Implementation Correctness of a Real-Time Operating System

Matthias Daum*, Norbert W. Schirmer†, and Mareike Schmidt*
*Computer Science Dept., Saarland University
66123 Saarbrücken, Germany
Email: {md11,mareike}@wjpserver.cs.uni-saarland.de
†German Research Center for Artificial Intelligence (DFKI)
66041 Saarbrücken, Germany
Email: Norbert.Schirmer@dfki.de

***Abstract*—In the modern car, electronic devices are even employed for safety-critical missions like brake control, where failures might cost human lives. Among various approaches to increase the reliability of those devices, pervasive formal verification most securely rules out all systematic failures. The main target of the Verisoft project is the development of technology for pervasive verification. Its application has been demonstrated in the automotive context by an exemplary distributed system consisting of hardware, a real-time operating system, and application programs. The contribution of this paper is a formal refinement proof linking an abstract specification of this real-time operating system to its C implementation.**

***Keywords*-Real-Time Operating System; Pervasive Verification; Refinement Proof; C Code Verification; Isabelle/HOL**

I. Introduction

The continually increasing number and variety of electronic components in cars results in an exponential growth for communication channels. Over time, adding more and more wires has led to space, complexity and maintenance problems. As an alternative, several components can share the same wire and use a communication protocol on this bus. For that purpose, Kopetz and Grünsteidl [1] 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.

The approach has, however, its disadvantages: Previously independent components meanwhile share the same bus, which requires clock synchronization and a reliance on former safety-uncritical components – a problem with the multi-media system should certainly not propagate to the brake system. The safest way to prevent such problems is the exclusion of systematic errors by formal verification. Certainly, this method cannot be limited to a single system layer but should span over as many layers as possible. While program verification has been known and used over decades, the pervasive verification of complete computer systems still remains a grand challenge [2].

Among others [3], 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. Pervasive verification means that all 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 hardware level. Our operating system OLOS has been implemented on a verified processor [4] using a generic programming framework for operating systems [5]. An emergency-call system [6], [7] serves as an example for its practicability. In our paper, we report on the refinement proof from an abstract specification layer down to the source code of this operating system.

Our contribution is an important milestone towards an evidence-based validation of safety-critical systems. We have shown that formal methods can indeed produce the evidence that an operating-system implementation meets its specification, providing the highest possible level of quality assurance for software. The developed verification technique and the overall proof architecture may be reused in similar contexts. In the long run, source-code verification should supplement software certification, which is currently limited to solely monitor the development process [3], [8]. For safety-critical software, both, formal methods and software engineering, can thus become two complementing disciplines aiming at the same target: 100% reliable software.

The paper is organized as follows: The next section explains the context of our work, including the design principles of OLOS and our verification environment. In Sect. III, we elaborate on the implementation, while Sect. IV presents the formal specification of our real-time system. Our key contribution constitutes the formal correctness theorem shown in Sect. V. We conclude in Sect. VI.

II. Background

Our distributed system comprises a number of components that are connected via a communication bus. These so-called *electronic control units* (ECUs) consist of a general-purpose RISC processor and an *automotive bus controller*

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.

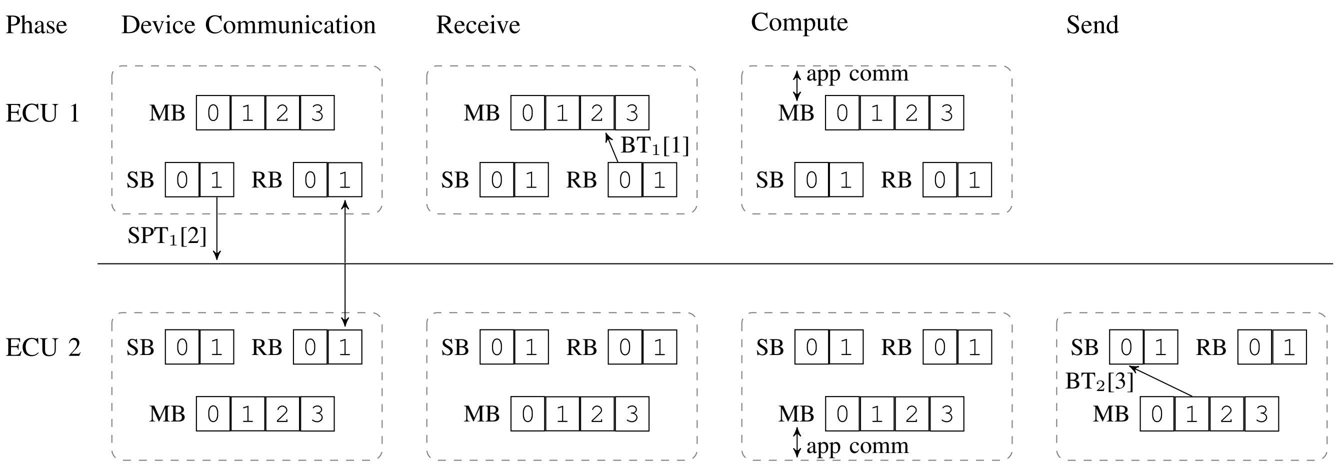

Figure 1. Transition phases during slot 2

(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. In its mediator role, the ABC buffers the messages from both sides, using pairs of send (SB) and receive buffers (RB) in order to deal with metastability, a common problem in electronic circuit design.

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).

In the following subsections, we introduce the principles of the time-sharing communication scheme, on the one hand, and our verification environment, on the other hand.

