The Embedded Machine

Christoph Meyer Kirsch

Report No. UCB//CSD-01-1137

March 21, 2001

Computer Science Division (EECS)
University of California
Berkeley, California 94720

The Embedded Machine

Christoph Meyer Kirsch*
University of California, Berkeley

March 21, 2001

Abstract

The embedded machine is a virtual machine in the spirit of the Java virtual machine with specific extensions for embedded real-time computing on distributed platforms. The embedded machine provides an abstract platform for generating distributed code from high-level embedded programming languages. The instruction set of the embedded machine has a formal synchronous (zero-delay) semantics which provides synchronous control of scheduled computation and communication with respect to the progress of real-time and the occurrences of events. The serialization of concurrent scheduled computation and communication is defined non-deterministically which makes the embedded machine compatible with any scheduling algorithm. A program of the embedded machine determines when to schedule task invocations and message delivery but not how. A scheduling algorithm is thus a parameter of a program of the embedded machine.

1 Introduction

The embedded machine or E machine for short is a virtual machine in the spirit of the Java virtual machine (JVM) [LY99] with specific extensions for embedded real-time computing on distributed platforms. The E machine provides an abstract platform for generating distributed code from high-level embedded programming languages such as synchronous reactive languages [Hal93], e.g., Lustre [HCRP91] or Esterel [Ber00], or time-triggered languages like Giotto [HHK00]. A virtual machine not only supports the generation of portable code but also helps to identify the key services of target platforms which support the execution of a given class of programming languages. The main objective of this paper is to define a minimal set of instructions which are essential for distributed code generation for event- and time-triggered programming languages with explicit real-time constructs.

The instruction set of the E machine is called E code. Unlike Java bytecode, E code allows to specify the computational behavior of a system relative to the progress of time and the occurrences of events. The E machine controls the execution of tasks and the delivery of messages. In fact, the E machine can be seen as a meta machine controlling the execution of other non-embedded machines which execute tasks and transmit messages. A *task* in this model is single-threaded code without any synchronization points but with known worst-case execution time (WCET) [TFW00]. The communication to and from a task is not performed by the task but by the E machine prior to task execution and after task completion. Similarly, we model messages as tasks with known worst-case latency (WCL) where, however, the emphasis is not on the code of a message but on its input and output interface which effectively determines sender and receiver.

*This work has been supported by Boeing on DARPA SEC grant F33615-99-C-1500.

The E machine abstracts from the services of a real-time operating system. In the semantics of the E machine we distinguish *synchronous* and *scheduled* computation. Synchronous computation is performed logically with zero time delay whereas scheduled computation takes time. In an implementation of the E machine, synchronous computation may be performed in the kernel context whereas scheduled computation may be done in the user context. From the perspective of scheduled computation the activity of the kernel, i.e., of the E machine, is instantaneous whereas the kernel sees user activity as scheduled computation which takes time. In an implementation, special care has to be taken to enforce this semantics, e.g., any portion of the memory accessed by a task should not be accessible to any other tasks.

Synchronous computation in the E machine is sequential whereas scheduled computation is concurrent. It is thus necessary to serialize scheduled computation which is usually done in online systems using priorities. A scheduling algorithm computes the required priority assignment. The E machine uses the synchronous semantics to specify the scheduled reaction of a system to the progress of time and the occurrences of events. A *scheduled reaction* is the result of scheduled computation or communication. Serialization of scheduled computational activity in the E machine is defined non-deterministically which makes the E machine compatible with any scheduling algorithm. A program of the E machine determines when to schedule task invocations and message delivery but not how. A scheduling algorithm is thus a parameter of E code.

The E machine has an *environment* and an *output* interface as well as an *internal* memory. Interfaces and memory are sets of ports. A *port* is a variable with finite type and a unique identifier. The value of a port in the environment interface is determined by the physical environment of the E machine. The port is read-only for the machine. On the other hand, the value of a port in the output interface is set by the E machine and may affect the physical environment. The port is read-only for the environment. The E machine has read and write access to its internal memory which is not observable by the environment. We distinguish signal and value ports. A *signal port* is an integer port whose valuations are non-decreasing with respect to the progress of time. A *signal counter* is a valuation of a signal port. We require that signal counters are increased at most by one at any instance. We speak of the occurrence of a *signal* whenever a signal counter is increased by one. A *value port* is an uninterpreted port with arbitrary type. Thus value ports in the environment interface may model state whereas signal ports may model change of state. If a signal port in the environment interface is driven by a real-time clock its signal counter corresponds to *absolute* time.

A configuration of the E machine contains a schedule and valuations for all ports of the machine. A schedule is a list of triggers which determines when to invoke E code of the machine. A trigger (s, m, l) consists of a signal port s, a signal counter m, and an E code program address l. A trigger is *active* when the signal counter of s reaches m. Then the E machine executes the E code at l. If s is driven by a real-time clock, m refers to the absolute time of this clock. The key instruction of the E machine is the *embedded jump*. It inserts new triggers into the schedule of the machine. An embedded jump has as arguments a signal port s, a signal counter n, and a program address l. The signal counter n determines relative to the current signal counter k of s how long to wait before invoking the E code at l. It will add a trigger $(s, k + n, l)$ to the schedule. Multiple active triggers are executed in the order of the execution of the according embedded jumps. If s is driven by a real-time clock, n refers to *relative* time. E code uses relative time which will be translated to absolute time during the execution of the code.

There are two call instructions to invoke tasks and to transmit messages, respectively. In order to execute tasks, the *synchronous* call instruction invokes a task and blocks until the execution of the task is completed. The *scheduled* call instruction invokes a task and then proceeds immediately to the next instruction. We assume that some scheduling algorithm assigns a priority to this task. The result of the scheduled invocation will be available some time later. We model message delivery in a similar way. Note that emitting signals by an E machine is only possible through scheduled invocations. A signal counter of a given signal port may be increased upon completion of scheduled computation but not synchronous computation. Note also that the E machine only checks the presence of signals but the absence as opposed to the synchronous reactive semantics, e.g., of Esterel.

The E machine is related to automata-based approaches like the object code (OC) [PS98] for generating code from Lustre and Esterel. An OC program is an automaton whose state transitions determine the reaction of a system with respect to the set of all signals in the system without keeping track of signal counters. The E machine, on other hand, may schedule reactions with respect to a single signal and its signal counter. Conceptually, the E machine implicitly partitions the states of an OC program into several states with possibly less transitions. In particular, in the context of real-time clocks, the E machine allows to code specific time-triggered reactions.

For distributed code generation we define the distributed E machine to be the parallel composition of E machines. The semantics of the E machine readily carries over to the distributed case. Communication between E machines is modeled by identifying ports of the output interfaces. Common output value ports of multiple E machines model frames send on the networks between the machines. A frame is the largest non-preemptive sequence of bits which can be send on a network. Writing to a common output value port corresponds to sending a frame.

