Towards the Formal Verification of Lower System Layers in Automotive Systems

Sven Beyer * Peter Böhm Michael Gerke Mark Hillebrand * Tom In der Rieden *
Steffen Knapp Dirk Leinenbach * Wolfgang J. Paul

Saarland University, Dept. of Computer Science, 66123 Saarbrücken, Germany
E-mail: wjp@cs.uni-sb.de

Abstract

The mission of the Verisoft project is (i) to develop techniques, which permit the pervasive formal verification of computer systems comprising hardware, system software, communication systems, and applications, (ii) to apply these techniques in an industrial context to verify prototypical systems. One such application is an emergency call, which is automatically placed on the mobile phone net after the sensors of a car have detected that it was involved in a crash. The application runs on a system of several electronic control units (ECUs). The local application programs of the ECUs run on top of a simple real time operating system kernel like described in the OSEKTime standard. ECUs are connected via a FlexRay bus. We outline the structure of an overall correctness proof for such a parallel system from the gate to the kernel level. For the communication system hardware one has to combine existing correctness proofs for components of time triggered architectures (e.g. clock synchronization) and arguments about hardware correctness into a single theorem. Results on processor, driver, and kernel correctness can to a large extent be imported from existing research in the Verisoft project. Worst case execution time bounds are derived with advanced industrial tools based on abstract interpretation.

1. Introduction

In recent years innovation in cars has come to an ever increasing extent from the use of advanced computer technology. The most innovative electronics is found in luxury cars. Unfortunately, complex computer systems have a tendency to fail from time to time, and failures in the computer system are presently *the* most frequent failures plaguing the owners of new cars [6]. Indeed luxury cars have become *less* reliable than other cars due to their more advanced—and hence more error prone—electronics systems.

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

Needless to say that industry is reacting with strong efforts to improve the reliability of car electronics. At first sight this does not look too promising: the computer systems in cars are distributed systems and the debugging of distributed systems is known to be very difficult. So increased testing alone will probably not solve the problem.

On the positive side, formal verification of distributed *algorithms* (even automatically) has been demonstrated repeatedly. Isolated correctness proofs for algorithms are not enough. They need to be embedded in a correctness proof for the entire computer system of a car comprising (i) processors, (ii) communication system and drivers, (iii) a real time operating system, and (iv) safety critical applications. This is a grand challenge problem in the sense of [17].

Based on results from Verisoft [26] we outline the structure of a pervasive correctness proof for all system layers below applications. We identify the main system layers in form of abstract computational models and sketch their main correctness statements. In Sect. 2 we handle serial interfaces and incorporate them in Sect. 3 into bus interfaces for a FlexRay like bus [8]. Sections 4 and 5 deal with a real time operating system and the programming model provided to the user. Section 6 surveys basic technology for worst case execution time analysis. In Sect. 7 we conclude.

2. Serial interfaces

2.1. Detailed Timing Analysis

Clocks. Each ECU ECU_u (with $u \in [1{:}n]$) has a clock signal $ck_u(t)$ with cycle time τ_u. Cycle times are assumed to be roughly equal; in each cycle clocks are allowed to drift by at most $\delta \leq 0.15\%$, i.e. $|\tau_u - \tau_v| \leq \tau_u \cdot (1 + \delta)$. Let $e_u(i)$ denote the time of the i-th rising edge of $ck_u(t)$; the interval $[e_u(i), e_u(i+1))$ is called cycle i of ECU_u. For signals $S_u(t)$ in the hardware of ECU_u we define their value S_u^i during cycle i as the value of signal S immediately before the end of cycle i when all hardware has stabilized.

As setup and hold times may be violated, copying register contents between ECUs is nontrivial in this scenario.

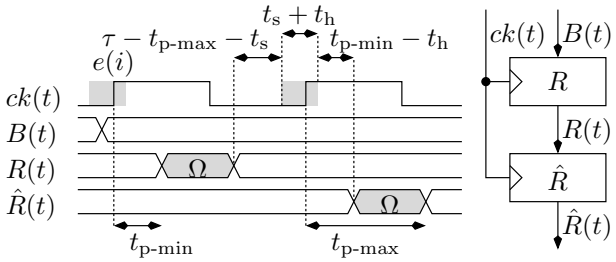

Figure 1. Sampling signals with two registers

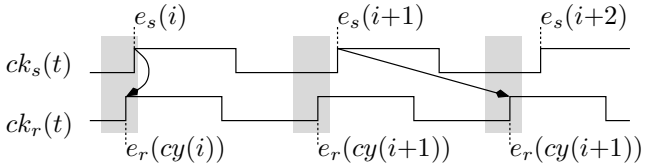

Figure 2. Sampling by the receiver

This problem is solved by *serial interfaces*. We present a serial interface in the spirit of the FlexRay standard [8].

Detailed timing. For the analysis of this interface we have to argue about *detailed timing diagrams* (cf. [13]). Formally we have to consider real valued time t. For the value $s(t)$ of a hardware signal at time t it suffices to consider values 0, 1, and Ω. Here Ω stands for voltages of the physical signal, which cannot be recognized as logical 1 or 0.

The clocking of bits into a register R is described by the detailed timing diagram of Fig. 1. We keep the input bus B at a stable value $x \in \{0,1\}$ during a setup time $t_s < \tau_s/10$ before clock edges $e(i)$ and a hold time $t_h < \tau_s/10$ after the edge: $B(t) = x$ for $t \in [e(i) - t_s, e(i) + t_h]$. This interval is called the *sampling interval* at edge $e(i)$. Then the register output R is guaranteed to show its old value y at least for a minimal propagation delay $t_{p\text{-}min}$ after the clock edge and to show the new value x at the latest after a maximal propagation delay $t_{p\text{-}max}$ after the clock edge. For $t \in [e(i) + t_{p\text{-}min}, e(i) + t_{p\text{-}max}]$ we have $R(t) = \Omega$.

