Pervasive Verification of Distributed Real Time Systems

Steffen Knapp* and Wolfgang Paul

Saarland University, Computer Science Dept., 66123 Saarbrücken, Germany
{sknapp, wjp}@wjpserver.cs.uni-sb.de

Abstract. In these lecture notes we outline for the first time in a single place a correctness proof for a distributed real time system from the gate level to the computational model of a CASE tool.

1 Introduction and Overview

The mission of the German Verisoft project [The05a] is (i) to develop tools and methods permitting the pervasive formal verification of entire computer systems including hardware, system software, communication systems and applications (ii) to demonstrate these methods and tools with examples of industrial complexity. In the automotive subproject the following distributed real time system is considered. The hardware consists of ECU's connected by a FlexRay [Fle] like bus. The ECU's comprise a VAMP processor [BJK+03,DHP05] and a FlexRay like interface. System software is a C0 compiler [LPP05] and an OSEKtime [OSE01b] like operating system OLOS [Kna05] realized as a dialect of the generic operating system kernel CVM [GHLP05]. Applications are compiled C0 programs communicating via an FTCom [OSE01a] like data structure. They are generated by a variant of the AutoFocus CASE tool; the computational model underlying this tool is a variant of communication automata. A pervasive correctness proof for this system was presented in the lectures of the second author at the summer school on 'Software System Reliability and Security' 2006 in Marktoberdorf. This survey paper contains the lecture notes.

In Section 2 we outline the specification of a DLX instruction set [HP96,MP00] including the handling of interrupts.

Using the VAMP processor [BJK+03] as an example we explain in Section 3 how the hardware design of complex processors with internal and external interrupts is verified. The resulting correctness proofs are based on the scheduling functions from [SH98,MP00].

Section 4 deals with a generic device theory. We show how to specify devices and how these specifications can be integrated into the instruction set architecture of a processor.

In Section 5 we extend our VAMP processor with memory management units (MMUs). This gives hardware support for multi processing operating system kernels and for virtual machine simulation [DHP05][1].

* 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.

[1] In real time systems one has of course to run virtual machine simulations in a restricted way such that no page faults occur.

In Section 6 we survey a formal correctness proof for a compiler from the C0 programming language [LPP05,**?**,**?**] to the DLX instruction set. In a nutshell C0 is PASCAL [HW73] with C syntax.

In Section 7 we extend the C0 language. We permit portions of in line assembler code and call the resulting language $C0_A$. Using the allocation function of the compiler from Section 6 we can define the semantics of $C0_A$ programs in a natural way.

In Section 8 we describe the semantics of the generic operating system kernel CVM (communicating virtual machines) [GHLP05]. The programmer sees a so called *abstract kernel* and a set of user processes. The user processes are virtual DLX machines. The abstract kernel is a C0 program; it can make use of certain so called CVM primitives which allow to transport data between the kernel and the user processes. The semantics of these primitives can be specified in the parallel user model.

The implementation of a CVM kernel requires to link some extra code to the abstract kernel as described in Section 9. This results in a so called *concrete kernel*. The concrete kernel contains necessarily in line assembler code, because machine registers and user processes are simply not visible in C0 variables alone. The correctness proof hinges on the virtual machine simulation from Section 3, the compiler correctness proof form Section 4 and on the in line assembler semantics from Section 5.

Next we would like to instantiate the abstract CVM kernel with a concrete OSEK-time like operating system kernel called OLOS [Kna05]. User processes running under OLOS will be C0 programs communicating via FTCom like message buffers with the local operating system *and* with processes running on remote processors. While the machinery available at the end of Section 9 permits effortlessly to define the application programmers model, for a pervasive correctness proof of the entire distributed system we lack an important ingredient: a correctness proof for a FlexRay like communication system between processors. Providing that ingredient takes us TODO more sections:

Because the ECUs are running with local oscillators of almost but not exactly equal clock frequency, one cannot guarantee that set up and hold times of registers are respected when one transmits data between ECUs. In such situations one uses serial interfaces; in Section 11 we review a correctness proof for a serial interface from [BBG$^+$05].

In Section 12 we construct I/O devices called *f-interfaces*, consisting among other things of message buffers, serial interfaces, and local timers. An ECU will consist of a processor together with such an interface. In time triggered protocols like FlexRay, communication between ECUs is in fixed time slots, in the simplest case via a single bus. In each time slot one ECU is allowed to broadcast one of its message buffers and the other ECU's must remain quiet. This only works, if local timers on the ECUs are kept roughly synchronized. The implementation and correctness proof of a non fault tolerant clock synchronization algorithm - built on top of the serial interfaces of Section 11 - is therefore part of Section 12. Extension of this section to the fault tolerant case is future work and has two parts: (i) clock synchronization in the fault tolerant case; this is an extremely well studied problem [Sch87,Rus94] (ii) a startup algorithm for the fault tolerant case. In view of results reported in [SK06], this might require some modifications in the start up algorithm from the FlexRay standard.

In Section 13 we use techniques from [HIP05] to integrate the f-interfaces with the ISA (instruction set architecture) model of the processor. Due to the (external) timer

interrupts we run into a problem which is both surprising and not so easy to overcome: timer interrupts occur in fixed time intervals. It is trivial to determine on the hardware level, in which cycle such an interrupt occurs. We have to define on the ISA level the corresponding instruction which gets interrupted. This can inherently not be done on the ISA level alone: the execution time of an instruction depends on cache hits and cache misses, but the memory hierarchy is invisible on the ISA level. On the pure ISA level we end up with a nondeterministic model of computation. The nondeterminism is formalized by oracle inputs indicating for each instruction, whether it is interrupted by a timer interrupt or not.

In Section 13 we revisit processor correctness proofs, this time with an f-interface as I/O device. The oracle inputs are determined as a byproduct of the processor correctness proof. This is intuitively plausible: if one is allowed to look inside the hardware at the register transfer language (RTL) level, then the occurrence of timer interrupts becomes deterministic. Technically we achieve this with the help of the scheduling functions introduced in Section 3.

In Section 14 we show how to combine classical program correctness proofs (on the ISA level), worst case execution time (WCET) analysis on the RTL level and hardware correctness proofs into pervasive correctness proofs for real time system from the gate level to the ISA level. The results of Sections 11 to 14 are from [KP06].

In Section **??** we define the distributed OLOS model (D-OLOS) from [Kna05]: a multi processing real time operating system OLOS is running on every ECU of the distributed system. User processes are compiled C0 programs. Using operating system calls they can communicate among each other by accessing an FTCom like data structure on their local ECU. A pervasive correctness proof for the implementation of D-OLOS outlined in section 16 hinges on the correctness of the CVM implementation from Section 8, the compiler correctness from Section 6 and the results from Section 14.

Optional: In Section **??** we introduce the automat theorec computational model of a CASE tool called AutoFocus Task Model (AFTM). Based on results from [**?**] we show in section 18 we show how to simulate this model by D-OLOS.

2 Specifying an Instruction Set Architecture (ISA)

2.1 Notation

For bit strings $a = a[n-1:0] \in \{0,1\}^n$ we denote by

$$\langle a \rangle = \sum_{i=0}^{n} a_i \cdot 2^i$$

the natural number with binary representation a. For numbers $x \in \{0, \ldots, 2^n - 1\}$ the binary representation of x of length n is the bit string $bin_n(x) \in \{0,1\}^n$ satisfying:

$$\langle bin_n(x) \rangle = x$$

The n bit binary addition function $+_n : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}^n$ is defineded by:

$$a +_n b = bin_n(\langle a \rangle + \langle b \rangle \bmod 2^n)$$

For bits x and natural numbers n we define x^n as the string obtained by concatenating x exactly n times with itself

$$x^n = x \circ \ldots \circ x$$

2.2 Configurations and Auxiliary Concepts

We outline how the DLX instruction set architecture (ISA) is formally specified. Processor configurations d have the following components:

1. $d.R \in \{0,1\}^{32}$ stores the current value of register R. For this paper, the most relevant registers are: the program counter pc, the delayed PC[2] dpc, the general purpose registers $gpr[x]$ with $x \in \{0,1\}^5$ and the status register sr containing the mask bits for the interrupts.
2. The byte addressable memory $d.m : A \to \{0,1\}^8$ where the set of addresses $A \subset \{0,1\}^{32}$ usually has the form $A = \{a \mid \langle a \rangle \leq d.b\}$ for some maximal available memory byte address $d.b$. The content of the memory at byte address a is given by $d.m(a)$.

The maximal available address $d.b$ does not change during an ISA computation. Therefore it is rather treated as a parameter of the model than as a component of a configuration. We will later partition memory into pages of $4K$ bytes. We assume that $d.b$ is a multiple of some page size:

$$d.b = d.ptl \cdot 4K$$

where $d.ptl$ is a mnemonic for the last index of page tables introduced later. For addresses a, memories m, and natural numbers x we denote by $m_x(a)$ the concatenation of the memory bytes from address a to address $a + x - 1$ in little endian order:

$$m_x(a) = m(a + x - 1) \circ \ldots \circ m(a)$$

The instruction executed in configuration d, denoted by $I(d)$, is the memory word addressed by the delayed PC:

$$I(d) = d.m_4(d.dpc)$$

The six high order bits of the instruction word constitute the opcode (opc):

$$opc(d) = I(d)[31:26]$$

Instruction decoding can easily be formalized by predicates on $I(d)$. In some cases it suffices to inspect the opcode only. The current instruction is for instance a 'load word' (lw) instruction if the opcode equals 100011:

$$lw(d) \leftrightarrow opc(d) = 100011$$

DLX instructions come in three instruction types as shown in Figure 1. The type of an instruction defines how the bits of the instruction outside the opcode are interpreted.

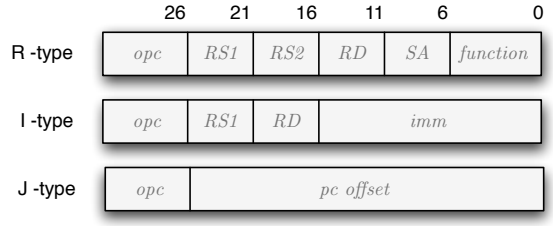

Fig. 1. Instruction Types

The occurrence of an R-type instructions, e.g. a add or a subtract instruction, is for instance specified by

$$rtype(d) \leftrightarrow opc(d) = 000000$$

Definitions of I-type and J-type instructions are slightly more complex. Depending on the instruction type, certain fields have different positions within the instruction. For the register 'destination' operand (RD) we have for instance

$$RD(d) = \begin{cases} I(d)[20:16] & itype(d) \\ I(d)[15:11] & \text{otherwise} \end{cases}$$

The effective address (ea) of load / store operations is computed as the sum of (i) the content of the register addressed by the $RS1$ field $d.gpr(RS1(d))$ and (ii) the immediate field $imm(d) = I(d)[15:0]$. The addition is performed modulo 2^{32} with two's complement arithmetic. Formally, we define the sign extension of the immediate constant as:

$$sxt(imm(d)) = imm(d)[15]^{16} \circ imm(d)$$

This turns the immediate constant into a 32-bit constant while preserving the value as a two's complement number. It is like adding leading zeros to a natural number. The effective address is defined as:

$$ea(d) = d.gpr(RS1(d)) +_{32} sxt(imm(d))$$

This definition is possible since n bit two's complement numbers and n bit binary numbers have the same value modulo 2^n. For details see e.g. Chapter 2 of [MP00].

2.3 Basic Instruction Set

With the above few preliminary definitions in place we specify the next configuration d', i.e. the configuration after execution of $I(d)$. This obviously formalizes the instruction set.

In the definition of d' we split cases depending on the instruction to be executed. As an example we specify the next configuration for a load word instruction.

[2] The delayed PC is used to specify the delayed branch mechanism detailed in [MP00].

The main effect of a load word instruction is that the general purpose register addressed by the RD field is updated with the memory word addressed by the effective address ea:

$$d'.gpr(RD(d)) = d.m_4(ea(d))$$

The PC is incremented by four in 32-bit binary arithmetic and the old PC is copied into the delayed PC:

$$\begin{aligned} d'.pc &= d.pc +_{32} bin_{32}(4) \\ d'.dpc &= d.pc \end{aligned}$$

This part of the definition is identical for all instructions except control instructions. Components which are not changed have to be specified, too:

$$\begin{aligned} d'.m &= d.m \\ d'.gpr(x) &= d.gpr(x) \qquad \text{for } x \neq RD(d) \\ d'.sr &= d.sr \end{aligned}$$

The main effect of store word instructions is that the general purpose register content addressed by RD is copied into the memory word addressed by ea

$$d'.m_4(ea(d)) = d.gpr(RD(d))$$

Completing this definition for all instructions results in the the definition of a DLX next state function:

$$d' = \delta_D(d)$$

2.4 Dealing with Interrupts

Interrupts are triggered by interrupt event signals which might be internally generated (like illegal instruction, misalignment, and overflow) or externally generated (like reset and timer interrupt). Interrupts are numbered using indices $j \in \{0, \ldots, 31\}$. We classify the set of these indices in two ways:

1. maskable / not maskable. The set of indices of maskable interrupts is denoted by M
2. external / internal. The set of indices of external interrupts is called E.

We denote external event signals by $eev[j]$ with $j \in E$ and we denote internal event signals by $iev[j]$ with $j \notin E$. We gather the external event signals into a vector eev and the internal event signals into a vector iev.

Formally these signals must be treated in a very different way. Whether an internal event signal $iev[j]$ is activated in configuration d is determined only by the configuration. For instance if we use the $j = 1$ for the illegal instruction interrupt and $LI \subset \{0,1\}^{32}$ is the set of bit patterns for which d' is defined if $I(d) \in LI$, then

$$iev(d)[1] \leftrightarrow I(d) \notin LI$$

Thus the vector of internal event signals is a function $iev(d)$ of the current processor configuration d. In contrast, external interrupts are external inputs for the next state function. We therefore get a new next state function

$$d' = \delta_D(d, eev)$$

The cause vector ca of all event signals is a function of the processor configuration d and the external input eev:

$$ca(d, eev)[j] = \begin{cases} eev[j] & j \in E \\ iev(d) & \text{otherwise} \end{cases}$$

The masked cause vector mca is computed from ca with the help of the interrupt mask stored in the status register: if interrupt j is maskable and $sr[j] = 0$, j is masked out:

$$mca(d, eev)[j] = \begin{cases} ca(d, eev)[j] \wedge d.sr[j] & j \in M \\ ca(d, eev) & \text{otherwise} \end{cases}$$

If any one of the masked cause bits is on, the JISR (jump to interrupt service routine) bit is turned on

$$JISR(d, eev) = \bigvee_j mca(d, eev[j])$$

If this occurs, many things happen. We mention only a few: The PCs are forced to point to the start addresses of the interrupt service routine. We assume it starts at (binary) address 0:

$$d'.dpc = bin_{32}(0)$$
$$d'.pc = bin_{32}(4)$$

All maskable interrupts are masked and the masked cause register is saved into a new exception cause register

$$d'.sr = 0^{32}$$
$$d'.eca = mca(d, eev)$$

Because many interrupt lines can become active simultaneously it is important to know the smallest index of an active bit of mca. This index is called the *interrupt level* and specifies the interrupts of highest priority which is also the one which will be serviced immediately.

$$il(d, eev) = \min\{j : mca(d, eev)[j] = 1\}$$

Auxiliary data for the intended interrupt handler is stored in an exception data register $edata$. We only specify the new content for the case of trap instructions. In the DLX instruction set the trap instruction has J-type format with opcode 111110 and an j. We give the trap instruction interrupt event line 5:

$$iev(d)[5] \leftrightarrow opc(d) = 111110$$

If this event line is active and no line with hogher priority is active, then a trap interrupt occurs

$$trap(d, eev) \leftrightarrow il(d, eev) = 5$$

In case of a trap interrupt, the sign extended (26 bit) immediate constant is saved in the exception data register

$$trap(d, eev) \rightarrow d'.edata = imm(d)[25]^6 imm(d)$$

For a complete definition see Chapter 5 of [MP00].

3 Processor Correctness

3.1 Hardware Model

The processor hardware is specified in a hardware model. A hardware configurations h consists of n bit registers $h.R \in \{0,1\}^n$ and $(a \times d)$-RAMs $h.r : \{0,1\}^a \to \{0,1\}^d$. Registers and RAMs are connected by Boolean circuits with the usual semantics from switching theory. We denote the value of a signal s in configuration h by $s(h)$. The hardware transition function δ_h depends on a reset signal and other external inputs ein. It maps a hardware configuration h to the hardware configuration $h' = \delta_H(h, reset, ein)$ after the next clock cycle. One defines for a register R with clock enable signal Rce and input Rin

$$h'.R = \begin{cases} Rin(h) & Rce(h) = 1 \\ h.R & \text{otherwise} \end{cases}$$

For a RAM r with address signal $addr$, data input din and write signal w one defines

$$h'.r(x) = \begin{cases} Din(h) & x = addr(h) \wedge w(h) \\ h.r(x) & \text{otherwise} \end{cases}$$

Hardware computations are defined in the usual way as sequences of configurations $h^0, h^1, \ldots$. A superscript t in this model is always read as 'during cycle t'. Hardware computations must satisfy for all cycles t:

$$h^{t+1} = \delta(h^t, reset^t, ein^t)$$

Processor correctness theorems state, that hardware defined in this model simulates in some sense an ISA next state function δ_D as defined in the previous sections.