In the following section we introduce preliminary definitions which will be used throughout the paper. In Section 3 we define the E machine and E machine configurations. Section 4 describes abstract and concrete syntax as well as the semantics of each instruction of the E machine. In the second part of Section 4 we formally define the semantics of the E machine. In Section 5 we define the distributed E machine which is a parallel composition of multiple E machines. We conclude the paper in Section 6 with a discussion of future work.

2 Preliminaries

In this section we define the notions of *ports*, *programs*, and *schedules* which will be used throughout the paper.

2.1 Ports

A port is a variable with finite type and a unique location in some shared memory. Note that we use shared memory as a logical concept not an implementation.

Definition 2.1 (Port)
A *port* is a tuple $(a, \mathbb{T})$ consisting of (1) an *address* a and (2) a *type* $\mathbb{T}$. An address is a non-negative integer. A type is a finite set of values. We require that any two distinct ports have different addresses.

Memory is a finite set of ports. We may call memory also interface depending on its usage.

Definition 2.2 (Memory)
Memory is a finite set mem of ports. We define a function *addresses* which returns the set of addresses of all ports in mem by $addresses(mem) = \{a | (a, \mathbb{T}) \in mem\}$. An address a is called *valid* in mem if $a \in addresses(mem)$. We use $*(a, mem)$ to denote the port $(a, \mathbb{T}) \in mem$ for a valid address a in mem.

We define valuations as relations between port addresses and values for notational convenience although valuations are actually functions assigning unique values to port addresses.

Definition 2.3 (Valuation)
Let $(a, \mathbb{T})$ be a port p. A *valuation* for p is a tuple (a, v) with $v \in \mathbb{T}$. Let *mem* be some memory. A *valuation* for *mem* is a set μ of valuations for all ports in *mem* such that for all $(a, v_1) \in \mu$ and for all $(b, v_2) \in \mu$ with $v_1 \neq v_2$ we have $a \neq b$. A valuation induces a function from addresses to values. If $(a, v) \in \mu$ then $\mu(a)$ maps a to the value v. We define a function *addresses* which returns the set of addresses of all ports in a valuation μ by $addresses(\mu) = \{a|(a, v) \in \mu\}$.

We define an update function for valuations in order to replace subsets of valuations by other valuations.

Definition 2.4 (Update)
Let μ be a valuation for some memory *mem* and let mem_r be a subset of *mem*. Let A be the set $addresses(mem_r)$ of addresses of mem_r. We define the restriction $\mu|_A$ of μ to be the valuation $\{(a, v)|(a, v) \in \mu, a \in A\}$. The extension $\mu|_{mem_r}$ to restrictions on memories is given by $\mu_{addresses(mem_r)}$. Let ν be a valuation for mem_r. We define the function *update* which replaces the valuations in μ for mem_r by valuations in ν by $update(\mu, \nu) = (\mu \setminus (\mu|_{addresses(\nu)})) \cup \nu$. Note that $update(\mu, \nu)$ is a valuation whenever μ and ν are valuations.

The following function allows to increase the signal counters of signal ports.

Definition 2.5 (Increase)
Let μ be a valuation for some memory *mem* and let mem_s be a subset of all signal ports in *mem*. Let S be the set $addresses(mem_s)$ of addresses of mem_s. The function *increase*, given by $increase(\mu, S) = update(\mu, \{(a, v + 1)|(a, v) \in \mu|_S\})$, increases the signal counters for mem_s in μ by one.

2.2 Programs

We restrict the number of maximum number of arguments of a machine instruction to three for notational convenience.

Definition 2.6 (Instruction)
An *instruction* is a tuple (ic, i_1, i_2, i_3) consisting of (1) an *instruction code ic*, (2) an *argument* i_1, (3) an *argument* i_2, and (4) an *argument* i_3. An instruction code is a three-letter string. We define a mapping of instruction codes to integer opcodes in the appendix in Section 7. An argument is a non-negative integer. We use $\texttt{str}(i_1)(i_2)(i_3)$ to denote an instruction $(\texttt{str}, i_1, i_2, i_3)$. If the instruction only uses one or two arguments we write $\texttt{str}(i_1)$ or $\texttt{str}(i_1)(i_2)$, respectively. If the instruction does not require an argument at all we write `str`.

A program is a finite list of instructions assigning a unique program address to each instruction. The program address of the first instruction may be any non-negative integer which we call the offset of the program.

Definition 2.7 (Program)
A *program eco* is a finite list $\langle ins_o, \ldots, ins_n\rangle$ of instructions ins_i for $o \leq i \leq n$. We call o the *offset* and n the *end* of *eco*. We use $*(i, eco)$ to denote the instruction ins_i. The index i is called a *program address*. A program address a is called *valid* in *eco* if there is an instruction ins_a in *eco*. A *program counter PC* is a non-negative integer.

The control state of a program associates non-negative integers with program addresses in the program. We use the control state for dereferencing the arguments of indirect jump instructions.

Definition 2.8 (Control State)
Let *eco* be a program and let l be a valid program address in *eco*. Let i be a non-negative integer. An *indirect jump address* for i in *eco* is a tuple (i, l). The *control state* of *eco* is a set ρ of indirect jump addresses in *eco*. Note that we will apply the update function for valuations to update the control state.

A stack is a list of non-negative integers. The E machine maintains a single LIFO stack.

Definition 2.9 (Stack)
A *list* lst is a finite sequence $\langle l_0, \ldots, l_n \rangle$ of elements l_i for $0 \leq i \leq n$. The empty list is denoted by $\langle\rangle$. We use $\circ$ to concatenate lists. We denote membership by $l \in lst$. We extend set exclusion to lists in a straight-forward way. For a given subset S of all elements of lst, the list $lst \setminus S$ denotes the list lst without the elements of S. A *stack* stk is a list $\langle int_0, \ldots, int_n \rangle$ of non-negative integers int_i for $0 \leq i \leq n$.

2.3 Schedules

A trigger associates E code with the state of a signal counter of a signal port. A trigger determines which E code is to be executed given the value of a signal counter of a signal port. A schedule is a list of triggers. It is possible that multiple triggers in a schedule are *active* at the same instance.

Definition 2.10 (Schedule)
Let *eco* be a program and let l be a valid program address in *eco*. Let $(s, \mathbb{T})$ be a signal port. Let n be a non-negative integer. A *trigger* on s is a tuple (s, n, l). A *schedule* on some memory *mem* of signal ports is a list τ of triggers on the ports in *mem*.

A scheduled reaction determines when to complete scheduled computation or communication. A distributed reaction is a set of scheduled reactions. It is possible that multiple scheduled reactions in a distributed reaction will be completed at the same instance.

