Separate Compilation of Hierarchical Real-Time Programs into Linear-Bounded Embedded Machine Code*

Arkadeb Ghosal
UC Berkeley
arkadeb@eecs.berkeley.edu

Daniel Iercan
"Politehnica" U. of Timisoara
daniel.iercan@aut.upt.ro

Christoph M. Kirsch
University of Salzburg
ck@cs.uni-salzburg.at

Thomas A. Henzinger
EPFL
tah@epfl.ch

Alberto Sangiovanni-Vincentelli
UC Berkeley
alberto@eecs.berkeley.edu

ABSTRACT

We have recently proposed a coordination language, called Hierarchical Timing Language (HTL), for distributed, hard real-time applications. HTL is a hierarchical extension of Giotto and, like its predecessor, based on the logical execution time (LET) paradigm of real-time programming. Giotto is compiled into code for a virtual machine, called the Embedded Machine (or E machine). If HTL is targeted to the E machine, the hierarchical program structure needs to be flattened which makes separate compilation difficult and may result in code of exponential size. In this paper, we propose a generalization of the E machine which supports a hierarchical program structure at runtime through real-time trigger mechanisms that are arranged in a tree. We present the generalized E machine, and a modular compiler for HTL that generates code of linear size. The compiler may generate code for any parts of a given HTL program separately in any order.

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and embedded systems

General Terms

Language, Compiler, Virtual Machine

Keywords

Real Time, Hierarchy, Code Generation

1. INTRODUCTION

Hierarchical Timing Language (HTL) is a hierarchical coordination language for distributed, hard real-time applications [6]. HTL programs determine portable and predictable real-time behavior of periodic software tasks running on a possibly distributed system of host computers. An HTL program specifies task-to-host mappings, task frequencies, mode switching, and I/O times and dependencies but not task implementations, which are assumed to be done in some general purpose language such as C or Java. A task in HTL is essentially sequential code that reads input, computes, and writes output. HTL offers two fully hierarchical programming constructs: (sequential, conditional, parallel) *composition* of tasks as well as *refinement* of abstract into concrete tasks. An abstract task has a frequency, specific I/O times and dependencies, and a worst-case execution time (WCET) but no implementation. An abstract task is a temporally conservative placeholder for a concrete task with an implementation. A concrete task refines an abstract task if the concrete task has the same frequency but at least as much time to compute, i.e., possibly relaxed I/O times and dependencies, and as much or smaller WCET than the abstract task. The result is that a concrete HTL program is time-safe (schedulable) if it refines a time-safe abstract HTL program [6]. In general, checking refinement in HTL is exponentially faster than checking time safety (schedulability). However, there are abstract HTL programs that are not time-safe but for which time-safe refinements exist.

After checking refinement and time safety, HTL programs are compiled into so-called E code of the Embedded Machine [9] or E Machine. E code is virtual machine code with specific instructions for timing I/O activity, native task computation, and host-to-host communication. Time-safe E code is portable and predictable, and therefore provides a hardware- and OS-independent target abstraction for compiling possibly distributed real-time programs. Further compiling E code into native code is possible but has so far not been necessary even for high-performance applications such as helicopter flight control [12, 11]. However, E code has originally been designed as target for compiling non-hierarchical programs written in Giotto [8], which is the predecessor of HTL. As a consequence, HTL compilation into conventional E code involves flattening the input HTL programs and may therefore result in exponentially larger output E code programs. Flattening also prohibits compiling parts of large HTL programs separately.

In this paper, we propose to extend E code by adding instructions for maintaining hierarchical program structure at runtime to enable separate compilation of (parts of) HTL programs into E code programs whose size linearly bounded by the size of HTL code. Our solution trades off runtime performance for compile-time convenience and E code size because execution of E code compiled from flattened HTL programs may result in lower runtime overhead than the execution of such *hierarchical E code*, or *HE code*. All original and most new instructions can be executed in constant time. However, a single new instruction that involves traversing hierarchical structure requires linear time with respect to the size of the original HTL program. This may only be avoided by again flattening

