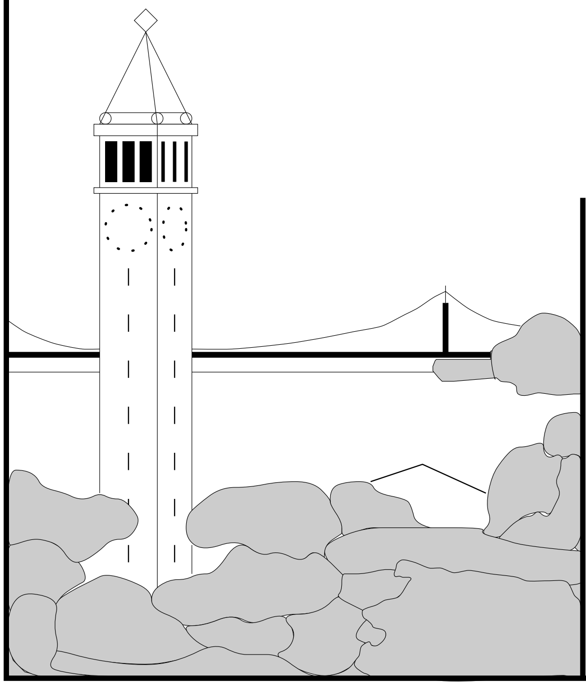
A Preliminary Report on the Embedded Virtual Machine

Arkadeb Ghosal, Marco A.A. Sanvido and Thomas A. Henzinger



Report No. UCB//CSD-04-1322

May 2004

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

A Preliminary Report on the Embedded Virtual Machine[1]

Arkadeb Ghosal, Marco A.A. Sanvido and Thomas A. Henzinger
University of California at Berkeley
Berkeley, 94704, CA
{arkadeb,msanvido,tah}@eecs.berkeley.edu

May 18, 2004

[1]This research is supported by the AFOSR MURI grant F49620-00-1-0327, the DARPA SEC grant F33615-C-98-3614, the MARCO GSRC grant 98-DT-660, and the NSF grants CCR-0208875 and CCR-0225610.

Abstract

xGIOTTO [5] is a domain specific language for the implementation of embedded software applications with hard temporal constraints. The language is an extension of the original GIOTTO language [6]. In this report we present the xGIOTTO tool chain, composed of the compiler and a specialized virtual machine, Embedded Virtual Machine (EVM). The compiler checks for determinism (absence of races) and time safety (schedulability within logical execution times) and generates code for EVM. The EVM integrates an event filter (which handles aperiodic asynchronous events and event scoping introduced in [5]) and a modified Embedded Machine [7]. The report presents the instruction set and the operational semantics of the virtual machine. The report also presents event calculus which is used to extend expressiveness of xGIOTTO. The report concludes with a case study of implementing an automotive engine controller.

Chapter 1

Introduction

Real-time systems for embedded applications are characterized by limited memory, distributed nodes, interprocess communication, fast context switches and concurrency [1]. However the most important features of such systems are *predictability* and *timing*. The execution of a safety critical system must be predictable and the evaluation of a task should be available when it is due (neither before the deadline nor after). Several programming paradigm has been used for implementing real-time controllers. In [2] real-time systems programming has been divided into three categories: scheduled, synchronous and timed. The first approach is the traditional scheduling based approach [3] where each task is assigned a priority. The second approach is programming with synchrony assumption [4] where all tasks are assumed to execute in logical zero time. In [5] it has been argued that while the first approach causes non-determinism making program verification difficult, the second approach is not suitable for applications with non-negligible task execution and distributed computing.
The third programming approach assumes that each task is associated with a *Logical Execution Time* (LET) [5, 6]. When a task is released on a platform its corresponding LET is specified by a termination event. The task output is available only when the termination event occurs. Even if the task completes its execution before the termination event arrives, the task output is not released. A trace of the execution is *time-safe* if all tasks released along the trace completes their execution before the arrival of the respective termination event. A program is *schedulable* if all the traces are time-safe. The LET model makes the program execution time-deterministic (no jitter) and value-deterministic (no race conditions) and thus make program verification and analyses easier than the traditional scheduling model. Details about the LET model and corresponding advantages in using LET model has been discussed in [5, 6, 7, 2, 8]. The LET programming model has been studied and implemented on different platforms. In [6] we present GIOTTO, a time-triggered version of the LET programming model. Our first implementation [9] was done on top of a conventional real-time operating system (Figure 1.1.A) like Osek [10] and HelyOS [11]. By using a programming paradigm where timing and functionality are separated [6, 7], we were able to focus on the compiler and the E Machine implementations leaving the platform specific problems to the real-time operating systems (RTOS). The E Machine (or the Embedded

Machine) [7] is the virtual machine on which GIOTTO was implemented. In [12] we extended the previous implementation by introducing the E Machine in the RTOS itself (Figure 1.1.B) in order to gain flexibility and to reduce the overhead of the operating system. In this paper we present another approach (Figure 1.1.C) for implementing the LET programming model, by integrating the E Machine directly into the instruction set of the running CPU. Since we are not able to produce a silicon for the associated high cost and our limitation of time, we choose to modify an instruction set simulator to implement a modified E Machine instructions. For its simplicity and popularity, the Java Virtual Machine [13] was chosen.