3.2 Scheduling Functions

The processor correctness proofs considered here hinge on the concept of scheduling functions s. The hardware of pipelined processors consists of many stages k, e.g. fetch stage, issue stage, reservation stations, reorder buffer, write back stage, etc. (see Figure 4). Stages can be full or empty due to pipeline bubbles. The hardware keeps track of this with the help of full bits $full_k$ for each stage as defined in [MP00]. Recall that $full_k(h^t)$ is the value of the full bit in cycle t. We will use the shorthand $full_k^t$.

For hardware cycles t and stages k which are full during cycle t, i.e. such that $full_k^t$ holds, the value $s(k,t)$ of the scheduling function is the index i of the instruction which is in stage k during cycle t. If the stage is not full, it is the index of the instruction which was in stage k in the last cycle before t, when stage k was full. Initially $s(k,0) = 0$ for all stages k.

The formal definition of scheduling functions uses an extremely simple idea: imagine that the hardware has registers which can hold integers of arbitrary size. Augment each stage with such a register and store in it the index of the instruction currently executed in that stage. These indices are computed exactly as the tags in a Tomasulo

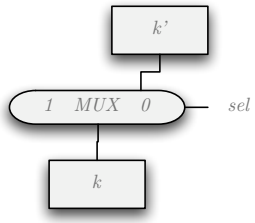

Fig. 2. Scheduling Functions

scheduler. The only difference is that they have unbounded size because we want to count up to arbitrarily large indices. In real hardware this is not possible and not necessary. In an abstract mathematical model there is no problem to do this.

Each stage k of the processors under consideration has an update enable signal ue_k. Stage k gets new data in cycle t if the update enable signal ue_k was on in cycle $t-1$. We fetch instructions in order and hence define for the instruction fetch stage IF:

$$s(IF,t) = \begin{cases} s(IF,t-1)+1 & ue_{IF}^{t-1} \\ s(IF,t-1) & \text{otherwise} \end{cases}$$

In general, a stage k can get data belonging to a new instruction from one or more stages k'. Examples where more than one predecessor stage k' exists for a stage k are: (i) cycles in the data path of a floating point unit performing iterative division or (ii) the producer registers feeding on the common data bus of a Tomasulo scheduler. In this situation one must define for each stage k' a predicate $trans(k,k',t)$ indicating that in cycle t data are transmitted from stage k' to stage k. In the example of Figure 2 we use the select signal sel of the multiplexer and define

$$trans(k,k',t) = ue_k^t \wedge sel^t$$

If $trans(k,k',t-1)$ holds for some k', then one sets $s(k,t) = s(k',t-1)$ for that k'. Otherwise $s(k,t) = s(k,t-1)$.

3.3 Naive Simulation Relations

For ECUs we first consider a 'naive' simulation relation $sim(d,h)$ between ISA configurations d and hardware configurations h. We require that the user visible processor registers R have identical values

$$h.R = d.R$$

For the addresses a in the processor we would like to make a similar definition, but this does not work, because the user visible processor memory is simulated in the hardware by a memory system consisting e.g. of an instruction cache $icache$, a data cache $dcache$ and a user main memory $mainm$. Thus there is a quite nontrivial function

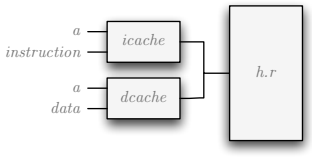

Fig. 3. Memory System

$m(h) : A \to \{0,1\}^8$ specifying the memory simulated by the memory system. One can define this functions in the following way: imagine you apply in configuration h at the memory interface (either at the $icache$ or at the $dcache$) address a. Considering a hit in the instruction cache, i.e. $ihit(h.p, a) = 1$, the $icache$ would return $icache(h.p, a)$. Similarly, considering a hit in the data cache $dhit(h.p, a) = 1$ the $dcache$ would return $dcache(h.p, a)$. Then one can define[3]:

$$m(h)(a) = \begin{cases} icache(h,a) & ihit(h,a) \\ dcache(h,a) & dhit(h,a) \\ h.mainm(a) & \text{otherwise} \end{cases}$$

Using this definition we require in the simulation relation for all addresses $a \in A$:

$$m(h)(a) = d.m(a)$$

In a pipelined machine this simulation relation almost never holds, because in one cycle different hardware stages k usually hold data from different ISA configurations; after all this is the very idea of pipelining. There is however an important exception: when the pipe is drained, i.e. all hardware stages except the instruction fetch stage are empty.

$$drained(h) \leftrightarrow \forall k : k \neq IF \to full_k^t = 0$$

This happens to be the case after interrupts, in particular initially after reset.

3.4 Basic Hardware Correctness Theorem

To begin with we ignore the external interrupts event signals (which brings us formally back to ISA computations defined by $d^{i+1} = \delta_D(d^i)$). Figure 4 shows in simplified form the stages of a processor with out of order processing and a Tomasulo scheduler

Each user visible register $d.R$ of the processor has a counter part $h.R$ belonging to some stage $k = stage(R)$ of the hardware. If the processor would have only registers R and no memory, one could show by induction over t for all cycles t and stages k:

If $k = stage(R)$, then the value $h^t.R$ of the hardware register R in cycle t is the value $d^{s(k,t)}.R$ of the ISA register R for the instruction scheduled in stage k in cycle t.

$$h^t.R = d^{s(k,t)}.R$$

[3] In the processors under consideration the caches snoop on each other; data of address a is only in at most one cache [Bey05,BJK+03]

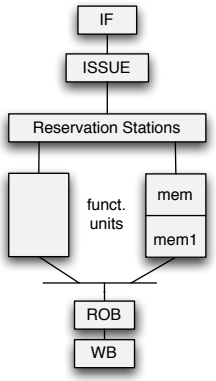

Fig. 4. Processor Pipeline

For the memory one has to consider the memory unit of the processor consisting of two stages mem and $mem1$. Stage mem contains hardware for the computation of the effective address. The memory $m(h^t)$ which is simulated by the memory hierarchy of the hardware in cycle t, is identical with the ISA memory $d^{s(mem1,t)}.m$ for the instruction scheduled in stage $mem1$ in cycle t.

$$m(h^t) = d^{s(mem1,t)}.m$$

We summarize this by stating a basic processor correctness theorem. It assumes that initially the pipe is drained and that the simulation relation between the first hardware configuration h^0 and the first ISA configuration d^0 holds.

Theorem 1 (Processor Correctness). *Assume $drained(h^0)$ and $sim(d^0, h^0)$ holds. Then for all t, for all stages k and for all registers R with $stage(R) = k$:*

$$\begin{aligned} h^t.R &= d^{s(k,t)}.R \\ m(h^t) &= d^{s(mem1,t)}.m \end{aligned}$$

Such theorems are proven by induction over t. For complex processors this requires hundreds of pages of paper and pencil proofs. For details see [MP00]. For a formal correctness proof see e.g. [Bey05,BJK$^+$03].

3.5 Dealing with External Interrupts

External interrupts complicate things only very slightly. The hardware now has external inputs $heev$ which we call the hardware interrupt event signals. Their value in hardware cycle t is $heev^t$. We have to construct from them a sequence eev^i of external ISA interrupt event signals such that the hardware simulates an ISA computation satisfying $d^{i+1} = \delta_D(d^i, eev^i)$. In order to support precise interrupts (for details see [MP00], Chapter 5) processor hardware usually samples interrupt event signals in the write back

stage WB. Because the write back stage is the last stage in the pipeline it cannot be stalled. Thus for every instruction i there is exactly one cycle $t = WB(i)$ such that $s(WB, t) = i \wedge full^t_{WB}$. The external ISA event signal observed by instruction i is therefore

$$eev^i = heev^{WB(i)}$$

Note that a hardware event signal $heev^t$ does not become visible to the ISA computation, if the write back stage in cycle t is empty. With this new definition of the ISA computation the processor correctness theorem 1 still holds. More details regarding a formal processor correctness proof dealing with external interrupts are given in [Bey05,Dal06].

4 Device Theory

4.1 Device Configurations

This section basically covers the device independent part of [HIP05]. In memory mapped I/O processors communicate with devices by read and write accesses to certain addresses x called *I/O ports*. In our treatment these addresses will be above the addresses in the processor memory: if b is the largest address in the processors memory, then $x > b$. For a hardware designer integrating a device with a processor a device therefore better should look in many respects like an ordinary RAM. However, a device has in general more state than is visible in the I/O ports. Thus configurations of devices f with N I/O ports have the following components

- a port RAM $f.m$. We assume that the device is word addressable and provides P bytes of port RAM $f.m : \{0,1\}^p \to \{0,1\}^8$ with $p = \lceil \log P \rceil - 2$.
- 'internal' state $f.Z$

Hardware devices take inputs from and produce output to the processor side and to the outside world. Inputs from the processor side are like inputs for the RAM and consist of: data in din, address $addr$ and write signal w. Outputs to the processor side are data out $dout$ (like in a RAM) and an external hardware interrupt event signal $heev$. Inputs $fdin$ from and outputs $fdout$ to the outside world are device dependent: network devices have inputs and outputs, monitors produce only outputs, keyboards take only inputs, disks neither produce outputs nor consume inputs.

I/O ports can be roughly divided in three categories: (i) control ports are only written from the processor side, (ii) status ports are only read by the processor side and (iii) data ports can be written or read both from the processor side and from the device side. Thus one must deal with the classical synchronization issues of shared memory. We postulate, that for each word address $addr$ of the device there is a device specific hardware predicate $hquiet(f, addr)$ acting like a semaphore. It indicates that the 4 ports belonging to that address are presently not being accessed from the device side and hence it is safe to access them from the processor side like ordinary RAM. Making the device configuration f a component $h.f$ of the hardware configuration we define for reading out the port RAM

$$fdout(h) = h.f.m(addr(h))$$

At quiet addresses the port RAM behaves like a RAM accessible only by the processor side.

$$\forall x : hquiet(h.f, x) \Rightarrow h'.f.m(x) = \begin{cases} din(h) & x = addr(h) \wedge fw(d) \\ h.f.m(x) & \text{otherwise} \end{cases}$$

When a data port is not quiet, it can be read or modified by the device side in a device specific way. The effect of writing a port which is not quiet from the processor side is left undefined. The processor side usually learns about changes in the quiet predicate either by an interrupt from the device either by (i) writing to a command register or (ii) by an interrupt from the device or (iii) by polling a status register. Here we will not consider polling[4].

4.2 Integrating Devices

Integration of a device into a memory system is an exercise in hardware design. The device is placed at some base address ba into the processors (byte addressable) memory space. An address decoder decides on read or write accesses whether the device is addressed by a load word or store word instruction ($ea \in \{ba, \ldots, ba + P - 1\}$). If the base address is a multiple of the page size $4K$ and if the device occupies exactly one page of memory, then the address decoder simply performs the test

$$ba[31:12] =^{?} ea[31:12]$$

If device f is accessed by a store word instruction, then write signal fw is activated. If the device is accessed by a load word instruction, then the output enable signal of a driver between $dout$ and some bus in the processors memory system is enabled. Moreover the cache system must be designed in a way that it does not cache accesses to I/O ports.

4.3 ISA Model with Devices

The assembler programmer then sees a system as shown in Figure 5. It is a distributed system because the non quiet ports of the device can change in a device specific way while the processor is working. Configurations have the form $e = (e.d, e.f)$ where $e.d$ is an ISA processor configuration and $e.f$ is the device state. Component $e.f$ might have the same form as the state from the hardware model or it might be more abstract. In the assembler model the programmer should have some means to keep track of the quiet status of ports. Thus, the hardware predicate $hquiet(h.f, addr)$ needs a device specific assembler level counter part $quiet(e.f, addr)$.

As we have argued in the introduction, in an assembler level model for a processor with a device the occurrence of external interrupts is inherently nondeterministic. We

[4] A reader experienced in hardware design will observe that our devices are unusually fast: they update a port in a single cycle of the processor hardware. Devices are usually slower and have a busy signal indicating that a read or write access is in progress. Extending the above definitions in this way poses no big difficulties.

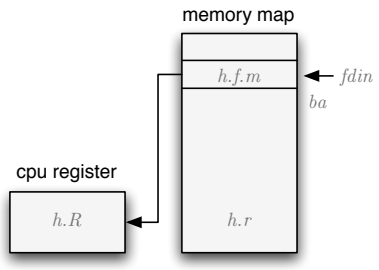

Fig. 5. Memory Mapped IO

model this nondeterminism here by an oracle input eev, which is used in the next state computation for the processor component $e'.d$ as explained in Section 3.5. In the case of accesses to the I/O ports the next processor state will also depend on the device state, thus we will define for an extension of the old next state function δ_D:

$$e'.d = \delta_D(e.d, e.f, eev)$$

The extension concerns load word instructions whose effective address is an I/O port and of course the occurrence of interrupts:

$$\neg JISR(e.d, eev) \wedge lw(e.d) \wedge ea(e.d) = ba + 4 \cdot addr \wedge quiet(e.f, addr) \Rightarrow \\ e'.d.gpr(RD(e.d)) = d.f.m(addr)$$

Moreover we can specify in a device independent way, that quiet portions of the port RAM behave like processor memory; they are only changed by store word instructions:

$$\forall x : quiet(e.f, x) \wedge \neg JISR(e.d, eev) \Rightarrow \\ e'.f.m(x) = \begin{cases} e.d.gpr(RD(e.d)) & sw(e.d) \wedge ea(e.d) = ba + 4 \cdot x \\ e.f.m(x) & \text{otherwise} \end{cases}$$

The remaining portions of the definition of $e'.f$ are device specific.

4.4 Hardware Correctness Theorem with Devices

In order to be able to use existing hardware correctness proofs for processors alone, we split the hardware h into a processor component $h.p$ and a device component $h.f$. Using the machinery already in place the extensions to the hardware correctness proof are remarkably easy as long as computations only access quiet I/O ports and the quiet predicate is stable for all ports. If we place in the hardware the devices port RAM parallel to the normal memory system, then we can use the same scheduling functions which work for the memory. We get a new induction hypothesis of the form

$$h^t.p.R = e^{s(k,t)}.d.R \\ m(h^t.p) = e^{s(mem1,t)}.d.m \\ h^t.f.m = e^{s(mem1,t)}.f.m$$

An external interrupt from the device will need device specific arguments. The hardware correctness proof works with the oracle inputs eev^i obtained from the hardware event signal $heev$ into an ISA event signal eev^t by the translation from Section 3.5.

$$eev^i = heev^{WB(i)}$$

5 Memory Management

5.1 Address Translation, Physical Machines and Virtual Machines

Physical machines consist of a processor operating on physical memory and on swap memory. Configurations d of physical machines have components $d.R$ for processor registers R, $d.m$ for the physical memory, and $d.sm$ for the swap memory. The physical machine has several special purpose registers not present in virtual machines, e.g. the mode register $mode$, the page table origin pto, and the page table length ptl.

In system mode, i.e. if $d.mode = 0$, the physical machine operates like the basic processor model from Section 2 with extra registers. In user mode, i.e. if $d.mode = 1$, the physical machine *emulates* the basic processor model from Section 2 using the page table for address translation. The simulated machine is called a *virtual machine*, and addresses generated by the virtual machines are called virtual addresses. We keep the notation d for configurations of the physical machine and we denote configurations of the virtual machine by vm. We split virtual addresses va into a page index $va.px = va[31:12]$ and into a byte index $va.bx = va[11:0]$. Thus pages size is $2^{12} = 4K$ bytes.

In user mode accesses to memory addresses va are subject to address translation: they either cause a page fault or are redirected to the translated physical memory address $pma(d, va)$. The result of address translation depends on the contents of the *page table*, a region of the physical memory starting at address $d.pto \cdot 4K$ with $d.ptl + 1$ entries of four bytes width[5].

Page table entries have a length of four bytes. The page table entry address for virtual address va is defined as $ptea(d, va) = d.pto \cdot 4K + 4 \cdot va.px$ and the page table entry of va is defined as $pte(d, va) = d.m_4(ptea(d, va))$. For our purposes a page table entry consists of two components as shown in Figure 6: the physical page index $ppx(d, va) = pte(d, va)[31:12]$ and the valid bit $v(d, va) = pte(d, va)[11]$.

On user mode memory access to address va, a page fault signals if the page index exceeds the page table length, $va.px > d.ptl$ or if the page table entry is not valid,

[5] The '+1' in this definition is awkward. It dates back to very old architectures. Because page table length is usually is a power of two it saves a bit in the page table length register

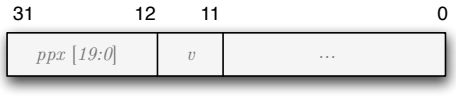

Fig. 6. Page Table Entry

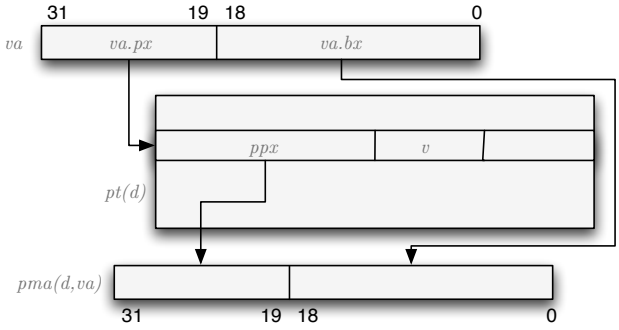

Fig. 7. Address Translation

$v(d, va) = 0$. On page fault the page fault handler, an interrupt service routine, is invoked.

Without a page fault, the access is performed on the (translated) physical memory address $pma(d, va)$ defined as the concatenation of the physical page index and the byte index,

$$pma(d, va) = ppx(d, va) \circ va.bx$$

Notice that the complete definition of a physical machine model involves the specification of the effect of a page fault handler. In pervasive system verification there is a model between the physical machine and the hardware: a processor with a disk, as an I/O device. In this model one can show that swap memory is a proper abstracting by proving the correctness of the page fault handler. For details see [HIP05]. The real time systems under consideration here have no disks and are programmed such that page faults do not occur; thus the omission of these details will not hurt us later.