A. *Our Time-Sharing Communication Protocol*

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*. The organization of a round is described by several scheduling tables. Each ECU features its own set of these tables. Firstly, the *send-permission table* (SPT) specifies for each slot, whether the ECU is allowed to send. Secondly, the *application scheduling table* (AST) determines for each slot, which application should compute. Finally, a *buffer-index table* (BT) identifies in each slot, which message buffer of Olos is exchanged with the device. Table I shows scheduling tables for the two communicating ECUs that we use as a running example in our paper.

We divide each slot into four phases: device communication, receive, compute, and send. Fig. 1 depicts them for slot 2 of our running example.

Device Communication: In this phase, the ABC device is communicating with the other ECUs via the bus, while the remaining phases are characterized by communication with Olos. In our example, ECU 1 broadcasts the message that Olos has written to the device in the previous slot. All ECUs receive this message from the bus.

Receive: The operating system reads the receive buffer from the ABC device into its own message buffers. Fig. 1 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.

B. *Verifying C Programs – the Isabelle/Simpl Framework*

The formalizations presented in this article are mechanized and checked within the interactive theorem prover Isabelle/HOL [9], implementing higher-order logic [10]. This paper is written using Isabelle's document generation facilities, which guarantees that the presented theorems correspond to formally proven ones. We distinguish formal entities typographically from other text. We use a sans-serif font for types and constants (including functions and predicates), e. g., map, a slanted serif font for free variables, e. g., *x*, and a slanted sans-serif font for bound variables,

Table I
EXAMPLE SCHEDULING TABLES

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

e. g., x. Type variables have a leading tick, e. g., $'a$. HOL (and Simpl) keywords are typeset in bold font, e. g., **let**.

The logical and mathematical notions follow the standard notational conventions biased towards functional programming. We only present the more unconventional parts here. We prefer curried function application, e. g., $f\ a\ b$ instead of $f\ (a,\ b)$. In this setting the latter becomes a function application to *one* argument, which happens to be a pair.

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., $'a\ \mathsf{list}$ is the list type with type variable $'a$. In HOL all functions are total, e. g., $\mathsf{nat} \Rightarrow \mathsf{nat}$ is a total function on natural numbers. There is, however, a type $'a\ \mathsf{option}$ 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 un(der)specified value. Partial functions can be represented by the type $'a \Rightarrow\ 'b\ \mathsf{option}$. The syntax and the operations for lists are similar to functional programming languages like ML or Haskell. The empty list is $[]$, with $x\ \#\ xs$ the element x is 'consed' to the list xs. With $\mathsf{map}\ f\ xs$, the function f is applied to all elements in xs. The n-th element of a list xs can be selected with $xs[n]$. A record is constructed by assigning all of its fields, e. g., $(\!|\mathsf{fld}_1 = v_1, \mathsf{fld}_2 = v_2|\!)$. Field fld_1 of record r is selected by $r.\mathsf{fld}_1$ and gets updated with a value x via $r(\!|\mathsf{fld}_1 := x|\!)$.

For the verification of C0, a fragment of C, we use a general program-verification framework for sequential imperative programming languages: Isabelle/Simpl [11], [12]. 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. Isabelle/Simpl is polymorphic over the state space; we use records but hide the details by an Isabelle syntax, such that $\{\sigma.\ {}^{\sigma}\mathsf{var} = 5\}$ denotes the assertion that the value of program variable var in state σ is five.

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. The procedure environment Γ is a partial function from procedure names to statements, which is consulted when calling procedures.

The framework includes a big-step semantics, a Hoare logic for partial as well as total correctness and an automated verification-condition generator for Simpl. Within this sequential core language, assembly fragments as well as C0 are embedded. The embedding is based on a compiler converting C0 constructs in terms of operations provided by the small-step semantics of the RISC processor. A correctness proof for this compiler, which links the small-step semantics to the Simpl big-step semantics, is also provided [13]. This correctness theorem about the embedding of C0 into Simpl allows for mapping low-level properties to more abstract ones formulated on the big-step semantics of C0. Alkassar *et al.* have dedicated a separate article [14] on the semantics stack in Verisoft.

III. Implementation

A. CVM: A Programming Framework for Operating Systems

Our operating system (OS) is implemented on the verified RISC processor *VAMP* [4] using a programming framework called *communicating virtual machines* (CVM) [5]. 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 and then 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.

We distinguish two interrupt sources: On the one hand, *external interrupts* might occur, which comprise the reset signal (upon power up) and interrupts generated by peripheral devices. On the other hand, the currently executed instruction may cause *exceptions* like an illegal instruction or a misalignment. Most notably, if an application is computing, it may cause an exception with the special instruction `trap` in order to explicitly call the operating system. The instruction features an immediate constant to identify a certain OS functionality. Upon such a *system call*, the processor immediately transfers control to the interrupt-service routine.

After saving the old processor context, the interrupt-service routine calls the C function `kdispatch` of OLOS. In OLOS, the function `kdispatch` and its subroutines call CVM primitives for the communication with the ABC device and the manipulation of application registers and memory. The return value of `kdispatch` determines, which application should resume its computation. In the next subsection, we describe the functionality of `kdispatch` in more detail.

