From Operating-System Correctness to Pervasively Verified Applications*

Matthias Daum[1], Norbert W. Schirmer[2], and Mareike Schmidt[1]

[1] Computer Science Dept., Saarland University
66123 Saarbrücken, Germany
{md11,mareike}@wjpserver.cs.uni-saarland.de
[2] German Research Center for Artificial Intelligence (DFKI)
66041 Saarbrücken, Germany
Norbert.Schirmer@dfki.de

Abstract. Though program verification is known and has been used for decades, the verification of a complete computer system still remains a grand challenge. Part of this challenge is the interaction of application programs with the operating system, which is usually entrusted with retrieving input data from and transferring output data to peripheral devices. In this scenario, the correct operation of the applications inherently relies on operating-system correctness. Based on the formal correctness of our real-time operating system Olos, this paper describes an approach to pervasively verify applications running on top of the operating system.

1 Introduction

Various electronic devices are embedded in the modern car, and some are even in charge of safety-critical tasks like accelerator control. In the past years, a failing accelerator control has caused several fatal accidents [1]. Though the manufacturer has attributed these failures to a blocked gas pedal, a software problem has recently been suspected for the sudden, unintended acceleration of a car from the same manufacturer while driven by cruise control [2]. The mere rumor of such a software flaw is economically troublesome, not to mention the tragedy of possibly resulting fatal accidents.

There are different approaches to increase the reliability of software. A rigorous way to prevent flaws is the exclusion of systematic errors by verification. If the proofs are checked by a computer, we speak of *formal verification*. Certainly, this method should not be limited to a single system layer but span as many layers as possible. *Pervasive verification* means that the system layers are coupled by formal soundness and simulation theorems, such that any verification result, obtained on a suitable layer, can ultimately be transferred down to a correctness theorem on the lowest level. While program verification has been known

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

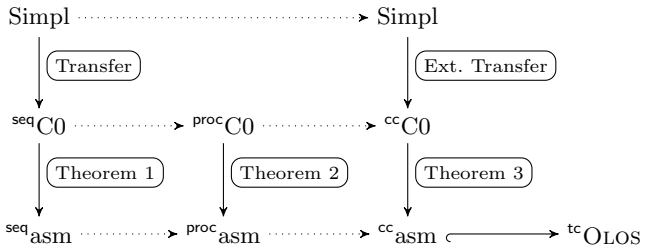

Fig. 1. Extending the Language Stack towards Concurrency

and used over decades, the pervasive formal verification of complete computer systems still remains a grand challenge [3].

Among others [4], the Verisoft project takes on this challenge. The goal of its automotive subproject is a pervasively verified distributed real-time system, consisting of hardware, a real-time operating system, and application programs. We have implemented the operating system OLOS on a verified processor [5] using a generic programming framework for operating systems [6]. Moreover, we have proven the correctness of OLOS [7] in the proof assistant Isabelle/HOL [8].

In this paper, we report on the formally proven foundation of a verification approach for applications running on top of OLOS. We present the necessary theorems to transfer verification results down to the operating-system level and thus, establish a formal link between the proofs on the application layer and those on the operating-system layer. More specifically, our verification approach is an extension of an existing language stack [9] based upon a verified C compiler [10] and a generic verification framework for sequential imperative programs [11].

Our overall proof architecture is depicted in Fig. 1: The original language stack (see Sect. 2.3) is shown on the left column. On the lower end, we have the sequential assembly language ($^{\mathsf{seq}}$asm), above is the C variant *C0*. Leinenbach & Petrova's [10] correct compiler translates a sequential C0 program ($^{\mathsf{seq}}$C0) to assembly. Schirmer [11] has developed the generic Simpl language together with Hoare logics and a transfer theorem stating that properties proven for Simpl actually hold in C0. The sequential semantics, however, have no means for communication. Thus, we extend $^{\mathsf{seq}}$C0 and $^{\mathsf{seq}}$asm in Sect. 3 to application processes (marked by $^{\mathsf{proc}}$ in the 2nd column), where the communication with OLOS is modeled by explicit inputs and outputs. We do not extend Simpl because it is solely used for reasoning in Hoare logic, which does not support inputs and outputs. Instead, we further extend the language stack in Sect. 4 to cooperative concurrent applications (marked by $^{\mathsf{cc}}$ in the 3rd column). By *cooperative concurrency*, we refer to the sequential execution of an application with calls to the operating system until a final call of a synchronization primitive. Finally, we embed the lowest layer of this stack into a true concurrent model ($^{\mathsf{tc}}$OLOS) of our operating system with an interleaved execution of applications (last column, Sect. 5).

In Sect. 6, we provide an application example demonstrating how our approach may be used in practice. We conclude in Sect. 7.

Notation. The formalizations presented in this article are mechanized and checked within the interactive theorem prover Isabelle/HOL [8]. This paper is written using Isabelle's document generation facilities, which guarantees that the presented theorems correspond to formally proven ones.[3] We distinguish formal entities typographically from other text. We use a sans-serif font for types and constants (including functions and predicates), e. g., map, 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., ExFinish. Type variables have a leading tick, e. g., $'a$. Keywords are typeset in bold font, e. g., **let**.

The logical and mathematical notation mostly follows standard conventions; we only present the more unconventional parts here. We prefer curried function application, e. g., $f\ a\ b$ instead of $f\ (a, b)$. We write f^n for the n-fold composition of function f.