5.2 Virtual Memory Simulation

A physical machine with appropriate page fault handlers can simulate virtual machines. For a simple page fault handler, virtual memory is stored on the swap memory of the physical machine and the physical memory acts as a write back cache. In addition to the architecturally defined physical memory address $pma(d, va)$, the page fault handler maintains a swap memory address function $sma(d, va)$. On page faults that do not violate the page table length check, the handler selects a physical memory page to evict and loads the missing page from the swap memory.

As in Section 2 we denote by b the maximal byte address accessible by the virtual machine. We use a simulation relation $B(vm, d)$ to indicate that a (user mode) physical machine configuration d encodes virtual machine configuration vm. Essentially, $B(vm, d)$ is the conjunction of the following two conditions:

1. For each of the $b/(4K)$ pages of virtual memory there is a page table entry in the physical machine, $b/(4K) = d.ptl$.

2. The content of virtual memory byte va is stored in physical memory at address $pma(d, va)$ if the corresponding valid bit is on; otherwise it is stored in swap memory.

$$vm.m(va) = \begin{cases} d.m(pma(d, va)) & v(d, va) \\ d.sm(sma(d, va)) & \text{otherwise} \end{cases}$$

Thus the physical memory serves as a write back cache for the swap memory.

The simulation theorem for a single virtual machine has the following form:

Theorem 2. *For all computations of the virtual machine* $(vm^0, vm^1, \dots)$ *there is a computation of the physical machine* $(d^0, d^1, \dots)$ *and there are step numbers for the physical machine* $(s(0), s(1), \dots)$ *such that for all* i *we have* $B(vm^i, d^{s(i)})$.

Thus step i of the virtual machine is simulated after step $s(i)$ of the physical machine. Even for simple handlers, the proof is not completely obvious since a single user mode instruction can cause two page faults. To avoid deadlock and guarantee forward progress, the page fault handler must not swap out the page that was swapped in during the last execution of the page fault handler. For details see [Hil05].

5.3 Synchronization Conditions

If the hardware implementation of a physical machine is pipelined or if it executes instructions out of order execution then an instruction $I(d^i)$ that is in the memory stage may modify a later instruction $I(d^j)$ for $j > i$ after it has been fetched, constituting a read after write (RAW) hazard. It may (i) overwrite the instruction itself, (ii) overwrite its page table entry, or (iii) change the mode.

In such situations instruction fetch (in particular translated fetch implemented by a memory management unit) would not work correctly. Of course it is possible to detect such data dependencies in hardware and to roll back the computation if necessary. Alternatively, the software to be run on the processor must adhere to certain *software synchronization conventions*. Let $iaddr(d^j)$ denote the address of instruction $I(d^j)$, possibly translated. If $I(d^i)$ writes to address $iaddr(d^j)$, then an intermediate instruction $I(d^k)$ for $i < k < j$ must drain the pipe. The same must hold if d^j is in user mode and $I(d^i)$ writes to $ptea(d^j, d^j.dpc)$. Finally, mode can only be changed to user mode by an `rfe` (return from exception) instruction (and the hardware guarantees that `rfe` instructions drain the pipe).

These conditions are hypotheses in the hardware correctness theorem in [DHP05]. It is be easy to show that they hold for the kernels constructed in Section 9.

6 Compilation

6.1 *C*0 Semantics

We summarize results from [LPP05]. Recall that C0 is roughly PASCAL with C syntax. Eventually we want to consider several programs running under an operating system.

The computations of these programs are interleaved. Therefore our compiler correctness statement is based on a small steps / structured operational semantics [NN99,Win93].

In $C0$ types are elementary ($bool$, int, ...), pointer types, or aggregate ($array$ or $struct$). A type is called simple if it is an elementary type or a pointer type. We define the (abstract) size of types for simple types t by $size(t) = 1$, for arrays by $size(t[n]) = n \cdot size(t)$, and for structures by $size(struct\{n_1 : t_1, \ldots, n_s : t_s\}) = \sum_i size(t_i)$. Values of variables with simple type are called *simple values*. Variables of aggregate type have *aggregate values*, which are represented as a flat sequence of simple values.

6.2 $C0$ Machine Configuration

A $C0$ machine configuration c has the following components:

1. The *program rest* $c.pr$, which is a sequence of $C0$ statements that still need to be executed. In [NN99] the program rest is called *code component* of the configuration.
2. The *recursion depth* $c.rd$.
3. The *local memory stack* $c.lms$. It maps numbers $i \leq c.rd$ to memory frames (defined below). The global memory is $c.lms(0)$. We denote the top local memory frame of a configuration c by $top(c) = c.lms(c.rd)$.
4. A *heap memory* $c.hm$. This is also a memory frame.

Parameters of the configuration which do not change during a computation are

- The *type table* $c.tt$ containing information about types used in the program.
- The *function table* $c.ft$ containing information about the functions of a program. It maps function names f to pairs $c.ft(f) = (c.ft(f).ty, c.ft(f).body)$ where $c.ft(f).ty$ specifies the types of the arguments, the local variables, and the result of the function, whereas $c.ft(f).body$ specifies the function body.

Memory frames. We use a relatively explicit, low level memory model in the style of [Nor98]. Memory frames m have the following components:

1. the number $m.n$ of variables in m (for local memory frames this also includes the parameters of the corresponding function definition),
2. a function $m.name$ mapping variable numbers $i \in [0 : m.n - 1]$ to their names (not used for variables on the heap),
3. a function $m.ty$ mapping variable numbers to their type. This permits to define the size of a memory frame $msize(m)$ as the number of simple values stored in it, namely: $msize(m) = \sum_{i=0}^{m.n-1} size(m.ty(i))$.
4. a content function $m.ct$ mapping indices $0 \leq i < msize(m)$ to simple values.

A *variable* v of configuration c is a pair $v = (m, i)$ where m is a memory frame of c and $i < m.n$ is the number of the variable in the frame. The type of a variable (m, i) is defined by $ty((m, i)) = m.ty(i)$.

Subvariables $S = (m, i)s$ are formed from variables (m, i) by appending a *selector* $s = (s_1, \ldots, s_t)$, where each component of a selector has the form $s_i = [j]$ for selecting

array element number j or the form $s_i = .n$ for selecting the struct component with name n. If the selector s is consistent with the type of (m, i), then $S = (m, i)s$ is a *subvariable* of (m, i). Selectors are allowed to be empty.

In $C0$, pointers p may point to subvariables $(m, i)s$ in the global memory or on the heap. The value of such pointers simply has the form $(m, i)s$. Component $m.ct$ stores the current values $va(c, (m, i)s)$ of the simple subvariables $(m, i)s$ in canonical order. Values of aggregate variables x are represented in $m.ct$ in the obvious way by sequences of simple values starting from the abstract base address $ba(x)$ of variable x.

With the help of visibility rules and bindings we easily extend the definition of va, ty, and ba from variables and subvariables to expressions e.

6.3 *C*0 Machine Computation

Due to space restrictions we cannot give the full definition of the (small-step) transition function δ_C mapping $C0$ configurations c to their successor configuration $c' = \delta_C(c)$. As an example we give a partial definition of the function call semantics.

Assume the program rest in configuration c begins with a call of function f with parameters $e_1, \ldots, e_n$ assigning the function's result to variable v, formally $c.pr = (v = f(e_1, \ldots, e_n); r)$. In the new program rest, the call statement is replaced by the body of function f taken from the function table, $c'.pr = (c.ft(f).body; r)$ and the recursion depth is incremented $c'.rd = c.rd + 1$. Furthermore, the values of all parameters e_i are stored in the new top local memory frame $top(c')$ by updating its content function at the corresponding positions: $top(c').ct_{size(ty(c,e_i))}(ba(c, e_i)) = va(c, e_i)$.

6.4 Compiler Correctness Theorem

The compiler correctness statement (for programs to be run on physical or virtual machines) depends on a simulation relation $consis(aba)(c, d)$ between configurations c of $C0$ machines and configurations d of ISA machines which run the compiled program. The relation is parameterized by a function aba which maps subvariables S of the $C0$ machine to their allocated base addresses $aba(c, S)$ in the ISA machine. The allocation function may change during a computation (i) if the recursion depth and thus the set of local variables change due to calls and returns or (ii) if reachable variables are moved on the heap during garbage collection (not yet implemented).

Notice however, that in the first case only the range of the allocation function is changed: for C0 configurations c and local or (sub) global variables x the allocated base address $aba(x, c)$ depends only on c.

Simulation Relation. The simulation relation consists essentially of five conditions:

1. Value consistency $v - consis(aba)(c, d)$: This condition states that reachable elementary subvariables x have the same value in the $C0$ machine and in the ISA machine. Let $asize(x)$ be the number of bytes needed to store a value of type $ty(x)$. Then we require $d.m_{asize(x)}(aba(c, x)) = va(c, x)$.

2. Pointer consistency $p - consis(aba)(c, d)$: This predicate requires for reachable pointer variables p, which point to a subvariable y, that the value stored at the allocated address of variable p in the ISA machine is the allocated base address of y, i.e. $d.m_4(aba(c, p)) = aba(c, y)$. This induces a subgraph isomorphism between the reachable portions of the heaps of the $C0$ and the ISA machine.
3. Control consistency $c - consis(c, d)$: This condition states that the delayed PC of the physical machine (used to fetch instructions) points to the start of the translated code of the program rest $c.pr$ of the $C0$ machine. We denote by $head(r)$ the first statement of statement sequence r and we denote by $caddr(s)$ the address of the first assembler instruction which is generated for statement s. We require $d.dpc = caddr(head(c.pr))$ and $d.pc = d.dpc + 4$.[6]
4. Code consistency $code - consis(c, d)$: This condition requires that the compiled code of the $C0$ program is stored in the physical machine d beginning at the code start address $cstart$. Thus it requires that the compiled code is not changed during the computation of the physical machine. We thereby forbid self modifying code.
5. Stack consistency $s - consis(_, d)$: this is a technical condition about stack pointers, heap pointers etc. which does not play an important role here.

Theorem 3. *For every C0 machine computation $(c^0, c^1, \dots)$ there is a computation of the physical machine $(d^0, d^1, \dots)$, step numbers $(s(0), s(1), \dots)$, and a sequence of allocation functions $(aba^0, aba^1, \dots)$ such that for all steps i the C0 machine and the physical machine are consistent $consis(aba^i)(c^i, d^{s(i)})$.*

A formal proof of this statement for a non optimizing compiler specified in Isabelle-HOL [**?**] (roughly speaking: in ML) is completed and will be reported in [**?**]. There is an implementation of the same compilation algorithm written in C0. A formal proof that the C0 implementation simulates the ML implementation is also completed and will be reported in [**?**]. In order to solve the bootstrap problem [The00] the C0 version of the compiler was translated by an existing compiler into DLX code. That the target DLX code simulates the source code will be shown using translation validation. This is work in progress.

7 Inline Assembler Code Semantics

Recall that processor registers, I/O ports and user processes are not visible in the C variables of an operating system kernel written in C. Hence we must necessarily permit in our language sequences u of in line assembler instructions (we do not distinguish here between assembler and machine language). We extend $C0$ by statements of the form $asm(u)$ and call the resulting language $C0_A$. In $C0_A$ the use of inline assembler code is restricted: (i) only a certain subset of DLX instructions is allowed (e.g. no load or store of bytes or half words, only *relative* jumps), (ii) the target address of store word instructions must be outside the code and data regions of the $C0_A$ program or it must be equal to the allocated base address of a subvariable of the $C0_A$ program with type int or $unsigned\ int$ (this implies that inline assembler code cannot change the stack layout

[6] For optimizing compilers this condition has in general to be weakened.

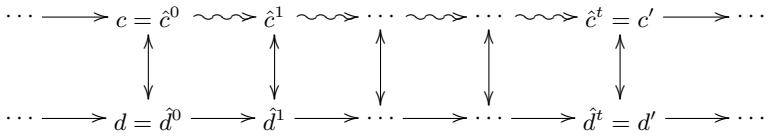

Fig. 8. Execution of Inline Assembler Code

of the $C0_A$ program), (iii) certain registers (e.g. the stack pointer) must not be changed, (iv) the last assembler instruction in u must not be a jump or branch instruction, (v) the execution of u must terminate, (vi) the target of jump and branch instructions must not be outside the code of u, and (vii) the execution of u must not generate misalignment or illegal instruction interrupts.

☞ termination depends on d

In order to argue about the correctness of $C0_A$ programs we must define the semantics of the newly introduced statements. Because a store word instruction of in line assembler code can overwrite a C variable x - for instance when a processor register is stored into a process control block - we have to specify the effect of that store instruction on the value of x the C0 configuration. This is easily done with the help of the allocated base address functions aba of the previous section (and impossible without them).

Thus consider a $C0_A$ configuration c with program rest $c.pr = asm(u); r$. When we enter the in line assembler portion, then the entire physical machine configuration d becomes visible. In this situation we make d an input parameter for the $C0_A$ transition function δ_{C0_A}. As pointed out above, another necessary parameter is an allocated base address function aba. Finally the in line assembler code will also produce a new DLX configuration d'. Thus we will define $(c', d') = \delta_{C0_A}(aba)(c, d)$. In all situations where we apply this definition we will have $consis(aba)(c, d)$.

The execution of u leads to a physical machine computation $(d = \hat{d}^0, \hat{d}^1, \ldots, \hat{d}^t = d')$ with $\hat{d}^t.dpc = caddr(head(r))$ and $\hat{d}^t.pc = \hat{d}^t.dpc + 4$ by the restrictions on inline assembler. We construct a corresponding sequence $(\hat{c}^0, \ldots, \hat{c}^t)$ of intermediate $C0$ machine configurations reflecting successively the possible updates of the C0 variables by the assembler instructions (see Figure 8). We set $\hat{c}^0 = c$ except for deleting the in line assembler portion $asm(u)$ from the program rest: $\hat{c}^0.pr = r$. Let $j < t$. If predicate $sw(\hat{d}^j)$ holds the instruction executed in configuration $\hat{d}^j$ writes the value $v = \hat{d}^j.gpr(RD(\hat{d}^j))$ to the word at address $ea(\hat{d}^j)$ by the definitions of Section 2. If this effective address is equal to the allocated base address of a $C0$ variable x, then we update the corresponding variable in configuration $\hat{c}^{j+1}$ such that $va(\hat{c}^{j+1}, x) = v$.

$$sw(\hat{d}^j) \wedge (ea(\hat{d}^j) = aba(c, x) \rightarrow va(\hat{c}^{j+1}, x) = \hat{d}^j.gpr(RD(\hat{d}^j))$$

Finally the result of the $C0_A$ transition function is defined by $c' = \hat{c}^t$ and $d' = \hat{d}^t$. This definition keeps configurations consistent:

Lemma 1. *If the program rest of c starts with an inline assembler statement we have:*

$$consis(aba)(c, d) \Rightarrow consis(aba)(\delta_{C0_A}(aba)(c', d'))$$

8 Communicating Virtual Machines (CVM)

8.1 CVM Semantics

We introduce communicating virtual machines (CVM), a model of computation for a generic abstract operating system kernel interacting with a fixed number of user processes. While the CVM is running, the kernel can only be interrupted by reset. Kernels with this property are called non-preemptive[7]. CVM uses the $C0$ language semantics to model computations of the (abstract) kernel and virtual machines to model computations of user processes. It is a pseudo-parallel model in the sense that in every step of computation either the kernel or one user process can make progress.

From a kernel implementor's point of view, CVM encapsulates the low-level functionality of a microkernel and provides access to it as a library of functions, the so-called CVM primitives. Accordingly, the abstract kernel may be 'linked' with the implementation of these primitives to produce the concrete kernel, a $C0_A$ program, that may be run on the target machine. This construction and its correctness will be treated in Section 9.

In the following sections we define CVM configurations, CVM computations, and show how abstract kernels implement system calls as regular $C0$ function calls.

8.2 CVM Configuration

A CVM configuration cvm has the following components:

- User processes are modeled by virtual machine configurations $cvm.vm(u)$ having indices $u \in \{1, \ldots, P\}$ (and P fixed, e.g. $P = 128$).
- Each user process has an individual page table 'lengths' $cvm.vm(u).ptl$. The memory available to a virtual machine can be increased or decreased dynamically.
- A C0 machine configuration $cvm.c$ represents the so-called *abstract kernel*. We require the kernel configuration, in particular its initial configuration, be in a certain form:
  1. Certain functions $f \in CVMP$, the CVM primitives, must be declared only, thus their body must be empty. Its arguments and effects are described below.
  2. In addition to the cvm primitives a special function called $kdispatch$ must be declared. It takes two integer arguments and returns an integer. An invocation of the $kdispatch$ function as being an initial program rest must eventually result in a function call of the CVM primitive $v = start(e)$, which passes control to the user processes determined by the current value $va(cvm.c, e)$ of expression e.
- The component $cvm.cp$ denotes the current process: $cvm.cp = 0$ means that the kernel is running while $cvm.cp = u > 0$ means that user process u is running.
- $cvm.f$ denotes the state of one[8] external device capable of interrupting user processes with an ISA interrupt signal eev.

[7] Premptive kernels require to deal with nested interrupts. A theory of nested interrupts is outlined in Chapter 5 of [MP00].

[8] Dealing with more devices is not necessary here; it is not much more difficult.