B. The Top-Level Function of OLOS

When an interrupt arrives, the CVM framework saves the old processor context and calls `kdispatch` with two parameters: the *interrupt cause*, a bit vector of occurred

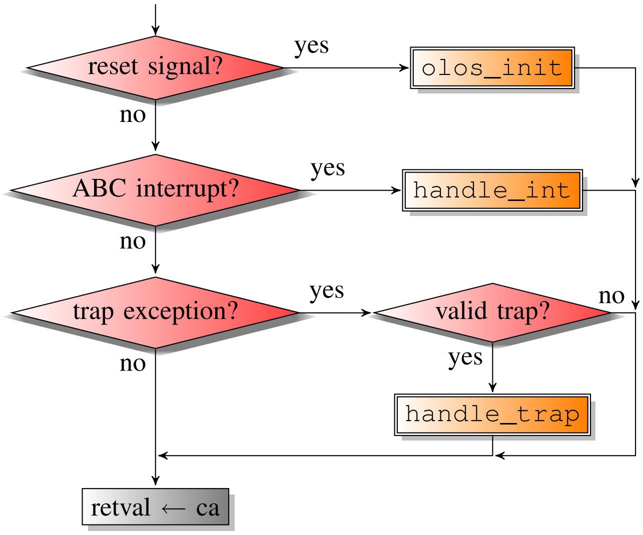

Figure 2. The program-flow diagram of function `kdispatch`

interrupts, and the *exception data*, which is the provided immediate constant if a trap has occurred.

Fig. 2 shows the control-flow graph of this function. From the interrupt cause, the function distinguishes three cases:

Initialization: After power-up, the processor generates a *reset* interrupt. In this case, the Cvm framework sets up its internal data structures and then passes the interrupt on to the `kdispatch` function. If called with the reset-interrupt bit set, `kdispatch` calls the function `olos_init`. This function initializes the internal C data structures of Olos, on the one hand, and configures the ABC device, on the other hand. The remaining interrupt vector is ignored.

ABC Interrupt: The ABC device raises its interrupt line when expecting data exchange with the operating system. There can be up to two such communications: If the ECU is scheduled to send a message on the bus in the next slot, the device raises its interrupt to signal that the compute phase has ended and Olos transfers the designated message to the device. As soon as the send phase has finished, the device raises its interrupt line again and the operating system reads the message that has been received in the slot before. If it is not the ECU's turn to send in the next slot, the send phase is omitted, i. e., there is only one interrupt in this slot and Olos starts reading the received message from the device when the interrupt at the end of the compute phase occurs. This functionality is encapsulated in the function `handle_int`.

Trap Exception: During the compute phase, the current application may communicate with the operating system 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 trap handler is implemented in the function `handle_trap` (see below).

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, determining the currently running application. The subroutines of `kdispatch` ensure that the variable has either the value of the AST table in the current slot or the special value IDLE. Upon return, the Cvm framework transfers control to the corresponding application or waits for interrupts, respectively.

C. Implementing System Calls – the Trap Handler

As sketched in Sect. III-A, the `trap` instruction is the designated hardware mechanism to request services from the operating system. We call these requests *system calls*. The instruction features an immediate constant, which allows us to distinguish different kinds of system calls. Furthermore, register values might serve as parameters to such a call. Olos implements three system calls: The calls *Send* and *Receive* are used for the message exchange between Olos and the applications. Moreover, an application should signal with the call *ExFinished* 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 *ExFinished*. Our operating system, however, works correctly even if this requirement is violated.

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`. Fig. 3 shows the control-flow diagram of this function.

The implementation of the *ExFinished* call is simple: Olos sets its global variable 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 require two parameters:

- a pointer *msg_ptr* into the application's memory indicating either the message value (for the *Send* call) or the designated buffer that should accommodate the message (for the *Receive* call), and
- a message identifier *msg_id* selecting the message buffer in Olos as source or destination of the message, respectively.

The parameters are held in registers. The implementation reads them using the Cvm primitive `cvm_get_gpr` and checks their validity. If these checks fail, Olos simply stores an error value in a designated register of the current application (using the Cvm primitive `cvm_set_gpr`).

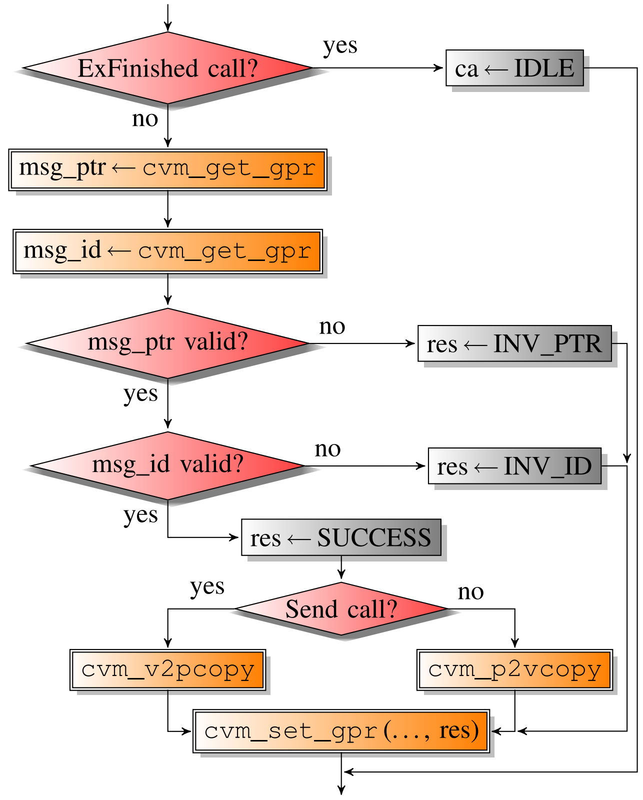

Figure 3. The program-flow diagram of function `handle_trap`

In the successful case, the message value is transferred either from OLOS's message buffer into the application's buffer using the CVM primitive `cvm_p2vcopy` (for *Send*), or from the application into OLOS using `cvm_v2pcopy` (for *Receive*). Finally, the designated error register is set to the value SUCCESS.