Definition 2.11 (Distributed Reaction)
Let $(a, \mathbb{T}_a)$ and $(b, \mathbb{T}_b)$ be two signal ports. Let S either be the empty set or the singleton set $\{b\}$. Let f be a function defined on a set *in* of ports and let ν be a valuation of *in*. Let n be a non-negative integer. A *scheduled reaction* on a is a tuple (a, n, f, ν, S). A *distributed reaction* on some memory *mem* of signal ports is a set δ of scheduled reactions on the ports in *mem*.

A scheduled reaction formalizes the completion of scheduled computation and communication. We call a set of scheduled reactions a distributed reaction because in an implementation of the E machine, multiple scheduled reactions at the same instance are only possible on different machines of a distributed E machine. On a single E machine there can only be a single scheduled reaction at any instance because there can only be a single task or message being completed at any instance.

3 The Embedded Machine

The E machine is an abstract stack machine which executes E code from a program *eco* on three different types of memories. The *environment interface* of the E machine is a set *env* of *environment ports* whose values are set by the physical environment of the E machine. For the E machine the

env interface is read-only. The environment interface is partitioned into the *signal interface* env_s of signal ports in env and the *value interface* env_v of value ports in env, i.e., env is the disjoint union $env_s \cup env_v$.

The *internal memory* of the E machine is a set int of *internal ports* of the E machine. We assume that the int memory is not observable by the environment or any other E machine. We partition int into the *signal memory* int_s and *value memory* int_v. A task scheduled by the eco program may only read from a subset of the int memory and write to a subset of the int_v memory.

The *output interface* of the E machine is a set out of *output ports*. For the environment the out interface is read-only. We partition out into the *signal memory* out_s and *value memory* out_v. A message scheduled by the eco program may either read from a subset of the env and out interface and write to a subset of the int_v memory, or else read from a subset of the int memory and write to a subset of the out_v interface.

We denote the overall E machine memory $env \cup int \cup out$ by mem which is partitioned into *signal memory* mem_s and *value memory* mem_v, i.e., mem is the disjoint union $mem_s \cup mem_v$ where $mem_s = env_s \cup int_s \cup out_s$ and $mem_v = env_v \cup int_v \cup out_v$. The E machine computes valuations for the int memory and out interface from valuations for all ports of the overall memory mem.

Definition 3.1 (Embedded Machine)
An *embedded machine* (E machine) M is a tuple (eco, env, int, out) which consists of (1) a program eco, (2) an interface env, (3) a memory int, and (4) an interface out. We require $env \cap int = \emptyset$, $env \cap out = \emptyset$, and $int \cap out = \emptyset$.

The E machine maintains a valuation μ for its memory mem, a schedule τ of its triggers, a distributed reaction δ of its scheduled reactions, and a control state ρ. The schedule accumulates the completed and pending embedded jumps. The distributed reaction specifies when scheduled tasks and messages have finished and will finish. An indirect jump instruction of the E machine uses ρ to dereference its arguments, which is a non-negative integer, to an absolute program address.

Definition 3.2 (Configuration)
Let M be an embedded machine (eco, env, int, out). A *configuration* C of M is a tuple $(\mu, \tau, \delta, \rho)$ which consists of (1) a valuation μ for $env \cup int \cup out$, (2) a schedule τ, (3) a distributed reaction δ, and (4) a control state ρ of eco.

The instruction set and the formal semantics of the E machine are defined in the next section.

4 Semantics

For the definitions in this section let M be an embedded machine (eco, env, int, out). Let mem be the overall memory of M. The E machine internally maintains a program counter PC and a stack stk of non-negative integers. The E machine uses the stk stack to maintain jump addresses, signal counters, and other miscellaneous values. Configurations of M are denoted by c.

4.1 The E Code Semantics

We call the instruction set of the E machine *E code*. The function $exec$ defines the semantics of E code. It computes sets of E machine configurations. We require that E code does not contain infinite loops. Note that we do not cover error handling in the definition of the E machine. We begin with a definition of *synchronous* E code which consists of all instructions except the instructions for scheduled computation. In the logical semantics of the E machine, we require that synchronous

E code is executed instantaneously similar to the logical semantics of synchronous reactive languages. Note that instructions like push, pop, add, the no operation, the comparison, conditional and absolute jump instructions are standard machine instructions. These instruction are convenient but not necessary. Other choices are possible. The unique features of the E machine are the embedded jump instruction in combination with the synchronous and scheduled call instructions which we will use to control task invocations and message delivery.

Synchronous Computation. The synchronous call instruction $\texttt{com}(f)$ invokes the synchronous computation of an external function f. We require that f is computable and is defined on a subset of the internal ports of M. Thus f can neither see nor affect the environment directly.

Definition 4.1 (Synchronous Computation)
Let f be a computable function which maps valuations of its *input ports* in which are a subset of the internal ports int to valuations of its *output ports* which are a subset of the internal value ports int_v. We define the semantics of the instruction $\texttt{com}(f)$. We assume that $*(PC, eco) = \texttt{com}(f)$. Then:

$$exec(PC, stk, (\mu, \tau, \delta, \rho)) = exec(PC + 1, stk, (update(\mu, f(\mu|_{in})), \tau, \delta, \rho))$$

The push instruction `psh` pushes its argument onto the stack.

Definition 4.2 (Push)
Let i be a non-negative integer. We define the semantics of the instruction $\texttt{psh}(i)$. We assume that $*(PC, eco) = \texttt{psh}(i)$. Then:

$$exec(PC, stk, c) = exec(PC + 1, \langle i\rangle \circ stk, c)$$

The pop instruction `pop` removes the top value from the stack.

Definition 4.3 (Pop)
We define the semantics of the instruction `pop`. Let i be a non-negative integer. We assume that $*(PC, eco) = \texttt{pop}$. Then:

$$exec(PC, \langle i\rangle \circ stk, c) = exec(PC + 1, stk, c)$$

The add instruction $\texttt{add}(j)$ adds j to the top value i of the stack. Note that j may be negative. $\texttt{add}(j)$ removes i from the stack, adds j to it, and then pushes the result r back onto the stack. r is equal to $i + j$ if $i + j \geq 0$, or else r is zero.

Definition 4.4 (Add)
Let i a non-negative integer and let j be an integer. We define the semantics of the instruction $\texttt{add}(j)$. We assume that $*(PC, eco) = \texttt{add}(j)$. Then:

$$exec(PC, \langle i\rangle \circ stk, c) = exec(PC + 1, \langle i + j\rangle \circ stk, c)$$

if $i + j \geq 0$, or otherwise

$$exec(PC, \langle i\rangle \circ stk, c) = exec(PC + 1, \langle 0\rangle \circ stk, c)$$

The comparison instruction $\texttt{neq}(j)$ pushes a 1 onto the stack whenever the top value i on the stack is not equal to j. Otherwise, it pushes 0 onto the stack.