8.3 CVM Computation

In every step of a CVM computation a new CVM configuration is computed from an old configuration cvm, an oracle input eev, and from the a device specific external input $fdin$:

$$cvm' = \delta_{CVM}(cvm, eev, fdin)$$

The external input only affects the device state $cvm'.f$. Updates of this state are device specific and are not treated here.

User computation. If the current process $u = cvm.cp$ in configuration cvm is non-zero then user process $vm(u)$ does a step.

$$cvm'.vm(u) = \delta_D(cvm.vm(u))$$

If no interrupt occurred, i.e. $\neg JISR(cvm.vm(u), eev)$ then user process $vm(u)$ keeps running:

$$cvm'.cp = u$$

Otherwise execution of the abstract kernel starts. Recall from Section 2.4 on interrupt semantics, that in case of an interrupt the masked cause register is saved into the exception cause register eca and that certain data necessary for handling the exception is stored in register $edata$. The kernels entry point is the function $kdispatch$ that is called with the saved exception cause register $cvm.vm(u).eca$ and the saved exception data register $cvm.vm(u).edata$ as parameters.

We set the current process component and the kernel's recursion depth to zero:

$$\begin{aligned} cvm'.cp &= 0 \\ cvm'.c.rd &= 0 \\ cvm'.c.pr &= kdispatch(cvm.vm(u).eca, cvm.vm(u).edata) \end{aligned}$$

Kernel computation. Initially (after power-up) and after an interrupt, as seen above, the kernel starts execution with a call of the function $kdispatch$. User process execution continues when the kernel calls the CVM primitive $start$.

If we have $cvm.cp = 0$ and the kernel's program rest does not start with a call to a CVM primitive, a regular $C0$ semantics step is performed:

$$cvm'.c = \delta_C(cvm.c)$$

Otherwise, we have $cvm.cp = 0$ and $cvm.c.pr = (v = f(e_1, \ldots, e_n); r)$ for a CVM primitive f, an integer variable v and integer expressions e_1 to e_n. Although the implementation of the CVM primitives involves in line assembler code, their semantics can be specified in the pseudo parallel CVM model by the effect they have on the user processes $vm(u)$ and on the device f.

Below we describe a few selected CVM primitives. We ignore any preconditions or border cases; these are straightforward to specify and resolve.

- $start(e)$ hands control over to the user process specified by the current value of expression e:

$$cvm'.cp = va(cvm.c, e)$$

  By this definition, the kernel stops execution and is restarted again on the next interrupt (with a fresh program rest as described before).
- $alloc(u, x)$ increases the memory size of user process $U = va(cvm.c, u)$ by $X = va(cvm.c, x)$ pages:

$$cvm'.vm(U).ptl = cvm.vm(U).ptl + X$$

  The new pages are cleared:

$$\forall y \in [cvm.vm(U).ptl : cvm.vm(U).ptl + 4K - 1] : cvm'.vm(U).m(y) = 0^8$$

- A primitive $free(u, x)$ which frees X pages of user process U is defined in a similar way.
- $copy(u_1, a_1, u_2, a_2, d)$ copies memory regions between user processes $U_1 = va(cvm.c, u_1)$ and $U_2 = va(cvmc, u_2)$. Start addresses in process U_1 resp. U_2 are $A_1 = va(cvm.c, a_1)$ resp. $A_2 = va(cvm.c, a_2)$. The number of bytes copied is $D = va(cvm.c, d)$:

$$cvm'.vm(U_2).m_D(A_2) = cvm.vm(U_1).m_D(A_1)$$

- Primitives copying data between user processes and I/O ports and between C variables of the kernel and I/O ports are defined in a similar way.
- $e = getgpr(r, u)$ reads general purpose register $R = va(cvm.c, r)$ of user process $U = va(cvm.c, u)$ and assigns it to the (sub)variable specified by expression e:

$$va(cvm'.c, e) = cvm.vm(U).gpr(R)$$

  As described below, this primitive is used to read parameters of system calls.
- $setgpr(r, u, e)$ writes the current value of expression e into general purpose register R of process U:

$$cvm'.vm(U).gpr(R) = va(cvm.c, e)$$

  This primitive is used to set return values of system calls.

8.4 Binary Interface of Kernels

Before we deal with the implementation of CVM and a proof for its correctness, we show how to build a kernel by appropriately specializing the generic abstract kernel of CVM.

The obvious means for a user process to make a system call is to use the trap instruction which causes an internal interrupt. If the kernel provides k trap handlers, then the user can specify the to be invoked handler using the immediate constant i within the trap instruction, where $i \in [0 : k - 1]$. We define a so called kernel call definition function kcd mapping immediate constants $i \in [0 : k - 1]$ to names of functions declared in the abstract kernel. Thus $kcd(i)$ is simply the name of the C function (including CVM

primitives) handling $trap(i)$. For each i, let $np(i) < 20$ be the number of parameters of function $kcd(i)$[9]. We require the user to pass the parameters for function $kcd(i)$ in general purpose registers $gpr[1 : n(i)]$. Together with the specification of the functions $kcd(i)$ this is the entire binary interface definition.

Implementation by specialization of the abstract CVM kernel is completely straight forward. First of all the kernel maintains a variable cu keeping track of the user process which is currently running or which has been running before the kernel started execution:

$$cvm.cp > 0 \Rightarrow va(cvm.c, cu) = cvm.cp$$

Assume $cvm.cp = u > 0$ and user $vm(u)$ executes the trap instruction with parameter i, i.e. $trap(cvm.vm(u), i)$. Assume that the trap instruction activates internal event line $iev(5)$, like described in Section 3.5, and that no interrupts with higher priority (lower index) are activated simultaneously. Then the masked cause vector $0^{26}10^5$ is saved into the exception cause register $eca[31 : 0]$ and index i is saved into the exception data register $edata$:

$$\begin{aligned} cvm'.vm(u).eca &= 0^{26}10^5 \\ cvm'.vm(u).edata &= i \end{aligned}$$

According to the CVM semantics the abstract kernel starts running with the function call $kdispatch(eca, edata)$ where $eca = 0^{26}10^5$ and $edata = i$. By a case split on eca the handler concludes that a trap instruction needs to be handled. Hence the handler invokes the function call $f(e_1, \ldots, e_{np(i)})$, where $f = kcd(i)$ using the parameters computed by the assignment $e_i = getgpr(i, cu)$.

Let $cvmd$ be the CVM configuration immediately after execution of the call of kcd above. Then one easily derives from the semantics of CVM and C0:

Lemma 2 (Intended Handler Called with the Intended Parameters).

$$\begin{aligned} cvmd.rd &= cvm'.rd + 1 = 1 \\ cvmd.c.pr &= ft(f).body; r \qquad \textit{for some } r \\ top(cvmd).ct(j) &= cvm.vm(u).gpr(j) \quad \textit{for all } j \in [1 : np(i)] \end{aligned}$$

This lemma formalizes the idea that an interrupt is something like a function call of the handler. Comparing with the C0 semantics in Section 6.1 we see that the trap instruction indeed formally causes a function call of the handler. The function call is however remote, because it is executed by a process (the abstract kernel) different from the calling process (virtual machine $vm(u)$).

9 CVM Implementation and Correctness

9.1 Concrete kernel and Linking

So far we have talked about the abstract kernel, but we have argued mathematically only about its configurations c. Now we also argue about its source code, which we denote by sak. We describe how to obtain the source code sck of the so called *concrete kernel*

[9] Assume for simplicity they are of type integer.

by linking sak with the source code of some CVM implementation $scvm$ by some link operator ld:

$$sck = ld(sak, scvm)$$

Note that sak is a pure C0 program, whereas $scvm$ and sck are $C0_A$ programs. The function table of the linked program sck is constructed from the function tables of the input programs. For functions present in both programs, *defined functions* (with a non-empty body) take precedence over *declared functions* (without a body). We do not formally define the ld operator here; it may only be applied under various restrictions concerning the input programs, e.g. the names of global variables of both programs must be distinct, function signatures must match, and no function may be defined in both input programs.

We require that the abstract kernel sak defines $kdispatch$ and declares all CVM primitives while the CVM implementation $scvm$ defines the primitives and declares $kdispatch$.

In analogy to the $consis$ relation of the compiler correctness proof we define a relation $kconsis(kalloc)(c, cc)$ stating that abstract kernel configuration c is coded by concrete kernel configuration cc. Function $kalloc$ maps subvariables x of abstract kernel configuration c to subvariables $kalloc(x)$ of concrete kernel configuration cc.

Linking is less complex than compilation. The definition of $kconsis$ has only three parts:

1. $e - kconsis(kalloc)(c, cc)$: All reachable elementary (sub) variables x of abstract kernel configuration c and the values of x in the concrete kernel coincide:

$$va(c, x) = va(cc, x)$$

2. $kalloc$ is a graph isomorphism between the reachable portions of the heaps. For all reachable pointer variables p of abstract kernel configuration c (pointing to subvariable p'):

$$(va(c, p) = p') \Rightarrow va(cc, kalloc(p)) = kalloc(p')$$

3. $c - kconsis$: The program rest of the concrete kernel is a prefix of the program rest of the abstract kernel. For technical reason there is a particular suffix r containing 'dangling returns'. This suffix is cleared when the kernel is started the next time (see Section 8.3).

$$cc.pr = c.pr; r$$

9.2 Data Structures

The CVM implementation maintains data structures for the simulation of the virtual machines, i.e. for the support of multiprocessing. These include:

1. An array of process control blocks $pcb[u]$ for the kernel ($u = 0$) and the user processes ($u > 0$). Process control blocks are structs with components $pcb[u].R$ for every processor register R of the *physical* machine.

2. A single integer array $ptarray$ on the heap holds the page tables of all user processes in the order of the process numbers u. The function $ptbase(u)$ defines the start index of the page table for process u:

$$ptbase(u) = \sum_{j<u} (pcb[j].ptl + 1)$$

Because the C array $ptarry$ is indexed by words and not by bytes we define the page table entry for virtual address va and process u as:

$$pte(u, va) = ptarray[ptbase(u) + va.px]$$

Notice that we have faked here pointer arithmetic on the page table array, but formally we just barely managed to dance around it. Physical page address and valid bit are defined by C expressions.

$$\begin{aligned} pma(u, va) &= pte(u, va)[31:12] \circ va.bx \\ v(u, va) &= pte(u, va)[11] \end{aligned}$$

Swap memory addresses $sma(u, va)$ are computed by C function in an analogous way. We require that the compiler computes the allocated base address of array $ptarray$ as a multiple of the page size $4K$.
3. Data structures (in the simplest case doubly-linked lists) for the management of physical and swap memory (including victim selection for page faults).
4. The variable cup keeping track of the current user process thus encoding the $cvm.cp$ component (unless the kernel is running).

9.3 Entering System Mode after an Interrupt

When the concrete kernel enters system mode, its program rest is initialized with $init_1; init_2$. In all cases except reset, the first part $init_1$ will (i) write all processor registers R to the process control block $pcb[cup].R$ of the process cup that was interrupted while it was running and (ii) restore the registers of the kernel from process control block $pcb[0]$.

In the second part $init_2$, the CVM implementation detects whether the interrupt was due to a page fault or for other causes. Page faults are handled silently without calling the abstract kernel (cf. below). For other interrupts, we call $kdispatch$ with the parameters already obtained from the C variables $pcb[cup]$.

$$kdispatch(pcb[cup].eca, pcb[cup].edata)$$

9.4 Leaving System Mode

User mode is entered again by a call of $start(cup)$. It is implemented using inline assembler. We write the physical processor registers to $pcb[0]$ in order to save the concrete kernel state. Then we restore the physical processor registers for process cup from $pcb[cup]$ and execute an `rfe` instruction (return from exception).

9.5 Page Fault Handler

The page fault handler maintains a simulation relation B as described in Section 5.2. With correct page fault handlers, user mode steps in the physical machine without interrupts simulate steps of a virtual machine. Note that a single user mode instruction can produce up to two page faults: one during instruction fetch and one during a load or store operation. In order to prevent even more page faults one must not choose the page most recently swapped in as the victim page to be swapped out (as is possible with pure random selection of the victim page).

To reason about multiple user processes u, we have to slightly modify and extend the B relation. Let u be an index of a user process/virtual machine. Let vm be a configuration of a virtual machine and let d be a configuration of the physical DLX machine on which the compiled concrete kernel and the user processes eventually are running. We define predicate $B(u)(cvm, d)$ stating that the configuration $cvm.vm(u)$ of user process u is coded by by configuration d.

1. Processor registers of $vm(u)$ are stored in the physical processor registers, if process u is running; otherwise they are stored in the process control blocks:

$$cvm.vm(u).gpr(r) = \begin{cases} d.gpr(r) & cvm.cp = u \\ va(cvm.c, pcb(u).gpr(r)) & \text{otherwise} \end{cases}$$

2. the memory content $vm(u)(a)$ are stored in the physical memory at the physical memory address, if the valid bit of virtual address a is 1, otherwise it is stored in swap memory at the swap memory address:

$$cvm.vm[u].m(a) = \begin{cases} d.m(va(cvm.c, pma(u, a)) & va(cvm.c, v(u, a)) = 1 \\ d.sm(va(cvm.c, sma(u, a)) & \text{otherwise} \end{cases}$$

9.6 Implementation of the CVM Primitives

The implementation of CVM primitives like $e = getgpr(u, r)$ and $e = setgpr(u, r, e')$ is straightforward:

$$va(cvm.c, e) = pcb[u].gpr(r)$$
$$pcb[u].gpr(r) = va(cvm.c, e')$$

For the CVM primitives $alloc$ and $free$ the page table length of the process has to be increased or decreased and - we have chosen a very simple implementation - lots of page table entries in $ptarry$ above the portion of the modified user process have to me moved around in the page table array. Various other data structures concerning memory management have to be adjusted as well. Such operations are closely interconnected with the page fault handler. Since the page tables are accessible as a C0 data structure, inline assembler is only required to clear newly allocated physical pages. Similarly, the *copy* implementation requires assembler code to copy pages of physical memory between user processes.

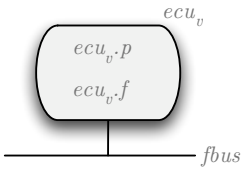

Fig. 9. Electronic Control Units

9.7 CVM Correctness Theorem

The correctness proof of the cvm deals simultaneously with comptations in three computational models:

1. the top model CVM consisting of a C0 machine and several virtual machines; configurations are denoted by cvm.
2. an intermediate model for the C0A computation of the concrete kernel; configurations are denoted by cc
3. the bottom model consisting of a physical machine; configurations are denoted by d

Theorem 4. *Consider an input sequence of external interrupts ($eev^0, eev^1, \ldots$) and a cvm computation ($cvm^0, cvm^1, \ldots$) defined with this input sequence. Then there exists (i) a concrete kernel computation ($cc^0, cc^1, \ldots$), (ii) a physical machine computation ($d^0, d^1, \ldots$), (iii) two sequences of allocation functions ($aba^0, aba^1, \ldots$ and $kalloc^0, kalloc^1, \ldots$) and (iv) two sequences of step numbers ($s^0, s^1, \ldots$ and $t^0, t^1, \ldots$) such that:*

1. *The abstract kernel component $cvm^i.c$ of the CVM computation after i steps is coded by the concrete kernel configuration $cc^{s(i)}$ after $s(i)$ steps:*

$$kconsis(kalloc^i)(c^i, cc^{s(i)})$$

2. *The concrete kernel configuration $cc^{s(i)}$ after $s(i)$ steps is coded configuration $d^{t(i)}$ of the physical machine after $t(i)$ instructions. Recall that on the physical machine the compiled concrete kernel is executed:*

$$consis(aba^i)(cc^{s(i)}, d^{t(i)})$$

3. *the user machines $cvm^i.vm(u)$ after i steps of the CVM computation are coded by configuration $d^{t(i)}$ of the physical machine after $t(i)$ instructions.*

$$B(u)(cvm^i, d^{t(i)})$$

The correctness theorem is proven by induction over the steps i of the CVM computations. Depending on the current process number $cvm^i.cp$ and the interrupts occurring, the proof uses compiler correctness (see Section 6.4), the correctness of memory management mechanisms (see Section 5.2), and detailed arguments about in line assembler code using $C0_A$ semantics (see Section 7).

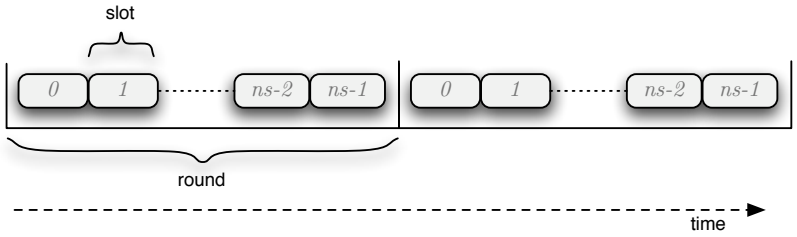

Fig. 10. Slots and Rounds

10 Parallel Hardware Overview

So far we only have considered systems with a single processor and a device. In what follows we construct particular hardware devices serving as interfaces to a FlexRay like bus or short: $fbus$. The devices will be called FlexRay like interfaces or short: f-interfaces. A processor together with a device will be called an electronic control unit (ECU).

We will consider p electronic control units ECU_v, where $v \in \{0, \ldots, p-1\}$, which are communicating over a common $fbus$. At the ISA level, an ECU configuration $ecu_v = (ecu_v.d, ecu_v.f)$, see Figure 9, therefore is a pair consisting of a processor configuration $ecu_v.d$, and a configuration $ecu_v.f$ of an f-interface.

