 152 predicate pcs_exec_value_correct is satisfied, the code region and the special purpose registers have not been modified, hence code_region_const and reg_pointers_const hold.
- The root g-variable of the message is either global or a heap variable, hence all its elementary g-variables are global or heap variables. The result variable is a local variable of the topmost stack frame so that is_top_local_gvar holds for it.

- All subvariables of an initialized root g-variable are initialized. The successful evaluation of the root g-variable of the message implies its initialization. The result variable is a root variable.
- From the given root g-variable of the message *msg_gvar* computes all elementary g-variables. The result g-variable is an elementary g-variable because its of type integer.
- The result variable is located in the topmost local memory frame, hence it is reachable. All subvariables of a reachable g-variable are reachable, too. The root g-variable of the message is reachable after the stack extension.
- After the execution of the inlined assembly instructions only the message and the result variable have been changed.

□

The memory update of the construction function is done sequentially for each changed elementary g-variable. Regarding each update separately is very tedious, hence we compressed the updates logically. The next lemma states that updating all elementary g-variables of the message with the corresponding elementary value equals a single memory update with the complex structural message type TYPE_Message_Struct'ty.

Lemma 6.5 (Write Message into C0 Memory)**.** *Assuming that*

- *the common preconditions hold i. e.,*
  $\mathsf{common\_preconds}\ S_{\mathsf{C0}}\ \mathsf{s}^{j}_{\mathsf{C0}}\ s'_{\mathsf{C0}}\ s^{1}_{\mathsf{asm}}$ *alloc mn msgval,*
- *the first statement of the program rest in state* $\mathsf{s}^{j}_{\mathsf{C0}}$ *is a call of function* "olosRecvMsg" *with the left variable lv and the parameter list es:*
  $\mathsf{fst\_stmt}\ \mathsf{s}^{j}_{\mathsf{C0}}.\mathsf{pstate.prog} = \textsc{Scall}\ lv$ "olosRecvMsg" *es sid, and*
- *state* s^{5}_{asm} *equals the final state after the execution of the inline assembly portion i. e.,* $s^{5}_{\mathsf{asm}} = \delta_{\mathsf{app}}\ \varepsilon_{\Sigma}\ (\delta_{\mathsf{app}}\ (\textsc{RecvSuccess}\ msgval)\ (\delta_{\mathsf{app}}\ \varepsilon_{\Sigma}\ (\delta_{\mathsf{app}}\ \varepsilon_{\Sigma}\ s^{1}_{\mathsf{asm}})))$.

Then, the sequential memcell construction of all elementary g-variables of the message type from a given final assembly state equals a single memory update of the root g-variable with the entire message. Formally:

$$
\begin{aligned}
&\mathsf{C0\_mem\_asm\_upd}\ \mathsf{s}^{j}_{\mathsf{C0}}.\mathsf{ttab}\ s'_{\mathsf{C0}}.\mathsf{pstate.mem}\ s^{5}_{\mathsf{asm}}\ alloc\\
&\ (\mathsf{msg\_gvars}\ \lceil \mathsf{eval_m}\ \mathsf{s}^{j}_{\mathsf{C0}}\ (\textsc{Deref}\ (es\ !\ 0))\rceil.\mathsf{ds\_lval}) =\\
&\mathsf{mem\_update}\ \mathsf{s}^{j}_{\mathsf{C0}}.\mathsf{ttab}\ s'_{\mathsf{C0}}.\mathsf{pstate.mem}\ \lceil \mathsf{eval_m}\ \mathsf{s}^{j}_{\mathsf{C0}}\ (\textsc{Deref}\ (es\ !\ 0))\rceil.\mathsf{ds\_lval}\\
&\ (\!|\mathsf{ds\_lval} = \mathsf{default},\ \mathsf{ds\_type} = \mathsf{TYPE\_Message\_Struct'ty},\\
&\ \ \mathsf{intermediate} = \mathsf{True},\ \mathsf{ds\_initialized} = \mathsf{True},\\
&\ \ \mathsf{ds\_data} = \mathsf{intlist2struct}\ msgval|\!)
\end{aligned}
$$

Proof. The root g-variable of the message is either a global or a heap g-variable. Hence, we distinguish between both cases. Note that all elementary g-variables inherit properties of their root g-variable in such a way that they are either all global or heap variables as well. The first value of the message type TYPE_Message_Struct'ty is an unsigned integer and updated successfully in the memory region determined by the root g-variable. The following values are integers and we show inductively that the memory update of an elementary variable succeeds. This includes to show that all addresses of the elementary

g-variables are disjoint and immediately consecutive. Furthermore, each single message value stored in the assembly memory is successively updated in the C0 memory. Recall that the correct assignment between both memory models is ensured because function C0_mem_asm_update_gvar uses the allocation function to construct the new C0 memory (recall Section 2.5). The allocation function takes care of the correct assignment between values at a certain address in the assembly main memory and the memory object in the C0 memory. Furthermore, the sequential updates of each elementary g-variable up to the n-th element of the message type should equal one single memory update of the entire message type TYPE_Message_Struct'ty up to the same element. For each memory update we have to prove that the symbol tables remain unchanged. This holds because memory updates do not affect the symbol tables. □

Finally, we can prove that the constructed C0 state s''_{C0} is equal to C0 state s'_{C0} after we updated the memory with an entire message and the result value. Note that the message is either written into the global memory or the heap memory according to the root g-variable of the message. The result variable on the topmost local memory frame holds the result value of the system call that is 0 after a successful message transmission.

Lemma 6.6 (Valid C0 State Construction)**.** *We assume that*

- *the common preconditions hold i. e.,*
  common_preconds S_{C0} s^j_{C0} s'_{C0} s^1_{asm} *alloc mn msgval,*
- *the first statement of the program rest in state* s^j_{C0} *is a call of function* "olosRecvMsg" *with the left variable lv,the parameter list es and the statement identifier sid:*
  fst_stmt s^j_{C0}.pstate.prog = SCall *lv* "olosRecvMsg" *es sid,*
- *state* s^5_{asm} *equals the final state after the execution of the inline assembly portion i. e.,* $s^5_{asm} = \delta_{app}\ \varepsilon_{\Sigma}\ (\delta_{app}\ (\text{RecvSuccess}\ msgval)\ (\delta_{app}\ \varepsilon_{\Sigma}\ (\delta_{app}\ \varepsilon_{\Sigma}\ s^1_{asm})))$
- *the data slices to update the entire message and the result value are given with*
  $msg_{ds} = (\!|$ds_lval = default, ds_type = TYPE_Message_Struct'ty, intermediate = True, ds_initialized = True, ds_data = intlist2struct *msgval*$|\!)$ *and*
  $res_{ds} = (\!|$ds_lval = default, ds_type = Integer, intermediate = False, ds_initialized = True, ds_data = λi. **if** $i = 0$ **then** memcellInt 0 **else** default$|\!)$ *respectively, and*
- *the root g-variable of the message and the local g-variable of the result variable are given with*
  $msg_g = \lceil$eval$_m$ s^j_{C0} (Deref (*es* ! 0))$\rceil$.ds_lval *and*
  res_g = gvar_lm (recursion_depth s^j_{C0}.pstate.mem) "result" *respectively.*

Thus, we conclude that the construction of a C0 state s''_{C0} *that is consistent to the final assembly state* s^5_{asm} *succeeds. More specifically, the memory of the constructed C0 state has been updated with the message on the one hand, and the result value on the other hand. Finally, the first statement of the program rest has been deleted. Formally:*

$$\begin{aligned}
&\text{C0_asm_upd}\ s'_{C0}\ s^1_{asm}\ s^5_{asm}\ alloc\ (\text{msg_gvars}\ msg_g \circledcirc [res_g]) = \\
&\lfloor s'_{C0}(\!|\text{pstate} := (\!|\text{mem} = \lceil \text{mem_update}\ s^j_{C0}.\text{ttab} \\
&\qquad\qquad \lceil \text{mem_update}\ s^j_{C0}.\text{ttab}\ s'_{C0}.\text{pstate.mem}\ msg_g\ msg_{ds}\rceil\ res_g\ res_{ds}\rceil, \\
&\qquad \text{prog} = \text{remove_fst_stmt}\ s'_{C0}.\text{pstate.prog}|\!)|\!)\rfloor
\end{aligned}$$

Proof. We know already from Lemma 6.5 on page 156 that the memory update of each individual elementary g-variable of the message type succeeds and can be comprised to a single memory update with the entire message. The remaining update of the local result variable succeeds as well because the g-variable is a root variable and the data slice is initialized. Furthermore, all restrictions of predicate preconds_C0_asm_upd are fulfilled due to Lemma 6.4 on page 155. Then, the new constructed C0 state stores the message on the one hand and the result value on the other hand. Moreover, the first statement of the program rest is removed. □

Note that both sides of the equation in the lemma above refer to the previously mentioned C0 state s''_{C0}. When no interrupt occurs during the execution of the inline assembly statement and the C0 state construction succeeds, the constructed state s''_{C0} and the final assembly state s^5_{asm} fulfill the sequential simulation relation consistent. A successful state construction implies that all the requirements of predicate preconds_C0_asm_upd that ensure code, data and control consistency are satisfied.

Note that the return destination in the topmost stack frame has not been affected after the construction of a new C0 state so that toprd s'_{C0} = toprd s''_{C0} holds. After the construction of the new C0 state s''_{C0} the first statement is the remaining RETURN statement:

fst_stmt s''_{C0}.prog = RETURN (VARACC "result") *sid'*

Execution of the RETURN Statement

The successful execution of a RETURN *expr sid* statement has the following effects:

- the result variable is updated with the evaluated return expression *expr*,
- the topmost stack frame is deleted and,
- the new program rest is set to SKIP

The following constraints have to hold to ensure that the transition function does not fail:

1. the evaluation of the return expression *expr* succeeds
2. the return destination stored in the topmost local stack frame is either globally defined or in the topmost frame of the stack (before the function call was executed)

We prove that the execution of the RETURN statement in the body of the system call function *olosRecvMsg* is successful.

Lemma 6.7 (Successful Execution of the RETURN Statement)**.** *We assume that*

- *the common preconditions hold i. e.,*
  common_preconds S_{C0} s^j_{C0} s'_{C0} s^1_{asm} *alloc mn msgval,*
- *the first statement of the program rest in state* s^j_{C0} *is a call of function* "olosRecvMsg" *with the left variable lv,the parameter list es and the statement identifier sid:* fst_stmt s^j_{C0}.pstate.prog = SCALL *lv* "olosRecvMsg" *es sid,*
- *the construction of a new C0 state* s''_{C0} *succeeds and the g-variable of the local variable* "result" *belongs to the list of elementary modified g-variables:*
  C0_asm_upd s'_{C0} s^1_{asm} s^5_{asm} *alloc* (*gl* ⊙ [gvar_lm (recursion_depth s^j_{C0}.pstate.mem) "result"]) = $\lfloor s''_{C0} \rfloor$,

- *the final assembly state* s^5_{asm} *is valid i. e.,* is_valid_asm s^5_{asm} *is satisfied, and*
- *the evaluation of the variable access of the local result variable* "result" *of the function call returns a data slice that is initialized, is of type integer and contains the result value of the system call:*
  $\mathsf{eval}_{\mathsf{m}}\ s''_{\mathsf{C0}}$ (VarAcc "result") =
  $\lfloor$(|ds_lval = gvar_lm (recursion_depth s^j_{C0}.pstate.mem) "result", ds_type = Integer,
  intermediate = False, ds_initialized = True,
  ds_data = λi. **if** $i = 0$ **then** memcellInt 0 **else** default|)$\rfloor$

With all these assumptions, we conclude that the execution of the Return *statement in state* s''_{C0} *succeeds. Moreover, the return destination of the topmost stack frame has been updated with the result variable of the system call and the stack frame has been deleted afterwards. Finally, the first statement of the program rest is the* Skip *statement. Formally:*

$\delta_{\mathsf{C0m}}\ s''_{\mathsf{C0}} \neq \bot \wedge$
$\lceil\delta_{\mathsf{C0m}}\ s''_{\mathsf{C0}}\rceil$.pstate.mem =
remove_toplm
$\lceil$mem_update s''_{C0}.ttab s''_{C0}.pstate.mem (toprd s''_{C0}.pstate.mem)
(|ds_lval = gvar_lm (recursion_depth s^j_{C0}.pstate.mem) "result",
ds_type = Integer, intermediate = False, ds_initialized = True,
ds_data = λi. **if** $i = 0$ **then** memcellInt 0 **else** default|)$\rceil \wedge$
fst_stmt $\lceil\delta_{\mathsf{C0m}}\ s''_{\mathsf{C0}}\rceil$.pstate.prog = Skip

Proof. The memory update does not fail because the data slice is initialized and the g-variable that is modified is a root g-variable. Recall that the execution of the Return statement updates the return destination of the topmost stack frame that stores the left variable of the function call. It has been set during the stack extension of the SCall statement execution and remained unchanged during the C0 state construction. Moreover, the local memory stack is not empty because it contains at least the stack frame of the system call. Thus, it is possible to remove the topmost stack frame. The execution of the Return statement does not fail, the memory has been changed in the way we described above and program rest is replaced by a Skip statement. □

Figure 6.4 on page 151 depicts the different C0 program states before (c) and after (d) the execution of the Return statement. Note that we sketch only one possible memory state. The message value may be either stored in the global or the heap memory whereas the updated result variable is located in the global memory or the topmost stack frame (before the stack has been extended for the system call).