Meta stability. It is extremely likely that the register output has a well defined digital value $z \in \{0,1\}$ after $t_{p\text{-}max}$. There is however a small likelihood that the register becomes *meta stable* in which case we can have $R(t) = \Omega$ for certain $t \geq e(i) + t_{p\text{-}max}$. This cannot be avoided [15]. The standard remedy is to pipeline the output of R into a second register $\hat{R}$ at the next clock edge of the same clock (cf. Fig. 1 where $B(t)$ violates the setup time). It is considered practically impossible that the second register turns meta stable, too. Thus we *always* assume $\hat{R}(e(i+1)+t_{p\text{-}max}) \in \{0,1\}$.

Bus connection. The receiver ECU_r is connected to the bus as described before. The sender ECU_s is connected via an edge triggered flip flop and an open collector driver, which is only enabled during transmission. The bus has value 1 if all drivers connected to it are disabled. The flip flop is only clocked if the sender puts a new logical value on the bus. This avoids spikes on the bus when the sender repeatedly sends the same bit. We assume the propagation delay from the sender to the receiver to be in $[0, \tau_s/2)$.

Assume the sender puts a new value x on the bus at edge $e_s(i)$. Consider the last receiver edge $e_r(j)$ whose sampling interval ends before $e_s(i)$, i.e. $j = \max\{j \mid e_r(j) + t_h < e_s(i)\}$. Sampling at $e_r(j)$ is not affected by the new value, but sampling at $e_r(j)+1$ is. We define $cy(i) = j+1$ as the index of this following edge. Note that $e_r(cy(i)) < e_s(i)$ is possible (e.g. $e_r(cy(i)) = e_s(i) - t_h/2$). Thus we cannot conclude $R_r^{cy(i)} = x$. However, the following lemmas hold.

Lemma 2.1 *If the sender transmits x in cycles i to $i+7$, then the receiver samples during at least 7 consecutive cycles the correct value x: $\hat{R}_r^{cy(i)+k+1} = R_r^{cy(i)+k} = x$ for $k \in [\beta{:}\beta+6]$. The value of β depends on the difference between sender and receiver clock and is 0 or 1.*

Proof. The sampling intervals of all receiver edges $cy(i)+k$ are in a region of time where the bus is stable.

Lemma 2.2 *If the sender transmits $\neg x$ in cycles $i-8$ to $i-1$ and x in cycles i to $i+7$, then for the cycle i' in which x occurs for the first time in $\hat{R}_r$ holds: $i' \in cy(i)+[0{:}1]+1$.*

Proof. Because clock drift is bounded to δ we know that for adjacent intervals of Lemma 2.1 the value of β is the same.

Usually, we have $cy(i+1) = cy(i) + 1$. But if τ_r is greater than τ_s we can have $cy(i+1) = cy(i)$ (as in Fig. 2) and $cy(i+1) = cy(i) + 2$ if it is less. This can happen at most once in $1/\delta > 600$ cycles, so we can show:

Lemma 2.3 *If $k < 600$ then $cy(i+k) \in cy(i)+k+[-1{:}1]$.*

2.2. Message assembly

Due to clock drift a nontrivial protocol is required to transmit via serial interfaces. We define the transmission start sequence $TSS = 0$, the frame start sequence $FSS = 1$, the byte start sequence $BSS = 10$, and the frame end sequence $FES = 01$. With $x[0{:}L-1]$ we denote a bit string consisting of L bits $x[i]$ and with $\circ$ bit string concatenation.

A message $m = m[0{:}\ell-1]$ consisting of ℓ bytes $m[i]$ is embedded into a message frame $f(m) \in \{0,1\}^{l'}$ (with $l' = 4 + 10 \cdot \ell$), which is defined by: $f(m) = TSS \circ FSS \circ BSS \circ m[0] \circ \cdots \circ BSS \circ m[\ell-1] \circ FES$. In the bit pattern $F(m)$ transmitted by the sender every bit of $f(m)$ is repeated eight times; thus it consists of $L = 8 \cdot l'$

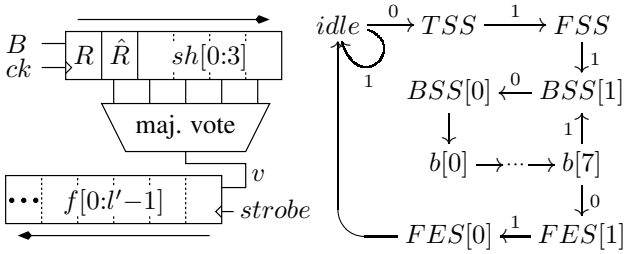

Figure 3. Receiver data paths and automaton

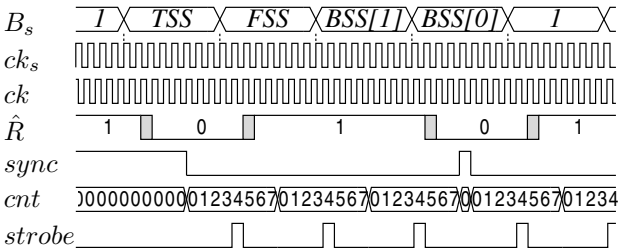

Figure 4. Local clock synchronization timing

bits. We number sender cycles such that $B_s^i = F(m)[i]$ for $i \in [0{:}L-1]$. For $i \geq L$ we have $B_s^i = 1$ due to the open collector drivers. Lemma 2.1 entails the following lemma.