From an interface configuration $ecu_v.f$ we define two user visible buffers: A send buffer $sb(ecu_v)$ and a receive buffer $rb(ecu_v)$. Each buffer is capable of holding a message of ℓ bytes.

In the distributed system all communications and computations proceed in rounds r where $r \in \mathbb{N}$. As depicted in Figure 10 each round is divided into an (even) number of slots s where $s \in \{0, \ldots, ns-1\}$. The tuple (r, s) refers to slot s in round r. On each ECU, boundaries between slots will be determined by local timer interrupts every T hardware cycles. At the beginning of each round the local timers are synchronized.

Given a slot (r, s) we define the predecessor $(r, s) - 1$ and the successor $(r, s) + 1$ according to the lexicographical order of slots. We denote by $d_v(r, s)$ the first and by $e_v(r, s)$ the last ISA configuration of ECU_v during slot (r, s).

ECUs of the system communicate according to a fixed schedule which is identical for each round: The function $send()$ specifies for all rounds r the electronic control unit ECU which owns the bus during slot (r, s):

$$send : \{0, \ldots, ns-1\} \to \{0, \ldots, p-1\}$$

During slot (r, s) the content of the send buffer of $ECU_{send(s)}$ at the end of the previous round $(r, s) - 1$ is broadcast to the receive buffers of all units ECU_u and becomes visible there at the beginning of the next round $(r, s) + 1$:

$$\forall u, r, s : sb(e_{send(s)}((r, s) - 1)) = rb(d_u((r, s) + 1))$$

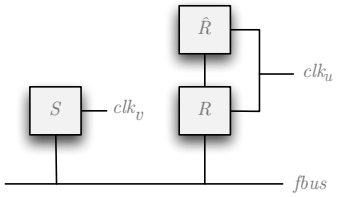

Fig. 11. Serial Interface

In Sections 11 to 14 we will outline the proof of a hardware correctness theorem for the entire distributed system justifying this programming model. This theorem establishes for each ECU_v at the start of each slot (r, s) the naive simulation relation sim from Section 3.4 between the ISA configuration $d_v(r, s)$ before the execution of the first instruction of the slot and the corresponding hardware configuration $h_v(r, s)$ during the first hardware cycle of the slot:

$$sim(d_v(r, s), h_v(r, s))$$

11 Serial Interface

The hardware of each ECU is clocked by an oscillator with a nominal clock period of say τ_{ref}, but for all v the individual clock periods τ_v of ECU_v is allowed to deviate from the nominal period by $\delta = 0.15\%$:

$$\mid \tau_v - \tau_{ref} \mid \leq \tau_{ref} \cdot \delta$$

With $\Delta = 2\delta/(1 - \delta)$ one easily bounds for all u and v the relative deviation of individual clock periods among each other by $\mid \tau_v - \tau_u \mid \leq \tau_v \cdot \Delta$.

Consider a situation, where a sending ECU puts data on the bus and these data are sampled into registers of receiving ECUs. Then, due to the clock drift between ECUs, one cannot guarantee that the set up and hold times of the receiving registers are obeyed at all clock edges. This problem occurs whenever computers without a common clock exchange data. It is solved by serial interfaces using a nontrivial protocol. Therefore we first need a hardware correctness proof of a serial interface as prescribed by the FlexRay standard.

11.1 Hardware Model with Continuous Time

The problems solved by serial interfaces can by their very nature not be treated in the standard digital hardware model with a single digital clock clk. Nevertheless, we can describe each ECU_v in a standard digital hardware model having its own hardware configuration h_v.

In order to argue about a sender register S of a sending ECU that is transmitting data via the $fbus$ to a receiver register R of a receiving ECU, as depicted in Figure 11, we have to extend the digital model.

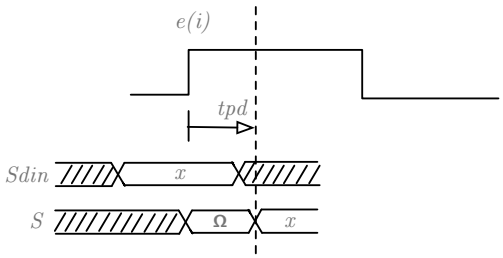

Fig. 12. Sender Register

For the registers –and only for the registers– connected to the $fbus$ we extend the hardware model such that we can deal with the concepts of propagation delay (tpd), set-up time (ts), hold time (th) and metastability of registers from hardware data sheets. In the extended model used near the $fbus$ we therefore consider time to be a real valued variable t. The date of the clock edge $e_v(i)$ which starts cycle i on ECU_v is defined by

$$e_v(i) = c_v + i \cdot \tau_v \tag{1}$$

for some offset $c_v < \tau_v$. In this continuous time model the content of the a sender register S at time t is denoted by $S(t)$.

We now have enough machinery to define in the continuous time model the output of a sender register S_v on ECU_v during cycle i of ECU_v, i.e. for $t \in (e_v(i), e_v(i+1)]$. If in cycle $i-1$ the digital clock enable $Sce(h_v^{i-1})$ signal was off, we see during the whole cycle the old digital value $h_v^{i-1}.S$ of the register. If the update enable signal was on, then during the propagation delay tpd we cannot predict what we see, which we denote by Ω. When the propagation delay has passed, we see the new digital value of the register, which is equal to the digital input $Sdin(h_v^{i-1})$ during the previous cycle (see Figure 12).

$$S_v(t) = \begin{cases} h_v^{i-1}.S & \neg Sce(h_v^{i-1}) \\ \Omega & Sce(h_v^{i-1}) \wedge t \leq e_v(i) + tpd \\ Sdin(h_v^{i-1}) & Sce(h_v^{i-1}) \wedge t > e_v(i) + tpd \end{cases}$$

The $fbus$ is an open collector bus modeled for all t by:

$$fbus(t) = \bigwedge_v S_v(t)$$

Now consider a receiver register R_u on ECU_u whose clock enable is continuously turned on; thus the register always samples from the $fbus$. In order to define the new digital value $h_u^j.R$ of register R during cycle j on ECU_u we have to consider the value of the $fbus(t)$ in the time interval $(e_u(j) - ts, e_u(j) + th)$, i.e. from the clock edge minus the set-up time until the clock edge plus the hold time. If during that time the $fbus$ has a constant digital value x, the register samples that value:

$$\exists x \in \{0,1\}\ \forall t \in (e_u(j) - ts, e_u(j) + th) : fbus(t) = x \Rightarrow h_u^j.R = fbus(e_u(j))$$

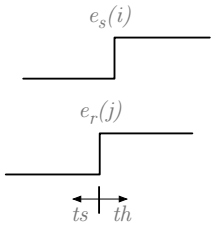

Fig. 13. Clock Edges

Otherwise we define $h_u^j.R = \Omega$.

Thus we still have to argue how to deal with unknown values Ω as input to digital hardware. We will use the output of register R only as input to a second register $\hat{R}$ whose clock enable is always turned on, too. If Ω is clocked into $\hat{R}$ we assume that $\hat{R}$ has an unknown but digital value:

$$h_u^j.R = \Omega \Rightarrow h_u^{j+1}.\hat{R} \in \{0, 1\}$$

Indeed, in real systems the counterpart of register $\hat{R}$ exists. The probability that R becomes metastable for an entire cycle *and* that this causes $\hat{R}$ to become metastable too is for practical purposes zero. This is exactly what has been formalized above. Note that our model uses different but fixed individual clock periods τ_v.

There is no problem to extend the model to deal with jitter. Let $\tau_v(i)$ denote the length of cycle i on ECU_v, then we require for all v and i:

$$\tau_v(i) \in [\tau_{ref} \cdot (1 - \delta), \tau_{ref} \cdot (1 + \delta)]$$

The time $e_v(i)$ of the i-th clock edge on ECU_j is then defined as:

$$e_v(i) = \begin{cases} c_v & i = 0 \\ e_v(i-1) + \tau_v(i-1) & \text{otherwise} \end{cases}$$

This does not complicate the subsequent theory significantly.

11.2 Continuous Time Lemmas for the Bus

Consider a pair of ECUs, where ECU_s is the sender and ECU_r is a receiver in a given slot. Let i be a sender $cycle$ such that $Sce(h_s^{i-1}) = 1$, i.e. the output of S is not guaranteed to stay constant at time $e_s(i)$. This change can only affect the value of register R of ECU_r in cycle j if it occurs before the sampling edge $e_r(j)$ plus the hold time th, i.e. $e_s(i) < e_r(j) + th$. Figure 13 shows a situation where due to a hold time violation we have $e_s(i) > e_r(j)$. The first cycle which can possibly be affected is denoted by:

$$cy_{r,s}(i) = \min\{j \mid e_s(i) < e_r(j) + th\}$$

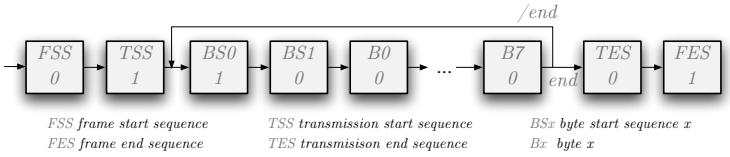

Fig. 14. Frame Encoding

In what follows we assume that all ECUs other than the sender unit ECU_s put the value 1 on the bus (hence $fbus(t) = S_s(t)$ for all t under consideration) and we consider only one receiving unit ECU_r. Because the indices r and s are fixed we simply write $cy(i)$ instead of $cy_{r,s}(i)$.

There are two essential lemmas whose proof hinges on the continuous time model. The first lemma considers a situation, where we activate the clock enable Sce of the sender ECU in cycle $i-1$ but not in the following seven cycles. In the digital model we then have $h_s^i.S = \ldots = h_s^{i+7}.S$ and in the continuous time model we observe $x = fbus(t) = S_v(t) = h_s^i.S$ for all $t \in [e_s(i) + tpd, e_s(i+8)]$. We claim that x is correctly sampled in at least six consecutive cycles

Lemma 3 (Correct Sampling Interval). *Let the clock enable signal of the S register be turned on in cycle $i-1$, i.e. $Sce(h_s^{i-1}) = 1$ and let the same signal be turned off in the next seven cycles, i.e. $Sce(h_s^j) = 0$ for $j \in \{i, \ldots, i+6\}$ then:*

$$h_r^{cy(i)+k}.R = h_s^i.S \quad \textit{for } k \in \{1, \ldots, 6\}$$

The second lemma simply bounds the clock drift. It essentially states that within 300 cycles clocks cannot drift by more than one cycle; this is shown using $\delta \leq 0.15\%$.

Lemma 4 (Bounded Clock Drift). *The clock drift in the interval $m \in \{1, \ldots, 300\}$ is bounded by:*

$$cy(i) + m - 1 \leq cy(i+m) \leq cy(i) + m + 1$$

Detailed proofs of very similar lemmas are to be found in [**?**,BBG+05], a formal proof is reported in [**?**].

11.3 Serial Interface Construction and Correctness

Recall that for natural numbers n and bits y we denote by y^n the string in which bit y is replicated n times, e.g. $0^4 = 0000$. For strings $x[0:k-1]$ consisting of k bits $x[i]$ we denote by $8 \cdot x$ the string obtained by repeating each bit eight times:

$$8 \cdot x = x[0]^8 \cdots x[k-1]^8$$

Our serial interface transmits messages $m[0:\ell-1]$ consisting of ℓ bytes $m[i]$ from a send buffer sb of the sending ECU to a receive buffer rb of the receiving ECU.

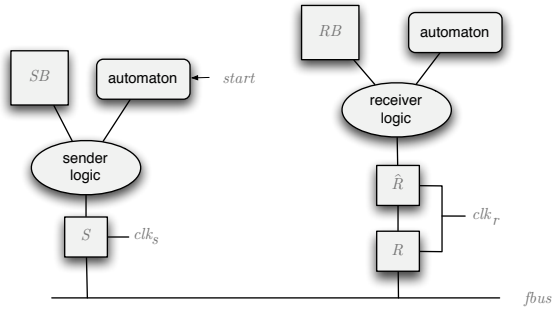

Fig. 15. Send and Receive Buffer

The following protocol is used for transmission (see Figure 14). One creates from message m a frame $f(m)$ by inserting falling edges between the bytes and adding some bits at the start and the end of the frame:

$$f(m) = 0110m[0] \cdots 10m[\ell - 1]01$$

In $f(m)$ one calls the first zero the transmission start sequence (TSS), the first one the frame start sequence (FSS), the last zero the frame end sequence (FES) and the last one the transmission end sequence (TES). The two bits producing a falling edge before each byte are called the byte start sequence $(BS0, BS1)$. The sending ECU broadcasts $8 \cdot f(m)$ over the $fbus$.

Figure 15 shows a simplified view on the hardware involved in the transmission of a message. On the sender side, there is an automaton keeping track which bit of the frame is currently being transmitted. This automaton inserts the additional protocol bits around the message bytes. Hardware for sending each bit eight times and for addressing the send buffer is not shown.

On the receiver side there is the automaton from Figure 14 (the automaton on the sender side is very similar) trying to keep track which bit of the frame is currently transmitted. That it does so successfully requires proof.

The bits sampled in register $\hat{R}$ are processed in the following way. The voted bit v is computed by applying a majority vote to the last five sampled bits. These bits are given by the $\hat{R}$ register and a 4-bit shift register as depicted in Figure 16.

According to Lemma 3 for each bit of the frame a sequence of at least six bits is correctly sampled. The filtering essentially maintains this property. If the receiver succeeds to sample that sequence roughly in the middle, he wins. For this purpose the receiver has a modulo-8 counter trying to keep track which of the eight identical copies of a frame bit is currently transmitted (see Figure 17). When the counter value equals four a strobe bit is produced. For frame decoding the voted bit is sampled with the strobed bit. The automaton trying to keep track of the protocol is also clocked with this strobe bit.

Clocks are drifting, hence the hardware has to perform a low level synchronization. The counter is reset by a $sync$ signal in two situations: at the beginning of a transmission

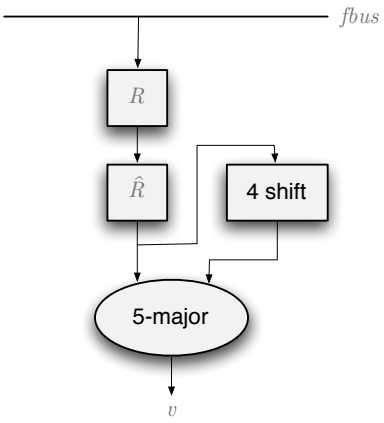

Fig. 16. Receiver Logic

or at an expected falling edge during the byte start sequence. Abbreviating signals $s(h_r^i)$ with s^i we write

$$sync^i = (idle^i \vee BS0^i) \wedge (\neg v^i \wedge v^{i-1})$$

The crucial part of the correctness proof is a lemma where one proves simultaneously three statements by induction over the receiver cycles:

1. the state of the automaton keeps track of the transmitted frame bit.
2. the $sync$ signal is activated at the corresponding falling edge of the voted bit between $BS0$ and $BS1$
3. sequences of identical bit are sampled roughly in the middle.

We sketch the proof of this lemma. Statement 1 is clearly true in the $idle$ state. From statement 1 follows that the automaton expects the falling edges of the voted signal exactly when the sender generates them. Thus the counter is well synchronized after these falling edges. This shows statement 2. Immediately after synchronization the receiver samples roughly in the middle. There is a synchronization roughly every 80 sender cycles. By Lemma 4 and because $80 < 300$, the sampling point can wander by at most one bit between activations of the $sync$ signal. This is good enough to stay within the correctly sampled six copies. This shows statement 3. If transmitted frame bits are correctly sampled, then the automaton keeps track of them. This shows statement 1. A formal proof of such a lemma in an abstract model obtained largely by automatic methods is reported in [**?**], a formal proof of the lemma in our hardware model with all gates and registers is reported in [**?**].

Let t_0 be the time (not the cycle) when the $start$ signal of the sender is activated. Let t_1 be the time, when all automata have reached the $idle$ state again and all write accesses to the receive buffer have completed. Let

$$tc = 45 + 80 \cdot \ell$$

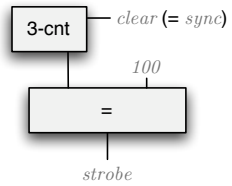

Fig. 17. Strobe Signal

be the number of 'transmission cycles'. The correctness of message transmission is stated as follows:

Lemma 5 (Correct Message Transfer With Time Bound). *Messages are correctly transmitted, and the transmission does not last longer than* tc *sender cycles:*

1. $rb(t_1) = sb(t_0)$
2. $t_1 - t_0 \leq tc \cdot \tau_s$

Intuitively, the product $80 \cdot \ell$ in the definition of tc comes from the fact that each byte produces 10 frame bits and each of these is transmitted 8 times. The four bits added at the start and the end of the frame contribute $4 \cdot 8 = 32$. The remaining 13 cycles are caused by delays in the receiver logic, in particular by delay in the shift register before the majority voter.

12 FlexRay Like Interfaces and Clock Synchronization

Using the serial interfaces from the last section we can proceed to construct the hardware of entire f-interfaces. The results from this section were first reported in [lecture notes, wilhelm paper]

12.1 Hardware Components

Recall that we denote hardware configurations of ECU_v by h_v. If the index v of the ECU does not matter, we drop it. The hardware configuration is split into a processor configuration $h.p$ and an interface configuration $h.f$. In addition to the registers of the serial interface, the essential components of the hardware configuration $h.f$ of our (non fault tolerant) FlexRay like interface are

1. double buffers $h.f.sb(par)$ and $h.f.rb(par)$, where $par \in \{0, 1\}$, implementing the user visible send and receive buffers,
2. the registers of a somewhat non trivial timer $h.f.timer$,
3. configuration registers.