Definition 4.5 (Comparison)
Let i be a non-negative integer. We define the semantics of the instruction $\texttt{neq}(j)$. We assume that $*(PC, eco) = \texttt{neq}(j)$. Then:

$$exec(PC, \langle i\rangle \circ stk, c) = exec(PC + 1, \langle 0, i\rangle \circ stk, c)$$

if $i = j$, or otherwise

$$exec(PC, \langle i\rangle \circ stk, c) = exec(PC + 1, \langle 1, i\rangle \circ stk, c)$$

The conditional jump instruction $\texttt{cmp}(l)$ jumps to the program address l whenever the top value i on the stack is equal to zero. More precisely, $\texttt{cmp}(l)$ reads and removes the top value i from the stack and then loads the program counter with its argument l if and only if i is equal to zero.

Definition 4.6 (Conditional Jump)
Let i be a non-negative integer. Let l be a valid address of the program eco. We define the semantics of the instruction $\texttt{cmp}(l)$. We assume that $*(PC, eco) = \texttt{cmp}(l)$. Then:

$$exec(PC, \langle i\rangle \circ stk, c) = exec(l, stk, c)$$

if $i = 0$, or otherwise

$$exec(PC, \langle i\rangle \circ stk, c) = exec(PC + 1, stk, c)$$

The absolute jump instruction $\texttt{jmp}(l)$ performs a jump to the program address l. It loads the program counter with l.

Definition 4.7 (Absolute Jump)
Let l be a valid address of the program eco. We define the semantics of the instruction $\texttt{jmp}(l)$. We assume that $*(PC, eco) = \texttt{jmp}(l)$. Then:

$$exec(PC, stk, c) = exec(l, stk, c)$$

We define the semantics of the return instruction `ret`. If the stack of the E machine is not empty, `ret` removes the top value from the stack and loads the program counter with the top value. We assume that this value is a program address which has been pushed onto the stack by a previous push instruction or an embedded jump instruction. In this case, `ret` behaves similar to a standard return instruction at the end of a procedure. If, however, the stack of the E machine is empty, `ret` stops the execution of the current program.

Definition 4.8 (Return)
Let l be a valid program address in eco. We define the semantics of the instruction `ret`. We assume that $*(PC, eco) = \texttt{ret}$. Then:

$$exec(PC, \langle\rangle, c) = \{c\}$$

or

$$exec(PC, \langle l\rangle \circ stk, c) = exec(l, stk, c)$$

The instruction `nop` just proceeds to the next instruction.

```
         psh(0)
While:   neq(10)
         cmp(End:)
         com(f)
         add(5)
         jmp(While:)
  End:   pop
         ret
```

Figure 1: An example of synchronous E code.

```
         prd(p)
         cmp(Else:)
         com(f)
 Else:   ret
```

Figure 2: An example of dynamic E code.

Definition 4.9 (No operation)
We define the semantics of the instruction `nop`. We assume that $*(PC, eco) = \texttt{nop}$. Then:

$$exec(PC, stk, c) = exec(PC + 1, stk, c)$$

Consider the basic E code in Figure 1. We use labels of the form `label:` to denote program addresses. The program implements a simple while loop. It invokes a function f two times before it leaves the while loop. We assume that each external function is associated to a unique non-negative integer. We overload f to denote this integer. Note that the execution of the program is assumed to happen in zero time.

Dynamic E code. In this paragraph we introduce an instruction to invoke external predicates. We call E code which contains predicate instructions *dynamic* E code. The instruction $\texttt{prd}(p)$ invokes the computation of an external predicate p. We require that p is computable and is defined on a subset of the internal ports of M.

Definition 4.10 (Predicate)
Let p be a computable function which maps valuations of its *input ports in* which are a subset of the internal ports *int* to 0 or 1. We define the semantics of the instruction $\texttt{prd}(p)$. We assume that $*(PC, eco) = \texttt{prd}(p)$. Then:

$$exec(PC, stk, (\mu, \tau, \delta, \rho)) = exec(PC + 1, \langle p(\mu|_{in})\rangle \circ stk, (\mu, \tau, \delta, \rho))$$

Consider the dynamic E code in Figure 2. The function f is only executed if the predicate p is true on the current valuation of the input ports of p.

Explicit control state. We use the control state ρ of the E machine to make the control state of E code available to computational activity in the future. The indirect jump instruction $\texttt{imp}(i)$ performs a jump to the absolute program address $\rho(i)$ by dereferencing its argument i using the control state ρ.

```
      set(0)(S1:)
      prd(p)
      cmp(Else:)
      set(0)(S0:)
Else: imp(0)
  S0: com(f)
  S1: ret
```

Figure 3: An example of E code with explicit control state.

Definition 4.11 (Indirect Jump)
Let i be a non-negative integer where $\rho(i)$ is a valid address in eco and ρ is the control state of the configuration c. We define the semantics of the instruction $\texttt{imp}(i)$. We assume that $*(PC, eco) = \texttt{imp}(i)$. Then:

$$exec(\_, stk, c) = exec(\rho(i), stk, c)$$

The set instruction $\texttt{set}(i)(l)$ modifies the control state ρ of the E machine. It associates its argument i with the program address l in ρ.

Definition 4.12 (Set)
Let i be a non-negative integer and let l be a valid address in eco. We define the semantics of the instruction $\texttt{set}(i)(l)$. We assume that $*(PC, eco) = \texttt{set}(i)(l)$. Then:

$$exec(PC, stk, (\mu, \tau, \delta, \rho)) = exec(PC + 1, stk, (\mu, \tau, \delta, update(\rho, \{(i, l)\})))$$

Consider the E code in Figure 3. Similarly as above, the function f is only executed if the predicate p is true on the current valuation of the input ports of p.

The E code introduced so far is still very limited. We can only express some restricted finite computational activity. The E machine can even neither see nor affect any state values of the environment. To connect the E machine to state values of its environment and of its output we introduce synchronous read and write instructions in the next paragraph.

Synchronous Communication. In order to allow external functions to see and affect state values of the environment we introduce read and write instructions which have the same semantics as the synchronous call instruction to invoke external functions but with different restrictions on their arguments.

Definition 4.13 (Read)
Let f be a computable function which maps valuations of a subset in of the environment and output ports $env \cup out$ to valuations of a subset of the internal value ports int_v. The semantics of the instruction $\texttt{red}(f)$ is exactly the same as the semantics of the instruction $\texttt{com}(f)$.

Definition 4.14 (Write)
Let f be a computable function which maps valuations of a subset in of the internal ports int to valuations of a subset of the output value ports out_v. The semantics of the instruction $\texttt{wrt}(f)$ is exactly the same as the semantics of the instruction $\texttt{com}(f)$.

```
           psh(0)
While:     neq(10)
           cmp(End:)
           red(f_r)
           com(f)
           wrt(f_w)
           add(5)
           jmp(While:)
  End:     pop
           ret
```

Figure 4: An example of closed E code.