Lemma 2.4 *For all $f(m)[i]$ there is a $\beta \in \{0,1\}$ such that for $k \in [\beta{:}\beta+6]$ the bit $f(m)[i]$ is correctly sampled in cycle $cy(8 \cdot i)+k$: $\hat{R}_r^{cy(8\cdot i)+k+1} = R_r^{cy(8\cdot i)+k} = f(m)[i]$.*

2.3. Message decoding

For message decoding three things are done *simultaneously*: (i) The bits sampled on the bus are filtered by a majority vote (Fig. 3). This is supposed to compensate for missed setup and hold times and for certain spikes (we ignore spikes in this paper). (ii) At certain strobe points the voted signal is clocked into a shift register f which reconstructs $f(m)$. (iii) Strobe points are mostly 8 receiver clock ticks apart. Their position is adjusted by a *low level clock synchronization* to compensate for clock drift.

Hardware. We store the last four values of $\hat{R}$ in a shift register $sh[0{:}3]$. After reset we want $sh = 1^4$ and $\hat{R} = 1$. By a majority vote we obtain the voted signal v^j. The voted signal matches each message bit during at least 7 cycles.

Lemma 2.5 *For all $f(m)[i]$ there is a $\beta \in \{0,1\}$ such that for $k \in [\beta+2{:}\beta+8]$ it holds: $v^{cy(8\cdot i)+k+1} = f(m)[i]$.*

Proof. For $k \in [\beta{:}\beta+6]$ Lemma 2.4 entails that in cycles $cy(8 \cdot i)+k+1$ we correctly receive $f(m)[i]$ in $\hat{R}$. Thus, for $k \in [\beta+2{:}\beta+8]$ at least three copies of $f(m)[i]$ are in $\hat{R}$ and $sh[0{:}3]$ and we have $v^{cy(8\cdot i)+k+1} = f(m)[i]$.

A 3-bit counter cnt is increased (modulo 8) every cycle unless it is reset by the synchronization signal $sync$. A signal $strobe$ is activated when $cnt = 4$ (cf. Fig. 4). Assume that after reset the bus is idle and we start with an empty reconstructed frame $\hat{f}^0$. In every cycle t with $strobe^j$, we sample the voted signal v^t and append it to the portion of the frame $\hat{f}^t$ reconstructed so far.

The automaton in Fig. 3 keeps track of the bits the receiver expects. In states $x \notin \{idle, b[i]\}$, the automaton is meant to sample bit x of the message frame $f(m)$; in states $b[i]$ it is meant to sample a bit $m[x][i]$ of the message. The transition function Δ of the automaton induces a transition function of a hardware automaton with state $state$. After reset the hardware automaton is in state $idle$. State transitions only occur in reaction to an active strobe signal. Thus, $state^{t+1} = \Delta(state^t, v^t)$ if $strobe^t$ and $state^{t+1} = state^t$ otherwise. We activate the $sync$ signal in all cycles j when a falling edge on $\hat{R}^j$ is expected and observed: $sync^j = v^{j-1} \wedge ((state^j = idle) \vee (state^j = BSS[1]) \wedge \neg v^j)$.

Correctness. Let $str(h)$ denote the cycle of activation $(h+1)$ of the strobe signal and $sy(h)$ the (last) cycle of activation $(h+1)$ of the sync signal. Let $nb(h)$ be the number of bits of $f(m)$ sent in synchronization interval $[sy(h){:}sy(h+1)]$ and set $NB(h) = \sum_{h'<h} nb(h')$. Because $nb(h) \cdot 8 \ll 600$, sender and receiver clock may drift by at most one cycle in one synchronization interval which implies $|sy(h+1) - sy(h) + 8 \cdot nb(h)| \leq 1$.

In the first interval, TSS, FSS, and $BSS[1]$ are meant to be sent, in the next $\ell - 1$ intervals, $BSS[0]$, eight bits of m, and $BSS[1]$ are meant to be sent, and in interval ℓ, $BSS[0]$, eight bits of m, and $FES[1{:}0]$ are meant to be sent. Hence, we define $nb(h)$ by $nb(0) = 3$, by $nb(h) = 10$ for $h \in [1{:}\ell-1]$, and by $nb(\ell) = 11$.

Inside synchronization interval $[sy(h){:}sy(h+1)]$, a message is sampled in each cycle $sy(h) + 8 \cdot y + 4$ for $y \in [0{:}nb(h)-1]$. After transmission of $NB(h)$ bits, the receiver is meant to synchronize on the sender, which, by definition of $F(m)$, transmits a falling edge in cycles $8 \cdot NB(h)$, i.e., $B_s^{8\cdot NB(h)-1} \wedge \neg B_s^{8\cdot NB(h)}$. Assuming that sender cycles $NB(h)$ and corresponding receiver cycles are not too far apart we can show the following lemma.

Lemma 2.6 *Let $k' = NB(h')$. If h' maximal such that $str(k) = sy(h') + 8 \cdot (k-k') + 4$ and if $sy(h') \in cy(8 \cdot k') + [2{:}3] + 1$, then (i) $v^{str(k)} = f(m)[k]$ and (ii) $str(k) + 1 < cy(8 \cdot (k+1)) + [2{:}3] + 1$.*

Proof. Using the assumption and Lemma 2.3 we conclude $str(k) = sy(h') + 8 \cdot (k-k') + 4 \in cy(8 \cdot k) + [5{:}8] + 1$. Then Lemma 2.5 implies Part (i): $v^{str(k)} = f(m)[k]$.

We show Part (ii) using the assumption and Lemma 2.3: $str(k)+1 \in cy(8 \cdot k') + 8 \cdot (k-k') + [8{:}9] \in cy(8 \cdot (k' + k - k' + 1)) + [-1{:}2] < cy(8 \cdot (k+1)) + [2{:}3] + 1$.

Next, we proof the main technical lemma that states that (i) the automaton correctly keeps track of the expected bits, (ii) the message bits are correctly sampled, (iii) transitions of the automaton occur fast enough, i.e. before the next bit can be sampled, and (iv) sync and strobe signals are activated at the expected times.