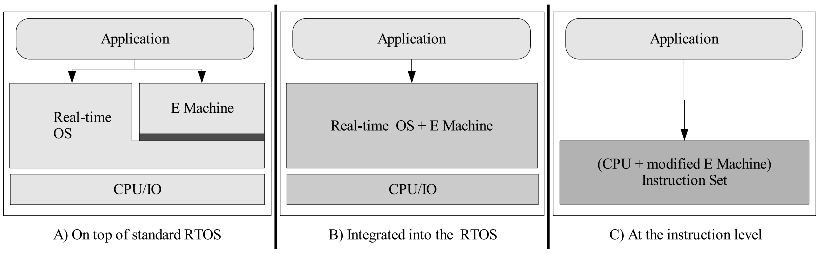


Figure 1.1: Implementing the LET programming model

XGIOTTO [5] is a high-level programming language for control applications. GIOTTO is the predecessor of XGIOTTO. While GIOTTO can express only time-triggered systems, XGIOTTO handles aperiodic, asynchronous events and hence can implement event-triggered applications. Besides, XGIOTTO introduces a new approach based on the notion of *event scoping*, which allows us to encode an *implicit* environment assumption in XGIOTTO programs. Event scoping temporarily disables event monitoring for a subset of the observed events. In this way, the environment assumption is reflected by the control structure of the program itself. An XGIOTTO program is compiled and checked whether the program is free of races and is schedulable. Next the code generator produces machine level code which can run on a virtual machine. GIOTTO runs on the virtual machine E Machine. However the E machine is not capable of handling the event scoping and functionality code of XGIOTTO. This paper discusses the Embedded Virtual Machine (EVM) which implements an event filter (for event scoping) and a modified E Machine which runs parallel to a scheduler. In the future the scheduler will be integrated with the EVM. The code (or the instruction set) for the EVM is called EVM code.
In the next chapter a brief discussion of XGIOTTO and its notion of event scoping has been presented. This is followed by the description of an event calculus. Event calculus complements the notion of event scoping introduced in [5]. The chapter concludes by an example of an XGIOTTO program and the corresponding EVM code. Chapter 3 introduces the instruction set for EVM and details the semantics of the operation of the EVM. Chapter 4 presents the compile- and run-time implementation of the XGIOTTO tool chain and presents the possible future extensions. Chapter 5 presents the implementation of a controller for an automobile engine in XGIOTTO. Chapter 6 concludes the report.

Chapter 2

xGiotto

xGIOTTO (like its predecessor GIOTTO) is built around the notion of LET model of task execution. In GIOTTO an event is always a clock tick and thus the release and termination of a task is possible only at clock ticks. However xGIOTTO being event-triggered an event can be either a clock tick or an arbitrary sensor interrupt. GIOTTO does not allow simultaneous execution of multiple instances of the same task; however xGIOTTO does not have this limitation. An event in xGIOTTO has two possible actions: invoking a reaction block or terminating a reaction block. A *reaction block* defines a set of triggers and a set of tasks and has a termination event. A *trigger* maps an event to a set of reaction blocks such that when the event occurs the reaction blocks are activated. A *task* is a functional code similar to a procedure. When a reaction block is *activated* it enables the triggers and releases the tasks. When a reaction block is *terminated* (on arrival of its termination event) the associated tasks are also terminated. Thus the active-time-span of a reaction block denotes the LET for the tasks released by it. The execution of a reaction block is done synchronously, i.e. executed in logical zero time. However the execution may take place at different time instants if there are nested reaction blocks.
The events associated with a reaction block are the events of the triggers and the termination event of the block. This is the event scope for the reaction block. When a reaction is invoked (one of the triggers is triggered) the called reaction becomes the active scope and the callee reaction (the passive scope) is pushed onto a stack. However as parallel invocation of reactions are allowed, a tree of scopes may exists at any instant of the execution. The leaves of the tree are the *active scopes* and the non-leaf nodes are the *passive scopes*. An event in passive scope can be handled as follows: ignored, remembered or acted upon as soon as possible; they are specified by the keywords: `forget`, `remember`, and `asap`. If an event is remembered, the associated action is taken when the corresponding scope becomes active again. If an `asap` event occurs the trigger queues of the sub-tree rooted by the scope is emptied so new trigger action can take place. The scopes are not preempted immediately to avoid unsafe termination of tasks.

Language Constructs.

An XGIOTTO program consists of a set of ports P, events E, tasks T, and reaction blocks R. A *port declaration* for a port (or a program variable) consists of a name, a fixed type and an initial value. An *event declaration* for an event consists of an event name, type, and the external interrupt triggering the event. The events `time` and `now` are predefined: `time` corresponds to the system clock while `now` is a placeholder for current event and cannot be used as a termination event for reaction blocks. Multiple events can be combined to express environment assumptions in an efficient and succinct manner in an XGIOTTO program. However it is assumed that no two events can occur simultaneously. The expressiveness of event has been extended by an event calculus presented below. A *task declaration* for a task consists of a task name, input parameters, output parameters and local variables. The task body is a standard sequential program. A *reaction block declaration* for a reaction block consists of a name, a body and an termination event parameter. The body of the reaction block consists of trigger statements, release statements and sequential reaction statements. The construct for *trigger* statement is `when [e] r` where event, $e \in E$ and r is a reaction. The statement denotes that when e arrives the reaction r is invoked. The reaction r may be a single reaction block or multiple reaction blocks composed in parallel. The types of parallelism are: `wait`- and `asap`-parallelism. If a scope bound in `asap`-parallelism terminates, the triggers of the sibling scopes and the sub-tree rooted by them are removed. If a scope bound in `wait`-parallelism terminates, no further action is taken. The construct for `release` statements is `release t(in)(out)`. It releases a new task instance for task `t`. When the task is released the values of input ports `in` are copied to the local ports of the task instance and at task termination the output ports `out` are updated. The trigger and release statements can be made conditional by attaching a predicate on the ports. If the predicate is true at the instance of reaction block invocation, the corresponding statement is executed.

xGiotto Analysis.