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. Isabelle allows polymorphic types, e. g., $'a$ list is the list type with type variable $'a$. In HOL all functions are total, e. g., nat $\Rightarrow$ nat is a total function on natural numbers. There is, however, a type $'a$ option to formalize partial functions. It is a data type with two constructors, one to inject values of the base type, e. g., $\lfloor x \rfloor$, and the additional element $\bot$. 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 still a valid yet unspecified value. Partial functions can be represented by the type $'a \Rightarrow 'b$ option.

2 Background

2.1 On A Simple Real-Time Operating System

The continually increasing number and variety of electronic components in cars result in an even faster growing demand for communication channels. Over time, adding more and more wires has led to space, complexity and maintenance problems. Alternatively, several components can share the same wire and use a communication protocol on this bus. For that purpose, Kopetz and Grünsteidl [12] have developed the *time-triggered protocol*, which schedules fixed transmission times for each component on the bus. Variations of this protocol are nowadays widely accepted in industry.

We adopt this idea and assume a distributed system comprising a number of components that are connected via a communication bus. The components are called *electronic control units* (ECUs). Each ECU consists 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, decoupling the processor from the communication bus.

On each processor there runs an instance of the operating system Olos, providing a virtual processor abstraction to the applications that share the same

[3] For the theory files, see `http://www.verisoft.de/VerisoftRepository.html`

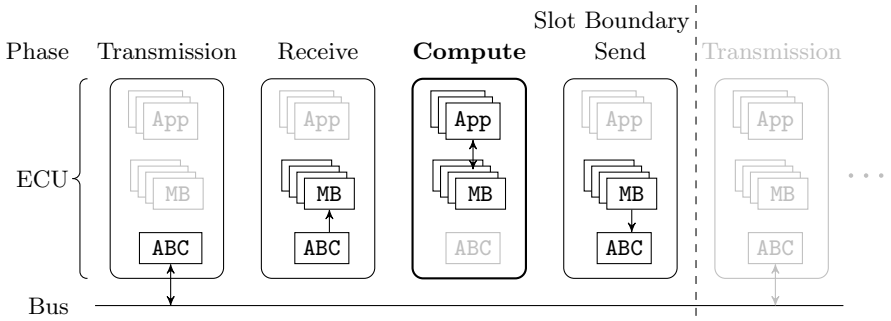

Fig. 2. The transition phases in each slot of our time-triggered computer system

physical processor. OLOS features its own message buffers (MB) for the communication between applications (on the same as well as on different physical processors). 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*.

Each slot is divided into four transition phases. Fig. 2 illustrates the data flow between the ECU components in the different phases:

Transmission In this phase, the ABC device of one ECU transmits a message to the communication bus. According to a predefined schedule, exactly one ECU has the send permission in each slot. All ECUs listen on the bus to receive the transmitted message.

Receive The operating system reads the receive buffer from the ABC device into one of its own message buffers.

Compute A statically fixed table specifies, which application is executed during this phase of the current slot. The application may compute locally or exchange messages with OLOS. When the application has finished its computation for the current slot, it informs the operating system by a special system call EXFINISH.

Send If the ECU is permitted to send, the operating system writes the corresponding message into the ABC's send buffer.

2.2 Formally Specifying OLOS – the True Concurrent ECU Model

Correctness is usually defined as the compliance with a specification. In our case, this specification is an automaton. Note that OLOS relies on a specific protocol with the ABC. Hence, the abstract automaton describes the behavior of the whole ECU consisting of the processor with its running operating system and applications together with the ABC device. A state s of this ECU automaton comprises the application states $\mathsf{s.AM}$, the message buffers $\mathsf{s.MB}$, an ABC state ($\mathsf{s.abc\_dev}$), and an *idle flag* ($\mathsf{s.idle\_flag}$). The latter determines whether the application scheduled in the current slot has finished its computation.

The transitions of the abstract ECU automaton concisely specify our informal description of the ECU behavior as depicted in Fig. 2. In Isabelle/HOL, we have formalized the transitions by the function $\delta_{\mathsf{ECU}}\ t\ i$. The static schedule t determines for each slot, which message buffer should be sent, and which application should be scheduled. The input i distinguishes external device steps ($i = \lfloor e \rfloor$) from processor computation ($i = \bot$).

In this paper, we are primarily interested in the compute phase. The beginning of the compute phase is marked by the ECU turning into the *computing* state. The name is derived from the fact that all transitions starting in this state involve a computation of the application scheduled in this slot. In this state, the idle flag as well as the ABC's interrupt flag are unset. Several transitions are possible from this state:

- An external device input might raise the interrupt line of the ABC device. In this case, the currently scheduled application has exceeded its execution time. OLOS reacts exactly as if it had been waiting for the interrupt.
- If the application issues an EXFINISH call, the operating system acknowledges the call and waits for an interrupt. Formally, the ECU transition raises the idle flag. It thereby turns into an idle state waiting for an input $\lfloor e \rfloor$.
- Otherwise, the ECU simply remains in the computing state.

2.3 On a Correct Compiler – the Sequential Language Stack

ANSI C [13] has a complex and highly underspecified semantics. Low-level functionality like the communication with the operating system, however, inherently *relies* on properties of a particular compiler and a specific target hardware, e. g., register bindings or the internal representation of data types. They can therefore not be verified based only on the vague ANSI C semantics. Hence, Leinenbach and Petrova [10] specified the C-like imperative language *C0*, which has sufficient features to implement low-level software but is interpreted by a more specific semantics. Due to lack of space, we omit the details of the language and only glance at its formal semantics.