Lemma 2.7 *For any receiver cycle j, for any $k = NB(h') + k'$ with $str(k) \leq j$ and $k' \in [0{:}nb(h') - 1]$, and for any h with $sy(h) \leq j$ it holds: (i) If $j \in [str(k) + 1{:}str(k+1)]$ then the automaton is in a state $state^j$ as expected (cf. Fig. 3). In particular we have $state^j = BSS[1]$ if $h' < \ell$ and $k' = nb(h') - 1$. (ii) The sampled signals satisfy $v^{str(k)} = f(m)[k]$. (iii) $str(k)+1 < cy(8 \cdot (k+1)) + [2{:}3] + 1$, (iv) $sy(h) \in cy(8 \cdot NB(h)) + [2{:}3] + 1$, and (v) $str(k) = sy(h') + 8 \cdot (k - NB(h')) + 4$.*

Proof. In the induction step one concludes Parts (i), (ii), and (iii) of the induction hypothesis from Lemma 2.6; Parts (iv) and (v) are needed to show the hypotheses of Lemma 2.6.

To conclude Part (iv) we have to show that (i) the falling edge, which triggers the sync signal $sy(h)$ is seen by the receiver in the right cycle j and (ii) the automaton is in state $BSS[1]$ in cycle j. Lemma 2.2 combined with Lemmas 2.4 and 2.5 shows that the falling edge, which triggers $sy(h)$ is seen in v^j for $j \in cy(r; s, 8 \cdot NB(k)) + [2{:}3] + 1$. This proves the first subgoal. Part (i) implies $state^j = BSS[1]$ if $k' = nb(h') - 1$; outside these time intervals the sync signal cannot become active. Part (iii) implies $str(k)+1 < cy(r; s, 8 \cdot (NB(h') + k' + 1)) + [2{:}3] + 1$, thus the automaton is in state $BSS[1]$ one cycle before the first zero of $BSS[0]$ can possibly be sampled. This proves the second subgoal and completes Part (iv).

For $k' > 0$ Part (v) is obvious. For $k' = 0$ we know from above that $sy(h') \in sy(h'-1) + 8 \cdot (nb(h'-1) - 1) + [7{:}9]$. The induction hypothesis implies $str(k-1) = sy(h'-1) + 8 \cdot (nb(h'-1) - 1) + 4$. Thus $str(k-1)$ is definitely before $sy(h')$ and there is no additional strobe between them.

Lemma 2.7 allows to show:

Theorem 2.8 $\hat{f}^{\lceil (1+\delta) \cdot L \rceil + 8} = f(m)$.

3. Bus interface

Now we show how to integrate the results of Sect. 2 into a bus interface for a FlexRay like bus. As we have omitted all features regarding fault tolerance, the interface does not comply with the standard. However, we will show which theorems become more complex in a fault tolerant design.

The interface is connected to a bus via a serial interface as described in Sect. 2. The interface is capable of sending messages m from a word addressable send buffer sb to the bus or of storing messages from the bus into a word addressable receive buffer rb. Pointers sbp and rbp point into the send and receive buffers. Each ECU has a hardware timer ti_u^i, which is incremented every 8 clock ticks. Based on it we define an abstract timer $ati_u(t)$ for continuous time t by $ati_u(t) = ti_u^i$ if $t \in [e_u(i), e_u(i+1))$. Timers on different ECUs are periodically synchronized by another (high level) hardware clock synchronization algorithm, as described below. Timer interrupts for the processors come from *these* timers, which are kept synchronized.

The interface is connected to the processor's memory system by a set of I/O-ports, each 32 bits wide. *Command and status registers* are used to issue special requests and check the status of the interface.

Send and receive buffer are accessed by the processor via the *data port*. Writing to this port copies the store operand into the send buffer at address sbp and increments sbp. Reading this port returns the content of the receive buffer at address rbp and increments rbp. Hence, successive accesses to the data port fill or read out the buffer.

Configuration registers are written only during the startup phase of the system; they are considered constant here. Register u stores the number of the ECU that the interface is attached to. The others store the components of the *global bus schedule* $S = (ns, ecu, st, mlen)$.

Global communication proceeds in so called *bus rounds*, each with an identical schedule. If we have n processors and ns bus *slots* per round, then the ECU sending during slot $\sigma \in [0{:}ns-1]$ is specified by $ecu(\sigma) \in [1{:}n]$. During slot σ the sending ECU starts the transmission of a message frame at time $st(\sigma)$, i.e., if $ti_u^t = st(\sigma)$ and $ecu(\sigma) = u$. The transmitted frame contains the first $mlen(\sigma)$ bytes in the send buffer of ECU_u. Message lengths have to be bounded by the buffer size. In every slot every serial interface writes the transmitted data into its receive buffer.

This mechanism will only work if (i) a processor does not access its data port while its interface is transmitting or receiving a message and (ii) transmission time of different slots does not overlap. Condition (i) arises in similar form for all kinds of I/O devices [12]. Additional configuration registers $wakeup_u(\sigma)$ help satisfy this constraint efficiently. They serve to activate local timer interrupts at cycle t if $ti_u^t = wakeup_u(\sigma)$. Message transmission on the bus is meant to take place in the interval from $st(\sigma)$ to $wakeup_u(\sigma)$; the processor is only allowed to access the data port from $wakeup_u(\sigma)$ to $st(\sigma+1)$. Times $st(\sigma)$ and $wakeup_u(\sigma)$ must be carefully chosen to account for clock drift. High level clock synchronization and restrictions on the global bus schedule will guarantee Condition (ii).

High level clock synchronization. We do not handle fault tolerance; thus we assume that all ECUs are working and can use a simple clock synchronization algorithm. It is triggered by the transmission of the last message in each round. The sending ECU in the last slot resets its timer after it has put the last copy of $FES[0]$ on the bus. All other ECUs reset their timers 3 cycles after they sample $FES[0]$. At the beginning of slots, the automata from Sect. 2 are forced into $idle$ state.