After the successful execution of the Return statement, the entire body of the system call function has been executed and we are in the successor state s^{j+1}_{C0}. Its memory contains the message on the one hand and the result value of the system call on the other hand. The first statement of the program rest is given with

fst_stmt s^{j+1}_{C0}.pstate.prog = Skip

System Call Simulation Theorems

In this subsection, we show that one abstract C0 transition of a system call can be simulated by a well-defined sequence of C0 small-steps combined with several assembly transitions. More specifically this sequence resembles the execution of a system-call implementation. A C0 program state consists of two components: the memory and the program rest. We consider both of them separately during all transitions belonging to the execution of a system-call implementation and then combine the results for state equality.

Equality of Program Rests. We know that the execution of a system-call implementation succeeds. Furthermore, the three OLOS system calls have the same structure. Thus, we can formulate a lemma that generally states that the program rest after the execution of all system call statements equals the function that simply replaces the first statement with a SKIP.

Lemma 6.8 (Equality of Program Rests)**.** *Assuming that*

- *a non-empty valid C0 application state* S_{C0} *i. e.,* $\mathsf{is\_valid}_{\mathsf{app}}\ S_{\mathsf{C0}} \wedge S_{\mathsf{C0}} = \lfloor \mathsf{s}^{j}_{\mathsf{C0}} \rfloor$ *holds,*
- *the application neither intends to perform a local step nor does the output function* ω_{app} *signal a runtime error i. e.,* $\neg\ \mathsf{is\_runtime\_error}\ (\omega_{\mathsf{app}}\ S_{\mathsf{C0}}) \wedge \omega_{\mathsf{app}}\ S_{\mathsf{C0}} \neq \varepsilon_{\Omega}$ *is fulfilled,*
- *the* SCALL *statement is executed successfully:* $\delta_{\mathsf{C0m}}\ \mathsf{s}^{j}_{\mathsf{C0}} = \lfloor s'_{\mathsf{C0}} \rfloor$*,*
- *the constructed C0 state* s''_{C0} *after the executed portion of inline assembly does not fail i. e.,* $\mathsf{C0\_asm\_upd}\ s'_{\mathsf{C0}}\ d\ d'\ alloc\ gl = \lfloor s''_{\mathsf{C0}} \rfloor$*, and*
- *the* RETURN *statement succeeds i. e.,* $\delta_{\mathsf{C0m}}\ s''_{\mathsf{C0}} = \lfloor \mathsf{s}^{j+1}_{\mathsf{C0}} \rfloor$.

Then, we conclude that the program rest of $\mathsf{s}^{j+1}_{\mathsf{C0}}$ *equals the program rest that simply replaces the first statement of state* $\mathsf{s}^{j}_{\mathsf{C0}}$ *with a* SKIP *instruction. Formally:*

$$\mathsf{s}^{j+1}_{\mathsf{C0}}.\mathsf{pstate.prog} = \mathsf{rm\_scall}\ \mathsf{s}^{j}_{\mathsf{C0}}.\mathsf{pstate.prog}$$

Proof. If the application neither intends to perform a local step nor does the output function ω_{app} signal a runtime error, it requests a service from the operating system. Thus, we know by unfolding the definitions of the output functions ω_{app} and $\mathsf{get\_output}$ the first statement of the program rest is an SCALL statement. More specifically, the called functions are "olosExFinished", "olosSendMsg" or "olosRecvMsg" with precise function definitions (recall Section 4.2.3). Figure 6.4 on page 151 depicts the progress of the first statement in a program rest during single executions of the system call implementation. The small-step semantics of a successful function call substitutes the SCALL statement by the body of the called function. From the function definitions we know that for all system calls the function body consists of exactly two statements: an inlined assembly and a RETURN statement. After the execution of the inlined assembly statement, a new C0 state s''_{C0} is constructed. The construction function removes the first statement of

the program rest in state s'_{C0}. Hence, the first statement of the new C0 state s''_{C0} is the RETURN statement. Finally, the successful execution of the RETURN statement sets the program rest to SKIP. These changes are equal to the substitution of a function call with the SKIP statement. □

Equality of Memory States. After we have shown the equivalence of the program rests for OLOS system calls in general, we prove that the abstract transition function δ_{app} of system call *olosRecvMsg* computes the same memory state as the execution of the entire function body of the system call implementation.

Lemma 6.9 (Equality of Memory States)**.** *We assume that*

- *the common preconditions hold i. e.,*
  common_preconds S_{C0} s^j_{C0} s'_{C0} s^1_{asm} *alloc mn msgval,*
- *the first statement of the program rest in state* s^j_{C0} *is a call of function* "olosRecvMsg" *with the left variable lv and the parameter list es:*
  fst_stmt s^j_{C0}.pstate.prog = SCALL *lv* "olosRecvMsg" *es sid, and*
- *state* s^5_{asm} *equals the final state after the execution of the inline assembly portion i. e.,*
  $s^5_{asm} = \delta_{app}\ \varepsilon_\Sigma\ (\delta_{app}$ (RECVSUCCESS *msgval*) $(\delta_{app}\ \varepsilon_\Sigma\ (\delta_{app}\ \varepsilon_\Sigma\ s^1_{asm})))$

We conclude that the memory state after the execution of the system call implementation of olosRecvMsg *equals the one we obtain after performing the corresponding abstract transition function* δ_{app}*. Formally:*

$$\lceil\delta_{C0m}\ \lceil C0\_asm\_upd\ s'_{C0}\ s^1_{asm}\ s^5_{asm}\ alloc\ (msg\_gvars\ \lceil eval_m\ s^j_{C0}\ (\text{DEREF}\ (es\ !\ 0))\rceil.ds\_lval \circledcirc [gvar\_lm\ (recursion\_depth\ s^j_{C0}.pstate.mem)\ \text{"result"}])\rceil\rceil.pstate.mem = \lceil\delta_{app}\ (\text{RECVSUCCESS}\ msgval)\ S_{C0}\rceil.pstate.mem$$

Proof. In this proof, we have to examine the memory changes of each step within the system call function that is illustrated for one particular case in Figure 6.4 on page 151. The effects of the SCALL statement execution is the extension of the local memory stack. From Lemma 6.6 on page 157 we know that the C0 state construction after the execution of the inline assembly code does not fail and more specifically, the message has been written either in the global or in the heap memory. In addition, the constructed C0 state has an updated result variable in the topmost stack frame. During the execution of the RETURN statement this value is copied into the left variable of the function call before the topmost stack is removed. Note that the left variable of the function call is either located in the global memory or the topmost stack frame of the local memory *before* the system call has extended the stack. Thus, the deletion of the topmost stack frame neither affects the message value nor the result value. We learn in Section 4.2.3 that the abstract transition function δ_{app} in the successful receive case manipulates the C0 application state with three functions: receiving, set_result and del_syscall. receiving stores the message value either in the global or in the heap memory, depending on the root g-variable of the message. Obviously, the global and the heap memory after the C0 state construction are equal. Function set_result updates the left variable of the

function call directly with the result value and function del_syscall does not affect the memory state of the C0 application state. Extending the stack, copying the updated result variable of the topmost stack frame into the left variable of the function and removing the topmost stack frame have the same effects as the direct update of the left variable with the result value. □

Now we have all prerequisites in place to prove the simulation of the successful system call *olosRecvMsg*.

Theorem 6.10 (System Call Simulation of *olosRecvMsg*)**.** *We assume that*

- *the common preconditions hold i. e.,*
  $\mathsf{common\_preconds}\ S_{\mathsf{C0}}\ \mathsf{s}^{j}_{\mathsf{C0}}\ s'_{\mathsf{C0}}\ s^{1}_{\mathsf{asm}}\ alloc\ mn\ msgval$
- *the first statement of the program rest in state* $\mathsf{s}^{j}_{\mathsf{C0}}$ *is a call of function* "olosRecvMsg" *with the left variable lv and the parameter list es:*
  $\mathsf{fst\_stmt}\ \mathsf{s}^{j}_{\mathsf{C0}}.\mathsf{pstate.prog} = \textsc{SCall}\ lv$ "olosRecvMsg" *es sid, and*
- *state* s^{5}_{asm} *equals the final state after the execution of the inline assembly portion i. e.,* $s^{5}_{\mathsf{asm}} = \delta_{\mathsf{app}}\ \varepsilon_{\Sigma}\ (\delta_{\mathsf{app}}\ (\textsc{RecvSuccess}\ msgval)\ (\delta_{\mathsf{app}}\ \varepsilon_{\Sigma}\ (\delta_{\mathsf{app}}\ \varepsilon_{\Sigma}\ s^{1}_{\mathsf{asm}})))$.

If an application successfully receives a message the transition function of C0 applications δ_{app} *simulates the execution of the implementation of function* "olosRecvMsg"*. Formally:*

$$
\begin{aligned}
&\delta_{\mathsf{app}}\ (\textsc{RecvSuccess}\ msgval)\ S_{\mathsf{C0}} = \\
&\delta_{\mathsf{C0m}} \\
&\ \lceil \mathsf{C0\_asm\_upd}\ s'_{\mathsf{C0}}\ s^{1}_{\mathsf{asm}}\ s^{5}_{\mathsf{asm}}\ alloc \\
&\quad (\mathsf{msg\_gvars}\ \lceil \mathsf{eval_m}\ \mathsf{s}^{j}_{\mathsf{C0}}\ (\textsc{Deref}\ (es\ !\ 0))\rceil.\mathsf{ds\_lval} \circledcirc \\
&\quad\ [\mathsf{gvar\_lm}\ (\mathsf{recursion\_depth}\ \mathsf{s}^{j}_{\mathsf{C0}}.\mathsf{pstate.mem})\ \text{"result"}])\rceil
\end{aligned}
$$

Proof. We know that the function table, the type table and the program size are never changed during all execution steps. The program state consists of the memory state and the program rest. We obtain the equivalence of both memory states by applying Lemma 6.9 on the preceding page. Now, we draw our attention to the program rests: From Section 4.2.3 we know that the abstract transition function δ_{app} in the successful receive case modifies the C0 application state with three functions: receiving, set_result and del_syscall. The former two functions do not affect the program state, the latter uses function rm_scall to replace the first function call with a Skip statement. In order to prove the equivalence of both program rests, we have to discharge some preconditions: We know from common_preconds that the C0 application state S_{C0} is valid and non-empty and that the output function neither returns an empty output ε_{Ω} nor does it indicate an occurred runtime error. Moreover, this predicate includes that the SCall statement execution has succeeded. Lemma 6.2 on page 152 and Lemma 6.6 on page 157 state that the C0 state construction does not fail. Finally, the execution of the Return statement succeeds due to Lemma 6.7 on page 158. Therefore, the abstract transition function indeed simulates the execution of the implementation system call function *olosRecvMsg* in the successful receive case. □

Considering the Remaining Cases. We showed simulation of the *olosRecvMsg* function execution in the successful receive case. As we mentioned before, this case is the

most complex one because it updates the memory on the one hand and the return variable on the other hand. Now, we sketch the proofs for the remaining cases. An application announces its termination for the compute phase in the actual slot by calling *olosExFinished*. In contrast to the other system call primitives, this function has neither parameters nor local variables. The inline assembly portion only consists of the TRAP instruction that does not modify the application's memory. The theorem proving the simulation of system call *olosExFinished* is given below:

Theorem 6.11 (System Call Simulation of *olosExFinished*)**.** *Assuming that*

- *we start in a non-empty C0 application state* $S_{\mathsf{C0}} = \lfloor \mathsf{s}^{j}_{\mathsf{C0}} \rfloor$,
- *that is valid i. e.,* $\mathsf{is\_valid}_{\mathsf{app}}\ S_{\mathsf{C0}}$ *holds,*
- *the application signals its termination for this slot i. e.,* $\omega_{\mathsf{app}}\ S_{\mathsf{C0}} = \textsc{ExFinish}$
- *after the successful execution of the* SCALL *statement we are in state* s'_{C0}*:* $\delta_{\mathsf{C0m}}\ \mathsf{s}^{j}_{\mathsf{C0}} = \lfloor s'_{\mathsf{C0}} \rfloor$,
- *there is an assembly state* s^{1}_{asm} *that can be related to* s'_{C0} *via the C0 simulation relation* consistent *i. e.,*
  $\mathsf{consistent}\ \mathsf{s}^{j}_{\mathsf{C0}}.\mathsf{ttab}\ \mathsf{s}^{j}_{\mathsf{C0}}.\mathsf{ftab}\ s'_{\mathsf{C0}}.\mathsf{pstate}\ alloc\ s^{1}_{\mathsf{asm}}$,
- *the assembly state* s^{1}_{asm} *is valid i. e.,* $\mathsf{is\_valid}_{\mathsf{app}}\ s^{1}_{\mathsf{asm}}$ *is satisfied, and*
- *the application sizes of the C0 application state* S_{C0} *and the assembly state* s^{1}_{asm} *are equal:* $\mathsf{size}_{\mathsf{app}}\ S_{\mathsf{C0}} = \mathsf{size}_{\mathsf{app}}\ s^{1}_{\mathsf{asm}}$

Then, if the application signals its termination in the compute phase of the actual slot the abstract transition function δ_{app} *simulates the C0$_A$ execution of the system call function* "olosExFinish"*.Formally:*

$$\delta_{\mathsf{app}}\ \textsc{FinishSuccess}\ S_{\mathsf{C0}} =$$
$$\delta_{\mathsf{C0m}}\ \lceil \mathsf{C0\_asm\_upd}\ s'_{\mathsf{C0}}\ s^{1}_{\mathsf{asm}}\ (\delta_{\mathsf{app}}\ \textsc{FinishSuccess}\ s^{1}_{\mathsf{asm}})\ alloc\ [] \rceil$$

Proof. The stack is extended after the successful execution of the SCALL statement. The inline assembly code does not change a g-variable in such a way that the list of modified, elementary g-variables required by the update function C0_asm_upd remains empty. This means for the construction of a C0 state that the memory remains unchanged and only the program rest has been modified. Finally, the expression of the RETURN statement is a literal value and is directly written into the left variable of the function. Afterwards the topmost stack frame is removed. The abstract function δ_{app} internally sets the result variable set_result and deletes the function call with del_syscall. We obtain memory equality because extending the stack, writing the left variable of a function directly and removing the topmost stack frame afterwards leads to the same memory state that we obtain by writing the left variable directly. We know from the preconditions that the C0 application state S_{C0} is valid and non-empty, that the output function neither returns an empty output ε_Ω nor does it indicate an occurred runtime error and the SCALL statement execution succeeds. After the TRAP transition, we construct a C0 state with an unchanged memory state. The execution of the RETURN statement succeeds because the memory update of the result variable does not fail and the literal value in the expression can be evaluated. After the execution of the system calls function body the states are equal. □

Finally, we consider all remaining cases where only the result variable of the application is changed. These cases are, in particular, system call *olosSendMsg* and the cases when *olosSendMsg* or *olosRecvMsg* are called with an invalid message number. Then, the error value is stored in the result variable . We formalize the respective theorem below.

Theorem 6.12 (System Call Simulation of the Remaining Cases)**.** *We assume that*

- *we start in a non-empty C0 application state* $S_{C0} = \lfloor s^j_{C0} \rfloor$,
- *this state is valid i. e.,* $\mathsf{is\_valid}_{app}\ S_{C0}$ *holds,*
- *the application requests to exchange a message with the operating system:* $\omega_{app}\ S_{C0} = \textsc{SendMsg}\ msgval\ mn \vee \omega_{app}\ S_{C0} = \textsc{RecvMsg}\ mn$,
- *the first statement of the program rest in state* s^j_{C0} *is a function call with the left variable* lv*, the function name* fn*, the parameter list* es *and the statement identifier* sid *i. e.,* $\mathsf{fst\_stmt}\ s^j_{C0}.\mathsf{pstate.prog} = \textsc{SCall}\ lv\ fn\ es\ sid$ *is fulfilled,*
- *the execution of the* SCall *statement succeeds:* $\delta_{C0m}\ s^j_{C0} = \lfloor s'_{C0} \rfloor$,
- *there is an assembly state* s^1_{asm} *that can be related to* s'_{C0} *via the C0 simulation relation* consistent *i. e.,*
  $\mathsf{consistent}\ s^j_{C0}.\mathsf{ttab}\ s^j_{C0}.\mathsf{ftab}\ s'_{C0}.\mathsf{pstate}\ alloc\ s^1_{asm}$ *holds,*
- *the assembly state* s^1_{asm} *is valid i. e.,* $\mathsf{is\_valid}_{app}\ s^1_{asm}$ *is satisfied,*
- *the application sizes of the C0 application state* S_{C0} *and the assembly state* s^1_{asm} *are equal:* $\mathsf{size}_{app}\ S_{C0} = \mathsf{size}_{app}\ s^1_{asm}$*, and*
- *the response* i *from the operating system is either* InvalidMsgNr *or* SendSuccess*:* $i = \textsc{InvalidMsgNr} \vee i = \textsc{SendSuccess}$*, and*

Then, we conclude that an abstract transition δ_{app} *simulates the execution of the implementation of the corresponding system call function. Formally:*

$$
\begin{aligned}
&\delta_{app}\ i\ S_{C0} = \\
&\delta_{C0m} \\
&\ \lceil \mathsf{C0\_asm\_upd}\ s'_{C0}\ s^1_{asm}\ (\delta_{app}\ \varepsilon_\Sigma\ (\delta_{app}\ i\ (\delta_{app}\ \varepsilon_\Sigma\ (\delta_{app}\ \varepsilon_\Sigma\ s^1_{asm}))))\ alloc \\
&\quad [\mathsf{gvar\_lm}\ (\mathsf{recursion\_depth}\ s^j_{C0}.\mathsf{pstate.mem})\ \text{“result”}]\rceil
\end{aligned}
$$

Proof. In the case that an application successfully sends a message to the operating system or the application requests a message exchange with an invalid message number the modifications of the application state are similar. Then, the operating system only writes the result value into the applications result variable. First, the local stack frame is extended. After the execution of the inline assembly statement the result register has been written. The C0 state construction updates the local result variable in the topmost stack frame with this new value. This value is copied from the Return statement to the functions left variable address and the topmost local stack frame is removed. The abstract transition δ_{app} consists of the functions set_result and del_syscall. With similar, but simpler considerations as in Theorem 6.10 on page 162 we conclude that a direct memory update equals the memory state after extending the stack, writing a local result variable, copying this value to the left variable of the function call and removing the stack. From the preconditions and proved lemmata that the single executions succeed, we derive derive the equality of the program rests by applying Lemma 6.8 on page 160. □

6.1.2 Extending Compiler Correctness to Applications

At the beginning of this chapter we aimed to prove the induction step of the extended compiler-correctness theorem. With the results we derived in the last subsection we can close the gap in the proof.

Theorem 6.13 (Extended Compiler-Correctness Induction Step)**.** *We assume a valid C0 application state* S_{C0} *and a valid assembly state* s_{asm} *that can be related with the consistency relation* $\mathsf{consis}_{\mathsf{app}}$*. Moreover, we obtain a successor state* S'_{C0} *after one successful C0 application step. If these assumptions hold, there exists an output-input sequence* ois'*, an allocation function* alloc *and a final assembly state* s_{asm} *so that*

- *the normalized output-input* $[oi]$ *and the corresponding normalized output-input sequence on the assembly layer* ois' *are equal,*
- *the assembly process successfully advances from the assembly state* s_{asm} *under the output-input sequence* ois' *to a final state* s'_{asm}*,*
- *the final states* S'_{C0} *and* s'_{asm} *are valid and* s'_{asm} *is not in a delay slot, and*
- *the final state* S'_{C0} *simulates* s'_{asm} *under the the allocation function* alloc'*.*

Formally:

$$
\begin{aligned}
&[\![\mathsf{is\_valid}_{\mathsf{app}}\ S_{\mathsf{C0}};\ \mathsf{is\_valid}_{\mathsf{app}}\ s_{\mathsf{asm}};\ s_{\mathsf{asm}}.\mathsf{pcp} = s_{\mathsf{asm}}.\mathsf{dpc} + 4;\\
&\mathsf{consis}_{\mathsf{app}}\ S_{\mathsf{C0}}\ \mathit{alloc}\ s_{\mathsf{asm}};\ \vdash^{\mathsf{proc}}_{\mathsf{C0}} S_{\mathsf{C0}} \xrightarrow{[oi]} S'_{\mathsf{C0}}]\!]\\
&\Longrightarrow \exists \mathit{ois}'\ \mathit{alloc}'\ s'_{\mathsf{asm}}.\\
&\qquad \gg [oi] \ll\ =\ \gg \mathit{ois}' \ll\ \wedge\\
&\qquad \mathsf{code\_range}\ S_{\mathsf{C0}} \vdash^{\mathsf{proc}}_{\mathsf{asm}} s_{\mathsf{asm}} \xrightarrow{\mathit{ois}'} s'_{\mathsf{asm}}\ \wedge\\
&\qquad \mathsf{is\_valid}_{\mathsf{app}}\ S'_{\mathsf{C0}}\ \wedge\\
&\qquad \mathsf{is\_valid}_{\mathsf{app}}\ s'_{\mathsf{asm}}\ \wedge\\
&\qquad s'_{\mathsf{asm}}.\mathsf{pcp} = s'_{\mathsf{asm}}.\mathsf{dpc} + 4 \wedge \mathsf{consis}_{\mathsf{app}}\ S'_{\mathsf{C0}}\ \mathit{alloc}'\ s'_{\mathsf{asm}}
\end{aligned}
$$

Proof. We show this theorem by a case distinction over all output-input pairs: The empty pair $(\varepsilon_\Omega, \varepsilon_\Sigma)$ signals that the application intends to perform a local step. Local steps in C0 consist of all possible statements of the sequential C0 semantics. In this case, we apply the sequential induction step (Lemma 2.2 on page 31). The remaining output-input pairs belong to the system calls. Therefore, we examine the implementation of the corresponding system-call functions. We have already shown that the execution of the implementation code equals an abstract C0 application transition. The implementation code of all system calls consists of a SCall statement followed by an inline assembly statement Asm and a Return statement. From the induction hypothesis we know that there exists a corresponding assembly state s_{asm} that satisfies the simulation relation $\mathsf{consis}_{\mathsf{app}}$. Using the sequential C0 semantics, the SCall statement is executed. From the sequential compiler theorem, we conclude that the execution of the corresponding compiled code yields an assembly state s^1_{asm} that satisfies the sequential consistency relation consistent. Starting in this state, we execute the inlined assembly code of the function body. After several transitions we reach the final state s^5_{asm}. During these executions the code reaches the trap instruction where the transition function δ_{app} on the assembly layer uses an input i from Olos. Note that all the other transitions in

the VAMP assembly layer are sequential i. e., the input-output sequence consists of pairs $(\varepsilon_\Omega, \varepsilon_\Sigma)$. From s^1_{asm}, the final assembly state s^5_{asm} and the C0 state s'_{C0} immediately before the execution of the inlined assembly statement, we construct the corresponding C0 state s''_{C0} directly after the assembly statement. For the assembly state s^5_{asm} and the constructed C0 state s''_{C0} the sequential consistency relation consistent holds. Now, we employ the sequential C0 semantics to execute the RETURN statement and obtain the successor state $\mathsf{s}^{j+1}_{\mathsf{C0}}$. From the compiler theorem, we know that there exists a corresponding assembly state in such a way that consistent is fulfilled. Furthermore, the application sizes have not been changed during all these execution steps. From Theorem 6.10 on page 162, Theorem 6.11 on page 163 and Theorem 6.12 on page 164 we conclude, that the successor state S'_{C0} of the abstract application transition equals state $\mathsf{s}^{j+1}_{\mathsf{C0}}$ so that $\mathsf{consis}_{\mathsf{app}}$ holds. The validity of the assembly state, the well-formedness of the program counters and the non-modified code range follow directly from the compiler correctness theorem. The validity of the final C0 state s''_{C0} can be concluded from the validity axioms Theorem 4.4 on page 101. □

We now may extend the theorem above to output-input sequences on the C0 layer.