The read instruction allows the E machine to observe whereas the write instruction allows to change state values of the environment. E code containing write instructions but no read instructions is called *open* E code. We call E code containing read and write instructions *closed* E code.

Consider the closed E code in Figure 4. Let f_r be a function which reads from an environment port and updates an internal port p_r. Let f_w be a function which takes the value from an internal port p_w and writes it to an output port. Let f be a function mapping values from the internal port p_r to values in the internal port p_w. The program implements a simple while loop. Anytime before it invokes f the instruction $\texttt{red}(f_r)$ copies the value of some environment port to the internal port p_r. Thus f can see the environment of the E machine. Each time f is finished, the instruction $\texttt{wrt}(f_w)$ copies the result to some output port. Note that since the execution of the program is assumed to happen in zero time, both invocations of f will see the same value in the environment.

Embedded Jump. In this paragraph we define the semantics of the embedded jump instruction and the deschedule instruction. We require for the execution of an embedded jump instruction $\texttt{emp}(s)(l)$ that the stack contains at least one value. s is the address of a signal port and l is a program address. $\texttt{emp}(s)(l)$ reads the top value n from the stack. If n is equal to zero it pushes the program address of the next instruction onto the stack and then jumps immediately to the program address l. In this case the embedded jump corresponds to a procedure call which saves a return address on the stack.

If n is greater than zero it inserts a trigger (s, m, l) into the schedule τ where $m = \mu(s) + n$ and $\mu(s)$ is the current signal counter of s. An embedded jump with $n > 0$ corresponds to an absolute jump performed in the future upon the m-th occurrence of a signal in s. Note that a return address is not saved on the stack.

Definition 4.15 (Embedded Jump)
Let s be the address of a signal port. Let l be a valid address of the program eco. We define the semantics of the instruction $\texttt{emp}(s)(l)$. Let n be a non-negative integer. We assume that $*(PC, eco) = \texttt{emp}(s)(l)$. Then:

$$exec(PC, \langle 0 \rangle \circ stk, c) = exec(l, \langle PC + 1, 0 \rangle \circ stk, c)$$

or, if $n > 0$:

$$exec(PC, \langle n \rangle \circ stk, (\mu, \tau, \delta, \rho)) = exec(PC + 1, \langle n \rangle \circ stk, (\mu, \tau \circ \langle (s, \mu(s) + n, l) \rangle, \delta, \rho))$$

```
  Invoke_f:  psh(0)
             emp(clk)(Function_f:)
             pop
             ret
Function_f:  red(f_r)
             com(f)
             wrt(f_w)
             ret
```

Figure 5: An example of synchronous E code with a procedure call.

```
  Invoke_g:  psh(0)
     While:  neq(10)
             cmp(End:)
             emp(clk)(Function_g:)
             add(5)
             jmp(While:)
       End:  pop
             ret
Function_g:  com(g)
             ret
```

Figure 6: An example of synchronous E code with a loop around an embedded jump.

An example of synchronous E code with a procedure call is depicted by Figure 5. clk is the address of a signal port of the environment interface. We assume that the signal counter of clk is increased periodically by a 1ms tick of a real-time clock. We first push a zero onto the empty stack and then perform an embedded jump to `Function_f:`. In this case, the jump behaves like a procedure call pushing the program address of the following pop instruction onto the stack and then jumping to `Function_f:`. The following three instructions are similar to the example depicted by Figure 4. The final return instruction jumps to the program address which has been saved on the stack by the previous embedded jump. Finally, the zero on the stack is removed from the stack and the program exits.

An example of synchronous E code with a loop around an embedded jump is depicted by Figure 6. The first invocation of the embedded jump behaves like a procedure call to `Function_g:` similar to the example depicted by Figure 5. However, the second invocation sees a five on the stack. In this case, the embedded jump adds a trigger ($clk, k+5,$`Function_g:`) to the schedule of the E machine where k is the current time in clk. Thus the code at `Function_g:` will be executed exactly in 5ms. At the current instance, however, the E machine proceeds immediately and finishes the loop and exits.

Consider the synchronous E code in Figure 7. The program invokes the function f every 20ms and the function g every 5ms as long as the predicate p returns zero which is checked every 10ms. As soon as p returns a one the E machine stops. This example has been motivated by the time-triggered semantics of Giotto [HHK00].

The deschedule instruction complements the embedded jump instruction in the sense that a trigger which has previously been inserted into the schedule τ by an embedded jump can be removed

```
      Minor1:  psh(0)
               emp(clk)(Invoke_f:)
               emp(clk)(Invoke_g:)
               add(10)
               emp(clk)(Check_Minor1:)
               pop
               ret
Check_Minor1:  prd(p)
               cmp(Minor2:)
               ret
      Minor2:  psh(0)
               emp(clk)(Invoke_g:)
               add(10)
               emp(clk)(Check_Minor2:)
               pop
               ret
Check_Minor2:  prd(p)
               cmp(Minor1:)
               ret
```

Figure 7: An example of a Giotto implementation.

by the deschedule instruction. An application of the deschedule instruction is the implementation of timeouts.

Definition 4.16 (Deschedule)
Let s be the address of a signal port. Let l be a valid address of the program *eco*. We define the semantics of the instruction $\mathtt{des}(s)(l)$. Let n be a non-negative integer. We assume that $*(PC, eco) = \mathtt{des}(s)(l)$. Then:

$$exec(PC, \langle n\rangle \circ stk, (\mu, \tau, \delta, \rho)) = exec(PC + 1, \langle n\rangle \circ stk, (\mu, \tau \setminus \{(s, \mu(s) + n, l)\}, \delta, \rho))$$

if $n > 0$.

Scheduled Computation. Synchronous computation is logically instantaneous computation. An implementation of the E machine can only approximate this assumption on the semantics. *Scheduled computation*, on the other hand, is computation which strictly takes time. We define the scheduled call instruction $\mathtt{cal}(s^e)(f)(s^i)$ for scheduled computation of external functions which complements the synchronous call instruction $\mathtt{com}(f)$. We call E code containing instructions for scheduled computation *scheduled E code*. We require that f is computable and is defined on a subset of the internal ports of M. Thus f can neither see nor affect the environment directly. We assume that the worst-case execution time of f is known and is available to any scheduling algorithm used by the E machine.