To estimate the local time on interface v at local time T on interface u, we define $time(v; u, T)$ as the smallest value taken by the abstract timer $ati_v(t)$ for a t with $ati_u(t) = T$. After clock synchronization we have $time(v; u, 0) \in \{0, 1\}$ for all u and v. The following lemma holds.

Lemma 3.1 *For all u and v and times T the clock drift is bounded by $|time(v; u, T) - T| \leq T \cdot \delta + 2$.*

In case one deals with fault tolerance, more complex clock synchronization algorithms must be used. Correctness proofs of such algorithms [22–24] can be integrated into the theory in a similar way as the simple argument above. However, this is complicated by the fact that the interaction of the clock synchronization with membership algorithms has to be considered (cf. [23, Sect. 5]).

Start and wakeup times. We define the *intended* abstract start and end times of message transmission on the bus. The abstract start time of the first slot is set to zero. The abstract end time in slot σ is set to $et_{\mathrm{a}}(\sigma) = st_{\mathrm{a}}(\sigma)+10\cdot mlen(\sigma)+4$. Before transmission of the next message, we leave $tp(\sigma)$ timer ticks for each ECU to access the interface: $st_{\mathrm{a}}(\sigma + 1) = et_{\mathrm{a}}(\sigma) + tp(\sigma)$. On the hardware we take clock drift into account. We set $et = et_{\mathrm{a}}\cdot(1+\delta)$ and $st = st_{\mathrm{a}}\cdot(1+\delta)$.

In Sect. 6, we use wakeup timer interrupts to notify the processor that it may access the serial interface. In Lemma 3.2 we state that the interface is idle at these times if we have $wakeup_u(\sigma) \geq et(\sigma) + 3$. In Lemma 3.3 we state that message transmission is completed in these intervals and the receive buffers hold the message just transmitted. This can be shown using Theorem 2.8 and standard techniques from hardware correctness proofs [2]. In Lemma 3.4 we show that bus contention is avoided if $tp(\sigma) \geq 1$.

Lemma 3.2 *On any ECU in the timer interval $[et(\sigma) + 3{:}st(\sigma + 1)]$ the serial interface is in state $idle$.*

Lemma 3.3 *For any cycle t with $timer_u^t \in [et(\sigma) + 3{:}et(\sigma) + tp(\sigma) + 2]$ the receive buffer of ECU_u holds the message last transmitted in slot σ. Formally, for $\ell = mlen(\sigma)$ we have $rb_u^t[0{:}\ell - 1] = sb_{ecu(\sigma)}^{8\cdot st(\sigma)}[0{:}\ell - 1]$.*

Lemma 3.4 *If $tp(\sigma) \geq 1$, then transmission times of adjacent slots are separated by at least 8 cycles on any interface.*

Integration into ECUs. One obtains a verified ECU by connecting a verified bus interface to a verified processor as an I/O device. As processor we use the VAMP processor with memory management units and DLX instruction set [10, 18]. The formal verification of this processor is already complete [2, 5]. The mathematical theory needed for the integration of I/O devices is outlined in [12].

4. The generic operating system kernel CVM

A simple operating system kernel named Kit was formally verified already back in 1989 as part of the CLI stack project [1]. The complexity of this kernel is not so far away from what we need here. Unlike Kit, which was written in machine language and did not provide virtual memory simulation, the kernel, on which we base our work here is written in C and provides virtual memory simulation making use of the address translation hardware of the VAMP. Dealing formally with this requires serious machinery.

$C0$ semantics and inline assembler. Formal semantics for very large subsets of C exist [19, 21]. They are complex and the generation of correctness proofs based on these semantics seem to be difficult. In the Verisoft project, program verification is based on a subset of C called $C0$ [14], which is similar to MISRA C [16].

An operating system kernel written in C or $C0$ cannot access processor registers, user processes, or I/O devices with its own variables. Hence, if not written altogether in assembler language, implementations of operating system kernels and device drivers necessarily contain portions of inline assembler code. One can keep these portions down to a few hundred instructions. Nevertheless they are present, and in order to argue about their effect we define the semantics of inline assembler code and call the extended language $C0_A$. Because inline assembler code can change the value of $C0$ variables we need to know how the compiler allocates variables in order to define the semantics of $C0_A$ (cf. [9]).

For a compiler for common LISP, the correctness of both the code generation algorithm and its implementation was formally shown in the Verifix project [25]. In the Verisoft project a corresponding effort for a (nonoptimizing) $C0$ compiler is under way [14]. The target machine for this compiler is the verified VAMP processor.

CVM semantics. While the implementation of an operating system kernel necessarily contains inline assembler code, its semantics can be described without reference to inline assembler code: we make the user processes explicitly visible by the introduction of a pseudo parallel model of computation called communicating virtual machines (CVM). Configurations c_{cvm} of this model have four

components: (i) a $C0$ configuration $c_{\mathrm{cvm}}.ca$ describing the current state of the so-called *abstract kernel*, (ii) a fixed number p of DLX machine configurations $c_{\mathrm{cvm}}.vm(u)$ describing the current state of the user processes, (iii) the number $c_{\mathrm{cvm}}.cp \in [0{:}p]$ of the current process (for $c_{\mathrm{cvm}}.cp = 0$ the kernel is active), and (iv) the state $c_{\mathrm{cvm}}.d(j)$ of a fixed number of I/O devices. For our purposes here the memory size of each user process is fixed and all user memories fit into the physical memory of the processor. Thus, we have address translation in user mode but no page faults.