Theorem 6.14 (Application Simulation)**.** *Assuming that the parameters img and pages are well-formed wrt.* (valid_initial)*, there is an initial C0-process state S_{C0} satisfying these parameters, and a successful execution leads from S_{C0} using the output-input sequence ois into the state S'_{C0}. If these assumptions hold, there exists an output-input sequence ois′, an allocation function alloc and a final assembly state s_{asm} in such a way that*

- *the normalized output-input* [oi] *and the corresponding normalized output-input sequence on the assembly layer ois′ are equal,*
- *the assembly process successfully advances from the initial assembly state that is uniquely identified by img and pages under the output-input sequence ois′ to a final state s'_{asm},*
- *the final state S'_{C0} simulates s'_{asm} under the the allocation function alloc,*
-

C0 processes simulate assembly processes. Formally:

$$
\begin{aligned}
&[\![\forall s_{\mathsf{asm}} \in \mathsf{set}\ img.\ \mathsf{asm\_int}\ s_{\mathsf{asm}};\ 0 < pages;\ pages \leq \mathsf{TVM\_MAXPAGES};\\
&\ \mathsf{is\_init}_{\mathsf{app}}\ img\ pages\ S_{\mathsf{C0}};\ \vdash^{\mathsf{proc}}_{\mathsf{C0}} S_{\mathsf{C0}} \xrightarrow{ois} S'_{\mathsf{C0}}]\!]\\
&\Longrightarrow \exists ois'\ alloc\ s'_{\mathsf{asm}}.\\
&\qquad \gg ois \ll\ = \gg ois' \ll\ \wedge\\
&\qquad \mathsf{code\_range}\ S_{\mathsf{C0}} \vdash^{\mathsf{proc}}_{\mathsf{asm}} \mathsf{init\_asm}\ img\ pages \xrightarrow{ois'} s'_{\mathsf{asm}} \wedge\\
&\qquad \mathsf{consis}_{\mathsf{app}}\ S'_{\mathsf{C0}}\ alloc\ s'_{\mathsf{asm}}
\end{aligned}
$$

Proof. We prove the theorem by induction on the output-input sequence. We use the sequential compiler correctness (Theorem 2.1 on page 30) to establish the the sequential consistency relation for a successor of the initial assembly state. This state is well-formed and its special purpose registers have not been modified. Thus, the application consistency $\mathsf{consis}_{\mathsf{app}}$ can be established as well. For the induction step, we employ Theorem 6.13 on the preceding page. □

6.2 Computation Step Simulation

In the previous section, we presented a computational model for applications that modelled communication with Olos by using outputs and inputs. For the verification of applications, however, the system calls are distracting. Most notably, Isabelle/Simpl cannot deal with outputs and inputs. Hence, we introduce a *cooperative concurrent* execution model, where we reason sequentially about a complete computation phase i. e., between two calls to the *olosExFinished* primitive. This execution model is the basis of the application semantics in Simpl. Recall that the current scheduled application executes in the compute phase and Olos has the sole task to exchange messages with it. Hence, we model the computation of Olos and the application as a single, sequential program with two separate states: on the one hand the current application and on the other hand the message buffers of the operating system. The internal application state can be manipulated via normal C0 statements whereas the message buffers are only accessible through the message transferring primitives. We define a state of the new computational model as a pair (s_{app}, mb) consisting of a process state s_{app} and a file of message buffers mb. The transitions emulate the behaviour of Olos during the computation phase. Thus, we can reuse the transition function δ_{cc} of Section 4.3.

This model uses a generic application interface, i. e., s_{app} may refer to a C0 state as well as a Vamp assembly state. Thus, application simulation Theorem 6.14 on the preceding page can easily be lifted to cooperative concurrently executing applications:

Theorem 6.15 (Cooperative Concurrent Simulation)**.** *We assume a valid C0 application state S_{C0} and a valid assembly state s_{asm} that can be related with the consistency relation $\mathsf{consis}_{\mathsf{app}}$. Moreover, s_{asm} is not in a delay slot i. e., $s_{\mathsf{asm}}.\mathsf{pcp} = s_{\mathsf{asm}}.\mathsf{dpc} + 4$ holds. Neither the C0 application state S_{C0} nor the successor state S'_{C0} signal a runtime error i. e., $\neg\ \mathsf{is\_runtime\_error}\ (\omega_{\mathsf{app}}\ S_{\mathsf{C0}}) \wedge \neg\ \mathsf{is\_runtime\_error}\ (\omega_{\mathsf{app}}\ (\mathsf{fst}\ (\delta_{\mathsf{cc}}\ (S_{\mathsf{C0}}, mb))))$ is satisfied. Finally, the message buffer mb is valid so that $\mathsf{is\_valid\_MB}\ mb$ is fulfilled. Then, we conclude that a C0 step in the cooperative concurrent model simulates a number of cooperative concurrent assembly steps. Formally:*

$$
\begin{aligned}
&[\![\mathsf{consis}_{\mathsf{app}}\ S_{\mathsf{C0}}\ \mathit{alloc}\ s_{\mathsf{asm}};\ \neg\ \mathsf{is\_runtime\_error}\ (\omega_{\mathsf{app}}\ S_{\mathsf{C0}});\\
&\ \neg\ \mathsf{is\_runtime\_error}\ (\omega_{\mathsf{app}}\ (\mathsf{fst}\ (\delta_{\mathsf{cc}}\ (S_{\mathsf{C0}}, mb))));\ \mathsf{is\_valid}_{\mathsf{app}}\ S_{\mathsf{C0}};\\
&\ \mathsf{is\_valid}_{\mathsf{app}}\ s_{\mathsf{asm}};\ s_{\mathsf{asm}}.\mathsf{pcp} = s_{\mathsf{asm}}.\mathsf{dpc} + 4;\ \mathsf{is\_valid\_MB}\ mb]\!]\\
&\Longrightarrow \exists t\ \mathit{alloc}'\ s'_{\mathsf{asm}}.\\
&\qquad \textbf{let}\ (S'_{\mathsf{C0}}, mb') = \delta_{\mathsf{cc}}\ (S_{\mathsf{C0}}, mb)\\
&\qquad \textbf{in}\ (s'_{\mathsf{asm}}, mb') = (\delta_{\mathsf{cc}}{}^{t})\ (s_{\mathsf{asm}}, mb) \wedge \mathsf{consis}_{\mathsf{app}}\ S'_{\mathsf{C0}}\ \mathit{alloc}'\ s'_{\mathsf{asm}}
\end{aligned}
$$

Proof. First, we show that the inputs i chosen by function δ_{cc} always match the application output. Thus, the transition $\delta_{\mathsf{app}}\ i\ S_{\mathsf{C0}}$ succeeds. Theorem 6.14 on the facing page states that there exists a corresponding output-input sequence ois' in such a way that this sequence obtains exactly one pair $(\omega_{\mathsf{app}}\ S_{\mathsf{C0}}, i)$ and a number of empty pairs $(\varepsilon_\Omega, \varepsilon_\Sigma)$ i. e., $\gg ois' \ll\ = [(\omega_{\mathsf{app}}\ S_{\mathsf{C0}}, i)]$.

Moreover, there is a successful assembly execution under ois' with a final state s'_{asm} that starts in state s_{asm}, where $\mathsf{consis}_{\mathsf{app}}\ S'_{\mathsf{C0}}\ \mathit{alloc}'\ s'_{\mathsf{asm}}$ holds. When δ_{cc} gets output

ε_Ω of the currently scheduled application, it responds with the empty input ε_Σ. Then we apply $|ois'|$ times function δ_{cc} to yield the successor state s'_{asm}. Furthermore, the message buffers are equal in both cases. □

The centerpiece of our cooperative concurrent execution model is that it allows us to reuse Isabelle/Simpl. The necessary adaptation of the existing technology for application verification is straightforward: It implies to specify the effects of the primitives for *olosSendMsg* and *olosRecvMsg* in the Simpl language and show that this specification corresponds with the definition of δ_{cc} for these cases. Using the Hoare logic of Isabelle/Simpl, we can efficiently reason about functional correctness of applications between two calls of the *olosExFinished* primitive as well as conveniently establish the freedom of runtime errors.

6.3 Embedding Applications into the Overall ECU Model

In Section 6.2, we claimed that the transition function δ_{cc} emulates the OLOS transitions of a scheduled application during the computation phase. Now, we prove that assertion. Therefore, we define a projection function Π that extracts the scheduled application and the file of message buffers of an ECU state. Formally:

$$\Pi\ pid\ s_{ecu} \equiv (s_{ecu}.\mathsf{AM}\ pid,\ s_{ecu}.\mathsf{MB_a})$$

Furthermore, we define an injection function inj_{pid}, which updates the state of application with identifier *pid* in the ECU state s_{ecu} by s_{app}.

$$\mathsf{inj}_{pid}\ s_{app}\ s_{ecu} \equiv s_{ecu}(\!|\mathsf{AM} := s_{ecu}.\mathsf{AM}(pid := \mathsf{fst}\ s_{app}),\ \mathsf{MB_a} := \mathsf{snd}\ s_{app}|\!)$$

Predicate calls_finish encapsulates the fact that an application with identifier *pid* in an ECU state s_{ecu} calls EXFINISH. Formally:

$$\mathsf{calls\_finish}\ pid\ s_{ecu} \equiv \omega_{app}\ (s_{ecu}.\mathsf{AM}\ pid) = \text{EXFINISH}$$

Finally, predicate is_computing holds for the ECU state s_{ecu} during the compute phase when an application with identifier *pid* is currently scheduled and has not terminated its execution i. e., the idleflag has not ($\neg\ s_{ecu}$.idleflag) been raised yet:

$$\begin{aligned}&\mathsf{is\_computing}\ pid\ (AST, BT)\ s_{ecu} \equiv\\ &\quad AST\ !\ s_{ecu}.\mathsf{abc\_dev.CSN} = \mathsf{pid2nat}\ pid \wedge \neg\ s_{ecu}.\mathsf{idleflag} \wedge \neg\ s_{ecu}.\mathsf{abc\_dev.INT}\end{aligned}$$

The following theorem states that transition function δ_{cc} indeed emulates the ECU transitions of a scheduled application during the computation phase.

Theorem 6.16 (Application Embedding)**.** *Assuming that application pid is scheduled in the current slot and computing in state s_{ecu}. Moreover, it did not terminate its execution yet i. e., there has no call for* EXFINISH*. Then, the transition δ_{cc} of the projection with pid followed by an injection into s_{ecu} is equal to a transition δ_{ECU} executing a kernel computation. Formally:*

$[\![\mathsf{is\_computing}\ pid\ \Theta\ s_{\mathsf{ecu}};\ \neg\ \mathsf{calls\_finish}\ pid\ s_{\mathsf{ecu}}]\!]$
$\Longrightarrow \mathsf{inj}_{pid}\ (\delta_{\mathsf{cc}}\ (\Pi\ pid\ s_{\mathsf{ecu}}))\ s_{\mathsf{ecu}} = \delta_{\mathsf{ECU}}\ \Theta\ \bot\ s_{\mathsf{ecu}}$

Proof. From the assumptions we know that the ECU transition function δ_{ECU} is in the compute phase on the one hand and that the result of the compute phase Compute_Phase only modifies the currently scheduled application state and the message buffers on the other hand. Thus, extracting the scheduled application and the file of message buffers of an ECU state, performing transition δ_{cc} and embedding the result into the ECU state again equals the effects of transition δ_{ECU} under the given conditions. □

6.4 Reasoning about Applications – a Practical Example

In this section, we give an outlook on a practical example to demonstrate the applicability of our approach. For this purpose, we regard a simple application program for a cruise control. [3] Note that all OLOS applications share a common control flow, which is depicted in Figure 6.5: After an application-specific set-up, they implement an infinite loop. The loop body contains a function call to a function `compute`, which implements the actual functionality of the application, followed by a system call of `olosExFinished`.

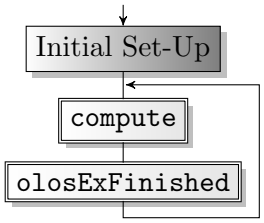

Figure 6.5: General control-flow of applications

Our example program features a global variable `target_speed` storing the speed that the regulator is aiming for. Furthermore, there is a global Boolean variable `enabled` that is true iff the speed control is enabled.