*This work was supported in part by the GSRC grant 2003-DT-660, the NSF grant CCR-0208875, HYCON, the Artist II Network of Excellence on Embedded Systems Design, the European Integrated Project SPEEDS, the SNSF NCCR on Mobile Information and Communication Systems, the Austrian Science Fund Project P18913-N15 and the Center for Hybrid and Embedded Software Systems (CHESS) at UC Berkeley, which receives support from the National Science Foundation (NSF award #CCR-0225610), the State of California Micro Program, and the following companies: Agilent, DGIST, General Motors, Hewlett Packard, Infineon, Microsoft, and Toyota

HTL programs prior to compilation. For simplicity and clarity, we have chosen to define new instructions in RISC style where most instructions have rather simple, "atomic" semantics. So far, runtime performance has not been an issue but may easily be improved using CISC-style macro instructions.

The contributions of this paper are the design of HE code (Section 5), the design of compile-time support for separate HTL compilation (Section 6, Section 7) into HE code, the implementation of runtime support for HE code as part of an existing E Machine implementation, and the implementation of compile-time support for separate HTL compilation into HE code in an existing HTL compiler implementation [1]. Throughout the paper we use a case study (Section 2) to illustrate the contributions of the paper. Sections 3 and Section 4 discuss the key features of HTL and E Machine, respectively. Section 8 compares our approach with related work.

2. CASE STUDY

The case study implements a distributed real-time controller for a three-tank system (3TS in short). There are three tanks `T1`, `T2` and `T3` (Fig. 1) each with an evacuation tap `tap1`, `tap2` and `tap3` respectively. The tanks are interconnected via taps `tap13` and `tap23`. Two pumps, `P1` and `P2`, feed water in the tanks `T1` and `T2` respectively. The goal of the controller is to maintain the level of water in tanks `T1` and `T2` under the presence and absence of perturbations (simulated by the evacuation taps). If there is no perturbation, a P (proportional) controller is used; under perturbations, a PI (Proportional Integral) controller is used [10]. The modeling generates four possible scenarios: (1) both pumps controlled by P controllers, (2) `P1` and `P2` controlled by P and PI controllers respectively, (3) `P1` and `P2` controlled by PI and P controllers respectively, and (4) both pumps controlled by PI controllers.

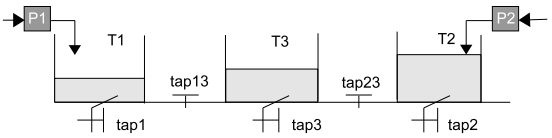

Figure 1: Overview of three tanks system

The controller is implemented in a distributed fashion on three E machines (Fig. 2). Each E Machine is implemented in C on a Unix machine. The three E machines implement the controller for `P1`, the controller for `P2`, and the interface controller. The tasks are implemented in C. The schedulers in the Unix machines are used for scheduling the released tasks. The E Machines communicate with each other through UDP. Communication with the 3TS plant (reading heights of water in the tanks and sending fill debit for each pump) is done via a TCP server implemented on a Windows 98 machine. Refer to *http://htl.cs.uni-salzburg.at/HEcode* for implementation details and online demo.

3. HIERARCHICAL TIMING LANGUAGE

HTL is centered around two constructs: the core computation and communication model, and the hierarchal programming structure. The first deals with task specification and communication between tasks, while the second deals with composition and refinement of tasks.

Computation and Communication Model. The computation model is the *Logical Execution Model* (LET) of task execution. A LET task is a sequential code block with no internal synchronization points. Each task has a release event and a termination event specified by clock ticks or completion events of other tasks. The task reads the inputs at the release event (even if the task starts executing later) and the task updates the outputs at the termination

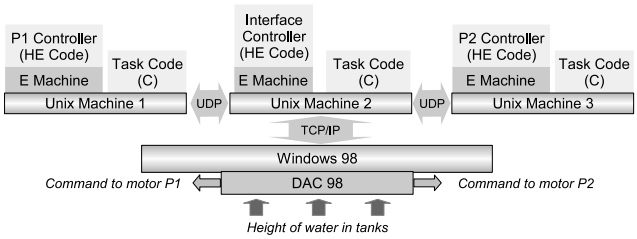

Figure 2: Overview of implementation

event (even if the task terminates earlier). The LET model decouples time when the input is read and output is written from actual execution which makes the model time- and value-deterministic, portable and composable [9].