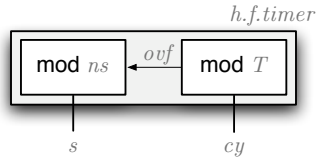

Fig. 18. Hardware Timer

The construction of the hardware timer $h.f.timer$ is sketched in Figure 18. The low order bits $h.f.timer.cy$ count the cycles of a slot. Unless the timer is synchronized, slots have locally T cycles, thus the low order bits are part of a modulo-T counter. The high order bits $h.f.timer.s$ count the slot index s of the current slot (r, s) modulo ns. The timer is initialized with the value $(ns - 1, T - 1)$.

The timers on all ECUs but $ECU_{send(0)}$ stall when reaching the maximum value $(ns-1, T-1)$ and wait for synchronization. The timer on $ECU_{send(0)}$ always continues counting. Details regarding the synchronization mechanism are given in Section 12.2.

The overflow signal $ovf(h)$ between the low order and the high order bits of the counter can essentially serve as the timer interrupt signal $ti(h)$ generated by the interface hardware[10]:

$$ti(h^i) = ovf(h^i) \wedge \neg ovf(h^{i-1})$$

The low order bit of the slot counter keeps track of the parity of the current slot and is called the hardware parity signal:

$$par(h) = h.f.timer.s[0]$$

In general the $fbus$ side of the interface will see the copies $h.f.sb(par(h))$ and $h.f.rb(par(h))$. Messages are always transmitted between these copies of the buffers. The processor on the other hand writes to $h.f.sb(\neg par(h))$ and reads from $h.f.rb(\neg par(h))$. This does not work at boundaries of rounds unless the number of slots ns is even.

The configuration registers are written immediately after reset / power-up. They contain in particular the locally relevant portions of the scheduling function. Thus if ECU_v is (locally) in a slot with slot index s and $send(s) = v$ then ECU_v will transmit the content of the send buffer $h.f.sb(par(h))$ via the $fbus$ during some transmission interval $[ts(r, s), te(r, s)]$. A serial interface which is not actively transmitting during slot (r, s) puts by construction the idle value (the bit 1) on the bus.

If we can guarantee that during the transmission interval *all* ECUs are locally in slot (r, s), then transmission will be successful by Lemma 5. The clock synchronization algorithm together with an appropriate choice of the transmission interval will guarantee exactly that.

[10] In general one needs to keep an interrupt signal active until it is cleared by software; the extra hardware is simple.

12.2 Clock Synchronization

The idea of clock synchronization is easily explained: Imagine one slot is one hour and one round is one day. Assume different clocks drift by up to $drift = 5$ minutes per day. ECUs synchronize to the first bit of the message transmission due between midnight and 1 o'clock. Assume adjusting the clocks at the receiving ECUs takes up to $adj = 1$ minute. Then the maximal deviation during 1 day is $off = drift + adj = 6$ minutes. $ECU_{send(s)}$, which is the sender in hour s, is on the safe side if it starts transmitting from s o'clock plus off minutes until off minutes before $s+1$ o'clock, i.e. somewhen in between $s : 06$ o'clock and $s + 1 : 54$ o'clock.

At midnight life becomes slightly tricky: $ECU_{send(0)}$ waits until it can be sure that everybody believes that midnight is over and hence nobody is transmitting, i.e. until its local time $0 : 06$. Then it starts sending. All other ECUs are waiting for the broadcast message and adjust their clocks to midnight + off = $0 : 06$ once they detect the first falling bit. Since that might take the receiving ECUs up to 1 minute it might be $0 : 07$ o'clock on the sender when it is $0 : 06$ o'clock at the receiver; thus after synchronization the clocks differ by at most $adj = 1$ minute.

We formalize this idea in the following way: Assume without loss of generality that $send(0) = 0$. All ECUs but ECU_0 synchronize to the transmission start sequence (TSS) of the first message of ECU_0. When ECU's waiting for synchronization ($d.f.timer = (ns-1, T-1)$) receive this TSS, they advance their local slot counter to 0 and their cycle counter to off. Analysis of the algorithm will imply that for all $v \neq 0$, ECU_v will be waiting for synchronization, when ECU_0 starts message transmission in any slot $(r, 0)$.

First we define the start times $\alpha_v(r, s)$ of slot (r, s) on ECU_v. This is the start time of the first cycle t in round r when the timer in the previous cycle had the value:

$$h^{t-1}.f.timer = ((s-1 \mod ns), T-1)$$

This are the cycles immediately after the local timer interrupts. For every round r, we also define the cycles $\beta_v(r)$ when the synchronization is completed on ECU_v. Formally this is defined as the first cycle $\beta > \alpha_v(r, 0)$ such that the local timer has value

$$h^{\beta}.f.timer = (0, off)$$

Timing analysis of the synchronization process in the complete hardware design shows that for all v and y adjustment of the local timer of ECU_v to value $(0, off)$ is completed within an adjustment time $ad = 15 \cdot \tau_y$ after $\alpha_0(r, 0)$

$$\begin{aligned} \beta_0(r) &= \alpha_0(r, 0) + off \cdot \tau_0 \\ \beta_v(r) &\leq \beta_0(r) + 15 \cdot \tau_y \end{aligned}$$

For $s \geq 1$ no synchronization takes place and the start of new slots is only determined by the progress of the local timer:

$$\alpha_v(r, s) = \begin{cases} \beta_v(r) + (T - off) \cdot \tau_v & s = 1 \\ \alpha_v(r, s-1) + T \cdot \tau_v & s \geq 2 \end{cases}$$

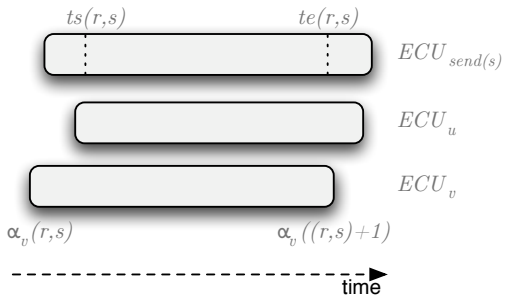

Fig. 19. Schedules

ECU_0 synchronizes the other ECUs. Thus the start of slot $(r, 0)$ on ECU_0 depends only on the progress of the local counter:

$$\alpha_0(r, 0) = \alpha_0(r - 1, ns - 1) + T \cdot \tau_0$$

An easy induction on s bounds the difference between start times of the same slot on different $ECUs$:

$$\begin{aligned} \alpha_x(r, s) - \alpha_v(r, s) &\leq 15 \cdot \tau_v + (s \cdot T - off) \cdot (\tau_x - \tau_v) \\ &\leq 15 \cdot \tau_v + (ns \cdot T \cdot \Delta \cdot \tau_v) \\ &= \tau_v \cdot (15 + (ns \cdot T \cdot \Delta)) \\ &= \tau_v \cdot off \end{aligned} \tag{2}$$

Thus we have $off = ad + drift$ with $ad = 15$ and $drift = ns \cdot T \cdot \Delta$.

Transmission is started in slots (r, s) by $ECU_{send(s)}$ when the local cycle count is off. Thus the transmission start time is

$$ts(r, s) = \alpha_{send(s)}(r, s) + off \cdot \tau_{send(s)}$$

By Lemma 5 the transmission ends at time

$$\begin{aligned} te(r, s) &= ts(r, s) + tc \cdot \tau_{send(s)} \\ &= \alpha_{send(s)}(r, s) + (off + tc) \cdot \tau_{send(s)} \end{aligned}$$

The transmission interval $[ts(r, s), te(r, s)]$ must be contained in the time interval, when all ECUs are in slot (r, s), as depicted in Figure 19. Thus we need for all indices v and u of ECUs:

Lemma 6 (No Bus Contention).

$$\begin{aligned} \alpha_x(r, s) &\leq ts(r, s) \\ te(r, s) &\leq \alpha_u((r, s) + 1) \end{aligned}$$

The first inequality holds because of (2). Let $v = send(s)$:

$$\begin{aligned}\alpha_x(r,s) &\leq \alpha_v + \tau_v \cdot off \\ &= ts(r,s)\end{aligned}$$

The second inequality determines the minimal size of T:

$$\begin{aligned}te(r,s) &\leq \alpha_v(r,s) + (off + tc) \cdot \tau_v \\ &\leq \alpha_u(r,s) + off \cdot \tau_u + (off + tc) \cdot (1 + \Delta) \cdot \tau_u \\ &\leq \alpha_u((r,s)+1) \\ &= \alpha_u(r,s) + T \cdot \tau_u\end{aligned}$$

Further calculations are necessary at the borders between rounds. Details can be found in [**?**].

From the local start times of slots $\alpha_v(r,s)$ we get to numbers of local start cycles $t_v(r,s)$ using Equation 1.

$$\alpha_v(r,s) = c_v + t_v(r,s) \cdot \tau_v$$

and then solving for $t_v(r,s)$. Trivially the number $u_v(r,s)$ of the locally last cycle on ECU_v is

$$u_v(r,s) = t_v((r,s)+1) - 1$$

Consider slot (r,s) and let $v = send(s)$. Lemma 5 and Lemma 6 then imply that the value of the send buffer of ECU_v on the network side ($par = s \bmod 2$) at the start of slot (r,s) is copied to the all receive buffers on the network side by the end of that slot

Lemma 7 (Message Transfer With Cycles). *Let $v = send(s)$. Then for all u:*

$$h_v^{t_v(r,s)}.f.sb(s \bmod 2) = h_u^{u_v(r,s)}.f.rb(s \bmod 2)$$

This lemma talks only about digital hardware and hardware cycles. Thus we have shown the correctness of data transmission via the bus *and* we are back in the digital world.

13 Integrating f-Interfaces into the ISA

In Section 2 we have developed an ISA model for processors with devices. So far we have collected many device specific results for ECUs connected by an $fbus$. Thus there is not terribly much left to be done in order to integrate f-Interfaces into the ISA.

13.1 Specifying Port RAM

If a processor accesses a device with K I/O ports, then for $k = \lceil \log K \rceil - 2$ the device configuration (here $ecu.f$) must contain a memory

$$ecu.f.m : \{0,1\}^k \rightarrow \{0,1\}^8$$

In our case the memory of the device contains the send buffer, the receive buffer –each with ℓ bytes– and say c configuration registers. Thus we have

$$K = 2 \cdot \ell + 4 \cdot c$$

We use the first ℓ bytes of this memory for the send buffer, the next ℓ bytes for the receive buffer and the remaining bytes for the configuration registers. We formalize this by defining for all indices of message bytes $y \in \{0, \ldots, \ell - 1\}$:

$$sb(d)(y) = d.f.m(y)$$
$$rb(d)(y) = d.f.m(\ell + y)$$

In the absence of timer interrupts the ports are quiet. Thus, as long as no timer interrupt occurs, we can use for the ECU the generic ISA model from Section 2.

13.2 Timer Interrupt and I/O

As pointed out earlier, at the ISA level the timer interrupt must betreated as an oracle input eev. Furthermore we have to deal with external data input $fdin$ from the f-interface. Thus –ignoring reset– the next state function for the device has on the ISA level the format

$$ecu' = \delta_D(ecu, eev, fdin)$$

If we denote by eev^i and $fdin^i$ the oracle input and the input from the $fbus$ for the i-th executed instruction, then we get computations $ecu^0, ecu^1, \ldots$ by defining (again straight from the automata theory textbooks):

$$ecu^{i+1} = \delta_D(ecu^i, eev^i, fdin^i)$$

In our distributed system we have configurations ecu_v from many ECUs. Within this programming model we now introduce names $j_v(r, s)$ for certain indices of local *instructions* on ECU_v. Intuitively, the timer interrupts the instruction executed in local configuration $d_v^{j_v(r,s)}$ of ECU_v, and this locally ends slot (r, s). By the results of Section 3, this is the instruction scheduled in the write back stage WB in the last cycle $u_v(r, s)$ (defined in Section 12.2) of slot (r, s) on ECU_v.

$$j(r, s) = s(WB, u_v(r, s)) \tag{3}$$

Note that in every cycle an instruction is scheduled in every stage. Nevertheless, due to pipeline bubbles, the write back stage might be empty in cycle $u_v(r, s)$. In this situation the scheduling functions, by construction, indicates the next instruction to arrive which is not there presently. We require interrupt event signals be only cleared by software, hence the hardware interrupt signal will stay active in cycles following $u_v(r, s)$. Thus Equation 3 also holds in this case.

This was the crucial step to get from the cycle level to the instruction level. Purely within the ISA model we continue to define:

1. $i_v(r, s) = j_v((r, s) - 1) + 1$: the index of the first local instruction in slot (r, s)

2. $d_v(r,s) = ecu_v^{i_v(r,s)}$: the first local ISA configuration in slot (r,s)
3. $e_v(r,s) = ecu_v^{j_v(r,s)}$ the last local ISA configuration in slot (r,s)

We can even define the sequence $eev(r,s)$ of oracle timer inputs eev^i where $i \in \{i_v(r,s), \ldots, j_v(r,s)\}$. It has the form

$$eev(r,s) = 1^a 0^b 1$$

where the timer interrupt is cleared by software instruction $i_v(r,s)+a-1$ and $a+b+1 = j_v(r,s) - i_v(r,s) + 1$ is the number of local instructions in slot (r,s).

Indeed we can complete, without any effort, the entire ISA programming model. The effect of an interrupt on the processor configuration has been defined in the previous section, thus we get for instance

$$\begin{aligned} ecu_v(r,s).d.dpc &= 0^{32} \\ ecu_v(r,s).d.pc &= 0^{30}10 \end{aligned}$$

Also for the transition from $e_v(r,s)$ to $d_v((r,s)+1)$ and only for this transition we use the external input

$$fdin^{j_v(r,s)} \in \{0,1\}^{8 \cdot \ell}$$

Thus we assume that it consists of an entire message and we copy that message into the user visible receive buffer

$$rb(ecu_v((r,s)+1)) = fdin^{j_v(r,s)}$$

Of course we also know what this message should be: the content of the user visible send buffer of $ECU_{send(s)}$ at the end of slot $(r,s)-1$:

$$fdin^{j_v(r,s)} = sb(ecu_{send(s)}((r,s)-1)$$

Thus

Theorem 5.

$$rb(ecu_v((r,s)+1)) = sb(evu_{send(s)}((r,s)-1))$$

This completes the user visible ISA model. And with Lemma 7 we essentially already completed the hardware correctness proof of the implementation of (**??**). The nondeterminism is completely encapsulated in the numbers $j_v(r,s)$ as it should be, at least if the local computations are fast enough. All we need to do is to justify the model by a hardware correctness theorem and to identify the conditions under which it can be used.

13.3 Hardware Correctness of the Parallel System

For a single slot (r,s), and a single processor with an f-interface, the generic hardware correctness statement from Section 4.4 translates into Theorem 6 below. It is proven by induction over the cycles of the slot. Recall from Sect. 12.2 that we know already

the start *cycles* $t_v(r,s)$ for all ECUs. The statement of the theorem is identical for all ECU_v. Thus we drop the subscript v.

The theorem assumes that at the start of the theorem the pipe is drained and that the simulation relation between the first hardware configuration $h(r,s) = h^{t(r,s)}$ and the first ISA configuration $d(r,s) = d^{i(r,s)}$ of the slot holds.

Theorem 6 (Hardware Correctness for One Slot). *Let the pipeline of the processor be drained ($drained(h(r,s))$) and let the simulation relation hold at the beginning of a round, i.e. $sim(d(r,s), h(r,s))$.*

Then for all $t \in \{t(r,s), \ldots, t((r,s)+1)-1\}$, for all stages k and for all registers R with $stage(R) = k$:

$$\begin{aligned} h^t.p.R &= ecu^{s(k,t)}.p.R \\ m(h^t.p) &= ecu^{s(mem1,t)}.p.m \\ h^t.f.sb(\neg par(h^t)) &= sb(ecu^{s(mem1,t)}) \\ h^t.f.rb(\neg par(h^t)) &= rb(ecu^{s(mem1,t)}) \end{aligned}$$

Then we can show

Theorem 7 (Hardware Correctness for System).

$$\forall (r,s), v : drained(h_v(r,s)) \wedge sim(d_v(r,s), h_v(r,s))$$

Theorem 7 is proven by induction over the slots (r,s). In order to argue about the boundaries between two slots Theorem 6 and Lemma 6 must be applied on the last cycle of the previous slot.

14 Pervasive Correctness Proofs

Next, we show how pervasive correctness proofs for computations with timer interrupts can be obtained from (i) correctness proofs for ISA programs which cannot be interrupted (ii) hardware correctness theorems and (iii) worst case execution time (WCET) analysis. As one would expect, the arguments are reasonably simple, but the entire formalism of the last sections is needed in order to formulate them.

We consider only programs of the form[11]:

```
{
        : P;
  a     : jump a;
  a+4   : NOP;
}
```

The program does the useful work in portion P and then waits in the idle loop for the timer interrupt. P initially has to clear and then to unmask the timer interrupt, which is masked when P is started (see Sect. 13.2).

[11] Note that we have an byte addressable memory and that in an ISA with delayed branch the idle loop has two instructions.