The definition of the next state function $\delta_{\mathrm{CVM}}(c_{\mathrm{cvm}}) = c'_{\mathrm{cvm}}$ formalizes the user manual of the kernel. Typical portions of the definition are: (i) If $c_{\mathrm{cvm}}.cp = 0$, then the kernel executes the next $C0$ statement (as prescribed by the $C0$ semantics) unless it is one of several special functions. The user processes are frozen. (ii) If $c_{\mathrm{cvm}}.cp = u > 0$ and no interrupt occurs, then user process u executes one instruction: $c'_{\mathrm{cvm}}.vm(u) = \delta_{\mathrm{DLX}}(c_{\mathrm{cvm}}.vm(u))$. The kernel and the other user processes are frozen. Thus the parallelism in the CVM model is not real, because at any time either the kernel or exactly one user is making progress.

The kernel contains some special functions whose semantics can only be defined in the parallel model. Most important are (i) copy functions permitting to copy regions of memory between user processes or between an user process and I/O devices and (ii) a function $startp()$, which suspends the kernel and activates user process u, where u is the current value of a $C0$ variable CP of the kernel.

User processes can invoke *kernel calls* by executing trap instructions. This produces an interrupt and thus turns control over to the kernel, which executes, depending on the immediate parameter of the trap instruction, a function f.

Concrete kernel and correctness. The kernel is implemented by a $C0_A$ program called the *concrete* kernel, which is obtained by linking the source code of the abstract kernel together with code for (i) the special functions (using inline assembler), (ii) for context switching and data structures for saving processor registers, and (iii) for a so called dispatcher, which, after an interrupt, calls a kernel function f that is supposed to handle the interrupt. In [9] a correctness proof for CVM is presented, which uses compiler correctness [14] and a theory for virtual memory simulation [5,11].

C0 programs and CVM. It is not hard to integrate $C0$ user programs into CVM, which yields a new model, called CCVM. In CCVM the $C0$ user programs can execute certain special functions, which allow to call kernel functions and are implemented using inline assembler code.

We have not yet a correctness proof for a simulation of CCVM by CVM. Besides compiler correctness [14] this proof will require an extra argument concerning interrupts. User $C0$ programs are either interrupted (i) by kernel calls or (ii) by external interrupts. Case (i) is safe as long as the compiler does not optimize across kernel calls. For Case (ii) extra arguments are needed as the compiled program may be interrupted in the middle of the code of a $C0$ statement.

5. Application programs

In this section we first specify the application programmer's view on an OSEKTime / FlexRay like system. Then we construct, based on the generic CVM kernel, an OSEKTime like operating system kernel [20] by using a specific scheduler and adding the kernel calls and a driver for the interface. As we go along we state the main invariants for the correctness proof. The invariants hold only if certain timing constraints are met (cf. Sect. 6).

Specification. On each ECU ECU_u runs a fixed number na_u of C0 applications $A(u,a)$ with $a \in [1{:}na_u]$. Global computation is done in *system rounds* with identical schedule. Here we assume that one system round equals a bus round. Applications $A(u,a)$ are scheduled to run for a schedule interval of $scnt(u,a)$ subsequent slots; they are assumed to signal the end of their computation for such an interval by calling the $C0$ function $ttDone$. The function $scd(u,\sigma)$ designates the application to be run on ECU_u during slot σ. The end of an interval is indicated by the predicate $lsl(u,\sigma) = (scd(u,\sigma) \neq scd(u,\sigma+1))$. The functions scd and $scnt$ must correspond in a natural way.

Applications communicate via a fixed number nsv of shared variables $\hat{V}[x]$. They are allowed to sample the value of *any* variable $\hat{V}[x]$ into a $C0$ variable Y and to update a *single* shared variable with index $z = sv(u,a)$. This is done via $C0$ functions $ttRcv(x,Y)$ and $ttSnd(z,Y)$, resp. Updates are *delayed* and only visible two slots after the last slot of the schedule interval of the updating application. Only one shared variable may be updated in a given slot; thus at most one application may have its last slot in a given slot. Here, we assume that in each slot σ *exactly* one application $A(u,a)$ ends its schedule interval; we denote this 'sending' application by $SA(\sigma) = (u,a)$.

For applications $A(u,a)$ with $scnt(u,a) > 1$ we still might have nondeterminism at the implementation level: it might be hard to predict whether a call $ttRcv(x,Y)$ is near the end of slot σ or near the beginning of slot $\sigma+1$. As we aim for a deterministic communication model, we impose additional schedule restrictions. If $[\sigma'{:}\sigma]$ is a schedule interval for an application, which samples variable $\hat{V}[x]$, then for all $\sigma'' \in [\sigma'-1{:}\sigma-2]$ we require $sv(SA(\sigma'')) \neq x$. For $scnt(u,a) = 1$, this condition trivially holds.

For the formal definition of the semantics of our model, we introduce an intermediate set $V[x]$ of shared variables. In slot σ, the result of running application $SA(\sigma)$ for its

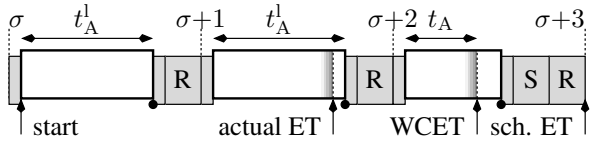

Figure 5. Schedule interval of an application

schedule interval is computed. Updates to the shared variable x in cycle σ take effect for $V[x]$ in cycle $\sigma+1$ and for $\hat{V}[x]$ in cycle $\sigma+2$. We set $\hat{V}[x]^{\sigma+1} = V[x]^{\sigma}$.

The paper [3] reports about correctness proofs of applications in a model very similar to the one we just explained.