The communication model of HTL is based on *communicator* [6], a typed variable that can be accessed (read from or written to) with a specified periodicity. Communicators are used to exchange data with environment (sensors and actuators are special cases of communicators) or between tasks. A task in HTL reads from certain instances of some communicators, computes a function and writes to certain instances of other communicators. Fig. 3 shows three communicators, `h1` (period 100ms), `u1` (period 100ms) and `p1` (500 ms): `h1` denotes the height in tank `T1`, `u1` denotes the motor current (for pump `P1`) computed by the controller and `p1` denotes the perturbation in tank `T1`. Task `t1` reads the fourth instance of `h1`, computes control law for tank `T1` and writes to the fifth instance of `u1`. The latest read and earliest write time implicitly specify the LET of the task; in case of `t1`, the LET is from 300 to 400 ms. The sequential code of the task is not expressed in HTL but in a "foreign" language (e.g. C in our example).

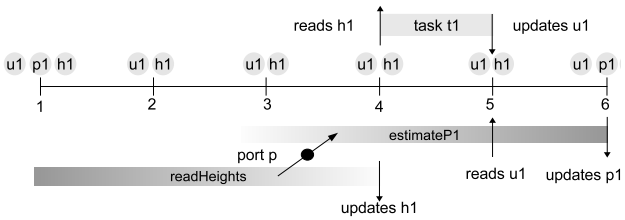

Figure 3: Interaction between tasks and communicators

Tasks can also communicate with one another through untimed variables referred as *ports*. Fig. 3 shows two tasks `readHeights` and `estimateP1` communicating via port `p`. Task `readHeights` reads the sensors and writes to the fourth instance of communicator `h1`. Task `estimateP1` reads port `p` and fifth instance of `u1`, computes perturbation for tank `T1` and writes to the second instance of `p1`. Task `estimateP1` reads the port `p` and hence reads the output of the task `readHeights` as soon as the task `readHeights` completes execution and does not have to wait until the fourth instance of `h1`.

Hierarchical Programming Structure. A set of interacting tasks with the same frequency form an HTL *mode* with a specified mode period. For example, the tasks `readHeights` and `estimateP1` belong to mode `imode`, which has a period of 500ms. All tasks in a mode execute with the periodicity of the mode. The tasks within a mode interact through ports and communicators; tasks from different modes interact only through communicators. For example, `readHeights` and `t1` are in different modes and interact through communicator `h1`; `readHeights` writes `h1` while `t1` reads `h1`. HTL allows mode switching (at the end of mode periods) to model changes in real-time controllers. In the complete specification we define two modes `oneP` and `onePI` invoking P and PI control tasks for pump `P1` respectively; the modes switch between themselves based on the perturbation in tank `T1`(i.e. the

value of the communicator `p1`). A network of modes (with one being the start mode) and mode switches is an HTL *module*; e.g. modes `oneP` and `onePI` are grouped in one module. An HTL *program* is a set of modules and a set of communicators. The modes within a module are composed sequentially while modes from different modules are composed in parallel. The communicators are used to exchange data between tasks in same module (but possibly different modes) and between tasks in different modules. The HTL program (Fig. 4) for the controller, `3TS_Controller`, consists of three modules `pumpOne`, `interface` and `pumpTwo` and six communicators. Refer to *http://htl.cs.uni-salzburg.at/HEcode* for the full specification and the complete program.