The scheduled call instruction $\mathtt{cal}(s^e)(f)(s^i)$ schedules the computation of f whose execution strictly takes time. s^e is the address of a signal port in the environment interface. s^i is the address of a signal port in the internal memory. We require for the execution of the instruction that the stack contains at least one value $n > 0$. The computation of f may finish non-deterministically before or

```
Function_f:  red(f_r)
             psh(20)
             cal(clk)(f)
             emp(clk)(Write:)
             pop
             ret
    Write:   wrt(f_w)
             ret
Function_g:  psh(5)
             cal(clk)(g)
             pop
             ret
```

Figure 8: An example of E code with scheduled computation.

at the m-th occurrence of a signal in s^e where $m = \mu(s^e) + n$. Note that whenever the computation of f finishes after the m-th occurrence its result will be discarded by the E machine. We require that f reads its input values upon execution of the scheduled call instruction. The result of the computation is written to the machine memory when the computation of f is finished on time. Then also the signal counter in s^i is increased by one, effectively emitting a signal in the internal memory at the completion of the computation of f. Technically, the instruction $\texttt{cal}(s^e)(f)(s^i)$ inserts a scheduled reaction $(s^e, \mu(s^e)+i, f, \mu|_{in}, \{s^i\})$ for some i with $0 < i \leq n$ into the distributed reaction of the E machine. The scheduled reaction will become effective when the signal counter of s^e reaches $\mu(s^e) + i$. Note that, in general, the scheduled call instruction leads to non-deterministic runs of the E machine because any choice of i with $0 < i \leq n$ is valid.

As opposed to synchronous reactive languages only scheduled computation in the E machine is allowed to emit signals. Note that the instruction $\texttt{cal}(s^e)(f)$ is a special case in which no signal counter is increased. The non-determinism of the duration of scheduled computation has to be resolved by an arbitrary scheduling algorithm which is a parameter of the E machine. Scheduled computation which does not finish before its scheduled reaction has to be discarded by a valid implementation of the E machine.

Definition 4.17 (Scheduled Computation)
Let s^e be the address of an environment signal port and let s^i be the address of an internal signal port. Let f be a computable function which maps valuations of a subset in of the internal ports int to valuations of a subset of the internal value ports int_v. Let n be a non-negative integer. We define the semantics of the instruction $\texttt{cal}(s^e)(f)(s^i)$ and $\texttt{cal}(s^e)(f)$. If $*(PC, eco) = \texttt{cal}(s^e)(f)(s^i)$ then let $S = \{s^i\}$, or else if $*(PC, eco) = \texttt{cal}(s^e)(f)$ then let $S = \emptyset$. Then:

$$exec(PC, \langle n \rangle \circ stk, (\mu, \tau, \delta, \rho)) = C^{cpl} \cup exec(PC + 1, \langle n \rangle \circ stk, (\mu, \tau, \delta, \rho))$$

with $C^{cpl} = \bigcup_{0<i\leq n} exec(PC + 1, \langle n \rangle \circ stk, (\mu, \tau, \delta \cup \{(s^e, \mu(s^e) + i, f, \mu|_{in}, S)\}, \rho))$.

Consider the scheduled E code in Figure 8. This example replaces the code labelled `Function_f:` and `Function_g:` depicted by Figure 5 and Figure 6, respectively. Now, the computation of g may take up to 5ms whereas the computation of f may take up to 20ms. Note that the write instruction subsequent to the computation of f has to be delayed until the computation of f is completed. The semantics of the E machine guarantees that the result of the computation of f is available before

```
Function_f:  red(f_r)
             psh(20)
             cal(clk)(f)(fin)
             pop
             psh(1)
             emp(fin)(Write:)
             pop
             ret
```

Figure 9: An example of E code with scheduled computation.

or at the next 20ms instance and strictly before any other synchronous E code scheduled to begin at the next 20ms instance is invoked.

Another way to replace the code labelled `Function_f:` depicted by Figure 8 is shown in Figure 9. Now, we use an additional internal signal port with address fin whose signal counter is increased by one as soon as the computation of f finishes. This implies that the write instruction may be performed earlier than within the next 20ms.

The termination instruction complements the scheduled call instruction in the sense that a scheduled reaction which has previously been inserted into the distributed reaction δ by a scheduled call instruction can be removed by the termination instruction. Termination corresponds to terminating scheduled computation and communication within a given interval of signal counters.

Definition 4.18 (Termination)
Let s^e be the address of an environment signal port and let S either be the empty set or a singleton set $\{s^i\}$ where s^i is the address of a signal port. Let f be a function and let ν be a valuation. We define the semantics of the instruction $\mathtt{trm}(s^e)(f)$. Let n be a non-negative integer. We assume that $*(PC, eco) = \mathtt{trm}(s^e)(f)$. Then:

$$exec(PC, \langle n\rangle \circ stk, (\mu, \tau, \delta, \rho)) = exec(PC+1, \langle n\rangle \circ stk, (\mu, \tau, \delta \setminus \delta^{trm}, \rho))$$

with $\delta^{trm} = \{d | d = (s^e, \mu(s^e) + i, f, \nu, S) \in \delta, 0 < i \leq n\}$.

Scheduled Communication. Similar to scheduled computation, we introduce instructions for *scheduled communication* which strictly takes time as opposed to synchronous communication. Note that the termination instruction may also be used to terminate scheduled communication. We define a polling instruction $\mathtt{pol}(s^e)(f)(s^i)$ and a send instruction $\mathtt{snd}(s^e)(f)(s^i)$ for scheduled communication which complement the synchronous read instruction $\mathtt{red}(f)$ and write instruction $\mathtt{wrt}(f)$, respectively. We assume that the worst-case latency of f is known and available to any scheduling algorithm used by the E machine.

Definition 4.19 (Poll)
Let s^e be an environment signal port and let s^i be an internal signal port. Let f be a computable function which maps valuations of a subset in of the environment and output ports $env \cup out$ to valuations of a subset of the internal value ports int_v. The semantics of the instruction $\mathtt{pol}(s^e)(f)(s^i)$ and $\mathtt{pol}(s^e)(f)$ is the same as the semantics of the instruction $\mathtt{cal}(s^e)(f)(s^i)$ and $\mathtt{cal}(s^e)(f)$, respectively.

```
Function_f:  red(f_r)
             psh(15)
             cal(clk)(f)
             emp(clk)(Write:)
             pop
             ret
    Write:   psh(5)
             snd(clk)(f_w)
             pop
             ret
```

Figure 10: An example of E code with scheduled computation and communication.

Definition 4.20 (Send)
Let s^e be an environment signal port and let s^o be an output signal port. Let f be a computable function which maps valuations of a subset *in* of the internal ports *int* to valuations of a subset of the output value ports out_v. The semantics of the instruction $\mathtt{snd}(s^e)(f)(s^o)$ and $\mathtt{snd}(s^e)(f)$ is the same as the semantics of the instruction $\mathtt{cal}(s^e)(f)(s^o)$ and $\mathtt{cal}(s^e)(f)$, respectively.