IV. Formal Specification

Correctness is usually defined as compliance with a specification. In our case, this specification is an automaton. 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 operating system's correctness with respect to a model of the ABC device, which specifies the device's expected behavior. The specification then comprises the complete behavior of an ECU, consisting of the operating system and the device. This section proceeds with the description of the ABC automaton and the ECU automaton.

A. ABC Automaton

As sketched in Sect. II, the ABC device is responsible for the timely transmission and reception of messages. For this purpose, the ABC device features a two-placed ring buffer for outgoing messages (*send buffer*) and a second one for incoming messages (*receive buffer*). In addition, it requires a considerable amount of system-specific information. This information includes time data like the slot length in hardware cycles, the SPT table, 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 written by OLOS in an initialization phase right after power up. This phase is identified by an *initialization flag*, which is initially raised together with the reset signal and remains enabled until OLOS signals the completion of the initialization by writing the *set-ready command*. 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, which directly represents the device's interrupt line.

Accordingly, we have formalized the state space of the ABC automaton in Isabelle/HOL by a record with the fields s.SB and s.RB for the buffers, s.CR for the set of configuration registers, s.init_flag, s.send_flag, and s.INT for the flags, as well as s.CSN for the current slot number.

The device interacts with the processor, on the one hand, and the external environment, on the other hand. Fig. 4 visualizes the state transitions of the device with respect to the inputs from both entities. Note that the diagram shows suggestive names instead of a flag combination from our formalization: A raised flag corresponds to the *init* state, the interrupt line is raised in the *read* and the *write* state, and the send flag is set for $\mathit{idle}_{\mathsf{w}}$ and *write*. We annotate flowing data with e for external environment and p for the processor. Inputs are denoted by ↓, and outputs by ↑. 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*).

After this command, the device idles ($\mathit{idle}_{\mathsf{r}}$) expecting communication with the bus ($e{\updownarrow}\mathit{msg}$). 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 send buffer to the bus and writes the same message to its receive buffer (here we assume ideal hardware). Otherwise, it awaits a message from the bus and stores it into its receive buffer. We abbreviate "if sp then ($e{\downarrow}\mathit{timer}$, $e{\uparrow}\mathit{msg}$) else $e{\downarrow}\mathit{msg}$" by $e{\updownarrow}\mathit{msg}$.

When the receive buffer has been written, the device enters the *read* state, where the processor requests ($p{\downarrow}\mathit{read}$) the message from the receive buffer ($p{\uparrow}\mathit{msg}$). 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 ($\mathit{idle}_{\mathsf{r}}$ or $\mathit{idle}_{\mathsf{w}}$, respectively). The former has been discussed before and the latter analogously awaits the beginning of the send phase. This phase is deterministically

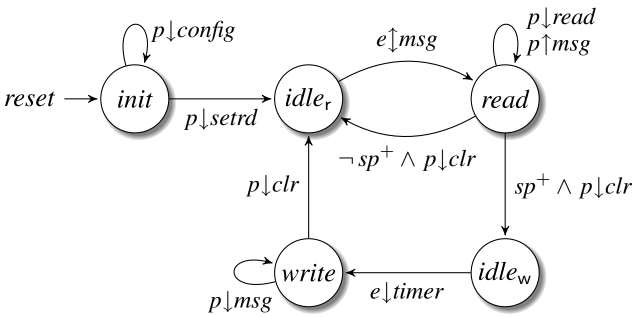

Figure 4. State diagram for the ABC

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, where the processor writes ($p{\downarrow}msg$) the send buffer. When finished, the processor sends the clear-interrupt command just like in the receive case. With this command, the device transfers back to the $idle_{\mathsf{r}}$ state.

In Isabelle/HOL, we have formalized the transitions of the ABC automaton in the function δ_{abc}.

B. Abstract ECU Automaton

In this subsection, we consider the abstract automaton describing the behavior of the whole ECU consisting of the processor with its running operating system and applications together with the ABC device. From the operating system, the state space of this model inherits the application mapping s.AM, the message buffers s.MB, and an *idle flag* (s.idle_flag), which abstracts the current application identifier from the implementation state. Furthermore, we literally embed the state of the ABC automaton above into our state space (s.abc_dev).

In Fig. 5, the state diagram of the ECU automaton, it is possible to recognize the transition phases of a slot as sketched in Fig. 1. The states *recv* and *send* in the diagram directly correspond to the beginning of the receive phase and the send phase. During the slot's compute phase, the automaton is in one of the states *comp*, $idle_{\mathsf{r}}$, and $idle_{\mathsf{s}}$. Finally, the device communication is depicted by $e{\updownarrow}msg$.

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 according transition resembles the device communication phase.

The device communication $e{\updownarrow}msg$ triggers a new slot and the ECU changes to the receive state (*recv*). In this state, the ABC interrupt is raised and the send flag is not set. The transition labeled *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).