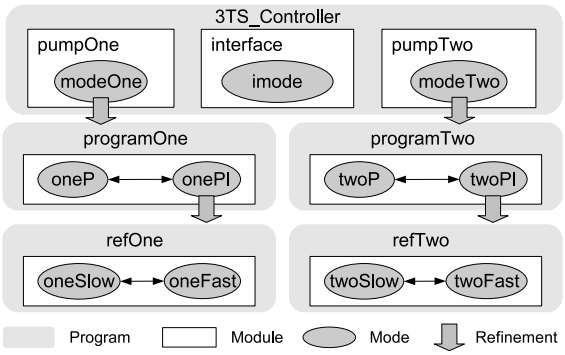

Figure 4: HTL program for 3TS controller

A mode (referred as *parent mode*) can be *refined* by another HTL program (referred as *refinement program*); any mode in the refinement program is a *child mode* of the parent mode. The mode `modeOne` is refined (Fig. 4) by program `programOne` which has a single module with two modes `oneP` and `onePI`. Both the modes `oneP` and `onePI` are child modes to parent mode `modeOne`. Each task (referred as *child task*) in a child mode maps to a unique task (referred as *parent task*) in the parent mode. Modes `modeOne`, `oneP` and `onePI` invokes tasks `t1`, `t1P` and `t1PI` respectively with `t1` being the parent of the other two tasks. During execution, the parent task is replaced by the child task i.e. instead of `t1`, either `t1P` or `t1P1` executes. While `t1` represents a control task for pump `P1`, `t1P` and `t1PI` are the P and PI version of the controller respectively. In other words, parent task is an *abstract* specification while children tasks are *concrete* implementations of the specification. Instead of specifying a functional behavior, a parent task specifies the timing behavior of the concrete task e.g. parent mode and child mode have identical periods, child task cannot be released (resp. terminated) later (resp. earlier) than the parent task and the WCET of the child task is bounded by that of the parent task; these constraints are referred as *refinement constraints*. There can be tasks (in parent mode) which are not parent to any child task and will execute in parallel with tasks in child mode. Mode refinement does not add expressiveness; an HTL program with multiple levels of refinement can be translated into an equivalent *flat* program without refinement. Mode `modeOne` can be replaced by the switching modes `oneP` and `onePI`. However mode refinement helps in a *structured and concise* specification. In the top-level, mode `modeOne` invokes control task for pump `P1`; however no distinction is made for different scenarios. In the second level (program `programOne`) the distinction is made between absence and presence of perturbations and thus requiring the use of P and PI control tasks. There can be subsequent refinement (e.g. `refOne`) which distinguishes slower and faster invocations of PI control task. Refinement helps in succinct expression of *choice* (a task is parent to several chidren tasks in different sequential modes), *change* (parent and child task have different I/O), *space* (empty parent tasks that can be refined later) and *replacement* (replacing a refinement program with another). Mode refinement helps in conservatively simplifying program analysis: e.g. schedulability check can only be done for the top-level program, and the refinement constraints preserve [6] the schedulability across the hierarchy.

Distribution. HTL modules can be distributed over several hosts. Distribution is specified through a mapping of top-level modules to hosts. All refinements of all modes in a top-level module are bound to the same host to which the module is mapped. The distribution is implemented by replicating shared communicators on all hosts, and then have the tasks that write to shared communicators broadcast the outputs. For this purpose, the LET model is extended to include both WCETs as well as the worst-case output transmission times (WCTTs). The semantics (i.e., the real time behavior) of an HTL program is independent of the number of hosts, but code generation and program analysis take the distribution into account. In the case study, the three modules `pumpOne`, `interface` and `pumpTwo` are implemented on three different hosts.

4. THE EMBEDDED MACHINE

The Embedded Machine or E Machine controls the release of tasks and the time when variable values are exchanged (i.e. copied or initialized). The variables are accessed through so called *drivers*. A task or a driver is implemented in any other language e.g. C. In the original E Machine definition there are six E code instructions. There are three non-control flow instructions: $call$, $release$ and $future$. The instruction $call(d)$ executes a driver d. The instruction $release(t)$ releases a task t for execution. The task may not be immediately executed; the actual execution of the task will depend on the real-time schedular being used. The instruction $future(e, a)$ marks E code at address a for future execution when the predicate e evaluates to true, i.e., when e is *enabled*. The pair (e, a) is a trigger: predicate e observes events such as time tick events (raised by the real-time clock) and completion events of tasks (raised by the executing platform) and is enabled when all observed events have occurred. The E machine maintains a FIFO queue of triggers. If multiple triggers in the queue are enabled at the same instant, the corresponding E code is executed in FIFO order, i.e., in the order in which the $future$ instructions that created the triggers were executed. There are two control flow instructions: if and $jump$. The conditional instruction $if(cnd, a)$ branches to the E code at address a if predicate cnd evaluates to true. A *condition* cnd observes variable states. The non-conditional control flow instruction $jump(a)$ executes an absolute jump to E code address a. There is one termination instruction $return$ which completes the execution of an E code sequence.