The C0 Small-Step Semantics. A *C0 program* is statically defined by a type-name table *tt*, a function table *ft*, and a symbol table *gst* of global variables.

In contrast to this static program definition, the program state evolves during the execution of a C0 program. A state s_{C0} comprises: (a) the statement $s_{\mathsf{C0}}.\mathsf{prog}$ of the program that remains to be executed, and (b) the current state $s_{\mathsf{C0}}.\mathsf{mem}$ of the program variables and the heap objects. The transition relation δ_{C0} of the C0 semantics is deterministic, i. e., a partial function.

The Target Language. Leinenbach & Petrova's verified C0 compiler translates C0 programs into the assembly language developed for the VAMP architecture [5]. VAMP assembly abstracts from the paging mechanism of the processor and employs a linear memory model. An assembly state s_{asm} is a record compris-

ing two program counters[4] ($s_{\mathsf{asm}}.\mathsf{pcp}$ and $s_{\mathsf{asm}}.\mathsf{dpc}$), general-purpose as well as special-purpose register files ($s_{\mathsf{asm}}.\mathsf{gprs}$ and $s_{\mathsf{asm}}.\mathsf{sprs}$), and memory ($s_{\mathsf{asm}}.\mathsf{mm}$).

Assembly transitions are modeled by function δ_{asm}. Again, we omit the details of the semantics because of space restrictions. Note that the effects of hardware exceptions like accessing unavailable memory cannot be fully determined from an assembly-machine state. In this case, δ_{asm} gets stuck. With sufficient memory, however, there are no exceptions generated when a well-formed C0 program is compiled and executed.

On a Correct Compiler. Compiler correctness is formulated as a simulation theorem. The simulation relation consistent holds for corresponding C0 and assembly states. In essence, the compiler-simulation theorem states that every step i of the source program executed on the C0 semantics simulates a certain number s_i of steps of the VAMP assembly machine executing the compiled code.

The memory requirements can directly be checked on the C0 semantics. We assume that memory is available from 0 to an address $x \leq 2^{32}$. The predicate $\mathsf{sufficient\_memory}\ x\ tt\ ft\ s_{\mathsf{C0}}$ holds iff x is large enough such that the C0 state s_{C0} of the program (tt, ft) can be encoded in assembly. Furthermore, we denote the successful (i. e., fault-free) execution from an assembly state s_{asm} in t steps to state s'_{asm} by $(crange, arange) \vdash_{\mathsf{asm}} s_{\mathsf{asm}} \rightarrow^t s'_{\mathsf{asm}}$, where instructions are only read from range *crange* and only memory addresses in range *arange* are accessed. We can compute the ranges for a given C0 program by the functions code_range and address_range, respectively.

Theorem 1 (Stepwise Compiler Simulation). *We assume:*

- *The C0 state s_{C0} is well-formed, i. e.,* $\mathsf{is\_valid}_{\mathsf{C0}}\ tt\ ft\ s_{\mathsf{C0}}$.
- *The program counters of the well-formed assembly state s_{asm} do not start in a delay slot,*[5] *i. e.,* $\mathsf{is\_valid}_{\mathsf{asm}}\ s_{\mathsf{asm}}$ *and* $s_{\mathsf{asm}}.\mathsf{pcp} = s_{\mathsf{asm}}.\mathsf{dpc} + 4$.
- *The simulation relation holds for s_{C0} and s_{asm} under an allocation function alloc, i. e.,* $\mathsf{consistent}\ tt\ ft\ s_{\mathsf{C0}}\ alloc\ s_{\mathsf{asm}}$.
- *The C0 transition from s_{C0} is defined to s'_{C0}, i. e.,* $\delta_{\mathsf{C0}}\ tt\ ft\ s_{\mathsf{C0}} = \lfloor s'_{\mathsf{C0}} \rfloor$.
- *there is sufficient memory before and after the transition, i. e.,* $x \leq 2^{32}$, $\mathsf{sufficient\_memory}\ x\ tt\ ft\ s_{\mathsf{C0}}$ *and* $\mathsf{sufficient\_memory}\ x\ tt\ ft\ s'_{\mathsf{C0}}$.

Under these assumptions, there exists a step number n, an allocation function $alloc'$, and an assembly state s'_{asm} such that (a) the assembly machine successfully advances in n steps from s_{asm} to s'_{asm}, (b) the final C0 state s'_{C0} simulates s'_{asm} under the allocation function $alloc'$, and (c) no special-purpose registers have been changed. Formally:

$$\exists n\ alloc'\ s'_{\mathsf{asm}}.$$
$$(\mathsf{code\_range}\ tt\ (\mathsf{gm\_st}\ s_{\mathsf{C0}}.\mathsf{mem})\ ft, \mathsf{address\_range}\ x) \vdash_{\mathsf{asm}} s_{\mathsf{asm}} \rightarrow^n s'_{\mathsf{asm}} \wedge$$
$$\mathsf{consistent}\ tt\ ft\ s'_{\mathsf{C0}}\ alloc'\ s'_{\mathsf{asm}} \wedge s'_{\mathsf{asm}}.\mathsf{sprs} = s_{\mathsf{asm}}.\mathsf{sprs}$$

[4] We need two program counters because branches are delayed by one instruction.

[5] When a C0 statement has been completely executed, the assembly machine should certainly not be about to execute a previously seen branch.