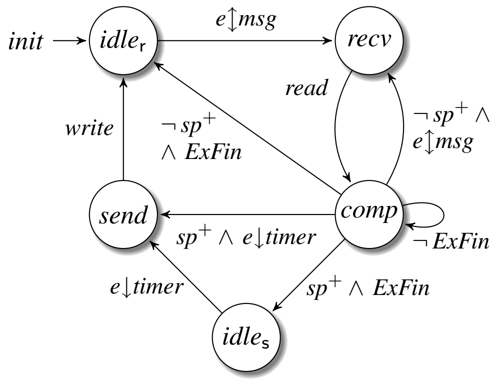

Figure 5. State diagram for the ECU

Immediately afterwards, the ECU is in the compute state *comp*. The idle flag as well as the ABC's interrupt flag are unset. 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 *ExFinished* call, the ECU remains in the *comp* state. In case of an *ExFinished* call, the idle flag is set and the ECU descends into an idle state, $idle_{\mathsf{s}}$ or $idle_{\mathsf{r}}$, 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.

In Isabelle/HOL, we have formalized the transitions of the abstract ECU automaton in the function δ_{ECU}. An optional external input i distinguishes external device steps ($i = \lfloor e \rfloor$) from processor computation ($i = \bot$).

V. Formal Refinement

This section reports on a formal correctness proof of the OLOS implementation. In contrast to a normal application program, an operating system is neither entirely written in C nor does it usually terminate. These circumstances demand further investigation of the verification framework and the system's architecture.

Firstly, we cannot prove the correctness of hardware-specific assembly code in Isabelle/Simpl. The code of CVM primitives has been verified on the processor level [15]. In Simpl, we can nevertheless express the effects of those low-level computations that are visible to the C0 programmer. We abstract these computations into atomic *XCalls* (extended

$$\begin{tikzcd}
 & s_1 \arrow[r, "\delta_{\mathsf{ECU}}\ i_1"] & s_2 \arrow[r, "\delta_{\mathsf{ECU}}\ i_2"] & s_3 \arrow[r, dotted, "\cdots"] & s_{\mathsf{n}} \\
\sigma_0 \arrow[r, "\delta^{\mathcal{I}}_{\mathsf{ECU}}\ \bot"'] & \sigma_1 \arrow[u, "\mathsf{abs}_{\mathsf{ECU}}"] \arrow[r, "\delta^{\mathcal{I}}_{\mathsf{ECU}}\ i_1"'] & \sigma_2 \arrow[u, "\mathsf{abs}_{\mathsf{ECU}}"] \arrow[r, "\delta^{\mathcal{I}}_{\mathsf{ECU}}\ i_2"'] & \sigma_3 \arrow[u, "\mathsf{abs}_{\mathsf{ECU}}"] \arrow[r, dotted, "\cdots"'] & \sigma_{\mathsf{n}} \arrow[u, "\mathsf{abs}_{\mathsf{ECU}}"]
\end{tikzcd}$$

Figure 6. Simulation between implementation and model

calls), which are similar to conventional function calls. In particular, we augment the state space of the overall program with an additional program variable cvmX representing the external, hardware-specific state. Semantically, an XCall is a function call performing a transition on this external state and communicating with C0 via parameter passing and return values. In our case, the external state comprises an identifier of the current application, the processor state of all applications, and the ABC device configuration.

Secondly, operating-system verification deals with non-termination. Observe, however, that our top-level routine of OLOS itself terminates while the endless loop is realized in the low-level hardware-specific parts. Hence, we simply formulate a single OLOS step – in analogy to the CVM primitives – as an ordinary Simpl function. Then, we expand the correctness theorem over unbounded execution traces.

In particular, we express correctness in terms of simulation between the execution traces of the implementation and the abstract ECU automaton, which both have been introduced in the previous two sections. Fig. 6 depicts the situation in detail: The implementation starts in the state σ_0 directly after power up. Its first transition $\delta^{\mathcal{I}}_{\mathsf{ECU}}$ yields in state σ_1 that can be abstracted via $\mathsf{abs}_{\mathsf{ECU}}$ to an initial specification state. This simulation is retained by all further transitions. Below, we define prerequisites like $\mathsf{abs}_{\mathsf{ECU}}$ and outline the induction proof for our correctness theorem.

A. Prerequisites

For our correctness theorem, we need a number of prerequisites. Firstly, there is the Simpl function `cvmstep` representing a combined step of CVM and OLOS. We model the reset interrupt by the global variable reset and assume that its initial value is True. The initial value corresponds to the processor state right after power-up. If reset is True, `cvmstep` changes it to False, initializes the applications, and invokes `kdispatch` with an interrupt vector of 1, i. e., exactly the reset interrupt is raised. In further computations, a `cvmstep` execution consists of the following parts: If there is a current application, it executes a single step. Then, CVM computes the bit vector of occurred interrupts, combining the exceptions caused by the current application and occurred device interrupts. If there are any interrupts, CVM calls the `kdispatch` function with the bit vector and determines from the function's return value, which application will become the current one in future steps. Note that the return value might not be a valid application identifier, in which case CVM will idle in future steps until the next device interrupt.