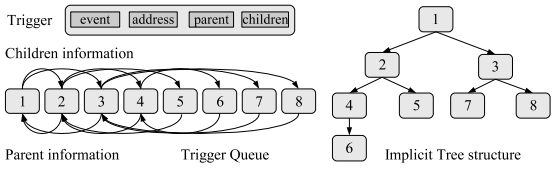

Figure 5: Triggers, queue of triggers and implicit tree

We make the following changes to allow execution of hierarchical code on E Machine. *First*, the trigger definitions are modified. Each trigger in addition to an event predicate and E code address, tracks a parent trigger and a set of children triggers. With the new trigger definition, a trigger queue is an implicit tree (Fig. 5). *Second*, two stacks are added to track the hierarchy of the program. The stacks are used to remember the position of code being executed in the hierarchy of the whole program and to add parent and children information to newly created triggers. *Third*, the modified E machine maintains three trigger queues instead of one. While one FIFO queue order the actions of simultaneously ordered triggers,

parallel FIFO queues provide second ordering on simultaneously enabled triggers. In case of code generated for HTL programs, the multiple queues are used to order communicator updates, mode switche checks, communicator reads and task releases. *Fourth*, E code instructions are modified/ added to operate on the new triggers and to access the stacks and the queues. The new E code is referred as *Hierarchical E code* (HE code).

5. SEMANTICS OF HE CODE

The semantics of an HE code program can be represented as a set of traces where each trace is a sequence of configurations. Each configuration tracks the following: state of program variables, set of released tasks, queues of triggers, address of the current instruction being executed, set of registers storing trigger names, stack of trigger names and stack of addresses. Formally, a *trace* is a (possibly infinite) sequence of configurations $u_0, u_1, \cdots$ where u_0 is the starting configuration. Each configuration is a tuple $(state, writeQ, switchQ, readQ, tasks, PC, R0, R1, R2, R3, parent\_stack, address\_stack)$, where $state$ is variable state, $writeQ$, $switchQ$ and $readQ$ are FIFO queues of triggers, $tasks$ is a set of tasks, PC is a program counter, $R0$, $R1$, $R2$, and $R3$ are registers to store trigger names, $parent\_stack$ is a stack of trigger names, and $address\_stack$ is a stack of addresses. For any two consecutive configurations u_{i-1}, u_i where $i > 0$, u_i is the result of progress of clock (time tick event), completion of task (task completion event) or execution of an instruction (see below) at configuration u_{i-1}.

The variable state $state$ tracks the values of program variables; e.g. for HTL programs the variables are communicators and ports. The task set $tasks$ tracks the set of tasks released for execution; once a task completes execution the task is removed from $tasks$. The program counter PC is the address of the current instruction being executed. The set of program addresses is $adrset \cup \{\bot\}$; $PC = \bot$ signifies there is no instruction being executed and the E machine is either checking for enabled triggers or waiting for an event. We will denote the instruction at address a as $ins(a)$ and the next address following a as $next(a)$.

A trigger g is a tuple $(e, a, par, clist)$, where e is an event, a is an address, par is a trigger name, and $clist$ is a list of trigger names. An *event* is a pair $(n, cmps)$, where $n \in \mathbb{N}_{\geq 0}$ and $cmps$ is a set of task names. The positive integer n denotes the number of time tick events being waited for. The set $cmps$ denotes the tasks whose completion event is being waited for. A trigger is *enabled* when $n = 0$ and $cmps = \emptyset$. When a trigger is created, it is assigned an unique name until the trigger is removed. A *trigger name* is the reference to a trigger; a trigger can be accessed through trigger names. The registers store trigger names. A register can be copied and/or reset without affecting the trigger unless the trigger is removed or modified by HE code instructions. The triggers are unique identities and are not duplicated; however they can be modified when events occur. A trigger may be *modified* by updating the associated event, changing the parent, or by modifying the children list. The trigger queues $writeQ$, $switchQ$ and $readQ$ are FIFO queues of triggers. A trigger can be present in at most one queue.