Figure 10 shows scheduled E code which replaces the code labelled `Function_f:` depicted by Figure 8 and Figure 9. Now, we shorten the deadline of the computation of f to 15ms and send, at the next 15ms instance its result to some output port within the next 5ms. The send instruction may also be triggered by an additional signal port similar as the example of Figure 9.

4.2 The E Machine Semantics

We define the formal semantics of the E machine in terms of sequences of configurations. We distinguish the deterministic *dispatcher* and the non-deterministic *scheduler* of the E machine. The scheduler is non-deterministic in the sense that it does not specify the completion times of scheduled computation and communication which effectively makes it compatible with any scheduling algorithm. In order to compute a new configuration, the scheduler first calls the dispatcher to update the memory according to all computations and communication which have been scheduled to finish now. Note that in an implementation of the E machine the dispatcher is usually not called by the scheduler but upon completion of scheduled computation and communication by the completed tasks and messages themselves. Based on the updated memory μ^{dsp} and the current set σ^{act} of signal counters the scheduler executes embedded jumps which were waiting for σ^{act} resulting in new embedded jumps in the future and possibly new scheduled computation and communication.

Given a set σ of signal counters from the environment interface, the dispatcher collects the results of all *completed* scheduled reactions which have been scheduled to finish at σ. The dispatcher updates value ports and may increase the signal counters of signal ports. Recall that, e.g., the scheduled call instruction $\mathtt{cal}(s^e)(f)(s^i)$ modifies, upon completion, the valuation of value ports to which f writes and also increases the signal counter of s^i. In general, there might be multiple completed scheduled reactions at the same instance which, however, in practice are only possible on different E machines of a distributed E machine.

Definition 4.21 (Dispatcher)
Let M be an embedded machine (eco, env, int, out). The *dispatcher* of M is a function *dispatch* which maps a set σ of signal counters, a valuation μ for the overall memory *mem* of M, and a

distributed reaction δ of M to a valuation μ' for *mem* as follows:

1. We assume that the set σ contains signal counters of all environment signal ports which just have been increased by one indicating the occurrence of a signal. We compute the set δ^{cpl} of *completed* scheduled reactions from the distributed reaction δ: $\delta^{cpl} = \{d | (s,n) \in \sigma, d = (s,n,f,\nu,S) \in \delta\}$.

2. In S^{cpl} we collect all signal ports from the completed scheduled reactions δ^{cpl} whose counters will be increased by one: $S^{cpl} = \{s^t | (s,n,f,\nu,S) \in \delta^{cpl}, s^t \in S\}$.

3. We compute inductively the updated valuation of the E machine memory by first initializing: $\delta_0 = \delta^{cpl}$ and $\mu_0 = \mu$.

4. For all $i \geq 0$ for which there is, non-deterministically, a completed scheduled reaction $d = (s,n,f,\nu,S) \in \delta_i$, let us remove d in $\delta_{i+1} = \delta_i \setminus \{d\}$ and update $\mu_{i+1} = update(\mu_i, f(\nu))$ by the valuation $f(\nu)$ computed by the function f on the valuation ν for the input ports of f.

5. Finally, we increase the signal counters of S^{cpl} by one and return the last valuation μ' with: $\mu' = increase(\mu_k, S^{cpl})$ and $k = min(\{i | i \geq 0, \delta_i = \emptyset\})$.

The scheduler of the E machine computes for a given configuration a set of new configurations which contains all possible schedules. Whenever an implementation of the E machine uses a scheduling algorithm which is deterministic with respect to occurrences of signals, i.e., the progress of time and the occurrences of events, the scheduler will become deterministic up to changes at the environment interface.

The scheduler first calls the dispatcher to collect the results of the completed scheduled reactions. Then it determines the set τ_0^{act} of active triggers in the schedule τ. A trigger (s,n,l) is *active* if the signal counter of s reached n. The scheduler executes the E code at the program address l. Multiple active triggers are executed in the order of the previous execution of the embedded jumps which created the triggers.

Definition 4.22 (Scheduler)
Let M be an embedded machine (eco, env, int, out). Let c and c' be two configurations of M of the form $(\mu, \tau, \delta, \rho)$ and $(\mu', \tau', \delta', \rho')$, respectively. The *scheduler scd* of M is a relation on pairs of configurations with $(c, c') \in scd$ if and only if:

1. In order to compute the completed scheduled reactions first, we begin with collecting all signal ports of the environment interface whose signal counters have been increased from c to c': $\sigma^{cpl} = \mu'|_{env_s} \setminus \mu|_{env_s}$. Recall that reactions of scheduled computation and communication may only be scheduled with respect to signal ports of the environment interface.

2. We compute the results of the completed scheduled reactions based on the new valuations of the environment interface and use the last valuations of the internal memory and output interface: $\mu^{cpl} = \mu'|_{env} \cup \mu|_{int \cup out}$.

3. $\mu^{dsp} = dispatch(\sigma^{cpl}, \mu^{cpl}, \delta)$ incorporates the results of all completed scheduled reactions written to μ^{cpl} by the dispatcher.

4. $S^{act} = env_s \cup int_s \cup out_s$ is the set of all signal ports of M which may now activate triggers.

5. We will compute all active triggers based on the set $\sigma^{act} = \mu^{dsp}|_{S^{act}} \setminus \mu|_{S^{act}}$ of the signal ports whose signal counters have been increased from c to c' at the environment interface as well as in the internal memory and at the output interface.

6. In $\tau_0^{act} = \tau \setminus \{t | t = (s, n, l) \in \tau, (s, n) \notin \sigma^{act}\}$ we collect all active triggers of the current schedule τ.

7. We compute inductively the updated versions of the valuation, the schedule, the distributed reaction, and the control state of the E machine by first initializing: $\mu_0 = \mu^{dsp}$, $\tau_0 = \tau$, $\delta_0 = \delta$, and $\rho_0 = \rho$.

8. For all $i \geq 0$ for which there is an active trigger t of the form (s, n, l) such that $\tau_i^{act} = \langle t \rangle \circ \tau_{i+1}^{act}$, we update $\mu_{i+1} = update(\mu_i, \mu_t)$, $\tau_{i+1} = \tau_t$, $\delta_{i+1} = \delta_t$, and $\rho_{i+1} = update(\rho_i, \rho_t)$ according to the execution of the E code at the program address l with $(\mu_t, \tau_t, \delta_t, \rho_t) \in exec(l, \langle\rangle, (\mu_i, \tau_i, \delta_i, \rho_i))$. Note that τ_{i+1} and δ_{i+1} are replaced by τ_t and δ_t, respectively, rather than being extended. This allows triggers and scheduled reactions not only to be added but also to be removed from the schedules. In particular, removing a scheduled reaction corresponds to terminating scheduled computation or communication.

9. Finally, a new configuration c' consists of $\mu' = \mu_k$, $\tau' = \tau_k$, $\delta' = \delta_k$, and $\rho' = \rho_k$ with $k = min(\{i | i \geq 0, \tau_i^{act} = \langle\rangle\})$.