Secondly, we need a simulation relation. More precisely, we use the function $\mathsf{abs}_{\mathsf{ECU}}$ that maps implementation states to model states:

$$\begin{array}{l}
\mathsf{abs}_{\mathsf{ECU}}\ \sigma \equiv \\
\quad (\!|\mathsf{AM} = \mathsf{cvm\_apps}\ \sigma.\mathsf{cvmX}, \\
\quad\ \mathsf{MB} = \mathsf{map}\ (\lambda n.\ \sigma.\mathsf{heap\_msg}\ \sigma.\mathsf{MB}[n]) \\
\qquad\qquad [0..{<}\mathsf{MSGCOUNT}], \\
\quad\ \mathsf{idle\_flag} = (\sigma.\mathsf{ca} = \mathsf{IDLE}), \\
\quad\ \mathsf{abc\_dev} = \mathsf{cvm\_abc}\ \sigma.\mathsf{cvmX}|\!)
\end{array}$$

The function constructs a state of the ECU automaton as described in Sect. IV-B. From the external CVM state, we extract the applications and the device configuration. They are stored in the components AM and $\mathsf{abc\_dev}$, respectively. Furthermore, the variable of the current application ca is abstracted to the idle flag, i. e., the flag is raised iff $\sigma.\mathsf{ca} = \mathsf{IDLE}$. Finally, the message buffers are gathered from the different memory objects in the implementation, which are scattered over the heap.

Thirdly, we formulate an invariant over the implementation states. This predicate excludes implementation states that cannot occur in an execution trace. Recall that we prove the simulation between implementation and specification by induction. Our induction is based on the fact that the state after the first `cvmstep` at power up can be abstracted via $\mathsf{abs}_{\mathsf{ECU}}$ to an initial specification state. The inductive step formalizes that the abstraction relation is preserved by any possible further `cvmstep`. We combine properties of these further steps in our invariant, which is established at the induction start and preserved by the inductive step.

This invariant comprises assumptions on variables containing the schedule, the current slot, the applications, and the device. For instance, OLOS and the device independently store information on the schedule or the current slot. Certainly, the redundant information must not be contradicting. We have already learned in Sect. III-B about another constraint: The variable ca has either the value of the application scheduling table in the current slot or the special value IDLE. Furthermore, we require the well-formedness of the message buffers and their content. These constraints arise from a weaker typing model in Isabelle/HOL, e. g., representing fixed-length arrays by lists. With all prerequisites in place, we can formalize our inductive proof.

B. 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 (see Sect. IV-A), on the other hand. We combine all these assumptions in the constant $\mathsf{at\_power\_up}\ \sigma$. After the initial `cvmstep`, the invariant $\mathsf{invariant}$ holds for the successor state τ, and the abstracted state $\mathsf{abs}_{\mathsf{ECU}}\ \tau$ is an initial specification state:

Theorem 1 (Induction start)**.** *After power-up, the initial* `cvmstep` *yields in a state, which can be related via* $\mathsf{abs}_{\mathsf{ECU}}$ *to an initial specification state. Moreover, the invariant holds for the yielded state. Formally:*

$$\begin{aligned}&\Gamma \vdash_{t} \{\sigma.\ \mathsf{at\_power\_up}\ \sigma\}\\ &\quad \textbf{CALL}\ \mathsf{cvmstep}()\\ &\quad \{\tau.\ \mathsf{invariant}\ \tau \wedge \mathsf{is\_init\_state}\ (\mathsf{abs}_{\mathsf{ECU}}\ \tau)\}\end{aligned}$$

Proof: The code, which initializes the required data structures and the device, can roughly be divided into three parts, each containing a loop. We have to show the correctness not only for the execution and the loop invariants of each part, but also the property preservation from one to the following piece of code. The problem of passing on invariants and interim results to discharge next preconditions does not only appear between separated pieces of code but also within a single loop. This is always the case when sequential instructions depend on each other, i. e., in the second loop where three different CVM primitives initialize registers, allocate memory and load applications. ■

C. Induction Step

In the induction step, we assume that the invariant initially holds, i. e., $\mathsf{invariant}\ \sigma$. Additionally, we require that the simulation relation between specification and implementation states hold. Note that the postcondition of the induction start establishes this assumption. After each execution of `cvmstep`, the invariant should hold for the successor state. We show that the invariant and the simulation are preserved by the execution of `cvmstep` on the implementation layer and a transition on the specification layer.

Theorem 2 (Induction step)**.** *The simulation relation and the invariant are preserved under a* `cvmstep`*. Formally:*

$$\begin{aligned}&\Gamma \vdash_{t} \{\sigma.\ \mathsf{invariant}\ \sigma \wedge s = \mathsf{abs}_{\mathsf{ECU}}\ \sigma\}\\ &\quad \textbf{CALL}\ \mathsf{cvmstep}()\\ &\quad \{\tau.\ \mathsf{invariant}\ \tau \wedge \mathsf{abs}_{\mathsf{ECU}}\ \tau = \delta_{\mathsf{ECU}} \perp s\}\end{aligned}$$

Proof: In our proof, we distinguish three possible phases: receive, compute and send (the device communication does not involve OLOS). In all these phases, the applications and the ABC device are strictly separated. The send and receive phase affect the operating system as well as the ABC devices whereas the compute phase only regards the current application and OLOS. Both phases concerning the communication with the ABC device require the validity of the device and the OLOS variables as well as the contents of send, receive and message buffers after a message transmission. In the compute phase we show the validity of the interaction between the operating system and the current application in case of an incoming exception or trap. If no timer interrupt occurs before the application finishes its execution, we distinguish between a "normal" execution step of the application, a system call or another exception. We have to assure in all cases that the application remains valid, i. e., that the values of the PCs, registers and memory are correct. Otherwise the timer occurs before the application has finished. Every execution step except a trap is finished first, before the receive or send phase is entered. Afterwards, the applications, message buffers and devices have to be valid. ■

D. Simulation

Concluding from the two previous theorems, we claim that the simulation relation continues to holds after an initial execution of `cvmstep` over all finite traces. Formally we obtain this simulation by employing the soundness theorem of the Hoare logic [12]. This theorem allows us to interpret the proven Hoare triples on the operational semantics, which results in a transition function, in our case, because we have proven termination and C0 is deterministic. Note, that within the transition function $\delta^{\mathcal{I}}_{\mathsf{ECU}}$, we combine the C0 computation and ABC transitions analogous to δ_{ECU}. In our formal theorem, we iterate transitions over a list of optional external inputs *is* using the function fold.