Implementation. On each ECU copies of the shared variables are locally maintained in $C0$ variables $\hat{V}[x]_u$ of the kernel. A set of intermediate copies $V[x]_u$ is also maintained for technical reasons. The functions $ttSnd(x, Y)$ and $ttRcv(x, Y)$ are realized as kernel calls, they copy the current value of Y into $V[x]_u$ or the current value of $\hat{V}[x]_u$ into Y, resp. The function $ttDone$ is implemented with a kernel call that passes control back to the kernel, which then schedules a special idle process. The kernel includes a function $R(x)$, which copies the content of the receive buffer rb_u to $\hat{V}[x]_u$ and a function $S(x)$, which copies the variable $V[x]$ for $x = sv(u, scd(u, \sigma))$ to the send buffer. The functions $R(x)$ and $S(x)$ form the essential part of the device driver for the interface. For details about (pervasive) correctness proofs of device drivers see [12]. Apart from being called by the application, the kernel is also entered if the $wakeup$ timers described in Sect. 3 trigger. In its last slot an application has to call $ttDone$ before the timer triggers (cf. 6).

Fig. 5 depicts the execution of an application over a schedule interval of length 3 starting in slot σ. The white boxes indicate application execution, the shaded boxed indicate kernel execution. Kernel execution starts and ends with context switches, indicated by the slim shaded boxes. Timer interrupts are indicated with small black circles. For all slots but the last in a schedule interval, the kernel is entered to execute $R(x)$ for certain x. In the last slot σ of a schedule interval the interrupt is caused earlier and the kernel additionally executes $S(sv(u, scd(u, \sigma)))$.

Let $TA(\sigma)$ denote a time interval in which $A(u, scd(u, \sigma))$ is scheduled to run, $TS(\sigma)$ an interval in which the send function $S(x)$ is called on ECU_u, i.e. $lsl(\sigma)$ and $x = sv(u, scd(u, \sigma))$, and $TR(\sigma+1)$ an interval in which the receive function $R(x)$ is called on ECU_u, i.e. $x = sv(SA(\sigma))$. Three invariants hold: (i) While an application is executed, i.e. $ti^t_u \in TA(\sigma)$, we have $\hat{V}[x]^t_u = \hat{V}[x]^{\sigma}$ for all x. (ii) If $ti^t_u \in TS(\sigma)$ we have, by the implementation of $ttSnd(x, Y)$, that $V[x]^t_u$ holds the last value written by $A(u, scd(u, \sigma))$ into $\hat{V}[x]$. In the specification, this will be the $(\sigma+1)$-value of $V[x]$ or the $(\sigma+2)$-value of $\hat{V}[x]$. Thus, $V[x]^t_u = V[x]^{\sigma+1} = \hat{V}[x]^{\sigma+2}$. By implementation of $S(x)$, the same holds for the last cycle t such that $ti^t_u \in TS(\sigma)$ for the send buffer: $sb^t_u = V[x]^{\sigma+1} = \hat{V}[x]^{\sigma+2}$. (iii) If $TR(\sigma)$ does start only after the transmission of the message on the bus is complete and if $ti^t_u = TS(\sigma+2)$, then we find the transferred message in all receive buffers, $rb^t_u = \hat{V}[x]^{\sigma+2}$ (cf. Theorem 2.8). By the implementation of $R(x)$ we can show Invariant (i) for slot $\sigma+2$.

6. Worst case execution times and scheduling

Worst case execution times. We want to talk about worst case execution times (WCET) of compiled programs on pipelined machines. We say that a sequence of DLX instructions starts in hardware cycle t_i, when the first instruction is in the issue stage and it completes in the cycle t_c, when the last instruction is in the write back stage. The execution time measured in timer ticks is then defined as $T_c - T_s + 1$ for $T_c = ti^{t_c}$ and $T_i = ti^{t_i}$.

Obtaining precise estimates for WCET of programs running on complex processors is highly nontrivial because one has to carefully analyze the pipeline structure and one has to be able to predict cache hits and misses very precisely. The company AbsInt is a member of the Verisoft consortium and is marketing tools for exactly this purpose [7]. These tools are based on the theory of abstract interpretation [4] and, once adapted to a processor, permit to obtain amazingly accurate estimates with a very high degree of automation.

We denote (i) by γ an upper bound of the WCET of a context switch, (ii) by t_{S} and t_{R} upper bounds of the WCET of function S and R, resp., (iii) by t_{misc} an upper bound of the WCET for miscellaneous kernel computations, and (iv) by t_{kc} an upper bound of the WCET of kernel calls. According to Sect. 5 and Fig. 5, we have an upper bound of the WCET for an application: $WCET(u, a) = (scnt(u, a) - 1) \cdot t^{\mathrm{l}}_{\mathrm{A}} + t_{\mathrm{A}}$. In this setting, $t^{\mathrm{l}}_{\mathrm{A}} = t_{\mathrm{A}} + t_{\mathrm{S}}$.

Satisfying timing constraints. Recall that a global bus schedule was defined as $S = (ns, ecu, st, mlen)$. Here, we consider the number of slots ns, the length $mlen(\sigma) = mlen$ of each message, and the times $tp(\sigma) = tp$ reserved for interface access (cf. Sect. 3) fixed. The sending ECU in slot σ corresponds to that of the sending application in the preceding slot $\sigma-1$. For $(u, a) = SA(\sigma-1)$, we set $ecu(\sigma) = u$. Let $l' = 10 \cdot mlen + 4$. Then, we can derive closed formulas for the start and end times of message transmission, $st(\sigma) = (1+\delta) \cdot ((l'+tp) \cdot \sigma)$ and $et(\sigma) = (1+\delta) \cdot ((l'+tp) \cdot \sigma + l')$; thus $et(\sigma) - st(\sigma) = (1+\delta) \cdot l'$.