We define the semantics of the embedded machine in terms of sequences of configurations. *Stuttering* with respect to signals at the environment interface occurs whenever there is no active trigger and no completed scheduled reaction.

Definition 4.23 (Run)
Let *scd* be the scheduler of an embedded machine M. Let C be a set of initial configurations of M. A *run* of M is a sequence $c_0, c_1, c_2, \ldots$ with $c_0 \in C$ and $scd(c_i, c_{i+1})$ for all $i \geq 0$.

5 The Distributed Embedded Machine

We define the parallel composition of multiple E machines where some output ports of one machine may be equal to some output ports of another machine. Simultaneous write access by multiple machines is scheduled non-deterministically by the E machine which effectively makes the machine compatible with any write access protocol. The ports of the environment interface, on the other hand, are required to be controlled exclusively by the physical environment rather than other E machines.

We formally define the parallel composition of E machines. The result of the parallel composition of E machines is the *distributed embedded machine*. Most importantly, the semantics of the non-distributed E machine carries over to the distributed E machine without further enhancements.

Definition 5.1 (Machine Composition)
Let M_1 be a (distributed) embedded machine $(eco_1, env_1, int_1, out_1)$ and let M_2 be a (distributed) embedded machine $(eco_2, env_2, int_2, out_2)$ such that either the end of eco_1 is strictly smaller than the offset of eco_2 or the end of eco_2 is strictly smaller than the offset of eco_1, $env_1 \cap out_2 = \emptyset$, $env_2 \cap out_1 = \emptyset$, $int_1 \cap (env_2 \cup int_2 \cup out_2) = \emptyset$, and $int_2 \cap (env_1 \cup int_1 \cup out_1) = \emptyset$. The composition $M_1 \| M_2$ of M_1 and M_2 is a tuple (eco, env, int, out) with:

$$eco = eco_1 \circ eco_2$$

$$env = env_1 \cup env_2$$

$$int = int_1 \cup int_2$$

$$out = out_1 \cup out_2$$

Note that $\|$ is associative and commutative up to the order of the E code of the composed machines which does not have any effects on the semantics of the composition.

Definition 5.2 (Distributed Embedded Machine)
A *distributed embedded machine* is the composition of (distributed) embedded machines.

6 Conclusion

We conclude with a discussion of particularly interesting platforms on which the embedded machine may be implemented. The *time-triggered machine* is a special case of the embedded machine in which we allow only a single signal port clk in the environment interface. We assume that a real-time clock increases periodically the signal counter of clk. Thus the physical environment can only trigger computational activity in the time-triggered machine through the signal port clk. In particular, events in the physical environment can only be observed in a time-triggered fashion by keeping track of the valuations of the value ports in the environment interface.

Definition 6.1 (Time-Triggered Machine)
The (*distributed*) *time-triggered machine* (TT Machine) is a (distributed) embedded machine with an environment interface which contains only a single signal port clk. We assume that a real-time clock increases periodically the signal counter of clk. We require that there are no shared output signal ports in the output interface among any of its component machines. We may call an embedded jump $\texttt{emp}(clk)(l)$ of the TT machine a *temporal jump* denoted by $\texttt{tmp}(l)$.

The time-triggered architecture (TTA) [Kop97] is an interesting platform to implement the distributed time-triggered machine. A TTA requires *implicit* clock synchronization which allows TTA applications to exploit the existence of a global clock. It is thus not necessary for a TTA application to use signals across machines to control computation and communication. Note, however, that a TTA on the lowest operational level of the time-triggered protocol (TTP) uses signals across machines because this is an inherent part of networking in general. It is also possible to model the operational level, in particular, of the TTP controllers by a more complex distributed embedded machine with a single signal port clk in the environment interface but with shared output signal ports in the output interfaces of its component machines.

Another interesting platform for the distributed embedded machine is the globally asynchronous locally synchronous (GALS) architecture [BCG00] in which implicit global synchronization is replaced by *explicit* and *directed* clock synchronization. A receiver synchronizes its clock with the clock of a sender only upon the arrival of a new message from the sender. All local clocks in a GALS architecture will effectively be synchronized if all nodes in the system are senders and receivers which engage in a periodic exchange of messages. The explicit and directed synchronization can be modeled with E machine signals crossing the networks of a GALS architecture.

Future work with the E machine includes the design of a compiler which generates E code for the time-triggered programming language Giotto [HHK00]. The compiler may be part of the software architecture Ptolemy II [DGH$^+$99] in which we have implemented Giotto as a model of computation. More complex code generation may result from combinations of Giotto with other models of computation in Ptolemy II like, e.g., the finite state machine and the synchronous data flow models.

Acknowledgements. We are grateful for many valuable comments and suggestions on the embedded machine from Thomas A. Henzinger, Benjamin Horowitz, Edward A. Lee, and Steve Neuendorffer.

References

[BCG00] A. Benveniste, B. Caillaud, and P. Le Guernic. Compositionality in dataflow synchronous languages: specification & distributed code generation. *Information and Computation*, 163:125–171, 2000.

[Ber00] G. Berry. The foundations of Esterel. In C. Stirling G. Plotkin and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000.

[DGH+99] J. Davis, M. Goel, C. Hylands, B. Kienhuis, E.A. Lee, J. Liu, X. Liu, L. Muliadi, S. Neuendorffer, J. Reekie, N. Smyth, J. Tsay, and Y. Xiong. Ptolemy II: Heterogeneous concurrent modeling and design in Java. Technical Report UCB/ERL M99/44, University of California, Berkeley, CA, July 1999.

[Hal93] N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer, 1993.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 79(9), 1991.

[HHK00] T.A. Henzinger, B. Horowitz, and C.M. Kirsch. Giotto: A time-triggered language for embedded programming. Technical Report UCB//CSD-00-1121, University of California, Berkeley, 2000. Available at: `www.eecs.berkeley.edu/~fresco/giotto`.

[Kop97] H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer, 1997.

[LY99] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification (2nd Ed)*. Addison-Wesley, 1999.

[PS98] J.A. Plaice and J.-B. Saint. The Lustre-Esterel portable format. Technical report, Sophia Antipolis, France, September 1998.

[TFW00] H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and precise WCET prediction by separated cache and path analyses. *Time-Critical Computing Systems*, pages 157–179, 2000.

7 Appendix: Opcodes

Instruction	Opcode	Instruction	Opcode
`nop`	0	`set`	10
`emp`	1	`ret`	11
`des`	2	`cal`	12
`com`	3	`pol`	13
`red`	4	`snd`	14
`wrt`	5	`trm`	15
`prd`	6	`psh`	16
`cmp`	7	`pop`	17
`jmp`	8	`add`	18
`imp`	9	`neq`	19