The XGIOTTO compiler performs three analyses on XGIOTTO programs. Race condition check and resource size prediction are *platform independent* analyses while schedulability is a *platform dependent* analyses. A program is said to have *race condition* if a port is updated by two or more terminating task instances at the same instance. Race conditions are undesirable as they give rise to non-determinism in the program execution. At present the XGIOTTO compiler checks for race by a reachability algorithm which is `PSPACE`-complete. The *resource size* analysis predicts the memory requirements for executing the program on a real-time platform. A conservative bound can be found in time linear to the size of the program. Schedulability for an XGIOTTO program checks whether a given program is schedulable or not with respect to a real-time platform (given by the *worst-case-execution-time* mapping of the tasks to the platform). Schedulability of an XGIOTTO program can be defined as a two-player safety game between the system and the environment. The program is schedulable if in this game the scheduler has a strategy to avoid time-safety violations forever. In theory the schedulability problem is complete for `EXPTIME`. Refer [5] for details of the analyses.

2.1 Event Calculus

The event calculus defines two types of events: atomic events and compound events. An *atomic event* is an event which is triggered by a sensor interrupt. For the event declaration `e AT sensor_interrupt`, `e` is an atomic event triggered by an external interrupt `sensor_interrupt`. A *compound event* is an expression on atomic events and compound events using the sequence operator ($\cdot$) and either operator ($\vee$) as follows:

$$E ::= e \mid E \cdot E \mid E \vee E \tag{2.1}$$

where E is an compound event and e is an atomic event. The *sequence* operator denotes that the associated action takes place only if the events occur serially in the order defined. The *either* operator denotes that either of the events has to occur for triggering the associated action. The order of precedence for evaluating a compound event is: brackets, sequence operator and either operator. A concrete compound event E can be expressed as a finite state machine whose edges are labelled with atomic events and the compound event is enabled if an accepting state of the state machine is reached. If a concrete compound event consists of only an atomic event then the corresponding state machine consists of one initial state, one accepting state and one transition from the initial state to the accepting state labelled by the atomic event. The event calculus is closed under the sequence and either operators and can be proved by construction. In the Figure 2.1 events `A, B` and `C` are atomic events triggered by interrupt_1, interrupt_2 and interrupt_3 respectively.

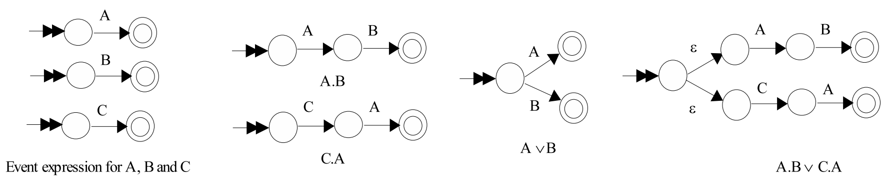


Figure 2.1: Event Calculus

Three derived operators are also defined in the event calculus for succinct representation. They are the repetition operator (*), in-any-order operator (#) and interleaving operator (:). Thus the extended calculus is

$$E ::= e \mid E \cdot E \mid E \vee E \mid k * E \mid E\#E \mid k_1 * e : k_2 * e \tag{2.2}$$

where k, k_1, k_2 are integer constants. The *repetition* operator denotes that an event occurs for k times and can be expressed by applying the sequence operator k times. The *in-any-order* operator denotes that the two events should occur but in any order and can be expressed with the sequence and either operators. Thus $E_1\#E_2 = E_1 \cdot E_2 \vee E_2 \cdot E_1$. The *interleaving* operator denotes the interleaving of k_1 instances of first atomic event and k_2 instances of second atomic event. Thus for the Figure 2.1 the compound event `2*A : 2*B` denotes any one of the sequences {`AABB, ABAB, BABA, BBAA`}.

2.2 Implementing an answering machine

We present a simple automated email and phone answering machine. The system replies to an email or answers the phone as follows: if either a phone or an email arrives when the system is idle the corresponding task is performed; if an email arrives during answering the phone call or reading another email it is remembered and is handled as soon as the phone call has been answered or the current email has been read; however if a phone call arrives when answering an email or a phone call, the new phone call is ignored.

The XGIOTTO code for implementing this machine has been shown in the Figure 2.2. The program implements two tasks, one for reading the emails and one for answering the phone. The input and output ports, and the body of the tasks have been omitted for simplicity. The code consists of two trigger statements: one for the `email` event and the other for the `phone` event. When an email arrives, an `email` event is raised and the task `readmail` is released. The `readmail` task processes the email and terminates at the arrival of the `done` event. Similarly at the arrival of a phone call — signaled by the `phone` event — the task `answerphone` answers the phone and terminates the execution when the caller hangs up (event `hangup`). The desired event handling is embedded by using the keywords `remember` with `email` and `forget` with `phone`. Thus event `email` will be remembered if the system is busy and would be handled as soon as the system becomes idle; however if an event `phone` arrives when the system is busy the event is ignored.

```
task readmail /*action*/
task answerphone /*action*/

react {
 whenever remember [email]
  react {
   release readmail()();
  } until [done];
 whenever forget [phone]
  react {
   release answerphone()();
  } until [hangup];
}
```