The address stack tracks the hierarchical position of the program, mode and module for which code is being executed. The parent stack remembers the hierarchy of the switch triggers. There are two operations to access the stacks: $push$ and pop. Operation $push(address\_stack, a)$ pushes address a on $address\_stack$. Operation $pop(address\_stack)$ returns the top value of $address\_stack$; the value is an address a if the stack is non-empty, $\bot$ otherwise. Operation $push(parent\_stack, Rx)$ pushes the trigger name stored in register Rx (where $x \in 0, 1, 2, 3$) on $parent\_stack$. Operation $pop(parent\_stack)$ returns the top value of $parent\_stack$; the value is a trigger name if the stack is non-empty, $\bot$ otherwise.

The E machine is *waiting* if none of the triggers in any of the queues are enabled, $PC = \bot$ and address stack is empty. The machine is in state *writing* if there exists at least one enabled trigger in the write queue. The machine is in state *switching* if there exists no enabled trigger in the write queue but there exists at least one enabled trigger in the switch queue. The machine is in state *post-switch* if there exists no enabled trigger in the write and the switch queue but there exists at least one enabled trigger in the read queue.

If the machine is waiting, a time tick or a task completion event updates the event for the triggers. For a time tick event: for all triggers $((n, \cdot), \cdot, \cdot, \cdot)$ where $n > 0$, the trigger is updated to $((n - 1, \cdot), \cdot, \cdot, \cdot)$. For a completion event for task `t`: for all triggers $((\cdot, cmps), \cdot, \cdot, \cdot)$ and $\texttt{t} \in cmps$, the trigger is updated to $((\cdot, cmps \setminus \{\texttt{t}\}), \cdot, \cdot, \cdot)$. If the E machine enters into non-waiting state (by enabling some triggers) after handling an event, the write queue is traversed in FIFO order until an enabled trigger is found and the trigger is handled. When a trigger $(\cdot, a, \cdot, \cdot)$ is *handled*, program counter PC is set to a, the name of the trigger is stored in register $R0$ and the trigger is removed from the queue. The E machine continues the execution at addresses following a until a $return$ instruction is executed. When a $return$ execution is executed, the trigger (which triggered the code execution) is deleted from the system and code execution starts from the address popped from the address stack. This is continued until the address stack is empty. At this point the control starts searching for other enabled triggers in the write queue; if no other trigger is enabled, the machine enters into switching state. If the E machine enters into *switching* state, the switch queue is traversed in FIFO order (and enabled triggers are handled) until the machine is in state *post-switch*. If the E machine enters into *post-switch* state, the read queue is traversed in FIFO order (and enabled triggers are handled) until the machine is in state *waiting*. The handling of triggers in all the three queues are identical.

Next we discuss the effect of executing the HE code instructions. Let the configuration be $(state, writeQ, switchQ, readQ, tasks, PC, R0, R1, R2, R3, parent\_stack, address\_stack)$ when an instruction at address a is being executed (i.e. $PC = a$). Once the instruction is executed, the new configuration be $(state', writeQ', switchQ', readQ', tasks', PC', R0', R1', R2', R3', parent\_stack', address\_stack')$. If $ins(a)$ is being executed, $PC' = next(a)$ unless otherwise mentioned. A parameter has the same value over the execution unless otherwise mentioned.

- $ins(a) = call(d)$: driver d is executed which updates variable state to $state'$
- $ins(a) = release(\texttt{t})$: $tasks' = tasks \cup \{\texttt{t}\}$
- $ins(a) = writeFuture(e, a)$: $writeQ' = writeQ \circ g'$ where $g' = (e, a, \bot, \emptyset)$ and $R1'$ stores the name of g'
- $ins(a) = switchFuture(e, a)$: $switchQ' = switchQ \circ g'$ where $g' = (e, a, \bot, \emptyset)$ and $R1'$ stores the name of g'
- $ins(a) = readFuture(e, a)$: $readQ' = readQ \circ g'$ where $g' = (e, a, \bot, \emptyset)$ and $R1'$ stores the name of g'
- $ins(a) = jumpIf(cnd, a)$: if condition cnd is true, then $PC' = a'$ else $PC' = next(a)$
- $ins(a) = jumpAbsolute(a')$: $PC' = a'$
- $ins(a) = jumpSubroutine(a')$: $PC' = a'$ and $address\_stack' = push(address\_stack, next(a))$
- $ins(a) = copyRegister(Rx, Ry)$ where $x, y \in \{0, 1, 2, 3\}$ and $x \neq y$: copy the content of register Rx to register Ry
- $ins(a) = pushRegister(Rx)$ where $x \in \{0, 1, 2, 3\}$: push the content of register Rx on to $parent\_stack$ i.e. $parent\_stack' = push(parent\_stack, Rx)$