The `compute` function (see Figure 6.6 on the following page) reads the message buffer 0 to receive one of the commands *ON*, *OFF*, *INCREASE*, and *DECREASE* as well as the message buffer 1 to receive the current speed. The function writes two buffers that are read by the accelerator unit (message buffer 2) and the brake unit (message buffer 3). Then, function `compute` adjusts the global variables wrt. the received command and subtracts the current from the target speed. If the difference is positive, the function sends the difference to message buffer 2 (which we assume to be read by the accelerator unit) and value 0 to message buffer 3 (which we assume to be read by the brake unit). If the difference is negative, the function sends the absolute value to buffer 3 and value 0 to buffer 2. Afterwards, `compute` returns and the program calls *olosExFinished*.

For the verification of the `compute` function, we employ the existing technology for sequential reasoning: At first, we mechanically translate the C0 code into Simpl. Moreover, all OLOS system calls are modelled in Simpl as XCalls. Then, we formally specify the functionality in terms of Hoare triples and, conveniently relying on the Hoare logics, prove the correctness of our specification. The loop body only calls the EXFINISH primitive after the execution of function `compute`. Obviously, this call does not alter the application's variables and the message buffer. Thus, the proved property holds for a

[3] We thank Matthias Daum for providing the example in a recent joint publication [DSS10].

```
int compute() {
  unsigned int command; unsigned int current_speed;

  dummy = olosRecvMsg(buffer, 0u); command = buffer->Field; // read command
  dummy = olosRecvMsg(buffer, 1u); current_speed = buffer->Field;     // read current speed

  // target_speed adjustment
  if (enabled) {
    if (command == CC_INCREASE) {
      if (target_speed < MAX_SPEED - 1u) { target_speed = target_speed + 2u; }
    }
    else if (command == CC_DECREASE) {
      if (target_speed > MIN_SPEED + 1u) { target_speed = target_speed - 2u; }
    }
    else if (command != CC_SET) { enabled = false; }
  }
  else if (current_speed >= MIN_SPEED && (command == CC_SET ||
           command == CC_INCREASE || command == CC_DECREASE)) {
    enabled = true;
    if (current_speed >= MAX_SPEED) { target_speed = MAX_SPEED; }
    else { target_speed = current_speed; }
  }

  // speed regulation
  *buffer = INIT_BUFFER;
  if (!enabled) { dummy = olosSendMsg(buffer, 2u); dummy = olosSendMsg(buffer, 3u); }
  else if (current_speed > target_speed) {
    dummy = olosSendMsg(buffer, 2u);
    buffer->Field = current_speed - target_speed; dummy = olosSendMsg(buffer, 3u);
  }
  else {
    dummy = olosSendMsg(buffer, 3u);
    buffer->Field = target_speed - current_speed; dummy = olosSendMsg(buffer, 2u);
  }

  return 0;
}
```

Figure 6.6: Function `compute` of our simple cruise-control application

complete computation phase (i. e., the loop body). Finally, we know from the property transfer theorem of Isabelle/Simpl that there is an equivalent property over the C0 semantics. The property transfer between Simpl and the small-step layer used for CVM correctness has been done before [ASS08], demonstrating the general applicability of our approach.

Using Theorem 6.15 on page 167, we can then infer that the property can be translated down to assembly level. Furthermore, Theorem 6.16 on page 168 allows us to infer properties about the whole ECU behavior by combining verification results from the applications running on the ECU. Recall that our example application relied on several assumptions: The message buffers 0 and 1 are assumed to stem from sensors, and the buffers 2 and 3 should be sent to other control units. Consequently, the static schedule should provide slots, where the messages received from the bus are stored into the buffers 0 and 1; moreover, the buffers 2 and 3 should be sent onto the bus. Furthermore, the

applications sharing the same ECU as our example application, should not alter the buffers after receiving or before sending, respecively. In addition, we have assumed that the example application calls ExFinish before the slot end. Within the Hoare logic, we can prove that our `compute` function terminates. Thus, the only remaining issue is to find an upper bound for the worst-case execution time, which can be done by static analysis [HF04]. Eventually, we can then argue that the values computed by our example application are indeed sent onto the bus several slots later.

7 Conclusion and Future work

> Achilles had overtaken the Tortoise, and had seated himself comfortably on its back. "So you've got to the end of our race-course?" said the Tortoise. "Even though it does consist of an infinite series of distances? I thought some wiseacre or other had proved that the thing couldn't be done?"
>
> *Lewis Carroll, in: "What the Tortoise Said to Achilles"*

Contents

The contributions of this thesis are the implementation of the real-time operating system Olos and the formal specification of an entire ECU as a part of the automotive subproject in Verisoft. We have formally verified functional correctness of Olos in terms of a simulation proof on the one hand and presented an approach to pervasively verify applications running on top of it on the other hand. In the following sections, we summarize our formal results, estimate the verification effort, and give a perspective for future work.

7.1 Formal Results

In the course of this thesis we have considered a distributed real-time system consisting of several ECUs and a bus in the context of pervasive formal verification. More specifically, we have focussed on a real-time operating system that was designed for an industrial context.

Figure 7.1 on the next page depicts an overview of our results. We implemented the Olos kernel as a C0 program. A syntax translator automatically generates Simpl code from the source code of Olos. Our operating system runs as a Cvm kernel machine. Therefore, we defined the effects of Cvm integrating Olos as an entire Cvm step in Simpl. This function combined with external device steps defines an overall implementation step. At the bottom right of Figure 7.1 on the following page, $^{\mathcal{I}}$ECU denotes the Simpl model that integrates the generated implementation code of Olos. On the abstract layer, we modelled the behaviour of a single ECU by integrating devices

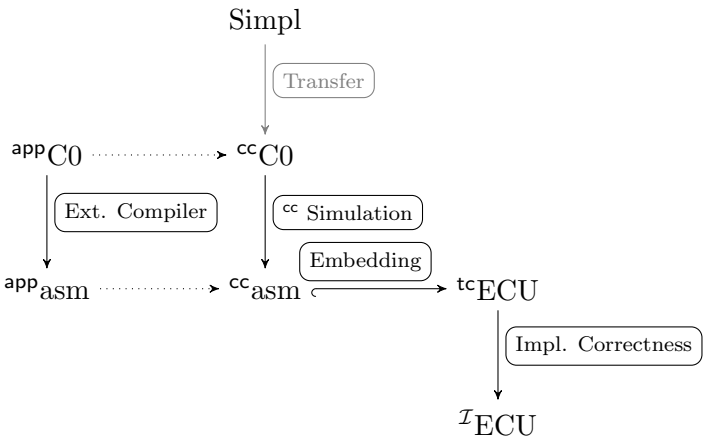

Figure 7.1: Formal Results

and applications to the specified Olos state. This specification is depicted by ^{tc}ECU. The implementation correctness of Olos is obtained by proving a simulation theorem Theorem 5.10 on page 141 between both layers.

Moreover, we defined an application abstraction influenced by previous work concerning the Vamos microkernel (Section 4.2). We instantiated it for C0 and Vamp assembly applications that are depicted by ^{app}C0 and ^{app}asm on the left side. Furthermore, we formulated a number of well-formedness constraints for application models specifying application-model validity. Then, we proved that these rules are fulfilled by both instances (Theorem 4.3 on page 94 and Theorem 4.4 on page 101). The validity of the application models is the premise for embedding them into our ECU model. The ECU specification uses the application abstraction that 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 system calls). For the verification of C0 applications, we extended the sequential compiler-correctness theorem of Dirk Leinenbach to applications and completed the formal proof of the extended theorem (Theorem 6.14 on page 166). Moreover, we introduced a cooperative concurrent execution model on the C0 and the Vamp assembly layer to reason sequentially about a complete computation phase. These models are denoted with ^{cc}C0 and ^{cc}asm. This approach allows to verify properties of applications in Isabelle/Simpl. Results obtained on this abstraction layer can be transferred down to the Vamp assembly layer when we combine the cooperative concurrent simulation theorem (Theorem 6.15 on page 167) with the transfer theorems of Norbert Schirmer [Sch06]. Finally, we have proved a theorem (Theorem 6.16 on page 168) that allows to infer properties about the entire ECU behavior.

The Olos implementation is completely integrated into the system stack. The real-time operating system runs on the verified Vamp processor [BJK+06] using the programming framework Cvm [GHLP05, IT08]. On the abstract level, the ECU model and the corresponding simulation proofs are seamlessly integrated into one coherent theory.

The Olos model is directly based on the Cvm layer and uses its definitions literally. Moreover, the ECU model employs the device framework. Our application model is fully embedded into the ECU model. Another archievement is that the ECU model is used to prove the functional correctness of the Olos implementation. This result ensures that the model can be safely used in the upper layers e. g., it can be related to the distributed ECU model of Steffen Knapp [Kna08]. Finally, our approach to verifiy communicating applications allows us to infer properties about the entire ECU.

We believe that our work responds to a long lasting grand challenge [Moo02] as well as to a general encouragement [Tic98] for more experimental work in computer science. Although the Olos kernel was developed with verification in mind and is quite small, we claim to have a complete pervasively verified real-time operating system.

7.2 Gained Insights from Pervasive Verification

The aim of pervasiveness has considerably influenced our verification approach and its result: Pervasive verification is inherently based on the integration of all verification results into one single, coherent theory. The tight integration of results from various authors with different backgrounds poses its own challenges [AHL+08]. We were able to rely on previous work, e. g., Vamp [BJK+06], the device framework [AH08, HIP05] Cvm [GHLP05, IT08], the verified C0 compiler [LPP05, Pet07, LP08], and Isabelle/Simpl [Sch06], which considerably increased the possible degree of reliance so that our verification result indeed holds for the overall system. Another advantage of the collaboration was the fact that some problems appeared earlier. Hence, we could profit from the experiences and solutions by adapting methods or approaches for our particular case. The formalizations of the application model and the extension of the sequential compiler-correctness theorem to the process semantics was strongly influenced by the results of Daum et al. [DDWS08] proposed for their microkernel Vamos. We took their general concept and adapted it for our applications running on our real-time operating system Olos. Then, we could completely prove the application model validity and embed all verified Olos system calls into the extended compiler-correctness theorem.

Due to the parallel proof development, our knowledge and experiences regarding the kernel library correctness could be reused in the corresponding correctness proof for Vamos [Dau10]. Furthermore, the proof of the extended compiler-correctness theorem has to deal with mixed-languages. We use the method of Gargano et al. [GHLP05] that has been refined by Starostin & Tsyban [ST08] to deal with inline assembly code in a C0 program. Even the refined method turned out to be inconvenient because the construction functions only support updates of elementary types. This fact became apparent in the simulation proof for the Olos system call *olosRecvMsg* where the application receives a complex message type.

Certainly, relying on various results has also its disadvantages: Within our work, we particularly perceived a high sensitivity to changes made by other verification engineers, on the one hand, and considerable friction losses because of different formalization styles and even duplicated definitions and proofs for similar problems, on the other hand.

Finally, another aspect has not to be underestimated. Whenever we encountered a problem for the first time, it took us considerable time to develop a suitable verification strategy. Then, we could cope with similar cases a lot easier. For the library correctness each of the three OLOS system calls belongs to a different degree of difficulty. Thus, we had to prove new aspects for every function of the library.

7.3 The Effort of Formal Verification

The effort of formal verification can be only roughly estimated. The implementation was subjected to several developments. These arose from improvements of the code or changed requirements. All changes in the OLOS source code always resulted in iterations on the theories depending on the implementation. The task of building a model stack extending over several abstraction levels – ideally from the gate-level implementation up to applications – proved to require much foresight and extreme prudence for the definition of the layer's interfaces because changes of the formalization usually spread over several layers and are thus very costly. Though this fact can certainly be expected, we considerably underestimated the necessary number of iterations, notably increasing the overall verification effort. Including all iterations, adaptions and several improvements, we approximate the effort for our implementation correctness proof with 2 years. Our OLOS implementation (without CVM) consists of roughly 300 lines of C0 code [1]. The functional correctness proof covers the entire OLOS implementation and comprises approximately 3000 lines of code. The largest portion of this code belongs to the Hoare triple proving the functional correctness of all OLOS functions in Simpl. Here, the most challenging proof was the correctness proof of the initialization function including three loops. It took about 3 person months and its proof size is about 490 lines of code.

The elaboration of the framework to pervasively verify applications also took about 16 months and the theories comprise about 11.000 lines of code. The work consuming most time were the simulation proofs between our abstract C0 transition and the execution of the system call implementation code. Each system call revealed new problems we had to solve. The easiest system call *olosExFinished* was done in 2 person months. The calls *olosSendMsg* and *olosRecvMsg* have many common properties concerning the structure of the calls, the compiled code and the message structure. We encapsulated all these lemmata into one theory. Without this preparatory work the most complex case (i. e., the successful receive case) took about 4 person months. The theory size of the latter case is more than three times as large as the case of *olosExFinished* (the theory sizes of all system call patterns range from 900 to 2900 lines of code). An overview of all theories and their sizes are given in the appendix. Furthermore, we extended several theories to prove important properties that we required for our verification work. We stored them directly together with the definitions of the underlying model and did not count them as part of our work.