Figure 2.2: xGiotto Code

```
Task Table:
0:  return  //readmail function
1:  return  //answerphone function

Code
0:  enterscope   //main reaction
1:  when repeat, remember,
         [email], @5, [done]
2:  when repeat, forget,
         [phone], @8, [hangup]
3:  exitscope
4:  enterscope   //reaction to email
5:  release #0   //task readmail
6:  exitscope
7:  enterscope   //reaction to phone
8:  release #1   //task answerphone
9:  exitscope
```

Figure 2.3: EVM Code

The XGIOTTO compiler generates code for the EVM. The EVM is a mediator between physical events and software processes: it executes EVM code which reacts to events and handles software processes. The EVM code for the XGIOTTO program presented in Figure 2.2 is presented in Figure 2.3. The EVM code begins with a table of all the tasks and their functionality code. For simplicity we have omitted the functionality code and the EVM code for the two tasks is composed only of return statements. The EVM code for the reactions follows the task table. The instruction at address `0` starts the execution of the main (starting) reaction. The instruction `enterscope` instructs the EVM to allocate a new scope for the reaction. The EVM interpreter (Algorithm 5) will then execute all instructions until the instruction `exitscope` is executed. The `when` instructions at address `1` and `2` declares two reaction invocations; one for the `email` event and one for the `phone` event. The `repeat` keyword implies that the trigger is to be repeated, i.e., belongs to a `whenever` statement. The `[email]` and `[phone]` specify the triggering event, followed by the address of the corresponding reaction code to be invoked. The last parameter of the instruction specifies the termination event. The `release` instruction releases a task; the parameter of the instruction specifies the index of the task in the task table. Thus the instruction at address `5` releases the task at index `0`, i.e. the task `readmail`.

Chapter 3

Embedded Virtual Machine

The Embedded Virtual Machine is a virtual machine, similar to the E Machine, that mediates between the physical and the software process of an embedded system through a control program. While E Machine interprets E code, EVM interprets EVM code. An EVM program consists of (1) a set P of program ports, (2) a set E of events, (3) a set T of task declarations, and (4) a set A of addresses and for each address a finite sequence of instructions. Besides interpreting the EVM instructions (described below), the EVM keeps a dynamical data structure of the *event scopes* defined in Section 2. The data structure is implemented as a tree of event scopes where each event scope stores: the termination event of the scope, a trigger queue, a set of released tasks, the mode of parallel invocation, i.e., `asap`-, or `wait`-parallel, and a link to all its children scopes. For efficiency, all active scopes, i.e., the leaves of the event scope tree, are linked together.

3.1 Instruction Set

The instruction set of the EVM is an augmented version of the Java Virtual Machine (JVM) instruction set. A list of the added instructions with the opcode in the parenthesis is provided below. The operational semantics of each instruction is given by the interpreter of Algorithm 5. For the full instruction set we refer to [13].

(opcode)	instruction	synopsis
(0xE3)	`trigger`	inserts a trigger in the trigger queue Q of the actual event scope
(0xE0)	`release`	releases a task to the task instances set T of the actual event scope
(0xE9)	`getport`	writes on the stack the value of a port
(0xEA)	`getevent`	writes on the stack the value of an event
(0xEB)	`putport`	writes the value on the stack to a port
(0xE4)	`enterscope`	generates a new scope, and initializes the trigger queue and the set of task instances: $Q = \emptyset$, $T = \emptyset$
(0xE5)	`exitscope`	returns from the scope

3.2 Operational Semantics

The execution of an XGIOTTO program yields a possibly infinite sequence of program configurations. A (program) configuration is the current state of the program execution and consists of a program counter, values of the ports, tree of scopes and a stack. Formally a *program configuration* is a tuple $(\Sigma, \Delta, PC, S, SP)$, where Σ is a function from the port set P to values, Δ is a labeled tree, where each node is labeled by a scope, PC is the program counter, S is a stack, and SP is the stack pointer. A *scope* is a tuple (U, Q, T, α), where U is the termination event instance, Q is a queue of triggers, T is a set of task instances, and $\alpha \in \{\texttt{asap}, \texttt{wait}\}$ is the parallelism of the scope (with respect to its siblings). An *event instance* is a pair $i = (e, s, f)$, where e is an *compound event*, s is a state of the automaton e, and $f : f \in \{A, R, F\}$ is the event qualifier, with A, R, F denoting `asap`, `remember`, and `forget`. An event instance is *enabled* when s is an accepting state of the compound event state machine e. Given an atomic event e' the operation $next(i, e') = i'$ where $i' = (e, s', f)$ such that (s, s') is a transition in e and labeled with e'; otherwise $i' = (e, s, f)$. The operation $reset(i)$ resets s to the initial state of the state machine e. A *trigger queue* is a queue of triggers. A *trigger* is a tuple (i, m, p, r), where i is an event instance, $m : m = \{\texttt{when}, \texttt{whenever}\}$ is the repetition parameter, $p \in \{||, \&\&\}$ is the parallel operator ($||$ denotes `asap`- and $\&\&$ denotes `wait`-parallelism) and r is a reaction, i.e., a set of reaction blocks to be invoked by the enabled event instance. A task instance consists of the tuple (t, p_o, s_{p_i}) which implies that the ports p_o are updated (at task termination) by the evaluation of the task t on the state of ports p_i at the instance of invocation (given by s_{p_i}).
A pseudo code description of the EVM is provided here. Algorithm 1 shows the main loop of the machine. Initially the active scope corresponds to the starting scope of the EVM code program. Whenever an event occurs, the trigger-related interrupts are disabled. Next three procedures: *UpdateEvent*, *TerminateScope* and *UpdateScope* are invoked in sequence. The EVM runtime system uses a discrete scheduler, i.e., the scheduler is invoked only at a certain periodic event, `tick`. If the input event corresponds to `tick` the scheduler is invoked.
The procedure *UpdateEvent* (Algorithm 2) updates the counter for the events in the event filter in the following way: for each event instance $i = (e, s, f)$, if either (1) $f \in$

$\{A, R\}$ or (2) $f = F$ and the event instance occurs in a leaf scope, then the event instance i is updated by `next`(i, e'). If the updated event instance i is enabled and the form of the event instance is `asap` then the trigger queues of all descendent scopes are emptied.