Verifying C Programs – the Isabelle/Simpl Framework. The verification environment Isabelle/Simpl [11] is implemented as a conservative extension of the higher-order logic (HOL) instance of the theorem-proving environment Isabelle [8]. Though the verification environment was motivated by C0, it is by no means restricted to C0. In fact, it is a self-contained theory development for a quite generic model of a sequential imperative programming language called *Simpl*. Part of this extensive framework are big- and small-step semantics as well as Hoare logics for both, partial and total correctness. In order to facilitate the usage of the Hoare logics within Isabelle/HOL, the application of the rules is automated as a verification-condition generator. Furthermore, proofs have been developed that the logics are sound and complete with respect to the operational semantics. Soundness is crucial for pervasive verification in order to formally link the results from the Hoare logics to the operational Simpl semantics. Correctness theorems about the embedding of C0 into Simpl then allow us to map these results to the small-step semantics of C0 [11,14]. Completeness can be viewed as a sophisticated sanity check for the Hoare logics, ensuring that verification cannot get stuck because of missing Hoare rules.

3 Application Processes

As their most basic feature, operating systems provide a processor abstraction to applications with (a) an 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*. As even most high-level programs are eventually compiled to run as a process, we consider any program semantics with primitives for the communication with an operating system as a process. In this section, we formally specify processes as automata with outputs and inputs, present two particular process semantics for C0 and VAMP assembly, and finally extend compiler correctness to processes.

3.1 Process Semantics

In general, we define:

Definition 1 (Process Semantics). *A process semantics is an automaton* $\mathsf{A}_{\mathsf{proc}}$ *specified by a tuple* $(\mathcal{S}_{\mathsf{proc}}, \mathsf{is\_valid}_{\mathsf{proc}}, \mathsf{is\_init}_{\mathsf{proc}}, \Sigma_{\mathsf{proc}}, \Omega_{\mathsf{proc}}, \delta_{\mathsf{proc}}, \omega_{\mathsf{proc}})$ *with a state space* $\mathcal{S}_{\mathsf{proc}}$*, a validity predicate* $\mathsf{is\_valid}_{\mathsf{proc}}$*, an initialization predicate* $\mathsf{is\_init}_{\mathsf{proc}}$*, an input alphabet* Σ_{proc}*, an output alphabet* Ω_{proc}*, a transition function* δ_{proc}*, as well as an output function* ω_{proc}*.*

The state space $\mathcal{S}_{\mathsf{proc}}$ depends on the underlying programming language – in our case, C0 or VAMP assembly. The communication interface, on the contrary, is determined by the operating system such that all OLOS processes possess the same in- and output alphabets. Table 1 presents both alphabets side by side. Note the strong correlation of outputs and inputs: When a process state features an output seen on the left, OLOS responds with one of the inputs shown in the

Table 1. Interface between OLOS and its application processes

Outputs Ω_{proc}	Inputs Σ_{proc}
ε_Ω (no call to a primitive)	ε_Σ (internal step)
SENDMSG *msgval msgnr*	SENDSUCCESS INVALIDMSGNR
RECVMSG *msgnr*	RECVSUCCESS *msgval* INVALIDMSGNR
EXFINISH	FINISHSUCCESS
INVPTRERR	INVPTRRESPONSE
REPEATERR	—
CONTINUEERR	CONTINUE

same table row on the right. Formally, we collect the matching output-input pairs (ω, i) in the set $\mathsf{olos\_responses}$.

The predicate $\mathsf{is\_init}_{\mathsf{proc}}$ mainly determines the set of initial states; it takes two parameters that constrain the initial memory of processes (effectively, specifying different subsets of initial states). We implicitly assume that the parameters fulfill basic validity constraints, which are formalized in the predicate $\mathsf{valid\_params}$. The predicate $\mathsf{is\_valid}_{\mathsf{proc}}$ formulates an invariant over the execution traces of processes.

Definition 2 (Validity of Process Semantics). *We call a process semantics valid, iff the invariant* $\mathsf{is\_valid}_{\mathsf{proc}}$ *holds for all initial states, i. e.,*

$$[\![\mathsf{valid\_params}\ \mathit{img}\ \mathit{pages};\ \mathsf{is\_init}_{\mathsf{proc}}\ \mathit{img}\ \mathit{pages}\ s_{\mathsf{proc}}]\!] \Longrightarrow \mathsf{is\_valid}_{\mathsf{proc}}\ s_{\mathsf{proc}}$$

and furthermore, the invariant is preserved under transitions with valid inputs:

$$[\![\mathsf{is\_valid}_{\mathsf{proc}}\ s_{\mathsf{proc}};\ (\omega_{\mathsf{proc}}\ s_{\mathsf{proc}},\ i) \in \mathsf{olos\_responses}]\!] \Longrightarrow \mathsf{is\_valid}_{\mathsf{proc}}\ (\delta_{\mathsf{proc}}\ i\ s_{\mathsf{proc}})$$

3.2 Specifying the Semantics for C0 and Assembly Processes

In this section, we shortly glance at the specification of our two particular process semantics. There are several runtime errors, namely INVPTRERR, REPEATERR, and CONTINUEERR. As correct programs do not feature these errors, we omit further details, here. VAMP assembly provides a special instruction TRAP n for the communication of a process with an operating system. The process-output function $\omega_{\mathsf{proc}}\ s_{\mathsf{asm}}$ uses the number n to distinguish between the SENDMSG, the RECVMSG, and the EXFINISH primitive; a number not assigned to a primitive results in CONTINUEERR, i. e., the instruction will simply be skipped. The parameter *msgval* is specified by a register pointing into the memory. Finally, the parameter *msgnr* is directly taken from a register. If neither a runtime error occurred nor the next instruction is TRAP n, the process output is ε_Ω.