Theorem 3 (Simulation)**.** *For an initial implementation state* σ_0*, where* $\mathsf{at\_power\_up}\ \sigma_0$ *holds, we obtain simulation on external inputs is between the implementation* $\delta^{\mathcal{I}}_{\mathsf{ECU}}$ *and its specification* δ_{ECU} *after the initial step* $\sigma_1 = \delta^{\mathcal{I}}_{\mathsf{ECU}} \perp \sigma_0$*. Formally:*

$$\mathsf{abs}_{\mathsf{ECU}}\ (\mathsf{fold}\ \delta^{\mathcal{I}}_{\mathsf{ECU}}\ \mathit{is}\ \sigma_1) = \mathsf{fold}\ \delta_{\mathsf{ECU}}\ \mathit{is}\ (\mathsf{abs}_{\mathsf{ECU}}\ \sigma_1)$$

Proof: Theorem 1 states that starting in the state σ_0 at power-up, the initialization step $\sigma_1 = \delta^{\mathcal{I}}_{\mathsf{ECU}} \perp \sigma_0$ establishes the implementation invariant as well as the simulation $\mathsf{abs}_{\mathsf{ECU}}$ between implementation and specification states. This simulation is preserved under transitions of both models because of Theorem 2 and an additional lemma stating that external ABC transitions δ_{abc} preserve the invariant. ■

VI. CONCLUSION

We have formally verified functional correctness of the real-time operating system OLOS in the context of pervasive verification. This operating system has been designed for an industrial context. By our extensive case study, we have challenged the current verification tools. Our complete formal work is available online.[1] With our work, we respond to a long lasting grand challenge [2] as well as to a general encouragement [16] for more experimental work in computer science. The section proceeds with related work, gained insights and future work.

A. *Related Work*

We are indebted to Klein for a comprehensive article [3] on past and present approaches to operating-system verification. Summarizing, Klein only presents a single, fully verified operating system: KIT, a small assembly program, that provides task isolation, device I/O, and single word message passing. This verification project can only be referred to as groundbreaking in the area of pervasive verification. The operating system is very far from any real system and the verification is based on a fairly abstract LISP execution model. 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, e. g., UCLA Secure Unix [17] or VFiasco/Robin [18]. Other projects, like Embedded Device [19], were successful in the verification but did not reach down to the code level. Furthermore, the Flint project [20] verified the correctness of certain low-level assembly fragments but did not aim at full code coverage. Embedded Device and Flint amend our work towards higher and lower layers, respectively. Finally, the L4.verified project [**?**], [**?**] completed the refinement proof for a general-purpose microkernel just days before this paper went on print. While their kernel is much more complex than Olos, they verified neither boot-up nor assembly code.

Beside these projects, there are several other verification attempts within the Verisoft project [22].[2] Our pervasive approach has been the subject of earlier papers [13], [14] as well. Most notably, the verification of the microkernel Vamos [23] has reached a mature state. While this kernel has more features than Olos, its current verification state does not cover the system's start up.

B. *Gained Insights from Pervasive Verification*

The aim of pervasiveness has considerably influenced our verification approach and its result: We were able to rely on previous work, namely VAMP, CVM, and Isabelle/Simpl, which considerably increased the possible degree of reliance that our verification result indeed holds for the overall system. Certainly this approach has also disadvantages.

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 [13]. 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.

Furthermore, several iterations were necessary in our verification process until we could verify our main theorem because earlier abstractions turned out to be insufficient. 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 correctness proof with 2 years.

C. *Future Work*

We foresee several possible extensions of our work. Firstly, our verification approach implicitly assumes that our real-time operating system is capable to timely handle the incoming device interrupts. We have proven 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 [24].

Secondly, Cvm is currently specified on an abstract level while its correctness proof is carried out on a lower level. The property transfer between Simpl and the small-step layer used for Cvm correctness has been done before [25], demonstrating the general applicability of our approach.

Furthermore, the model stack can be extended in the opposite direction: the applications are currently modeled on assembly level while usually written in C. Using Leinenbach and Petrova's [26], [27] theorem on compiler correctness, it is possible to provide a more abstract programming model for applications and their interaction with the operating system. A similar approach [28] has been taken for the Vamos microkernel, already.

Finally, 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 [29] and a comparatively small enhancement of Isabelle/Simpl.

Acknowledgment

We thank Burkhart Wolff for constructive criticism and helpful suggestions as well as the anonymous peer reviewers for a very detailed review report.

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

[2] There is also a successor project, Verisoft XT, see http://verisoftxt.de/.

References

[1] H. Kopetz and G. Grünsteidl, "TTP – A protocol for fault-tolerant real-time systems," *IEEE Computer*, vol. 27, no. 1, pp. 14–23, 1994.