Apart from that, the number of lines within a proof script or the elapsed time of a single proof does not necessarily correlate with its degree of difficulty. As an example,

[1] We present more detailed statistics in Table A.3 on page 199

structural proofs of the subcomponents of the message type have an unproportional number of proof script lines whereas other proofs involve much more creativity and finally appear rather small.

7.4 Future Work

We foresee several possible extensions of our work.

On the hardware layer Christian Müller verifies the message transmission between several ABC devices [EMST10]. More specifically, he completed the verification of the bus correctness and the correct message transmission between the ABC buffers and the bus. Linking our formal proofs we would obtain an impressive result.

Our correctness statement is yet limited to a single slot schedule because of restrictions in our tool chain: The schedule is currently specified via implementation constants in the code. These constants are formally fixed in the generated code; the current verification framework is incapable to instantiate a correctness proof for multiple schedules. We are confident, however, that this problem can be solved using recent improvements of Isabelle/HOL [HW09] and a comparatively small enhancement of Isabelle/Simpl.

Moreover, our verification approach implicitly assumes that our real-time operating system is capable to timely handle the incoming device interrupts. We have proved that our interrupt handler, the top-level function of Olos, terminates. In order to set up a correctly running system, however, we need an upper bound of its worst-case execution time. A possible approach for its computation is static worst-case execution-time analysis [FHW04, KP07a].

Finally, the correctness of the Cvm specification in Simpl has not been verified. For large parts of Cvm, however, there is a correctness proof up to the C0 small-step semantics and parts of this specification have been reused in the Simpl specification. Furthermore, a property transfer between Simpl and the small-step semantics has been done before [ASS08], demonstrating the general applicability of this approach. Finally, we specified the Olos system calls for the practical example Section 6.4 in Simpl but did not transfer the proven properties down to the small-step level yet.

Bibliography

> If I have been able to see further (than you and Descartes),
> it is because I have stood on the shoulders of giants.
>
> *Isaac Newton, quoted in a letter to Robert Hooke*

[AH08] Eyad Alkassar and Mark A. Hillebrand. Formal Functional Verification of Device Drivers. In *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 Schirmer, and Artem Starostin. The Verisoft Approach to Systems Verification. In *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, Special Issue on Operating System Verification*, 42(2-4):389–454, 2009.

[AHPP10] Eyad Alkassar, Mark A. Hillebrand, Wolfgang Paul, and Elena Petrova. Automated Verification of a Small Hypervisor. In Peter O'Hearn, Gary T. Leavens, and Sriram Rajamani, editors, *Verified Software: Theories, Tools, Experiments (VSTTE 2010)*, volume 6217 of *Lecture Notes in Computer Science*, pages 40–54, Edinburgh, UK, August 2010. Springer.

[Alk09] Eyad Alkassar. *OS Verification Extended - On the Formal Verification of Device Drivers and the Correctness of Client/Server Software*. PhD thesis, University of Saarland, 2009.

[Ame99] American National Standards Institute. *ANSI/ISO/IEC 9899-1999: Programming Languages — C*. American National Standards Institute, New York, USA, December 1999.

[AS97] Mark Anthony and Shawn Smith. Formal Verification of TCP and T/TCP, 1997.

[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 Rinard. Verifying a file system implementation. In *In Sixth International Conference on Formal Engineering Methods (ICFEM'04), volume 3308 of LNCS*, pages 8–12, 2004.

[BBBB09a] Christoph Baumann, Bernhard Beckert, Holger Blasum, and Thorsten Bormer. Better avionics software reliability by code verification – a glance at code verification methodology in the Verisoft XT project. In *In Embedded World 2009 Conference*, 2009.

[BBBB09b] Christoph Baumann, Bernhard Beckert, Holger Blasum, and Thorsten Bormer. Formal Verification of a Microkernel used in Dependable Software Systems. In *Proceedings of the 28th International Conference on Computer Safety, Reliability, and Security*, SAFECOMP '09, pages 187–200, Berlin, Heidelberg, 2009. Springer-Verlag.

[BBG+08] J. Botaschanjan, M. Broy, A. Gruler, A. Harhurin, S. Knapp, L. Kof, W. Paul, and M. Spichkova. On the Correctness of Upper Layers of Automotive Systems. *Formal Aspects of Computing*, 20(6):637–662, 2008.

[BCT04] William Bevier, Richard Cohen, and Jeff Turner. A Specification for the Synergy File System, 2004.

[Bev89] William R. Bevier. Kit and the short stack. *J. Autom. Reasoning*, 5(4):519–530, 1989.

[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.

[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.

[BLW] Sascha Böhme, K. Rustan M. Leino, and Burkhart Wolff. HOL/Boogie – An Interactive Prover for the Boogie Program-Verifier.

[BMSW] S. Böhme, M. Moskal, W. Schulte, and B. Wolff. HOL/Boogie: An interactive prover-backend for the Verifiying C Compiler. *Journal of Automated Reasoning*.

[Bog08] Sebastian Bogan. *Formal Specification of a Simple Operating System*. PhD thesis, Saarland University, Saarbrücken, 2008.

[Bor00] Richard Bornat. Proving pointer programs in Hoare Logic, 2000.

[Boy09] Andrew Boyton. A Verified Shared Capability Model. In Gerwin Klein, Ralf Huuck, and Bastian Schlich, editors, *Proceedings of the 4th Workshop on Systems Software Verification*, volume 254 of *Electronic Notes in Computer Science*, pages 25–44, Aachen, Germany, Oct 2009. Elsevier.

[Bur72] R. Burstall. Some techniques for proving correctness of programs which alter data structures. *Machine Intelligence 7*, pages 23–50, 1972.

[CAB+09] E. Cohen, A. Alkassar, V. Boyarinov, M. Dahlweid, U. Degenbaev, M. Hillebrand, B. Langenstein, D. Leinenbach, M. Moskal, S. Obua, W. Paul, H. Pentchev, E. Petrova, T. Santen, N. Schirmer, S. Schmaltz, W. Schulte, A. Shadrin, S. Tobies, A. Tsyban, and S. Tverdyshev. Invariants, Modularity, and Rights. In Amir Pnueli, Irina Virbitskaite, and Andrei Voronkov, editors, *Perspectives of Systems Informatics (PSI 2009)*, volume 5947 of *Lecture Notes in Computer Science*, pages 43–55. Springer, 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, *Theorem Proving in Higher Order Logics (TPHOLs 2009)*, volume 5674 of *Lecture Notes in Computer Science*, pages 23–42, Munich, Germany, 2009. Springer. Invited paper.

[CMST09] Ernie Cohen, Michał Moskal, Wolfram Schulte, and Stephan Tobies. A Precise Yet Efficient Memory Model for C. In *Workshop on Systems Software Verification (SSV 2009)*, volume 254 of *Electronic Notes in Theoretical Computer Science*, pages 85–103. Elsevier Science B.V., 2009.

[CMST10a] Ernie Cohen, Michał Moskal, Wolfram Schulte, and Stephan Tobies. Local Verification of Global Invariants in Concurrent Programs. In Byron Cook, Paul Jackson, and Tayssir Touili, editors, *Computer Aided Verification (CAV 2010)*, volume 6174 of *Lecture Notes in Computer Science*, pages 480–494, Edinburgh, UK, July 2010. Springer.

[CMST10b] Ernie Cohen, Michał Moskal, Wolfram Schulte, and Stephan Tobies. Local Verification of Global Invariants in Concurrent Programs. Technical Report MSR-TR-2010-9, Microsoft Research, January 2010.

[Dau10] Matthias Daum. *On the Formal Foundation of a Verification Approach for System-Level Concurrent Programs*. PhD thesis, Saarland University, Saarbrücken, 2010.

[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, *5th International Verification Workshop (VERIFY'08)*, volume 372 of *CEUR Workshop Proceedings*, pages 56–70. CEUR-WS.org, 2008.

[DDW09] Matthias Daum, Jan Dörrenbächer, and Burkhart Wolff. Proving Fairness and Implementation Correctness of a Microkernel Scheduler. *J. Autom. Reasoning, Special Issue on 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 *Verified Software: Theories, Tools, and Experiments*, volume 5295 of *LNCS*, pages 161–176. Springer, October 2008.

[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.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[DSS09] Matthias Daum, Norbert W. Schirmer, and Mareike Schmidt. Implementation Correctness of a Real-Time Operating System. In *7th IEEE International Conference on Software Engineering and Formal Methods (SEFM 2009), 23–27 November 2009, Hanoi, Vietnam*, pages 23–32. IEEE, 2009.

[DSS10] Matthias Daum, Norbert W. Schirmer, and Mareike Schmidt. From Operating-System Correctness to Pervasively Verified Applications. In Dominique Méry and Stephan Merz, editors, *Integrated Formal Methods, 8th International Conference*, Lecture Notes in Computer Science. Springer, 2010.

[Dör10] Jan Dörrenbächer. *Formal Specification and Verification of a Microkernel*. PhD thesis, Saarland University, Saarbrücken, 2010.

[EKE08] Dhammika Elkaduwe, Gerwin Klein, and Kevin Elphinstone. Verified Protection Model of the seL4 Microkernel. In Jim Woodcock and Natarajan Shankar, editors, *Second IFIP Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE 2008)*, volume 5295 of *LNCS*, pages 99–114, Toronto, Canada, October 2008. Springer.

[EMST10] Erik Endres, Christian Müller, Andrey Shadrin, and Sergey Tverdyshev. Towards the Formal Verification of a Distributed Real-Time Automotive System. In César Muñoz, editor, *Proceedings of the Second NASA Formal Methods Symposium (NFM 2010), NASA/CP-2010-216215*, pages 212–216, Langley Research Center, Hampton VA 23681-2199, USA, April 2010. NASA.

[End05] Endrawaty. Verification of the Fiasco IPC Implementation. Master's thesis, Technical University of Dresden, 2005.

[Eur03] European Commission (DG Enterprise and DG Information Society). eSafety Forum: Summary Report 2003. Technical report, eSafety, March 2003.

[FHW04] Christian Ferdinand, Reinhold Heckmann, and Reinhard Wilhelm. Analyzing the worst-case execution time by abstract interpretation of executable code. In *ASWSD*, pages 1–14, 2004.

[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.

[GHLP05] Mauro Gargano, Mark Hillebrand, Dirk Leinenbach, and Wolfgang Paul. On the Correctness of Operating System Kernels. In *TPHOLs 2005*, volume 3603 of *LNCS*, pages 1–16. Springer, 2005.

[Guy10] Jessica Guynn. Apple co-founder Steve Wozniak says his Toyota Prius accelerates on its own. *Los Angeles Times*, February 3, 2010.

[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.

[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.

[HHF+05] Hermann Härtig, Michael Hohmuth, Norman Feske, Christian Helmuth, Adam Lackorzynski, Frank Mehnert, and Michael Peter. The Nizza Secure-System Architecture. In *In IEEE CollaborateCom 2005*, 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.

[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.

[HTS02] Michael Hohmuth, Hendrik Tews, and Shane G. Stephens. Applying source-code verification to a microkernel: the VFiasco project. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, EW 10, pages 165–169, New York, NY, USA, 2002. ACM.

[HW09] Florian Haftmann and Makarius Wenzel. Local Theory Specifications in Isabelle/Isar. In *TYPES*, volume 5497 of *LNCS*, pages 153–168. Springer, 2009.

[IdR09] Thomas In der Rieden. *Verified Linking for Modular Kernel Verification*. PhD thesis, Saarland University, Saarbrücken, 2009.

[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.

[Kai07] Robert Kaiser. Combining Partitioning and Virtualisation for Safety-critical Systems. In *Embedded World Conference 2007, Nuremberg*, February 2007.

[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 *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, pages 207–220, Big Sky, MT, USA, Oct 2009. ACM.

[KG94] Hermann Kopetz and Günter Grünsteidl. TTP – A Protocol for Fault-Tolerant Real-Time Systems. *IEEE Computer*, 27(1):14–23, 1994.

[Kle09a] Gerwin Klein. Correct OS Kernel? Proof? Done! *USENIX ;login:*, 34(6):28–34, December 2009.