For a transition $\delta_{\mathsf{proc}}\ s_{\mathsf{asm}}$ with the empty input ε_Σ, the assembly process semantics employs the underlying, sequential assembly semantics δ_{asm} for its

transition. Otherwise, the response from OLOS is reflected by placing a corresponding value into a specified *response register*; and in case of RECVSUCCESS, the received message is additionally stored into the process memory.

For C0 processes, we have implemented functions with inline assembly that wrap the necessary assembly instructions for the communication with OLOS. For illustration, Table 2 shows the implementation of the function `olosRecvMsg`, which wraps the system call RECVMSG.

The C0-process semantics treats a call to these functions as a primitive, i. e., the output function $\omega_{\mathsf{proc}}\ s_{\mathsf{C0}}$ simply determines whether the next statement is a call to such a wrapper function, and the transition function $\delta_{\mathsf{proc}}\ i\ s_{\mathsf{C0}}$ directly removes the function-call statement from the remaining program and updates the C0 state s_{C0} according to the input i.

3.3 Extending Compiler Correctness to Processes

Recall that Leinenbach & Petrova [10] have shown compiler correctness for the sequential C0 semantics with respect to the sequential part of VAMP assembly (cf. Theorem 1). Below, we extend their result to the two corresponding process semantics, which we have defined above.

First, we extend the existing simulation relation $\mathsf{consistent}$ for processes to $\mathsf{consis}_{\mathsf{proc}}$. This is mainly a syntactic adaptation and therefore we omit the details. Second, we define predicates for the successful execution of processes analogous to the sequential counterpart for assembly machines. Intuitively, a successful execution is characterized by the absence of runtime errors (including the sufficiency of memory). Furthermore, process transitions take inputs from OLOS. We make the output-input sequence *ois* explicit and require that the outputs in the sequence equal the process output in the corresponding state as well as that the sequence only contains matching output-input pairs, i. e., all pairs in the sequence are contained in $\mathsf{olos\_responses}$. Successful execution, we denote as: $\vdash^{\mathsf{proc}}_{\mathsf{C0}}\ s_{\mathsf{C0}} \xrightarrow{ois} s'_{\mathsf{C0}}$ for C0 processes and $\mathit{crange} \vdash^{\mathsf{proc}}_{\mathsf{asm}}\ s_{\mathsf{asm}} \xrightarrow{ois} s'_{\mathsf{asm}}$ for assembly processes, respectively. We only need the code range *crange* to determine that the assembly code does not modify itself. The maximal address, in contrast, that we know from the sequential assembly semantics, is encoded in the process state.

Table 2. C0 implementation of the receive primitive

```
int olosRecvMsg (t_msg *msg_ptr, unsigned int msgnr) {
  int result;
  asm { lw(r11, r30, asm_offset(msg_ptr));
        lw(r12, r30, asm_offset (msgnr));
        trap(2);
        sw(r22, r30, asm_offset( result ));
    };
  return result;
}
```

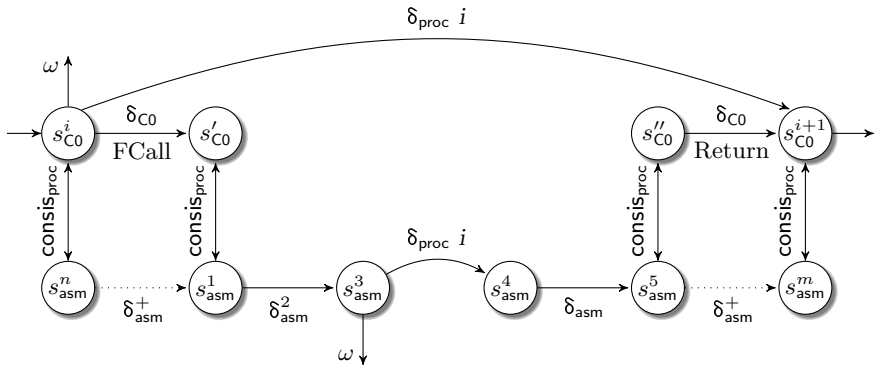

Fig. 3. Verification plan for the C0 implementation of the `olosRecvMsg` primitive

We overload code_range taking a C0 process state s_{C0}, which also contains the static C0 program.

Recall that multiple assembly transitions might simulate a single C0 transition. With our notions of successful process execution, we reflect this circumstance by output-input sequences of different length. Nevertheless, both process models should invoke the same primitives, i. e., all output-input pairs except for internal steps (ε_Ω, ε_Σ), remain equal. For this purpose, we define a normalization function $\gg ois \ll$ over the *ois* sequences that simply removes all internal steps.