- $ins(a) = popRegister(Rx)$ where $x \in \{0, 1, 2, 3\}$: pop content from $parent\_stack$ to register Rx i.e. $Rx' = pop(parent\_stack)$
- $ins(a) = getParent(Rx, Ry)$ where $x, y \in \{0, 1, 2, 3\}$ and $x \neq y$: load the name of parent of trigger pointed to by Rx into register Ry
- $ins(a) = setParent(Rx, Ry)$ where $x, y \in \{0, 1, 2, 3\}$ and $x \neq y$: the trigger name in Ry is stored as the parent of the trigger pointed to by register Rx
- $ins(a) = copyChildren(Rx, Ry)$ where $x, y \in \{0, 1, 2, 3\}$ and $x \neq y$: the children list of the trigger pointed to by Ry is stored as the children list of the trigger pointed to by register Rx
- $ins(a) = setParentOfChildren(Rx, Ry)$ where $x, y \in \{0, 1, 2, 3\}$ and $x \neq y$: set the trigger name in Ry as the parent of all the triggers in the children list of the trigger pointed by register Rx
- $ins(a) = deleteChildren(Rx)$ where $x \in \{0, 1, 2, 3\}$: for all trigger names in children list of trigger referred by register Rx: (recursively) delete the triggers pointed by the children list and remove the triggers from the queue
- $ins(a) = replaceChild(Rx, Ry, Rz)$ where $x, y, z \in \{0, 1, 2, 3\}$ and $x \neq y \neq z$: in the children list of trigger pointed to by register Rx, replace the trigger name identical to that in Ry by the trigger name in Rz
- $ins(a) = cleanChildren(Rx)$ where $x \in \{0, 1, 2, 3\}$: delete the children list of trigger pointed by register Rx
- $ins(a) = return()$: $PC' = pop(address\_stack)$

Once a trigger is handled and removed from the queue, the trigger is deleted from the system when the code block (started by the trigger) ends. For general HE code program, a garbage collector may be necessary to properly remove all de-referenced triggers and to ensure that there is no reference fault (trigger name is being used but the trigger itself has been deleted). Code generated from an HTL program does not create any such problem; so we avoid the definition of a formal garbage collector. All of the above instructions except $deleteChildren$ can be executed in constant time. The execution of $deleteChildren$ requires time linear in the size of the original HTL description of the involved children.

The E machine starts with the following configuration: $state$ is default value of each variable, $writeQ = \emptyset$, $switchQ = \emptyset$, $readQ = \emptyset$, $tasks = \emptyset$, $PC = \perp$, $R0 = \perp$, $R1 = \perp$, $R2 = \perp$, $R3 = \perp$, $address\_stack = \emptyset$, and $parent\_stack = \emptyset$.

6. HANDLING HIERARCHY IN HE CODE

There are two major concerns for handling HTL programs in HE code: tracking the current position in the hierarchy (i.e. which program, module or mode is being executed) and maintaining the hierarchical relation between modes. The first is done by subroutine-like calls to initialize and execute programs, modules and modes; refer Section 7 for details. Intuitively, the address stack stores the addresses of programs, modules and modes in a tree like fashion so that E Machine knows which program, module or mode is to be initialized/executed once the current one has been initialized/executed. Maintaining the hierarchical relation is more involved and is done through triggers and HE code instructions. For HTL programs, the compiler generates triggers as follows: all triggers associated with writing communicators are stored in the write queue, all triggers associated with mode switch checks are stored in the switch queue and all triggers associated with reading communicators (and subsequently releasing tasks) are stored in the read queue. The writing of communicators in a module, reading of communicators in a mode and releasing of tasks in a mode are independent of other modes, modules and programs. The above holds if the HTL program is race free (ensured by structural checks) and if all communicators are written before they are read (ensured by handling triggers in the write queue before that of the switch and the read queue). However checking switches (and subsequent actions) in a mode depend on other modes. For code generated from HTL, triggers in the write and the read queue have no parent and children information; in other words they do not carry any hierarchy information. Only triggers in the switch queue have hierarchy information.