To guarantee that the kernel does not access the serial interface during the message transmission on the bus we require $t_{\mathrm{A}} + 2 \cdot \gamma + t_{\mathrm{misc}} \geq (1+\delta) \cdot l' + 3$. The kernel runs noninterruptibly. Hence, if an application performs a

kernel call just before a timer interrupt, the delivery of that interrupt is delayed by the execution time of that kernel call, which is bounded by t_{kc}. To allow the kernel the execution of $S(x)$ and $R(x)$ in the remainder of the slot we have to set $tp = t_{\mathrm{S}} + t_{\mathrm{R}} + t_{\mathrm{kc}}$. Finally, all slot length t_{sl} are equal; we define $t_{\mathrm{sl}} = t_{\mathrm{S}} + t_{\mathrm{R}} + 2 \cdot \gamma + t_{\mathrm{kc}} + t_{\mathrm{A}} + t_{\mathrm{misc}}$.

7. Conclusion

We have built on a unified mathematical framework for pervasive correctness proofs already existing in the Verisoft project. Arguments concerning the verification of processors, memory management units, compilers, operating system kernels, devices, and device drivers have already been formulated in a single uniform mathematical theory. We have extended this theory by three results.

1. A mathematical correctness proof for a serial interface (Sect. 2). The construction of such an interface contains a low level clock synchronization algorithm. For this proof we had to argue about detailed timing diagrams.

2. In Sect. 3 we have analyzed and used a second clock synchronization algorithm. This is the kind of clock synchronization commonly studied in the literature about distributed systems. The existing correctness proofs for fault tolerant and more complex algorithms can be integrated into the theory and the hardware at exactly the same place.

3. Building only on components, which are already verified or for which a formal verification effort is under way, we have constructed from the gate level up a parallel real time system connected by a FlexRay like bus and using an OSEKTime like operating system kernel. We have outlined a proof that this system provides to the application programmer an OSEKTime like programming model.

The system described here is far from being perfect, mainly because the underlying components we are building on were not designed for real time applications and we have not yet optimized the design. Also notation and proofs are still less polished than in established theories. On the positive side we have for the first time outlined on paper and pencil a pervasive correctness proof of an entire parallel computer system covering hardware, system software, communication mechanism, and at least the programming model for the applications. We expect to build and formally verify a similar system within the next few years.

References

[1] W. R. Bevier. Kit and the short stack. *JAR*, 5(4), 1989.

[2] S. Beyer, C. Jacobi, D. Kröning, D. Leinenbach, and W. Paul. Instantiating uninterpreted functional units and memory system: Functional verification of the VAMP. In D. Geist and E. Tronci, editors, *CHARME'03*, LNCS. Springer, 2003.

[3] J. Botaschanjan, L. Kof, C. Kühnel, and M. Spichkova. Towards verified automotive software. In *Software Engineering for Automotive Systems'05*. ACM Press, 2005.

[4] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*. ACM Press, 1977.

[5] I. Dalinger, M. Hillebrand, and W. Paul. On the verification of memory management mechanisms. In D. Borrione and W. Paul, editors, *CHARME'05*, LNCS. Springer, 2005.

[6] F. Dudenhöffer, M. Krüger, and H. Schmaler. Ausfall-Sicherheit Fahrzeug-Elektronik. Technical report, Center of Automotive Research (CAR), 2002.

[7] C. Ferdinand and R. Heckmann. Verifying timing behavior by abstract interpretation of executable code. In *CHARME'05*, LNCS. Springer, 2005.

[8] FlexRay Consortium. `http://www.flexray.com`.

[9] M. Gargano, M. Hillebrand, D. Leinenbach, and W. Paul. On the correctness of operating system kernels. In J. Hurd and T. F. Melham, editors, *TPHOLs'05*, LNCS. Springer, 2005.

[10] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1996.

[11] M. Hillebrand. *Address Spaces and Virtual Memory: Specification, Implementation, and Correctness*. PhD thesis, Saarland University, Computer Science Department, June 2005.

[12] M. Hillebrand, T. In der Rieden, and W. Paul. Dealing with I/O devices in the context of pervasive system verification. In *ICCD '05*. IEEE Computer Society, 2005. To appear.

[13] J. Keller and W. Paul. *Hardware Design*. Teubner-Texte zur Informatik. Teubner Verlag, 1995.

[14] D. Leinenbach, W. Paul, and E. Petrova. Towards the formal verification of a C0 compiler: Code generation and implementation correctness. In *SEFM'05*, 2005.

[15] R. Männer. Metastable States in Asynchronous Digital Systems: Avoidable or Unavoidable? *Microelectronics Reliability*, 28(2):295–307, 1988.

[16] The Motor Industry Software Reliability Association. *MISRA-C:2004*. MIRA, Ltd., UK, 2004.

[17] J. S. Moore. A grand challenge proposal for formal methods: A verified stack. In B. K. Aichernig and T. S. E. Maibaum, editors, *10th Anniversary Colloquium of UNU/IIST*, volume 2757 of *LNCS*, pages 161–172. Springer, 2003.

[18] S. M. Mueller and W. J. Paul. *Computer Architecture: Complexity and Correctness*. Springer, 2000.

[19] M. Norrish. *C Formalised in HOL*. PhD thesis, University of Cambridge, Computer Laboratory, Dec. 1998.

[20] OSEK/VDX time-triggered operating system, 2001. `http://www.osek-vdx.org/mirror/ttos10.pdf`.

[21] N. Papaspyrou. *A Formal Semantics for the C Programming Language*. PhD thesis, Nat. Tech. Univ. of Athens, 1998.

[22] J. Rushby. A formally verified algorithm for clock synchronization under a hybrid fault model. In *Principles of distributed computing'94*, pages 304–313. ACM Press, 1994.

[23] J. Rushby. An overview of formal verification for the time-triggered architecture. In W. Damm and E.-R. Olderog, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, LNCS, pages 83–105. Springer, 2002.

[24] F. B. Schneider. Understanding protocols for byzantine clock synchronization. Technical report, 1987.

[25] The VERIFIX Project. `http://www.verifix.de/`.

[26] The Verisoft Project. `http://www.verisoft.de/`.