Algorithm 1: EVM-main	**Algorithm 2: UpdateEvent**

loop
  wait for input e'
  disable trigger-related interrupts
  UpdateEventCounter(*EventFilter*,e')
  TerminateScope (*EventFilter*,e')
  ReactScope (*EventFilter*,e')
  enable trigger-related interrupts
  if e' is `tick` event **then**
    invoke system scheduler on *Active-TaskSet*
  end if
end loop
	for all scopes $s \in$ *EventFilter* **do**
	  for all event instances $i \in s$ **do**
	    if $(f = A, R) \vee ((f = F) \wedge$ (*s is a leaf*)) **then**
	      next(i, e');
	      if (*i is enabled*) $\wedge(f = A)$ **then**
	        RemoveTriggers from sub-tree rooted by s
	      end if
	    end if
	  end for
	end for

Next the procedure *TerminateScope* (Algorithm 3) is invoked. The procedure terminates all terminating scopes and terminates the tasks that are released in the scope. A scope is *terminating* if it is a leaf scope and its terminating event instance is enabled. First, a terminating scope is terminated by removing the corresponding node. Second, for each task instance (t, p_o, s_{p_i}) of the removed scope, the port values of p_o in Σ are updated by applying task t to the port values of p_i given by s_{p_i}. Third, if the removed scope is `asap`-parallel, then the trigger queues of all its sibling scopes and their descendants are emptied. The procedure is iterated until there are no terminating reactions.

Algorithm 3: TerminateScope	**Algorithm 4: ReactScope**

while there exists a terminating scope $s \in$*EventFilter* **do**
  terminate the tasks released in s
  terminate s
  if s is bound by `asap` parallelism **then**
    RemoveTriggers from the siblings of s and the scopes in the sub-tree rooted by the siblings
  end if
end while
	while there is a reacting scope $s \in$ *Event-Filter* **do**
	  the enabled trigger be (i, m, p, r)
	  if $m =$ `whenever` **then**
	    reset(i)
	    trigger is reinserted in the trigger queue
	  end if
	  for all reaction block r' in r **do**
	    create a new scope s' for r'
	    Interpreter(address of r')
	  end for
	end while

Next the procedure *ReactScope* (Algorithm 4) invokes the enabled triggers for each reacting scope. A scope is *reacting* if it is a leaf scope and its trigger queue contains a trigger with an enabled invoking event instance i; this is called an *invoked* trigger. Let the first invoked trigger be $g = (i, m, p, r)$, then a set of scopes are added to the reacting scope as children — one for each reaction block of r. Moreover, the trigger g is removed from the queue, and if $m = \texttt{whenever}$, then the new trigger $(\text{reset}(i), m, p, r)$ is appended at the end of the trigger queue.

The scope of each new node is computed by executing the corresponding EVM code at the address given by r by the pseudo-code shown in Algorithm 5: the termination event instance of the new scope is determined by the until event of the corresponding reaction block; the trigger queue of the new scope contains one trigger for each `when` and `whenever` statement in the order of the statements; the ready set of the new scope contains one task instance for each `release` statement where the values of the task input ports are taken from Σ; and the parallelism of the new scope is determined by whether the invoked reaction blocks are composed with `asap`- or `wait`-parallelism. The conditionals associated with the trigger and the release statements are evaluated with standard JVM instructions and omitted here for simplicity. The procedures *PortValue* and *EventValue* access the memory location of a port and an event respectively. The procedure shown in Algorithm 4 is repeated until there exists no reacting scopes. The handling of an input event ends here and EVM starts waiting for the next event. However if the last event is a `tick` event, the system scheduler is invoked and a new task starts its execution until the next `tick` event occurs.

Algorithm 5: Interpreter

while PC$\neq \bot$ **do**
  op := *GetInstruction*(PC)
  if $op = \texttt{trigger}(i, m, p, r)$ **then**
    $Q := Q \cup (i, m, p, r)$
  else if $op = \texttt{release}(t)$ **then**
    $T := T \cup t$
  else if $op = \texttt{getport}(p, o)$ **then**
    $s := S[SP]$; $S[SP] := PortValue(p, o + s)$;
  else if $op = \texttt{putport}(p, o)$ **then**
    $s := S[SP]$; $v := S[SP - 1]$;
    $PortValue(p, o + s) := v$; $SP := SP - 2$
  else if $op = \texttt{getevent}(e, o)$ **then**
    $s := S[SP]$; $S[SP] := EventValue(e, o + s)$;
  else if $op = \texttt{enterscope}$ **then**
    $Q := \emptyset$; $T := \emptyset$;
  else if $op = \texttt{exitscope}$ **then**
    PC:= $\bot$
  else
    Java code interpreter
  end if
end while

Chapter 4

EVM code generation and run-time implementation

This section discusses the xGIOTTO tool chain comprising the compiler and the EVM run-time implementation. The prototype implementation has been shown in Figure 4.1. The compiler checks the syntactic requirements and perform the three analyses described earlier and the run-time implementation executes the program with the event filter, the modified E machine and the scheduler. The sections concludes with an discussion on the possible extensions of the xGIOTTO tool chain.