14.1 Computation Theory

We have to distinguish carefully between the transition function $\delta_D(ecu, eev, fdin)$ of the interruptible ISA computation and the transition function $\delta_u(ecu)$ of the non interruptible ISA computation, which we define as follows:

$$\delta_u(ecu) = \delta_D(ecu, 0, *)$$

Observe that this definition permits the non interruptible computation to clear the timer interrupt bit by software. Non interruptible computations starting from configuration ecu are obtained by iterated application of δ_u:

$$\delta_u^i(ecu) = \begin{cases} ecu & i = 0 \\ \delta_u(\delta_u^{i-1}(ecu)) & \text{otherwise} \end{cases}$$

For the ISA computation

$$d(r,s) = ecu^{i(r,s)}, ecu^{i(r,s)+1}, \ldots, ecu^{j(r,s)} = e(r,s)$$

which has been constructed in the hardware correctness theorem we get

Lemma 8. *For all t such that $0 \leq t \leq j(r,s) - i(r,s)$:*

$$ecu^{i(r,s)+t} = \delta_u^t(d(r,s))$$

This lemma holds due to the definition of $j(r,s)$ and the fact that the timer is masked initially such that the instructions of the interruptible computation are not interrupted.

We define the ISA run time[12] $T_u(ecu, a)$ simply as the smallest i such that δ_u^i fetches an instruction from address a:

$$T_u(ecu, a) = \min\{i \mid \delta_u^i(ecu).p.dpc = a\}$$

Furthermore we define the result of the non interruptible ISA computation as

$$resu(ecu, a) = \delta_u^{T_u(ecu,a)}(ecu)$$

Correctness proofs for non interruptible computations can of course be obtained by classical program correctness proofs. They usually have the form $d \in E \rightarrow resu(d,a) \in Q$ or, written as a Hoare triple $\{E\}P\{Q\}$.

We assume that the definition of Q does not involve the PC and the delayed PC. Because the idle loop only changes the PC and the delayed PC of the ISA computation we can infer on the ISA level that property Q continues to hold while we execute the idle loop:

$$\forall i \geq T_u(ecu, a) : \delta_u^i(ecu) \in Q$$

[12] This is the time until the idle loop is reached

14.2 Pervasive Correctness

Assume $sim(ecu, h)$ holds. Then the ISA configuration ecu can be decoded from the hardware configuration by a function:

$$ecu = decode(h)$$

Clearly, in order to apply the correctness statement $\{E\}P\{Q\}$ to a local computation in slot (r, s), we have to show for the first ISA configuration in the slot:

$$d(r, s) \in E$$

In order to apply the processor correctness theorem the simulation relation must hold initially:

$$sim(d(r, s), h(r, s))$$

Now consider the last hardware configuration $U(r, s) = h^{t((r,s)+1)-1}$ of the slot. We want to conclude

Theorem 8. *The decoded configuration obeys the postcondition Q:*

$$decode(u(r, s)) \in Q$$

This only works if portion P is executed fast enough on the pipelined processor hardware.

14.3 Worst Case Execution Time

We consider the set $H(E)$ of all hardware configurations h encoding an ISA configuration $d \in E$:

$$H(E) = \{h \mid dec(h) \in E\}$$

While the decoding is unique, the encoding is definitely not. Portions of the ISA memory can be kept in the caches in various ways.

For a hardware configuration $h = h^0$ we define the hardware run time $T_H(h, a)$ until a fetch from address a as the smallest number of cycles such that in cycle t an instruction, which has been fetched in an earlier cycle $t' < t$ from address a, is in the write back stage WB. Using scheduling functions this definition is formalized as

$$T_H(h, a) = \min\{t \mid \exists t' : s(WB, t) = s(IF, t') \wedge h^{t'}.dpc = a\}$$

Thus for ISA configurations satisfying E we define the worst case execution time $WCET(E, a)$ as the largest hardware runtime $T_H(h, a)$ of a hardware configuration encoding a configuration in E:

$$WCET(E, a) = \max\{T_H(h, a) \mid h \in H(E)\}$$

As pointed out earlier such estimates can be obtained from (sound!) industrial tools based on the concept of abstract interpretation [**?**]. Assume we have

$$WCET(E, a) \leq T - off$$

Now consider a local computation within slot (r, s) starting in hardware configuration $h(r, s) = h^{t(r,s)}$ and in ISA configuration $d(r, s) = ecu^{i(r,s)}$. If this computation is run for hardware run time many cycles $T_H(h(r, s), a) < T - off$, then the computation is not interrupted and the instruction in the write back stage (at the end of the computation) is the first instruction being fetched from a. By the definition of the ISA run time this is exactly instruction $i(r, s) + T_u(d(r, s), a)$, thus we conclude:

$$s(WB, t(r, s) + T_H(h(r, s), a)) = i(r, s) + T_u(d(r, s), a)$$

Let $h' = h^{t(r,s)+T_H(h(r,s),a)}$ be the hardware configuration in this cycle and let $d' = d^{i(r,s)+T_u(d(r,s),a)} = resu(d(r, s), a)$ be the ISA configuration of the instruction in the write back stage.

In this situation the pipe is almost drained. It contains nothing but instructions from the idle loop. Thus the processor correctness theorem $sim(ecu', h')$ holds for all components of the configuration but the PC and the delayed PC. Therefore we weaken the simulation relation sim to a relation $dsim$ by dropping the requirement that the PCs and delayed PCs should match:

$$dsim(ecu', h')$$

Until the end of the slot in cycle $t(r, s) + T$ and instruction $j(r, s)$, only instructions from the idle loops are executed. They do not affect the $dsim$ relation, hence

$$dsim(j(r, s), U(r, s))$$

Since $resu(d(r, s), a) \in Q$ and Q does not depend on the program counters we have $j(r, s) \in Q$. We derive that $decode(U(r, s))$ coincides with $j(r, s)$ except for the program counters. And again, because this does not affect the membership in Q, we get the desired theorem.

15 The Distributed OSEKTime Like Operating System D-OLOS

15.1 D-OLOS Configuration

As in the previous sections we consider p electronic control units ECU_i, where $i \in [0 : p - 1]$. On each ECU_i there are n_i user processes $vm(i, j)$, where $j \in [0 : n_i - 1]$, running under the real time operating system OLOS. These user programs are compiled C0 programs. We denote the source program for $vm(i, j)$ by $C(i, j)$.

On each ECU_i application programs $C(i, j)$ can access via system calls a set of messages buffers $MB(i)(k)$. Messages come in nm many different types. For $k \in [0 : nm - 1]$ messages of type k are stored in message buffers $MB(i)(k)$. Thus each ECU_i is capable of storing one message of each type in its message buffers $MB(i)(k)$. These message buffers are the direct counterparts of the $FTCom$ buffers in OSEK. However we do not support fault tolerance, yet. Messages between different ECU's are exchanged via an $fbus$ using f-interfaces. The drivers for these interfaces are part of OLOS.

As before time is divided into rounds r each consisting of a fixed number ns of slots s, The scheduling of all applications $C(i,j)$ as well as the inter ECU communication procedure via the $fbus$ is identical in each round r and only depends on the slot index s. From the standpoint of a C0 application programmer a D-OLOS configuration $dolos$ representing the global state of the distributed has the following components:

- $dolos.C(i,j)$ is the configuration of an abstract C0 machine representing application program $C(i,j)$ for $i \in [0:p-1]$ and $j \in [0:n_i-1]$.
- $dolos.MB(i)(k)$ is the k-th message in the message buffer of ECU_i.
- The index of the current slot is given by $dolos.s$.
- $dolos.bus$ holds the message value of the message currently being broadcast.

15.2 Scheduling and Communication

For slots (r,s) we denote by $D(r,s)$ resp. $E(r,s)$ the D-OLOS configuration at the start resp. at the end of slot (r,s). The message on the bus is constant during each slot (r,s). Thus it equals $D(r,s).bus$.

Scheduling of applications and communication between message buffers on different ECUs is identical for all rounds r and depends only on the slot index s. It is determined by three functions.

- The scheduling of all applications is defined by the global scheduling function run with $run(i,s) \in [0:n_i-1]$. For all i and s this function returns the index of the application being executed in slots (r,s) on ECU_i. Thus application $C(i,run(i,s)$ is running on ECU_i during slots (r,s). The state of applications, which are not running, does not change during a slot

$$j \neq run(i,s) \rightarrow E(r,s).C(i,j) = D(r,s).C(i,j)$$

- As before functions $send$ with $send(s) \in [0:p-1]$ gives the index of the ECU sending during slots (r,s).
- Function $mtype$ with $mtype(s) \in [0:nm-1]$ gives the type of the message transmitted over the $fbus$ during slots (r,s). The message $bus(r,s)$ is the content of message buffer with index $mtype(s)$ of $ECU_{send(s)}$ at the end of the previous slot

$$bus(r,s) = E((r,s)-1).MB(send(s), mtype(s))$$

  At the start of the next slot, message $bus(r,s)$ is copied into all message buffers with index $mtype(s)$

$$\forall i : D((r,s)+1).MB(i, mtype(s)) = D(r,s).bus$$

15.3 Local Computation

For each ECU_i and slot (r,s) we have to define the effect of the application $C(i,run(i,s)$ on the corresponding C0 configuration $dolos.C(i,run(i,s))$ and on the local message buffers $dolos.MB(i)(k)$. We define a *local configuration* lc as a pair with the following components

1. a C0 configuration $lc.c$ of a local application
2. a set of local message buffers $lc.M(k)$ with $k \in [0 : nm - 1]$.

and a local transition function $lc' = \delta_{LC}(lc)$. The C0 programs running under the local operating system (OLOS) can read and write $MB(k)$ using two system-calls:

1. $ttsend(k, msg)$: The execution of this function results in copying the value of the C0 sub-variable with identifier msg into $MB(k)$. Let $MSG = va(lc.c, msg)$ and $K = va(lc.c, k)$ be current values of msg and k. Then

$$lc.c.pr = ttsend(k, msg); r \rightarrow lc'.M(K) = MSG \wedge lc'.c.pr = r$$

2. $ttrec(k, msg)$: At invocation of this function the C0 sub-variable having the identifier msg is updated with the value of $MB(k)$. Let $K = va(lc.c, k)$. Then

$$lc.c.pr = ttrec(k, msg); r \rightarrow va(lc'.c, msg) = M(K) \wedge lc'.c.pr = r$$

OLOS offers a third call named $ttex$. An application invoking this system-call indicates that it has completed its computation for the current slot and wants to return the control back to the operating system. The execution of system call $ttex$ on the local configuration is like a NOOP:

$$lc'.c.pr = ttex; r \rightarrow lc'.pr = r$$

If the program rest does not start with one of the system calls, then an ordinary C0 instruction is executed and the message buffers stays unchanged:

$$lc' = \delta_C(lc.c, lc.M)$$

We define run tim (measured in C instructions) and result of a local computation in the usual way

$$\begin{aligned} T_C(lc) &= \min\{t : \exists r : \delta^t_{LC}(lc).pr = ttex; r\} \\ res_{LC}(lc) &= \delta^{T_C(lc)}_{LC}(lc) \end{aligned}$$

and complete the definition of the D-OLOS semantics with the help of the result of local computations. Let $j = run(i, s)$. Then

$$(E(r, s).C(i, j), E(r, s).MB(i)) = res_{LC}D(r, s).C(i, j), D(r, s).MB(i)$$

Let us consider a situation where the application code is wrapped by a $while$-$loop$ and that $ttExFinished()$ is invoked only once as the last statement of the loop body:

$$while(true)\ \{\ \text{“Application Code”}\ ;\ ttExFinished()\ \}$$

In this case we enforce the application code to be executed once each time the application is scheduled. Intuitively, from the applications programmers point of view, the $ttExFinished()$ system-call does nothing but to wait till the application is scheduled again.

16 D-OLOS Implementation

Consider any ECU. We implement the local version OLOS of D-OLOS by specializing the abstract kernel of CVM. The only device of CVM is an f-interface. The ISA programs of the user virtual machines are obtained by compiling the local application programs. Among others the abstract kernel uses the following variables and constants

- constant own of the kernel stores the index of the local ECU.
- C0 implementations of the functions run, $send$ and $type$
- An integer variable s keeps track of the current slot.
- The content of local message buffers are stored in an array $M[0 : nm - 1]$ of messages

16.1 Invariants

Let $cvm = (cvm(0), \ldots, cvm(p-1))$ be a sequence of CVM configurations. On cvm_i we will make use of n_i user virtual machines, one for each application on ECU_i. An obvious simulation relation $osim(aba)(dolos, cvm)$ is parameterized by a sequence aba of allocation functions $aba(i, j)$. For each ECU_i we require:

1. the kernel keeps track of the D-OLOS slot

$$va(cvm(i).c, s) = dolos.s$$

2. The application scheduled by D-OLOS is running

$$cvm(i).cp = run(i, dolos.s)$$

3. the user processes of CVM encode the applications of D-OLOS

$$\forall j < n_i : consis(aba(i, j))(dolos.C(i, j), cvm(i).vm(j))$$

4. the content of the message buffers of D-OLOS are stored in the corresponding variables of the abstract kernel

$$\forall k : va(cvm(i), MB[k]) = dolos.MB(i)(k)$$

For arguments at slot boundaries we need to define for ECU indices i and slots (r, s) the first CVM configuration $dcvm(i)(r, s)$ and the last CVM configuration $ecvm(i)(r, s)$ of $cvm(i)$ in slot (r, s). Slot boundaries are defined by timer interrupts.

Because the CVM primitive $wait$ is interruptible by timer interrupts, one has to extend the sequence $eev(i)^t$ of oracle interrupt event signals also for the situation, when the current process of CVM is the abstract kernel, i.e. a C0 program, and the program rest starts with $wait$; for a simulation theorem one then has - as in section x - to construct a sequence $eev(i)^t$ such that the simulation theorem works. Because the kernel computation gets stuck if the program rest starts with the $wait$ primitive one then can easily show: if user processes on ECU_i are not interrupted during slot (r, s) then $dcvm(i)(r, s)$ is the first configuration in slot (r, s) such that $cvm(i).c.pr = wait; r'$ for some r. The first configuration after the timer interrupt is defined in a similar way as the cvm configuration after a trap instruction in subsection y.

At slot boundaries we then can define two more 'communication' invariants:

1. If $i = send(s)$, then the send buffer $sb(ecvm(i)((r,s)-1.f))$ on ECU_i at the end of the previous slot is the message on $dolos.bus$ during slot (r,s)

$$i = send(s) \rightarrow sb(ecvm(i)((r,s)-1.f)) = D(r,s).bus$$

2. The receive buffer $rb(dcvm(i)(r,s)$ on every ECU_i at the beginning of slot (r,s) is the message on $dolos.bus$ during the previous slot

$$\forall i : rb(dcvm(i)(r,s)) = D((r,s)-1).bus$$

16.2 Construction of the Abstract OLOS Kernel

Assume that all invariants hold for slot $(r,s)-1$. We construct the abstract OLOS kernel such that they are maintained during slot (r,s). One round of a CVM computation on an ECU proceeds in three phases as shown in figure x. In phases 1 and 3 the kernel runs; in phase 2 a user process runs and makes system calls. The following happens in phase 1.

1. the kernel runs. It increments s. Then part 1 of $osim$ holds.
2. a driver using variants of the *copy* primitives of CVM copies the local receive buffer into variable $MB(type(own, s-1))$. This implies that part 4 of $osim$ holds after phase 1.
3. The next process to be started is computed by $cup = run(own, s)$ and the CVM primitive $start(cup)$ is executed. This implies that part 2 of $osim$ holds after phase 1.

During phase 2 we only have to worry about the running process. It is easy to implement the handlers for system calls $ttsend$ and $ttrec$ with the help of variants of the CVM $copy$ primitive such that parts 3 and 4 of $osim$ hold.

Phase 2 ends by a system call $ttex$ of the application returning control to the kernel again. The kernel determines whether its ECU is the sender in the next slot

$$send(s+1) = own?$$

If this is the case it copies the content of variable $MB(mtype(s+1))$ into the local send buffer. This implies part 1 of the communication invariant. In any case the kernel then executes the wait primitive and idles waiting for the end of the round.

Worst case execution time analysis for an ECU must consider all assembler programs runnig on the ECU: the compiled concrete kernel as well as the compiled user programs. Address a from subsection .. is the address, where the compiled concrete kernel starts waiting for the timer interrupt.

Theorem 4 then implies part 2 of the communication invariant.

17 The Auto Focus Task Model (AFTM)

17.1 Configurations

AFTM is a computational model for a restricted version of the AutoFocus CASE tool []. The restrictions were aimed at making the implementation of AFTM by D-OLOS

efficient. As many high level CASE tools AFTM programs can be modeled by a certain number M of communicating 'task' automata $T(i)$. For technical reasons we one adds a automaton $T(M+1)$ which always generates output. We number the automata with indices $i \in [1 : M+1]$. Each automaton with index i has $nip(i)$ input ports $IP(i,j)$ with $j \in [0 : nip(i)-1]$ as well as $nop(i)$ output ports $OP(i,j)$ with $j \in [0 : nop(i)-1]$. A function src (for source) specifies for each input port $OP(i,j)$ the index $(i',j') = src(i,j)$ of the output port such that $OP(i',j')$ is connected to $IP(i,j)$. An AFTM configuration $aftm$ has the following components:

1. $aftm.S(i)$: the state of the i'th task automaton. It is split into subcomponents i) a control component $aftm.S(i).con$ and ii) data components $aftm.S(i).x$, which are later stored in a set V of C0 variables. Each automaton has a control state called $idle$.
2. $aftm.IP(i,j)$: the current value of input port $IP(i,j)$.
3. $aftm.OP(i,j)$: the current value of output port $IP(i,j)$ Input and output ports can hold non empty values or a special empty value ϵ.

Initially (in configuration $aftm^0$) all automata are in idle state and all ports are empty. Indices i of tasks are partitioned into three classes: indices of AND-tasks, OR tasks and the special 'environment' automaton $T(M+1)$.

$$[1 : M+1] = T_{and} \uplus T_{or} \uplus \{M+1\}$$

17.2 Local and Global Next State computation

One defines when a task i is runnable in configuration $aftm$ by a predicate $runnable(aftm, i)$.