[Kle09b] Gerwin Klein. Operating System Verification — An Overview. *Sādhanā*, 34(1):27–69, February 2009.

[Kna08] Steffen Knapp. *The Correctness of a Distributed Real-Time System*. PhD thesis, Saarland University, July 2008.

[KP07a] Steffen Knapp and Wolfgang Paul. Realistic Worst Case Execution Time Analysis in the Context of Pervasive System Verification. In *Program Analysis and Compilation, Theory and Practice*, volume 4444 of *LNCS*, pages 53–81. Springer, 2007.

[KP07b] Steffen Knapp and Wolfgang J. Paul. Pervasive Verification of Distributed Realtime Systems. In T. Hoare M. Broy, J. Grünbauer, editor, *Software System Reliability and Security*, NATO Security Through Science Series. Sub-Series: Information and Communication Vol.9. IOS Press, 2007.

[Lei08] Dirk Carsten Leinenbach. *Compiler Verification in the Context of Pervasive System Verification*. PhD thesis, Saarland University, 2008.

[LP06] Markus Linnemann and Norbert Pohlmann. Schöne neue Welt?! - Die vertrauenswürdige Sicherheitsplattform Turaya. In *IT-Sicherheit Ausgabe*, volume 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.

[LS09] Dirk Leinenbach and Thomas Santen. Verifying the Microsoft Hyper-V Hypervisor with VCC. In *Formal Methods (FM 2009)*, volume 5850 of *Lecture Notes in Computer Science*, pages 806–809, Eindhoven, the Netherlands, 2009. Springer. Invited paper.

[MN05] Farhad Mehta and Tobias Nipkow. Proving Pointer Programs in Higher-Order Logic. *Information and Computation*, 199:200–227, 2005.

[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.

[MWTG00] W. Martin, P. White, F. S. Taylor, and A. Goldberg. Formal Construction of the Mathematically Analyzed Separation Kernel. In *Proceedings of the 15th IEEE international conference on Automated software engineering*, ASE '00, pages 133–, Washington, DC, USA, 2000. IEEE Computer Society.

[NF03] Peter G. Neumann and Richard J. Feiertag. PSOS Revisited. In *Proceedings of the 19th Annual Computer Security Applications Conference*, ACSAC '03, pages 208–, Washington, DC, USA, 2003. IEEE Computer Society.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[Pau94] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *LNCS*. Springer, 1994.

[Pau05] Wolfgang Paul. Towards a worldwide verification technology. In *In Proceedings of the Verified Software: Theories, Tools, Experiments Conference (VSTTE 2005)*, 2005.

[Pet07] Elena Petrova. *Verification of the C0 Compiler Implementation on the Source Code Level*. PhD thesis, Saarland University, 2007.

[PRS+01] Birgit Pfitzmann, James Riordan, Christian Stüble, Michael Waidner, and Arnd Weber. The PERSEUS System Architecture. Technical Report RZ 3335(#93381), IBM Research Division, 2001.

[Sam08] Thaima Samman. Verifying 50,000 lines of C code. In *Futures, Microsoft's European Innovation Magazine*, volume 21, 2008.

[Sch05] Norbert Schirmer. A Verification Environment for Sequential Imperative Programs in Isabelle/HOL. In *LPAR*, LNCS, pages 398–414. Springer, 2005.

[Sch06] Norbert Walter Schirmer. *Verification of Sequential Imperative Programs in Isabelle/HOL*. PhD thesis, TU Munich, 2006.

[Sch07] E. Schierboom. Verification of Fiasco's IPC implementation. Master's thesis, Radboud Universiteit Nijmegen, 2007.

[SDD+04] Jonathan Shapiro, Ph. D, Michael Scott Doerrie, Eric Northup, Swaroop Sridhar, and Mark 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.

[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.

[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.

[Tic98] Walter F. Tichy. Should Computer Scientists Experiment More? *IEEE Computer*, 31(5):32–40, 1998.

[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.

[Tsy09] Alexandra Tsyban. *Formal Verification of a Framework for Microkernel Programmers*. PhD thesis, Saarland University, Computer Science Department, November 2009.

[VB10] Ralph Vartabedian and Ken Bensinger. Doubt cast on Toyota's decision to blame sudden acceleration on gas pedal defect. *Los Angeles Times*, January 30, 2010.

[WKP80] B. Walker, R. Kemmerer, and G. Popek. Specification and verification of the UCLA Unix security kernel. *Commun. ACM*, 23(2):118–131, 1980.

[YTE04] Junfeng Yang, Paul Twohey, and Dawson Engler. Using Model Checking to Find Serious File System Errors. pages 273–288, 2004.

A Appendix

A.1 Source Code of the OLOS Implementation

We give a short overview where the OLOS implementation is located and the source code sizes. Afterwards, we present the content of the files containing the C0 source code.

Implementation: `implementation/software/*`

- OLOS Implementation: `olos/*`
  - `olos.c`: OLOS implementation (340 LOC)
  - `olos.h`: Static implementation information
    (e. g., system call names or register and port numbers)
  - `olos_config.h`: Configurable implementation information
    (e. g., initial arrays for scheduling tables and ABC configuration)
- OLOS System Call Library: `libolos/*`
  - `syscall_library.c`: system call implementation (50 LOC)
  - `syscall_library.h`: message type definition
  - directory `t/*`: small test programs

A.1.1 OLOS Constant Definitions

File `olos.h` contains all the OLOS specific constant definitions:

```
//General purpose register for System Calls
#define OLOS_GPR_ARG_01 11u
#define OLOS_GPR_ARG_02 12u
#define OLOS_GPR_ARG_RES 22u

//Length and initial values of the Configuration Buffer
#define CBLENGTH 8u
#define CB_CONF {SLOTCOUNT,L_MSG,OFF,T_SL,WAKEUP,IWAIT,SENDL1,
    SENDL2}

// Identifier of the ABC device
#define ABC_ID 13u

// Port addresses (in Words)
// Send buffer
#define SENDB_PORT 0u
// Receive buffer
#define RECVB_PORT 256u
// Configuration buffer
#define CONFB_PORT 512u
```

```
// Command register
#define CMD_PORT 767u

// Commands for the ABC (clear interrupt, set ready)
#define CMD_CLRINT 0u
#define CMD_SETRD 1u

/** \name Defines: OLOS return codes for errors **/
#define OLOS_RC_INVALID_MSGNUMBER -1
#define OLOS_RC_INVALID_POINTER   -2

/** \name Defines: OLOS system calls **/
#define OLOS_CALL_ExFinished 0u
#define OLOS_CALL_SendMsg    1u
#define OLOS_CALL_RecvMsg    2u

// Maximal Trap Number
#define MAX_CALL_ID 2u
```

A.1.2 Declarations

Global OLOS variables and data structures declared in `olos.c`:

```
/* Messages */
typedef int t_kmsg[MSGLENGTH];
typedef t_kmsg *p_kmsg;
p_msg MB[MSGCOUNT];
/* Processes */
unsigned int ca;
#define IDLE 0u
/* Round Scheduling */
unsigned int AST[SLOTCOUNT]; // Application Scheduling Table
unsigned int BT[SLOTCOUNT]; //  Message-Buffer Index Table
bool SPT[SLOTCOUNT]; // Sending Permission Table (Bus
    communication)
unsigned int csn; // current slotnumber
unsigned int PAGEC[PROCCOUNT + 1u]; // pagenumber per process
/* Interrupt type*/
/* Value of Sendflag indicates the type of timer interrupt:
   1) Sendflag = true   => wakeup interrupt
      Start send phase (only occurs to ECUs sends the message to
          the
                        bus in the next slot)
   2) Sendflag = false  => slotboundary is reached
      Start receive phase (occurs after a compute or a send phase
                           depending on the sending permission of
                              the
                           ECU)
 */
```

```
bool Sendflag;
// CVM functions for copy operations
DECLARE_CVM_PHYS_COPY(t_kmsg)
DECLARE_CVM_PHYS_IO_RANGE(t_kmsg)
```

A.1.3 The Interrupt Handler

C0 source code of function `handle_int` in `olos.c`:

```
// ************************************
// Internal-CODE (Bus communication)
// ************************************
int handle_int(){
    int dummy;
    unsigned int result_ui;

        if (Sendflag) { // send phase:
           // select MB with message that is sent to the bus next
              slot
           // write message to ABC send buffer
           result_ui = BT[(csn + 1u) % SLOTCOUNT];
           dummy = cvm_physIOOutRange_t_kmsg(MB[result_ui],ABC_ID,
              SEND_PORT);
           ca = 0u; // set current application to IDLE
           Sendflag = false; // lower send flag
        } else { // recv phase:
              // slot boundary: increase slot number
              csn = (csn + 1u) % SLOTCOUNT;
              // set sendflag to write permission of the next slot
              Sendflag = SPT[(csn + 1u)% SLOTCOUNT];
              // select MB to store message received from the bus
              // in the previous slot, read message from ABC
                 receive buffer
              result_ui = BT[(csn + SLOTCOUNT - 1u)% SLOTCOUNT];
              dummy = cvm_physIOInRange_t_kmsg(ABC_ID,RECV_PORT,MB
                 [result_ui]);
              ca = AST[csn]; //activate currently scheduled
                 application
        }
        // Clear ABC interrupt
        dummy = cvm_out_word(ABC_ID,COMR_PORT, COM_CLRINT);
return 0;
}
```

A.1.4 The Trap Handler

C0 source code of function `handle_trap` in `olos.c`:

```
// ************************************
```

```
// Dispatch and execution of kernel calls
// ************************************
int handle_trap (unsigned int n) {
    int dummy, result;
    unsigned int pmsg, mn;
    ;
  if (n == OLOS_CALL_ExFinished) {ca = IDLE;}
    else {
           pmsg = cvm_get_vm_gpr(ca,OLOS_GPR_ARG_01);
           mn = cvm_get_vm_gpr(ca,OLOS_GPR_ARG_02);
           if ((pmsg&3u) != 0u
               || pmsg >= PAGEC[ca] * PAGE_SIZE - 4u * MSGLENGTH)
              {result = OLOS_RC_INVALID_POINTER;}
    else if (mn >= MSGCOUNT) {result = OLOS_RC_INVALID_MSGNUMBER;}
           else {result = 0;
                 if (n == OLOS_CALL_SendMsg){
                       result = cvm_v2pcopy_t_kmsg(ca, pmsg,MB[mn])
                          ;
                       }
                 else {result = cvm_p2vcopy_t_kmsg(MB[mn], ca,
                    pmsg);
                                  }
                }
           dummy = cvm_set_vm_gpr(ca,OLOS_GPR_ARG_RES,UNSIGNED(
              result));
           }
    return 0;
}
```

A.1.5 The Initialization Function

C0 source code of the initialization function `olos_init` in `olos.c`:

```
// ************************************
// Initialization of OLOS
// ************************************

int olos_init() {

    unsigned int CONFB [CBLENGTH];

    int dummy;
    unsigned int n, next;

    // Initialization of schedules and PAGEC
    ASSIGN_ARRAY_VAL(AST,unsigned[],AST_CONF);
    ASSIGN_ARRAY_VAL(BT,unsigned[],BT_CONF);
    ASSIGN_ARRAY_VAL(SPT,bool[],SPT_CONF);
```

```
    ASSIGN_ARRAY_VAL(PAGEC,unsigned[],PAGEC_CONF);

    // Initialization of OLOS internal data
    csn = SLOTCOUNT - 1u;
    ca = IDLE;
    Sendflag = false;

    n = 0u;
    while (n < MSGCOUNT) {
        MB[n] = new(t_kmsg);
        n = n + 1u;
    }
// Initialize processes
// PID 0 is reserved for kernel computation and not used for
   applications
// PID 1 to PID PROCCOUNT are used for the applications
// In the boot image, PID 0 is at page offset 0.
// The application sizes are configured with the PAGEC_CONF
   configuration item.
    n = 1u;
    next = 0u;
    while (n <= PROCCOUNT) {
        dummy = cvm_reset(n); // Initialize registers
        dummy = cvm_alloc(n, PAGEC[n]); // Allocate memory
        dummy = cvm_load_os(PAGEC[n], n, next); // Load
            application
        next = next + PAGEC[n];
        n = n + 1u;
    }
    // Configure ABC device and send set-ready signal
    ASSIGN_ARRAY_VAL(CONFB,unsigned[],CB_CONF);
    n = 0u;
    while (n < CBLENGTH) {
          dummy = cvm_out_word(ABC_ID,CONFR_PORTu + n, CONFB[n]);
          n = n + 1u;
    }
    dummy = cvm_out_word(ABC_ID,COMR_PORT, COM_SETRD);
    return 0;
```

A.1.6 OLOS Mainfunction

C0 source code of the dispatcher function `kdispatch` in `olos.c`:

```
// ************************************
// Kernel Dispatcher
// ************************************
unsigned int kdispatch(unsigned int eca, unsigned int edata) {
    int dummy;
    unsigned int result_ui;
```

```
    // If there is reset, we do not care about the remaining bits
       of eca
    // Otherwise, we must handle no other interrupt than trap (5)
       or ABC (13)
    if ((eca & 1u) != 0u) { // Reset (0)
       dummy = olos_init();
    }
    else if ((eca&8192u)!= 0u) { // ABC interrupt
         dummy = handle_int ();
         }
    else if ((eca & 32u) != 0u) { // trap / system call
        if (edata <= MAX_CALL_ID) {
            dummy = handle_trap (edata);
        }
    }
    return ca;
}
```

A.1.7 Typed CVM Primitives

In file `cvmolos.c`, we define the following two macros to specify generic patterns for individually typed copy functions. The additional parameter `t_kmsg` defines that the implementation of the copy functions uses the kernel message type.

```
DEFINE_CVM_PHYS_COPY(t_kmsg)
DEFINE_CVM_PHYS_IO_RANGE(t_kmsg)
```

The preprocessor expands the first one to the typed functions `cvm_p2vcopy_t_kmsg` and `cvm_v2pcopy_t_kmsg` that copy data between the kernel and a user process. The second macro is expanded to the typed functions `cvm_physIOInRange_t_kmsg` and `cvm_physIOOutRange_t_kmsg` which are used to exchange messages between the kernel and a specified device.