4.1 Compile-time implementation.

An application is specified using the xGIOTTO program. The xGIOTTO compiler (Figure 4.1) checks for the syntactic correctness and then checks for presence of races; if race exists the trace of event leading to the race is provided. If no race is detected the time safety check is carried out relative to the worst-case-execution-time (WCET) mapping of the tasks and the real time platform. If the program is not schedulable, the information is passed for further iteration; otherwise a scheduling strategy is provided to the run-time system. The code generator produces code which can be divided into two parts, *reaction code* and *task code*. Reaction code is essentially augmented JVM instructions, whereas task code is JVM instructions.

4.2 Run-time Implementation.

The xGIOTTO run-time environment (Figure 4.1) consists of three interacting components: the event filter, the modified E machine, and the scheduler. The *event filter* implements the event-scoping mechanism and presents the filtered events to the E machine. The implementation of the event filter is same as discussed in the last section. At run-time, the occurrence of an event is processed by the event filter. The event filter computes the event transition and the termination transitions on the tree of event

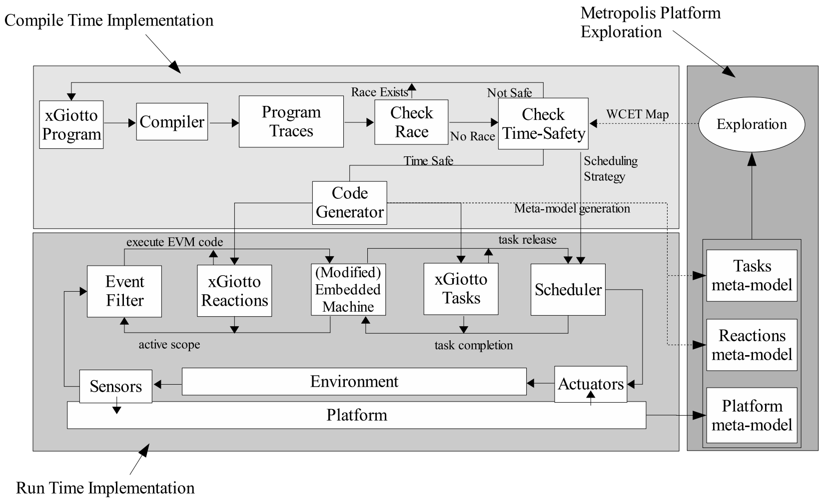


Figure 4.1: Implementation

scopes and gives to the E machine a set of EVM code addresses, which correspond to the invoked reaction blocks. The EVM interprets the EVM code, thus performing the reaction transitions. The EVM code instructions may release new tasks to the scheduler and enable new triggers. When all invoked reactions have been processed by the machine, the system scheduler chooses a task to execute from the ready set of the active event scopes, and whenever such a task completes, the EVM is notified. In addition, the EVM monitors the running tasks by detecting task overruns (time-safety violations). If a task overrun is detected (i.e. if a task termination event arrives before the task completes), a run-time exception is generated. The platform interacts with the environment through actuators and sensors. The actuators are driven by the task outputs and the sensors generate *atomic* events (interrupts), which are handled by the event filter. The prototype is implemented in Java and is able to run any XGIOTTO program and to interprets a subset of the JVM bytecode [13].
OSEKWorks is a real time platform from WindRiver which implements the OSEK standard [10] for embedded applications in the domain of automotive and industrial automation. Currently work is under progress to port the EVM on OSEKWorks and implement the AFR controller (discussed in the next section).

4.3 Extensions

XGIOTTO is being integrated with Metropolis [14]. Metropolis is a unified framework which allows the simulation and exploration of different runtime platforms. By incorporating XGIOTTO in the metropolis framework we allow the flexibility of studying and exploring the embedded software design. The XGIOTTO compiler generates Metropo-

lis meta-model descriptions (Figure 4.1) of the reactivity and of the tasks. Given a description of the platform and its environment, Metropolis is capable of analyzing the platform and generates a worst-case-execution-time mapping for the tasks.
The Ptolemy project [15] studies modeling, simulation, and design of concurrent, real-time, embedded systems. It focuses on the use of well-defined models of computation and how heterogeneous mixture of such models of computations can be used for modelling and analyzing embedded systems. In the future we are interested in implementing an XGIOTTO domain in Ptolemy to further investigate the embedded system applications design and development.
Simulink is a well known simulation and code-generation environment for control applications in automotive and avionics domain. Simulink is used in association with code generators for real-time systems like Real-Time Workshop [16]. In the future we would like to explore the integration of XGIOTTO and Simulink as has been done in Giotto-Simulink translator [17].