Finally, we extend the compiler theorem to processes:

Theorem 2 (Process Simulation). *As in Theorem 1, we assume well-formed C0 and assembly states s_{C0} and s_{asm}, where the latter is not in a delay slot. Moreover, we assume that the states are related by* $\mathsf{consis}_{\mathsf{proc}}$ *and there is a successful execution from s_{C0} to a state s'_{C0}.*

Then, it exists a sequence ois′*, a function* alloc′*, and a state* $\mathsf{s'}_{\mathsf{asm}}$ *such that the normalized sequences are equal, s_{asm} successfully advances under* ois′ *to* $\mathsf{s'}_{\mathsf{asm}}$*, which is not in a delay slot, and s'_{C0} and* $\mathsf{s'}_{\mathsf{asm}}$ *are related by* $\mathsf{consis}_{\mathsf{proc}}$*. Formally:*

$$\begin{aligned}&[\![\mathsf{is\_valid}_{\mathsf{proc}}\ s_{\mathsf{C0}};\ \mathsf{is\_valid}_{\mathsf{proc}}\ s_{\mathsf{asm}};\ s_{\mathsf{asm}}.\mathsf{pcp} = s_{\mathsf{asm}}.\mathsf{dpc} + 4;\ \mathsf{consis}_{\mathsf{proc}}\ s_{\mathsf{C0}}\ \mathit{alloc}\ s_{\mathsf{asm}};\\ &\ \vdash^{\mathsf{proc}}_{\mathsf{C0}} s_{\mathsf{C0}} \xrightarrow{ois} s'_{\mathsf{C0}}]\!]\\ \Longrightarrow\ &\exists \mathit{ois'}\ \mathit{alloc'}\ \mathsf{s'}_{\mathsf{asm}}.\ \gg ois \ll = \gg \mathit{ois'} \ll \wedge\ \mathsf{code\_range}\ s_{\mathsf{C0}} \vdash^{\mathsf{proc}}_{\mathsf{asm}} s_{\mathsf{asm}} \xrightarrow{\mathit{ois'}} \mathsf{s'}_{\mathsf{asm}} \wedge\\ &\quad \mathsf{s'}_{\mathsf{asm}}.\mathsf{pcp} = \mathsf{s'}_{\mathsf{asm}}.\mathsf{dpc} + 4 \wedge \mathsf{consis}_{\mathsf{proc}}\ s'_{\mathsf{C0}}\ \mathit{alloc'}\ \mathsf{s'}_{\mathsf{asm}}\end{aligned}$$

Proof. We prove the theorem by induction on the output-input sequence *ois*. The induction start is trivial. In the induction step, we distinguish the possible process inputs. For an empty input ε_Σ, we employ Theorem 1.

For the other inputs, we examine the implementation of the corresponding primitives. Fig. 3 shows the case that primitive `olosRecvMsg` is called in some C0-process state s^i_{C0}. From the induction hypothesis, we know that there exists a corresponding assembly state s^n_{asm} that satisfies the simulation relation $\mathsf{consis}_{\mathsf{proc}}$. Using the sequential C0 semantics, we execute the function-call statement. From

the sequential compiler theorem, we know that the execution ($\delta^{+}_{\mathsf{asm}}$) of the corresponding, compiled code yields an assembly state s^{1}_{asm} satisfying the simulation relation $\mathsf{consis}_{\mathsf{proc}}$.

Starting in this state, we execute the inlined assembly code (cf. Table 2) of the function body.[6] After 2 steps, the code reaches the trap instruction in state s^{3}_{asm}. If our implementation is correct, the assembly process signals the same output ω in this state as the C0 process does in s^{i}_{C0}. At this stage, the transition function δ_{proc} uses an input i from Olos to proceed to s^{4}_{asm}. After a further step, we arrive at the end of the inlined assembly code. From 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 s''_{C0} immediately after the assembly statement.[7] For this to work, we have to show that the assembly code did not disrupt the C0 execution environment (code, stack pointer, etc.). If the assembly code preserves the integrity of the execution environment, we can again establish the $\mathsf{consis}_{\mathsf{proc}}$ relation.

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

The proof for the other primitives proceeds very similarly. □

4 The Cooperative Concurrent Application Model

The previous section presented a computational model for applications featuring inputs and outputs for the communication with Olos. For the verification of applications, however, the communication primitives are distracting. Most notably, Isabelle/Simpl cannot deal with inputs and outputs. Hence, we prefer a cooperative concurrent execution model, where we can sequentially reason about a complete computation phase, i. e., between two calls to the ExFinish primitive. This execution model forms the basis of the application semantics in Simpl.

Recall that during the compute phase, the sole task of Olos is the exchange of messages with the application that is scheduled to compute in the current slot. Hence, we can perceive the computation of Olos and the application as a single, sequential program with two separate states: The internal process state, on the one hand, and the Olos message buffers, on the other hand. While the internal state can be accessed via normal C0 statements, the message buffers are only accessible through the Olos-communication primitives.

The definition of this new computational model is straightforward: Each state is a pair (s_{proc}, mb) of a process state s_{proc} and a file of message buffers mb. The transition function emulates the behavior of Olos during the computation

[6] Recall that a transition δ_{asm} is equal to $\delta_{\mathsf{proc}}\ \varepsilon_{\Sigma}$.

[7] For reasoning about inlined Vamp assembly, we have been able to reuse previous work [15]. Note, however, that the semantics of the trap instruction is Olos-specific.

phase (assuming the corresponding application is scheduled). It distinguishes the output of the process and computes the corresponding input. Formally:

$\delta_{cc}\ (s_{proc}, mb) \equiv$
case $\omega_{proc}\ s_{proc}$ **of**
SENDMSG *msgval msgnr* $\Rightarrow$
if *msgnr* $<$ MSGCOUNT
then $(\delta_{proc}$ SENDSUCCESS $s_{proc}, mb[msgnr := msgval])$
else $(\delta_{proc}$ INVALIDMSGNR $s_{proc}, mb)$
| RECVMSG *msgnr* $\Rightarrow$
if *msgnr* $<$ MSGCOUNT **then** $(\delta_{proc}$ (RECVSUCCESS $mb[msgnr])\ s_{proc}, mb)$
else $(\delta_{proc}$ INVALIDMSGNR $s_{proc}, mb)$
| EXFINISH $\Rightarrow (\delta_{proc}$ FINISHSUCCESS $s_{proc}, mb)$
| INVPTRERR $\Rightarrow (\delta_{proc}$ INVPTRRESPONSE $s_{proc}, mb)$
| REPEATERR $\Rightarrow (s_{proc}, mb)$
| CONTINUEERR $\Rightarrow (\delta_{proc}$ CONTINUE $s_{proc}, mb)$
| $\varepsilon_\Omega \Rightarrow (\delta_{proc}\ \varepsilon_\Sigma\ s_{proc}, mb)$

Note that this model uses the generic process interface, i. e., s_{proc} might equally refer to a C0 or assembly state. Thus, it is easy to lift process simulation (Theorem 2) to cooperative concurrently executing applications:

Theorem 3 (Cooperative Concurrent Simulation). *Assuming (a) well-formed process states s_{C0} and s_{asm}, where the latter is not in a delay slot, (b) the absence of runtime errors in s_{C0} and all its immediate successors (predicate* runtime_error *does not hold), and (c) a well-formed file of message buffers mb (predicate* is_valid_mb *holds), then a C0 transition in the cooperative concurrent model simulates a number of cooperative concurrent assembly steps. Formally:*

$[\![\mathsf{is\_valid}_{proc}\ s_{C0};\ \mathsf{is\_valid}_{proc}\ s_{asm};\ s_{asm}.\mathsf{pcp} = s_{asm}.\mathsf{dpc} + 4;\ \mathsf{consis}_{proc}\ s_{C0}\ alloc\ s_{asm};$
$\neg\ \mathsf{runtime\_error}\ s_{C0};\ \mathsf{is\_valid\_mb}\ mb]\!]$
$\Longrightarrow \exists n\ alloc'\ s'_{asm}.$
let $(s'_{C0}, mb') = \delta_{cc}\ (s_{C0}, mb)$
in $(s'_{asm}, mb') = (\delta_{cc}{}^n)\ (s_{asm}, mb) \wedge \mathsf{consis}_{proc}\ s'_{C0}\ alloc'\ s'_{asm}$

Proof. At first, we show that the inputs i chosen by function δ_{cc} always match the process output. Hence, the transition from s_{C0} to $\delta_{proc}\ i\ s_{C0}$ is a successful execution. With Theorem 2, we know that there is a corresponding output-input sequence ois' such that

- the sequence ois' contains exactly one pair $(\omega_{proc}\ s_{C0}, i)$ as well as a number of pairs $(\varepsilon_\Omega, \varepsilon_\Sigma)$, i. e., $\gg ois' \ll = [(\omega_{proc}\ s_{C0}, i)]$, and
- there is a successful assembly execution under ois' starting in s_{asm} and ending in a state s'_{asm}, where $\mathsf{consis}_{proc}\ s'_{C0}\ alloc'\ s'_{asm}$ holds.

As δ_{cc} reacts with an empty input whenever the process outputs ε_Ω, function δ_{cc} yields the assembly state s'_{asm} if it is applied $|ois'|$ times to s_{asm}. Furthermore, the message buffers are equal in both cases. □

Verifying Applications in Isabelle/Simpl. The chief attraction of our cooperative concurrent execution model is that it allows us to reuse Isabelle/Simpl

[14]. The necessary adaptation of the existing technology for application verification is straightforward: It amounts to specify the effects of the primitives for SendMsg and RecvMsg 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 conveniently establish the absence of runtime errors as well as efficiently reason about functional correctness of applications between two calls to the ExFinish primitive.

5 Embedding Applications into the Overall ECU Model

In the previous section, we have claimed that the transition function δ_{cc} of the cooperative concurrent application model emulates the Olos transitions during the computation phase iff the corresponding application is scheduled. In this section, we prove that claim. Formally, we express our claim with the help of a projection function Π_p, which extracts the state of the application p, and an injection $\mathsf{inj}_p\ s_{cc}\ s_{ECU}$, which updates the state of application p in the ECU state s_{ECU} by s_{cc}. We state:

Theorem 4 (Application Embedding). *We assume that (a) the application p is computing in the ECU state s according to the static schedule t and (b) that p does not call for* ExFinish. *Then, the projection of p followed by a transition of δ_{cc} and its injection into s is equal to a transition of δ_{ECU}. Formally:*

$$[\![\mathsf{is\_computing}\ p\ t\ s;\ \neg\ \mathsf{calls\_finish}\ p\ s]\!] \Longrightarrow \mathsf{inj}_p\ (\delta_{cc}\ (\Pi_p\ s))\ s = \delta_{ECU}\ t\ \bot\ s$$

Proof. We have proven this fact in Isabelle/HOL. □

Note that Olos leaves the computing state (cf. Sect. 2.2) when an application calls for ExFinish, i. e., predicate calls_finish *pid s* holds. In this case, the transition δ_{ECU} *tables* $\bot$ *s* only raises the idle flag and sends FinishSuccess to process *pid*. From the Olos specification, we know that at all other ECU states, the process states remain constant.

6 Reasoning about Applications – a Practical Example

So far, we have elaborated on the foundation of our verification approach. Now, we take our arguments a step further and venture a practical example. For this purpose, we use a simple application program for cruise control. Note that all Olos applications share a common control flow, which is depicted in Fig. 4: 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, and a call to the ExFinish primitive.