Definition:

1. the environment task is always runnable:

$$\forall aftm : runnable(aftm, M+1)$$

2. OR-tasks are runnable if one of their inputs are non empty:

$$i \in T_{or} \Rightarrow runnable(aftm, i) \Leftrightarrow \exists j : aftm.IP(i,j) \neq \epsilon$$

3. AND-tasks are runable. if all their inputs are non empty

$$i \in T_{and} \Rightarrow runnable(aftm, i) \Leftrightarrow \forall j : aftm.IP(i,j) \neq \epsilon$$

For AFTM configurations $aftm$ we define the configuration $aftm'$ after the next AFTM step. AFTM computations are then defined in the usual way by

$$aftm^{r+1} = (aftm^r)'$$

In AFTM, each step consists of two phases. In the first phase all runnable tasks make locally a number of micro steps until their control reaches the idle state again. In the

second phase values of non empty output ports are copied into the connected input ports and all output ports are cleared. [13] Formalization of this model is straight forward.

Assume local computation is specified by a 'local autofocus' transition function δ_{LAF} mapping states S and a vector IP of input port contents to states T' and a vector of output port contents OP'

$$(S', OP') = \delta_{LAF}(T, OP)$$

The local run time $T_{LAF}(aftm, i)$- in automata steps - of runnable task t in configuration $aftm$ is defined as

$$T_{LAF}(aftm, i) = \min\{t : \delta^t_{LAF}(aftm.S(i), aftm.IP(i)).S.con = idle\}$$

and the result of this local computation is

$$res_{LAF}(aftm, i) = \delta^{T_{AF}(aftm,i)}_{LAF}(aftm.S(i), aftm.IP(i))$$

We define the configuration $aftm$" after the local computations by:

1. for runnable tasks state and output ports are determined by the result of local computations. Input ports are cleared $runnable(aftm, i) \rightarrow$

$$(aftm".T(i), aftm".OP(i)) = res_{LAF}(aftm, i) \wedge$$
$$\forall j : aftm.IP(i)(j) = \epsilon$$

2. state and output ports of non runnable tasks don't change. Input ports are not cleared and (thus can continue to accumulate inputs for AND tasks in the communication phase). $\not runnable(aftm, i) \rightarrow$

$$(aftm".T(i), aftm".OP(i)) = (aftm.T(i), aft.OP(i)) \wedge$$
$$\forall j : aftm".IP(i, j) = aftm.IP(i, j)$$

In the communication phase non empty contents of output ports are copied into connected input ports. Let $src(i, j) = (i'j')$. Then

$$aftm'.IP(i, j) = \begin{cases} aftm".OP(i', j') & \text{if} aftm".OP(i', j') \neq \epsilon \\ aftm.IP(i, j) & \text{otherwise} \end{cases}$$

All output ports are cleared

$$\forall i, j : aftm'.OP(i, j) = \epsilon$$

Local state does not change during the communication phase

$$\forall i : aftm".T(i) = aftm'.T(i)$$

[13] An easy exercise shows that this model is equivalent to the model described in [].

18 Simulation of AFTM by D-OLOS

18.1 C0 Code Generation for Local Computation

We define a local configuration T of a task automaton TA as a triple with the following components: i) state $T.S$, content of input ports $T.IP[0 : nip - 1]$ and iii) content of output ports $T.OP[0 : nop - 1]$.

In order to implement a single task automaton TA as a process in OLOS, we first need a C0 program $prog(TA)$ which simulates local runs of the automaton in the following sense:

I/O: Inputs are read from a C array $IP[0 : nip - 1]$ and outputs are written to another C-array $OP[0 : nop-1]$. Access to these arrays is restricted to assignments of the form $e = IP[e']$ for input and $O[e'] = e$ for output operations; here e and e' are expressions. This restriction makes it later easy to replace these assignments by Operating system calls like $ttrec(e, e')$; the replacement will however be very slightly more involved.

Data: each data component $S.x$ of the state has its counter part in a C variables with the name x.

Simulation Relation: recall that for C0 configurations c and expressions e we denote by $va(c, e)$ the value of expression e in configuration c. A trivial simulation relation $afsim(T, c)$ between T and c is established by requiring for all i and x

$$\begin{aligned} T.OP[i] &= va(c, OP[i]) \\ T.IP[i] &= va(c, IP[i]) \\ T.x &= va(c, x) \end{aligned}$$

Specification of the local program: Program $prog(TA)$ is specified by the following requirement: assume $afsim(T, c)$ holds. Moreover assume that both the automaton and the C machine are in their initial states; for the C machine this means that the program rest is the body of the main function, formally to be found in the function table FT at argument $main$, component $body$ of the C-configuration.

$$\begin{aligned} T.con &= idle \\ c.pr &= FT(main).body \end{aligned}$$

Then we simply require that the simulation relation holds for the results of the computations

$$sim(res_C(c), res_{LAF}(T))$$

There are several ways to produce the program $prog(TA)$ from the task automaton TA. One can generate the program by hand or by a translation tool. Also, one can do the correctness proof by hand or one can get it by an automatic translation validation tool. For a program generated by a a verified program generation tool, one would of course need no further correctness proof. But to the best of our knowledge no such tool exists yet. Note that in any case we are dealing so far with plain C code verification only.

18.2 Deployment

We will simulate each step of AFTM by one round consisting of ns slots of D-OLOS. Thus, we will be interested to relate $aftm^r$ with $D(r,0)$. In order to deploy an AFTM machine on a COA machine we have to specify several things:

Task deployment for each AFTM task $T(i)$ we have to specify the CO application $C(i,j)$ which simulates the task. Let p be the number of ECUs and N the maximum number of task executable on an ECU. Then this will be done with an injective task deployment function

$$depl : [[1 : M+1] \to [0 : p-1] \times [0 : N-1]$$

Application Scheduling for every ECU number i and for every slot s we have to specify the CO application $run(i,s)$ running on $ECU(i)$ during slot s:

$$run : [0 : p-1] \times [0 : ns-1] \to [0 : N-1]$$

This defines for each task $T(k)$ and round r a slot $start(k) < ns$, such that task $T(k)$ is simulated in slot $start(k)$ of the round: if $depl(k) = (i,j)$ then

$$start(k) = s \leftrightarrow run(i,s) = j$$

Output Port Broadcasting Recall that for each AFTM task $T(i)$ we denote by $nip(i)$ resp. $nop(i)$ the number output ports resp. input ports of task $T(i)$. The set of all indices of output ports is denoted by

$$OP = \bigcup_i \{i\} \times [0 : nop(i) - 1]$$

We denote by N the cardinality of this set. For each pair of indices $(i,j) \in OP$ specifying output port j of task i we specify a function

$$broad : OP \to [0 : ns-1]$$

During each round r we plan to broadcast $(aftm^r)".OP(i,j)$ (i.e. the content of port $OP(i,j)$ after the local computation phase of macro step r)in slot $broad(i,j)$ of the half round.

We require in each round, that any output port $OP(i,j)$ of task i is only broadcast after the task has run:

$$\forall i,j : broad(i,j) > start(i)$$

Obviously we need $ns \geq N+1$. This is the only restriction we impose on schedules. Schedules will tend to be shorter if tasks with many output ports are scheduled earlier than tasks with few output ports.

18.3 Mapping Output Ports on Message Buffer Entries

We reuse function $broad$ to map the output ports $OP(i,j)$ with their two dimensional index structure to the D-OLOS message buffers $MB(k)$ with their one dimensional index structure. Thus, contents of output port $OP(i,j)$ will be stored in elements $MB[broad(i,j)]$. Equivalently: output ports broadcast in slot s are stored on each ECU in message buffers $MB(s)$.

18.4 Invariants

At the slot boundaries we maintain four invariants between the AFTM configurations $aftm^r, (aftm^r)"$ and the corresponding D-OLOS configuration $D(r,s)$. For all slots (r,s), for all indices $(i,j$ of output ports $OP(i,j)$ and for all indices e of ECUs:

1. consider an output port $OP(i,j)$ and the message buffers $MB(broad(i,j))$ reserved for storing values of port $OP(i,j)$. Before or while $OP(i,j)$ is scheduled for broadcast, the message buffers contain the value of $OP(i,j)$ before the local computation phase of step r. Afterwards they have the value after the local computation phase. There is however an exception. On the ECU_e where task i is deployed (formally: e is the first component of $depl(i)$) the new values are already in the local message buffers after the task has been simulated:

$$D(r,s)(e).MB(broad(i,j)) =$$
$$\begin{cases} (aftm^{r+1})".OP(i,j) & s > broad(i,j) \vee s > start(i) \wedge e = depl(i)[1] \\ (aftm^r)".OP(i,j) & \text{otherwise} \end{cases}$$

2. Consider a data component $aftm(i).S.x$ of task i and the C0 variable x of the application $C(depl(i))$ where task i is simulated. Until the task is scheduled for simulation, the value of the variable is the value of x before the step r. Otherwise it is the value after step r which is the same as the value before the next step:

$$va(D(r,s).C(depl(i)),x) = \begin{cases} aftm^r(i).S.x & start(i) \leq s \\ aftm^{r+1}(i).S.X & \text{otherwise} \end{cases}$$

3. The invariants given so far do not suffice to infer the input buffers $aftm(i).IP(i,j)$ from the message buffers for slots $s = start(i)$. Let $(i',j') = src(i,j)$ and assume that $broad(i',j') < start(i)$. Then the output port value $(aftm^r)".OP(i',j')$ needed for the computation of input port value $aftm^r(i).IP(i,j)$ is already overwritten in the message buffers. Therefor we save in the previous round the endangered value $(aftm^r)".OP(i',j')$ into a 'shadow message buffer' variable $SMB(i',j')$ of the application. We define a predicate $q(i',j')$ stating that a shadow message buffer is needed for $OP(i',j')$ by:

$$q(i',j') \leftrightarrow broad(i',j') < start(i)$$

and require

$$q(i',j') \rightarrow va(D(r,s).C(depl(i)), SMB(i',j') = (aftm^r)".OP(i',j')$$

Initially the shadow buffers must be set to ϵ

4. finally we must track the accumulation of values in the input ports $IP(i,j)$ of AND tasks. This is done in array elements $I[j]$ of application $C(depl(i))$. Even if task i is not runnable in step r the simulation must update array I. We require

$$va(D(r,s).C(depl(i)), I(j)) = \begin{cases} (aftm^{r-1})".IP(i,j) & s \leq start(i) \\ (aftm^{r})".IP(i,j) & \text{otherwise} \end{cases}$$

18.5 Construction of D-OLOS Applications

When application $C(depl(i)$ is started in slot $(r, start(i))$, we first simulate the communication phase at the end of the previous slot.

Communication phase: The input port values $aftm^r(i).IP(i,j)$ at the start of step r are computed in C0 array IP. Let $k(i,j) = broad(src(i,j))$ be the index of message buffers, where values of the output port $OP(src(i,j))$ connected to $IP(i,j)$ are stored. Then for each j the new values of $IP(j)$ is computed as follows. The current content of message buffer $MB(k(i,j))$ is accessed with a ttrec system call. If $q(i,j) = 0$ it is directly stored in $IP(j)$ by the system call

$$ttrec(k(i,j), IP(j))$$

Otherwise shadow register $SMB(i,j)$ is copied into $IP(j)$ and then $MB(k(i,j))$ is copied into the shadow register:

$$IP(j) = SMB(i,j); ttrec(k(i,j), SMB(i,j))$$

In the C0 configurations c after execution of these pieces of code we conclude from the invariants for the previous slot:

$$va(c, IP(j)) = aftm^r.IP(i,j)$$

and that invariant 3 holds. Next, we clear all entries $OP(j)$ in the C0 array of output values. For configurations c after this code holds

$$va(c, OP(j)) = aftm^r.OP(i,j)$$

Local computation Next we test wether task i is runnable; if so we run program $prog(T(i))$. For configurations c after execution of this piece of code we conclude that invariant 2 holds and that array OP holds the values of the output ports $OP(i,j)$ after the local computation phase

$$va(c, OP(j)) = (aftm^r)".OP(i,j)$$

For runnable tasks we clear the input array IP, for non runnable tasks, the input array stays unchanged. From this we conclude invariant 4.

Updating the message buffers Using the $ttsend$ system call the new values of the output ports are copied into their message buffers

$$ttsend(broad(i,j), OP(j))$$

After this invariant 1 holds.

References

[BBG+05] S. Beyer, P. Böhm, M. Gerke, M. Hillebrand, T. In der Rieden, S. Knapp, D. Leinenbach, and W.J. Paul. Towards the formal verification of lower system layers in automotive systems. In *23nd IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD 2005), 2–5 October 2005, San Jose, CA, USA, Proceedings*, pages 317–324. IEEE, 2005.

[Bey05] Sven Beyer. *Putting It All Together: Formal Verification of the VAMP*. PhD thesis, Saarland University, Computer Science Department, March 2005.

[BJK+03] Sven Beyer, Christian Jacobi, Daniel Kröning, Dirk Leinenbach, and Wolfgang Paul. Instantiating uninterpreted functional units and memory system: Functional verification of the VAMP. In D. Geist and E. Tronci, editors, *Proc. of the 12th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME)*, Lecture Notes in Computer Science (LNCS), pages 51–65. Springer, 2003.

[Dal06] Iakov Dalinger. *Formal Verification of a Processor with Memory Management Units*. PhD thesis, Saarland University, Computer Science Department, July 2006.

[DHP05] Iakov Dalinger, Mark Hillebrand, and Wolfgang Paul. On the verification of memory management mechanisms. In D. Borrione and W. Paul, editors, *Proceedings of the 13th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME 2005)*, volume 3725 of *LNCS*, pages 301–316. Springer, 2005.

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

[GHLP05] Mauro Gargano, Mark Hillebrand, Dirk Leinenbach, and Wolfgang Paul. On the correctness of operating system kernels. In J. Hurd and T. F. Melham, editors, *18th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2005)*, volume 3603 of *LNCS*, pages 1–16. Springer, 2005.

[Hil05] Mark Hillebrand. *Address Spaces and Virtual Memory: Specification, Implementation, and Correctness*. PhD thesis, Saarland University, Computer Science Department, June 2005.

[HIP05] Mark Hillebrand, Thomas In der Rieden, and Wolfgang Paul. Dealing with I/O devices in the context of pervasive system verification. In *ICCD '05*, pages 309–316. IEEE Computer Society, 2005.

[HP96] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Mateo, CA, second edition, 1996.

[HW73] C. A. R. Hoare and Niklaus Wirth. An axiomatic definition of the programming language PASCAL. *Acta Informatica (ACTA)*, 2:335–355, 1973.

[Kna05] Steffen Knapp. Towards the Verification of Functional and Timely Behavior of an eCall Implementation. Master's thesis, Universität des Saarlandes, 2005.

[KP06] Steffen Knapp and Wolfgang Paul. Realistic worst case execution time analysis in the context of pervasive system verification. 2006. To appear.

[LPP05] Dirk Leinenbach, Wolfgang Paul, and Elena Petrova. Towards the formal verification of a C0 compiler: Code generation and implementation correctness. In Bernhard Aichernig and Bernhard Beckert, editors, *3rd International Conference on Software Engineering and Formal Methods (SEFM 2005), 5-9 September 2005, Koblenz, Germany*, pages 2–11, 2005.

[MP00] Silvia M. Müller and Wolfgang J. Paul. *Computer Architecture: Complexity and Correctness*. Springer, 2000.

[NN99] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: A Formal Introduction*. John Wiley & Sons, Inc., New York, NY, USA, 1992, revised online version: 1999.

[Nor98] Michael Norrish. C formalised in HOL. Technical Report UCAM-CL-TR-453, University of Cambridge, Computer Laboratory, December 1998.

[OSE01a] OSEK group. *OSEK/VDX fault tolerant communication*, 2001. `http://portal.osek-vdx.org/files/pdf/specs/ftcom10.pdf`.

[OSE01b] OSEK group. *OSEK/VDX time-triggered operating system*, 2001. `http://portal.osek-vdx.org/files/pdf/specs/ttos10.pdf`.

[Rus94] John Rushby. A formally verified algorithm for clock synchronization under a hybrid fault model. In *PODC '94: Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing*, pages 304–313, New York, NY, USA, 1994. ACM Press.

[Sch87] Fred B. Schneider. Understanding protocols for byzantine clock synchronization. Technical report, Department of Computer Science, Ithaca, NY, USA, 1987.

[SH98] Jun Sawada and Warren A. Hunt. Processor verification with precise exceptions and speculative execution. In Alan J. Hu and Moshe Y. Vardi, editors, *CAV '98*, pages 135–146. Springer, 1998.

[SK06] Wilfried Steiner and Hermann Kopetz. The startup problem in fault-tolerant time-triggered communication. In *DSN '06: Proceedings of the International Conference on Dependable Systems and Networks (DSN'06)*, pages 35–44, Washington, DC, USA, 2006. IEEE Computer Society.

[The00] The VERIFIX Consortium. The VERIFIX Project. `http://www.info.uni-karlsruhe.de/~verifix/`, 2000.

[The05a] The Verisoft Consortium. The Verisoft project. `http://www.verisoft.de/`, 2005.

[The05b] The Verisoft Subproject 6. Subproject 6: Automotive. `http://www.verisoft.de/TeilProjekt6.html`, 2005.

[Win93] G. Winskel. *The formal semantics of programming languages*. The MIT Press, 1993.