Chapter 5

Case Study

The case study presents a problem from the automotive industry. While an engine is in motion the engine-fuel system needs to inject fuel into the combustion engine with the optimal amount of air [18]. This is commonly known as air-fuel ratio (AFR) control in the automotive domain. The operation is very sensitive to the timing of the AFR controller. The AFR controller updates the fuel-injectors (actuators) with the amount of fuel to be injected in the next engine cycle. The fuel is sprayed to the intake ports once the valves are closed and with enough time allowance so that the mixture forms before the valves open. The valve timing depends upon the engine speed; this forces the AFR controller to issue updates at aperiodic time intervals. The right half of Figure 5.1 shows the AFR controller. There are two shafts in the engine: crank and cam shaft. For every rotation in the wheel, the cam rotates once and crank rotates twice. There is only one teeth in the cam shaft. There are six teeth slots in the crank. However one slot is left empty to synchronize with the cam shaft. On receiving signal (which is synchronized with the arrival of the teeth), each injector injects a certain amount of precalculated amount of fuel. For each engine cycle an injector sprays once. Initially we assume that the engine is at top dead center (as illustrated in Figure 5.1). At this point the AFR controller computes how much fuel is needed for the cylinder in the next engine cycle – and the amount of fuel each injector needs to inject. In this discussion the injector response time and other non-linear effects have been omitted. The cam shaft pulse is used as a reference to obtain the numbering of the crank shaft teeth (which we have chosen to be 5 + 1 missing) as shown in the figure. For our case only three injectors have been considered; however they can be increased to any number allowed by the dynamics of the system. The crank shaft teeth is converted to pulses via the signal conditioning circuitry. The i-th injector injects at the i-th teeth pulse as shown. When the crank shaft reaches the end of the engine cycle, denoted by tooth number 10, the *calcInjPar* routine is initiated which calculates the fueling parameters for the next engine cycle. The computation has to be completed before the arrival of the tooth number 1. At this instance the injection for first injector is initiated. The fuel parameter is actually scaled to the length of a pulse proportional to the mass to be injected. Thus the fuel injection can be represented by a pulse of width proportional to the mass of fuel to be injected by the respective injector.

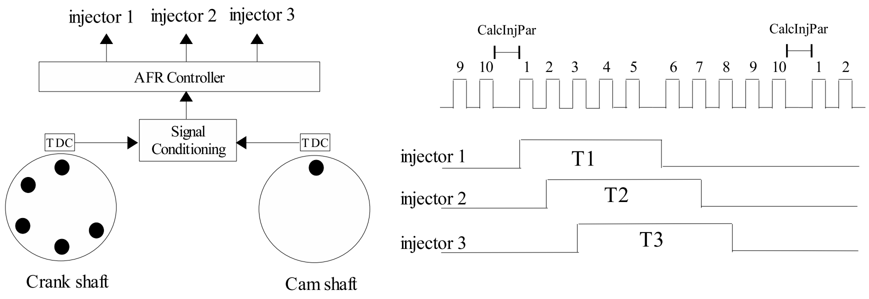


Figure 5.1: AFR Controller — Implementation

xGiotto Program

An XGIOTTO program that simulates the behavior of the controller has been shown in Figure 5.2. The controller presented is used when the engine is warm, running in fourth gear and the minimum speed of the system is 1000 rpm. This implies that between each slot of crank shaft there is a minimum time gap of 5ms (one engine revolution every 60ms). The event `tick` in the program denotes `1 ms` event. There are three fuel ports `f1, f2, f3` for the three injectors and are of type integer. The i-th fuel port is updated with the mass of fuel to be injected for the i-th injector. There are three (one for each injector) pulse ports and simulate the fuel injection. The ports `p1, p2, p3` are pulse ports and are boolean ports which are normally grounded (or reset). They are set to high (or `true`) for the time equal to the length of the pulse (i.e. the value stored in the corresponding fuel port) to simulate the injection. In actual implementation the tasks `set` and `reset` will correspond to the closing and opening of the valve for an injector. The task `dec` checks for the remaining length of the pulse; in real implementation it will simulate the pulse generation technique to inject the required mass of fuel. The controller starts with the signal `synch` which is invoked at the synchronization of TDC for the crank and cam shafts. At the signal, reaction mode `start` is invoked which calls the reaction `controller` once for each revolution of the wheel and thus for the arrival of every 10 teeth (whenever a teeth passes the TDC a `teeth` event is generated) of the crank shaft. Once `controller` is invoked, the task `CalcInjPar` is released to calculate the mass of fuel to be injected for each of the injector and updates the three fuel ports `f1`, `f2` and `f3` by the arrival of the next teeth. This implies that at the worst case it takes 10ms for its computation (the toothless slot is included here) and is expressed by the deadline `[10ms : teeth]`. At the arrival of the first tooth on the cycle the three injectors (the three reaction blocks `channel1, channel2, channel3`) are invoked in parallel. The reaction `channel`*i* handles the injection of the i-th injector. For the i-th injector, the injection should start at the arrival of the i-th teeth; so the task `set` sets the value of the pulse variable of the injector at the i-th teeth. For the block `channel2` an initial gap of a tooth is provided and then `set` is

```
program Air-Fuel-Ratio-Controller {

port
 /* fuel ports */
 int f1; int f2; int f3;
 /* pulse ports */
 bool p1; bool p2; bool p3;

event
 int synch;
 int stop;
 int teeth;

task set () output (int o)
 { o = true; }

task reset (int i) output (int o)
 { if (i == 0) o = false; }

task dec (int i) output (int o)
 { if (i > 0) o = i - 1; }

task calcFuelInj ()
 output (int f1, int f2, int f3)
 {/* Fuel Parameter Computation */}

react channel1() {
 when [now]
 {release set () (p1);} until [1time];
 loop
  react {release reset (f1) (p1);
          release dec (f1) (f1);
        } until [time]
  end;
} until [int e]

react channel2() {
 when [now] react {} until [5time : teeth];
 when remember [5time : teeth] react
  {release set () (p2);} until [time];
 react {
  loop react {release reset (f2) (p2);
               release dec (f2) (f2);
             } until [time];
  end; }
} until [int e]


react channel3() {
 when [now] react {} until [10time : 2teeth];
 when remember [10time : 2teeth] react
  {release set () (p2);} until [time];
 react {
  loop react {release reset (f3) (p3);
               release dec (f3) (f3);
             } until [time];
  end; }
} until [int e]

react calcFuel() {
 release calcFuelInj() (f1, f2, f3);
} until [int e]

react controller() {
 when [now] react calcFuel()
           until [10time : teeth];
 when remember [teeth]
  react channel1() until asap [50time : 9teeth] ||
  react channel2() until asap [50time : 9teeth] ||
  react channel3() until asap [50time : 9teeth];
} until [int e]

react start() {
 when [now] react controller()
           until remember [10teeth];
 whenever remember [10teeth]
   react controller() until remember [10teeth];
} until [int e]

{when [synch] react start() until asap [stop];}

}
```