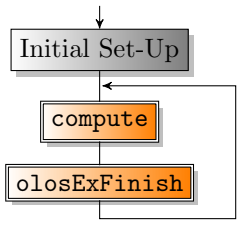

Fig. 4. General control-flow of applications

```
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;
}
```

Fig. 5. Function `compute` of our simple cruise-control application

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 Fig. 5) 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 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 ExFinish.

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. Then, we formally specify the functionality in terms of Hoare triples and, conveniently relying on the Hoare logics, prove the correctness of our specifica-

tion. Obviously, the containing loop alters neither the application's variables nor the message buffers. Thus, our proven property holds for a 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.

Using Theorem 3, we can then infer that the property can be translated down to assembly level. Furthermore, Theorem 4 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 is easily done by static analysis [16]. Eventually, we can then argue that the values computed by our example application are indeed sent onto the bus several slots later.

7 Conclusion

Based on existing technology for ordinary program verification, we formally proved the foundation of a pervasive verification approach for applications communicating with the operating system Olos. Additionally, we provided an application example illustrating that our approach can indeed be used in practice.

With our work, we respond to a long lasting grand challenge [3]. Despite many recent achievements in operating-systems verification [4], we only know of a single project that attempted pervasive systems verification: Bevier *et. al* [17] verified the correctness of KIT, a small assembly program that provides task isolation, device I/O, and single word message passing. Moreover, they ventured into the verification of applications but could not fully integrate their results. We can only refer to their work as groundbreaking because of KIT's simplicity.

Though even our computer system is simple, it is practically usable and the developed verification technique as well as the overall proof architecture may be reused for real computer systems. We implemented Olos as well as our example application in a C variant and employed a verified compiler for the mechanic translation into executable code. Thus, we are able to verify programs on the source code level—conveniently in a Hoare logic using the verification environment Isabelle/Simpl—and can then transfer the proven properties down to the assembly level (or even further [9]), e. g., to combine it with properties of the operating system or peripheral devices.

Integrating different layers of abstraction into a coherent theory is an important prerequisite for efficient reasoning. The verification engineer can then

choose a convenient abstraction layer for reasoning although the results might eventually be needed at a different abstraction layer. We see our contribution as an important milestone towards an evidence-based validation of safety-critical computer systems. Pervasive verification and software engineering should become two complementing disciplines aiming at the same target: perfectly reliable software.

Acknowledgments. We thank the anonymous peer reviewers for their detailed review reports with a very constructive criticism and many helpful suggestions.

References

1. Vartabedian, R., Bensinger, K.: Doubt cast on Toyota's decision to blame sudden acceleration on gas pedal defect. Los Angeles Times (January 30, 2010)
2. Guynn, J.: Apple co-founder Steve Wozniak says his toyota prius accelerates on its own. Los Angeles Times (February 3, 2010)
3. Moore, J.S.: A grand challenge proposal for formal methods: A verified stack. In: 10th Anniversary Colloquium of UNU/IIST, Springer (2002) 161–172
4. Klein, G.: Operating system verification — an overview. Sādhanā **34**(1) (2009) 27–69
5. Beyer, S., Jacobi, C., Kröning, D., Leinenbach, D., Paul, W.J.: Putting it all together: Formal verification of the VAMP. STTT **8**(4-5) (2006) 411–430
6. In der Rieden, T., Tsyban, A.: CVM – a verified framework for microkernel programmers. In: Systems Software Verification. Volume 217 of ENTCS., Elsevier Science B.V. (2008) 151–168
7. Daum, M., Schirmer, N.W., Schmidt, M.: Implementation correctness of a real-time operating system. In: SEFM, IEEE Computer Society (2009) 23–32
8. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic. Volume 2283 of LNCS. Springer (2002)
9. Alkassar, E., Hillebrand, M.A., Leinenbach, D.C., Schirmer, N.W., Starostin, A., Tsyban, A.: Balancing the load – leveraging a semantics stack for systems verification. J. Autom. Reasoning **42**(2-4) (2009) 389–454
10. Leinenbach, D., Petrova, E.: Pervasive compiler verification: From verified programs to verified systems. In: Systems Software Verification. Volume 217 of ENTCS., Elsevier Science B.V. (2008) 23–40
11. Schirmer, N.W.: Verification of Sequential Imperative Programs in Isabelle/HOL. PhD thesis, TU Munich (2006)
12. Kopetz, H., Grünsteidl, G.: TTP – A protocol for fault-tolerant real-time systems. IEEE Computer **27**(1) (1994) 14–23
13. American National Standards Institute: ANSI/ISO/IEC 9899-1999: Programming Languages — C. American National Standards Institute, New York, USA (1999)
14. Alkassar, E., Hillebrand, M.A., Leinenbach, D., Schirmer, N., Starostin, A.: The Verisoft approach to systems verification. In: Verified Software: Theories, Tools, and Experiments. Volume 5295 of LNCS., Springer (2008) 209–224
15. Starostin, A., Tsyban, A.: Correct microkernel primitives. In: Systems Software Verification. Volume 217 of ENTCS., Elsevier Science B.V. (2008) 169–185
16. Heckmann, R., Ferdinand, C.: Worst-case execution time prediction by static program analysis. White paper, AbsInt Angewandte Informatik GmbH (2004)
17. Bevier, W.R.: Kit and the short stack. J. Autom. Reasoning **5**(4) (1989) 519–530