[2] J. S. Moore, "A grand challenge proposal for formal methods: A verified stack," in *10th Anniversary Colloquium of UNU/IIST*. Springer, 2002, pp. 161–172.

[3] G. Klein, "Operating system verification — an overview," *Sādhanā*, vol. 34, no. 1, pp. 27–69, Feb. 2009.

[4] S. Beyer, C. Jacobi, D. Kröning, D. Leinenbach, and W. J. Paul, "Putting it all together: Formal verification of the VAMP," *STTT*, vol. 8, no. 4-5, pp. 411–430, 2006.

[5] T. In der Rieden and A. Tsyban, "CVM – a verified framework for microkernel programmers," in *Systems Software Verification*, ser. ENTCS, vol. 217. Elsevier Science B.V., 2008, pp. 151–168.

[6] European Commission (DG Enterprise and DG Information Society), "eSafety forum: Summary report 2003," eSafety, Tech. Rep., Mar. 2003.

[7] 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*, vol. 20, no. 6, pp. 637–662, 2008.

[8] CC Consortium, *Common Criteria for Information Technology Security Evaluation, Version 3.1*, Sep. 2007.

[9] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, ser. LNCS. Springer, 2002, vol. 2283.

[10] A. Church, "A formulation of the simple theory of types," *J. Symb. Log.*, vol. 5, no. 2, pp. 56–68, 1940.

[11] N. Schirmer, "A verification environment for sequential imperative programs in Isabelle/HOL," in *LPAR*, ser. LNCS. Springer, 2005, pp. 398–414.

[12] N. W. Schirmer, "Verification of sequential imperative programs in Isabelle/HOL," Ph.D. dissertation, TU Munich, 2006.

[13] E. Alkassar, M. A. Hillebrand, D. Leinenbach, N. W. Schirmer, and A. Starostin, "The Verisoft approach to systems verification," in *Verified Software: Theories, Tools, and Experiments*, ser. LNCS, vol. 5295. Springer, 2008, pp. 209–224.

[14] E. Alkassar, M. A. Hillebrand, D. C. Leinenbach, N. W. Schirmer, A. Starostin, and A. Tsyban, "Balancing the load – leveraging a semantics stack for systems verification," *J. Autom. Reasoning, Special Issue on Operating System Verification*, vol. 42, no. 2-4, pp. 389–454, 2009.

[15] A. Starostin and A. Tsyban, "Correct microkernel primitives," in *Systems Software Verification*, ser. ENTCS, vol. 217. Elsevier Science B.V., 2008, pp. 169–185.

[16] W. F. Tichy, "Should computer scientists experiment more?" *IEEE Computer*, vol. 31, no. 5, pp. 32–40, 1998.

[17] B. Walker, R. Kemmerer, and G. Popek, "Specification and verification of the UCLA Unix security kernel," *Commun. ACM*, vol. 23, no. 2, pp. 118–131, 1980.

[18] H. Tews, "Formal methods in the Robin project: Specification and verification of the Nova microhypervisor," in *C/C++ Verification Workshop, Tech. Rep. ICIS–R07015*. Radboud University Nijmegen, Jun. 2007, pp. 59–68.

[19] C. L. Heitmeyer, M. Archer, E. I. Leonard, and J. 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*. ACM, 2006, pp. 346–355.

[20] X. Feng, Z. Shao, Y. Dong, and Y. Guo, "Certifying low-level programs with hardware interrupts and preemptive threads," in *PLDI*. ACM, 2008, pp. 170–182.

[21] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "seL4: Formal verification of an OS kernel," in *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*. Big Sky, MT, USA: ACM, Oct 2009.

[22] M. A. Hillebrand and W. J. Paul, "On the architecture of system verification environments," in *Haifa Verification Conference*. Springer, 2007, pp. 153–168.

[23] M. Daum, J. Dörrenbächer, and B. Wolff, "Proving fairness and implementation correctness of a microkernel scheduler," *J. Autom. Reasoning, Special Issue on Operating System Verification*, vol. 42, no. 2-4, pp. 349–388, 2009.

[24] S. Knapp and W. Paul, "Realistic worst case execution time analysis in the context of pervasive system verification," in *Program Analysis and Compilation, Theory and Practice*, ser. LNCS. Springer, 2007, vol. 4444, pp. 53–81.

[25] E. Alkassar, N. Schirmer, and A. Starostin, "Formal pervasive verification of a paging mechanism," in *TACAS*, ser. LNCS, vol. 4963. Springer, 2008, pp. 109–123.

[26] D. Leinenbach, W. J. Paul, and E. Petrova, "Towards the formal verification of a C0 compiler: Code generation and implementation correctness," in *SEFM*. IEEE Computer Society, 2005, pp. 2–12.

[27] D. Leinenbach and E. Petrova, "Pervasive compiler verification: From verified programs to verified systems," in *Systems Software Verification*, ser. ENTCS, vol. 217. Elsevier Science B.V., 2008, pp. 23–40.

[28] M. Daum, J. Dörrenbächer, B. Wolff, and M. Schmidt, "A verification approach for system-level concurrent programs," in *Verified Software: Theories, Tools, and Experiments*, ser. LNCS, vol. 5295. Springer, Oct. 2008, pp. 161–176.

[29] F. Haftmann and M. Wenzel, "Local theory specifications in Isabelle/Isar," in *Types for Proofs and Programs*, ser. LNCS, S. Berardi, F. Damiani, and U. de'Liguoro, Eds., vol. 5497. Springer, 2009, pp. 153–168.