Figure 5.2: AFR Controller — xGIOTTO program

released on the pulse port p2 (with a LET of 1ms). Following this a loop is started which repeats every ms and schedules tasks dec and reset. The implementation of the controller on OSEK is currently under development.

Chapter 6

Conclusion

In this report we presented the Embedded Virtual Machine. The EVM simplifies implementing and porting of XGIOTTO to any other RTOS. The virtual machine code is simple enough to be ported to any RTOS. This is not the most optimal approach since it involves the interpretation on top of the hosting RTOS. An optimal but complex solution would be to implement the EVM "bare bone" on the target platform, reducing the overhead to the minimum. In this paper we propose a modified Java Virtual Machine (JVM). However the concept can be applied to any other instruction set. Choosing the JVM as a starting point for our investigation allowed us to have access to and be inspired by several different implementation of the original JVM, such as the JikesVM, JamaicaVM, aJile and JBed to name a few.
In particular we were inspired by the leJOS [19] and TinyOS [20] systems. The TinyOS system is targeted towards sensor-networks applications implemented in nesC and leJOS is targeted towards implementing applications for the Lego Mindstorms. Both the systems are characterized by their succinctness (15Kb for leJOS and 4Kb for TinyOS) and are targeted to run on microprocessors with a very small memory footprint (the Hitachi H8 for leJOS, and the Atmel AVR for TinyOS).
Moreover in this work we attempted to bring some of the threading functionality usually found in real-time operating system at the instruction level. This can be compared to the introduction of object-oriented functionality (such as object instantiation) on the JVM.

Bibliography

[1] Burns, A., Wellings, A.: Real-Time Systems and Programming Languages. 3rd edn. Addison Wesley (2001)

[2] Kirsch, C.M.: Principles of real-time programming. In: EMSOFT 02: Embedded Software. Lecture Notes in Computer Science 2491. Springer-Verlag (2002) 61–75

[3] Buttazzo, G.: Hard Real-Time Computing Systems. Kluwer Academic Publisher (1997)

[4] Halbwachs, N.: Synchronous Programming of Reactive Systems. Kluwer Academic Publisher (1993)

[5] Ghosal, A., Henzinger, T.A., Kirsch, C.M., Sanvido, M.A.A.: Event-driven programming with logical execution times. In: HSCC 04: Hybrid Systems Computation and Control. Lecture Notes in Computer Science 2993. Springer-Verlag (2004) 357–371

[6] Henzinger, T.A., Horowitz, B., Kirsch, C.M.: GIOTTO: A time-triggered language for embedded programming. Proceedings of the IEEE **91** (2003) 84–99

[7] Henzinger, T.A., Kirsch, C.M.: The Embedded Machine: Predictable, portable real-time code. In: Proceedings of Programming Language Design and Implementation, ACM Press (2002) 315–326

[8] Henzinger, T.A., Kirsch, C.M., Majumdar, R., Matic, S.: Time-safety checking for embedded programs. In: EMSOFT 02: Embedded Software. Lecture Notes in Computer Science 2491. Springer-Verlag (2002) 76–92

[9] Henzinger, T.A., Kirsch, C.M., Sanvido, M.A.A., Pree, W.: From control models to real-time code using GIOTTO. IEEE Control Systems Magazine **23** (2003) 50–64

[10] OSEK: (http://www.osek-vdx.org)

[11] Sanvido, M.A.A.: A computer system for model helicopter flight control. Technical Report Technical Memo Nr. 3: The Software Core, Institute for Computer Systems, Technical Report 317, ETH Zürich (1999)

[12] Kirsch, C.M., Henzinger, T.A., Sanvido, M.A.A.: A programmable microkernel for real-time systems. Technical Report UCB//CSD-03-1250, UC Berkeley (2003)

[13] Lindholm, T., Yellin, F.: Java Virtual Machine Specification, Second Edition. Addison-Wesley (1999)

[14] Balarin, F., Watanabe, Y., Hsieh, H., Lavagno, L., Passerone, C., Sangiovanni-Vincentelli, A.: Metropolis: An integrated electronic system design environment. IEEE Computer **36** (2003) 45–52

[15] Ptolemy: (http://ptolemy.eecs.berkeley.edu/)

[16] Real-Time Workshop: (http://www.mathworks.com/products/rtw/)

[17] Stieglbauer, G., Pree, W.: Visual and interactive development of hard real time code. In: Future Generation Software Architectures in the Automotive Domain. Lecture Notes in Computer Science. To be published by Springer-Verlag (2004)

[18] Simsek, T.: Technical overview of the expanded powertrain challange problem. Technical report (Mobies PI meeting, 2002)

[19] leJOS: (http://lejos.sourceforge.net/)

[20] TinyOS: (http://webs.cs.berkeley.edu/tos/)