The copy functions have a similar structure. Most notably, all these functions take a message pointer as parameter. Then, a portion of inline assembly is used to compute the start address and the size of the message (the start address is either the address where the message is stored in the memory or where it will be copied). The first two instructions load the address of the pointer and store this value into a local variable `addr`. Furthermore, an instruction computes the size of the referenced data with function `asm_sizeof`. This value is stored in the local variable `size`. Afterwards, the underlying copy functions are called with a boolean variable to distinguish between read and write accesses of the kernel (i. e., `true` denotes that the kernel sends and `false` it receives data from a device or user process, respectively). The source code of the expanded typed copy functions is given below:

```
int cvm_p2vcopy_t_kmsg(t_kmsg *ptr, unsigned int pid,
                       unsigned int va)
{
 int result; unsigned int addr; unsigned int size;
  asm { lw(r11,r30,asm_offset(ptr));
        sw(r11,r30,asm_offset(addr));
        addi(r11,r0,asm_sizeof(*ptr));
        sw(r11,r30,asm_offset(size)); };
  result = cvm_phys_copy(true, pid, va, addr, size);
  return 0;
 }

int cvm_v2pcopy_t_kmsg(unsigned int pid, unsigned int va,
                       t_kmsg *ptr)
{
 int result; unsigned int addr; unsigned int size;
  asm { lw(r11,r30,asm_offset(ptr));
        sw(r11,r30,asm_offset(addr));
        addi(r11,r0,asm_sizeof(*ptr));
        sw(r11,r30,asm_offset(size)); };
  result = cvm_phys_copy(false, pid, va, addr, size);
  return 0;
 }

int cvm_physIOInRange_t_kmsg(unsigned int device_id,
                             unsigned int port, t_kmsg *ptr)
{
 int result; unsigned int addr; unsigned int size;
  asm { lw(r11,r30,asm_offset(ptr));
        sw(r11,r30,asm_offset(addr));
        addi(r11,r0,asm_sizeof(*ptr));
        sw(r11,r30,asm_offset(size)); };
  result = cvm_phys_io_range(false, device_id, port, addr, size);
 return 0;
 }

int cvm_physIOOutRange_t_kmsg(t_kmsg *ptr, unsigned int device_id,
                              unsigned int port)
{
  int result; unsigned int addr; unsigned int size;
  asm { lw(r11,r30,asm_offset(ptr));
        sw(r11,r30,asm_offset(addr));
        addi(r11,r0,asm_sizeof(*ptr));
        sw(r11,r30,asm_offset(size)); };
  result = cvm_phys_io_range(true, device_id, port, addr, size);
  return 0;
}
```

A.2 Source Code of the OLOS System Call Library

A.2.1 Message Structure

The message type is defined in `syscall_library.h`:

```
  typedef int TYPE_CMV;
  typedef int TYPE_CCMV;
  typedef int TYPE_CSMV;
  typedef int TYPE_Signal;
  typedef int TYPE_Contr_Signal;

  struct TYPE_Coordinate{
    int xCoord;
    int yCoord;
  };

  struct TYPE_Message_Struct{
    unsigned int Field;
    TYPE_CMV crash19;
    TYPE_CSMV connection_status;
    TYPE_Signal c2eCall;
    struct TYPE_Coordinate coord;
    TYPE_Signal started23;
    struct TYPE_Coordinate outCoord;
    TYPE_CCMV connection_control;
    TYPE_Signal finished26;
    TYPE_CCMV connection_control27;
    TYPE_Signal Taskmodel_connection_failed_channel;
    struct TYPE_Coordinate coord29;
    TYPE_Signal c2MP;
    TYPE_Signal started31;
    TYPE_Signal finished32;
    TYPE_Signal c2GPS;
    int x;
    int y;
    TYPE_Signal finished36;
    TYPE_Signal started37;
    TYPE_Signal end_eCall2c;
    TYPE_Signal start_eCall2c;
    TYPE_Signal start_mP2c;
    TYPE_Signal start_GPS2c;
    TYPE_Signal end_GPS2c;
    TYPE_Signal end_mP2c;
    int dummy[11];
  };

typedef struct TYPE_Message_Struct t_msg;

typedef t_msg *p_msg;
```

A.2.2 System Call Functions

`syscall_library.c` contains the definitions of all system call functions:

```
int olosExFinished()
{
  assembler (trap(0););
  return 0;
}

int olosSendMsg(p_msg pmsg, unsigned int mn)
{
  int result;

    assembler(
            loadlocal(r11, pmsg);
            loadlocal(r12, mn);
            trap(1);
            storelocal(result, r22);
            );

  return result;
}

int olosRecvMsg(p_msg pmsg, unsigned int mn)
{
  int result;

    assembler(
            loadlocal(r11, pmsg);
            loadlocal(r12, mn);
            trap(2);
            storelocal(result, r22);
            );

  return result;
}
```

A.2.3 OLOS Program State

The tool `c0_check`, developed in the Verisoft project, generates from the OLOS source code a Simpl state representation. Table A.1 on the following page depicts the C0 variables and their generated counterparts side by side.

C0 variables		Simpl variables	
int	t_msg[MSGLENGTH]	´heap_msg	:: ref ⇒ int list
t_msg	*p_msg		
p_msg	MB[MSGCOUNT]	´MB	:: ref list
unsigned int	ca	´ca	:: nat
unsigned int	ST[SLOTCOUNT]	´AST	:: nat list
unsigned int	BT[SLOTCOUNT]	´BT	:: nat list
bool	SPT[SLOTCOUNT]	´SPT	:: bool list
unsigned int	csn	´csn	:: nat
unsigned int	PAGEC[SLOTCOUNT]	´PAGEC	:: nat list
bool	sendflag	´sendflag	:: bool

Table A.1: OLOS C0 variables and their Simpl representation

A.3 Theory Structure and Statistics

The following paragraph briefly shows the theory structure and its descriptions. Table A.3 on the next page depicts the number of code lines. The theorems of this thesis and the corresponding theorem names in the theories are presented in Table A.3 on page 201.

Specifications: `verification/spec/*`

- ABC: `Devices/abc.thy`
  specification of the ABC state and transition, appertaining lemmata
- Message Type: `AutoLI/syscall_library_ss.thy`
  generated C0 small-step code from the implementation (230 LOC)
- Applications: `AutoLI/*`
  - `UP.thy`: abstract specification of applications and their behaviour
  - `ASM_DELTA.thy`: assembly applications, validity proof (180 LOC)
  - `C0_DELTA.thy`: C0 applications, validity proof (240 LOC)
- ECU: `OLOSDevices/*`
  - `ECUModel.thy`: ECU specification
  - `ECUSimulation.thy`: simulation proof of the global compute phase

Implementation Correctness: `llverification/olos/*`

- `llverification/cvmHoare/cvmHoare.thy`:
  definition of extended state, Simpl specification of used untyped CVM primitives
- `olos_conf.thy`:
  definitions of OLOS specific configurable values, appertaining lemmata
- `cvmOlos.thy`:
  Simpl definitions of OLOS specific CVM primitives, appertaining lemmata (LOC: 255)

Theory name	LOC
ABC Device	Σ: 842
abc	842
Applications	Σ: 1857
UP	128
ASM_DELTA	376
C0_DELTA	1353
ECU Model	Σ: 1189
ECUModel	473
ECUSimulation	716
Impl. Correctness	Σ: 3303
cvmHoare	191
olos_conf	71
cvmOlos	255
olos_impl	188
olos_hoarespec	1163
interrupt_lemmata	315
cvmstep_simpl	850
olos_simulation	270
Ext. Compiler Correct	Σ: 10,917
syscall_sim_lemmata	484
send_recv_lemmata	2279
syscall_sim_ExFinished	899
syscall_sim_update_result	1643
syscall_sim_RecvMsg	2864
syscall_correctness	1921
compute_phase_sim	827

Table A.2: Verification Effort

- `olos_impl.thy`:
  generated Simpl code from C0 program of the Olos implementation
- `olos_hoarespec.thy`:
  Hoare triple proving the functional correctness of Olos
  (size of init code: 490 LOC)
- `interrupt_lemmata.thy`:
  definition of the masked cause interrupt vector for Olos,
  lemmata relating the output function of applications with interrupts
- `cvmstep_simpl.thy`:
  Simpl definition of an entire Cvm step, induction proofs
  (size of induction step proof: 508 LOC)
- `olos_simulation.thy`:
  simulation proof of overall implementation step

Library Correctness: `verification/proof/libolos/*`

- `syscall_sim_lemmata.thy`: useful lemmata for all system call proofs
- `send_recv_lemmata.thy`: lemmata for the message exchanging system calls
- `syscall_sim_ExFinished.thy`: Simulation of system call *olosExFinished*
- `syscall_sim_RecvMsg.thy`: Simulation of successful system call *olosRecvMsg*
- `syscall_sim_update_result.thy`: Simulation of calls where only the result variable is set
- `syscall_correctness.thy`: extended compiler correctness theorem, appertaining lemmata
  (size of induction step proof: 900 LOC)
- `compute_phase_sim.thy`: Simulation of the Olos compute phase

Finally, we give a small example to demonstrate that proof size, execution time and degree of difficulty do not correlate: A proof that shows the correctness of the relative base address of all the elementary message values has the size of 87 proof lines and takes about 1 minute to execute. In contrast, the computer executes 147 lines of code proving properties after a SCall execution of *olosSendMsg* and *olosRecvMsg* in about 15 seconds. Moreover, the former example was less demanding and very technically whereas the latter required more creativity and reflection.

Theorem Name	Reference	Name in Theory
Application Model Validity		
Assembly	Theorem 4.3	instance ASMcore_t_ext_type
C0	Theorem 4.4	instance processT
Global Compute Phase Simulation	Theorem 4.8	OlosDevSimulation
Functional Correctness of		
fun_olos_init	Theorem 5.1	fun_olos_init_spec
fun_handle_int	Theorem 5.2	fun_handle_timer_spec
fun_handle_trap	Theorem 5.3	fun_handle_trap_spec
fun_kdispatch	Theorem 5.4	fun_dispatcher_kernel_spec
Induction Start	Theorem 5.7	ind_start_weak
Induction Step	Theorem 5.8	ind_step
Invariant Preservation	Theorem 5.9	invariant_preserved_under_ δabc_ext
Simulation	Theorem 5.10	simulation
System Call Simulation of		
olosRecvMsg	Theorem 6.10	olosRecvMsg_sim_mono
olosExFinished	Theorem 6.11	olosExFinished_sim_mono
Remaining cases	Theorem 6.12	update_result_sim_mono
Ext. Compiler-Correct. (Ind. Step)	Theorem 6.13	c0proc_correct_step_oi
Application Simulation	Theorem 6.14	c0proc_correctness_oi
Cooperative Concurrent Simulation	Theorem 6.15	proc_consis_over_pi
Application Embedding	Theorem 6.16	cc_ecu_sim

Table A.3: Theorems and their corresponding names in the theories