In HTL, switches for a parent mode and its children modes are enabled simultaneously due to constraints on timing behavior. The HTL semantics prioritizes the mode switch check (and subsequent action) of the parent mode over those of the children. Consider an instance when modes `modeOne`, `onePI` and `oneSlow` are active (Fig. 6). Mode `modeOne` has no mode switches i.e. it is invoked repeatedly. There are three possible scenarios: (1) none of the modes switches, (2) only `oneSlow` switches to `oneFast` i.e. the new combination is `modeOne`, `onePI` and `oneFast`, and (3) `onePI` switches i.e. the new combination is `modeOne` and `oneP`; the switch of `oneSlow` does not matter in the transition.

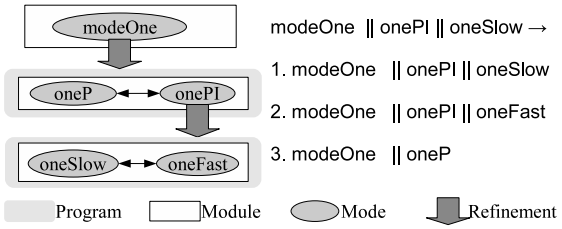

Figure 6: Mode switch for HTL programs

The switching action of HTL is reflected in the HE code as follows. The compiler generates code in such a way that there is exactly one trigger per mode in the switch queue i.e. the implicit tree in the switch queue is the hierarchy of the modes in the program. When a trigger in the switch queue is enabled, the corresponding mode switch is checked; if the mode switch is false then the mode is reinvoked, otherwise all triggers (in the switch queue) related to the modes in the refinement program of the mode are removed and the target mode is invoked.

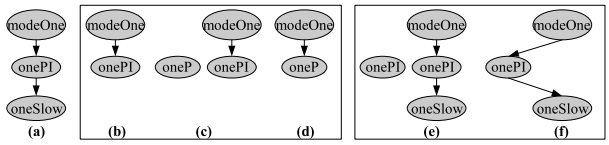

Figure 7: Handling switch checks in HE code

Consider the situation when modes `modeOne`, `onePI` and `oneSlow` are executing and switch condition for `OnePI` is true. Fig. 7.a shows the associated triggers in the switch queue; instead of the queue, the implicit tree structure has been shown. First, the triggers in the switch queue from refinement program of `onePI` are removed (Fig. 7.b). A new trigger for the target mode `oneP` is generated (Fig. 7.c), the parent information is transferred to the new trigger(Fig. 7.d) and the trigger for mode `onePI` is removed. The trigger for mode `oneSlow` is removed without even checking whether the switch condition is true or false. In another scenario, consider the mode switch condition of `onePI` is false i.e. the mode will be reinvoked. First a new trigger is created for mode `onePI` in the switch queue (Fig. 7.e) with no parent and children information. Next, the parent and children information of the old trigger

for onePI is redirected to the new trigger for onePI (Fig. 7.f) and the old trigger for onePI is removed from the switch queue. The E machine will next traverse the queue to check mode switch for oneSlow.

7. HTL COMPILER

The compiler (Fig. 8) for HTL ensures that the program satisfies the constraints on parallel composition of modules, refinement of modes and timing of tasks relative to the target platform [6]. The WCET/ WCTT information for tasks are provided by an external tool. If the checks go through, HE code generator generates code for a distributed implementation. The code generation is done by compiling the whole program for each host. Each host maintains its own copies of all communicators and ports; however tasks are executed on the host only if the corresponding mode (in which the task is invoked) is mapped onto that host. Whenever a task completes execution, the output is broadcast to all hosts and stored in local ports; when a communicator (on a host) is to be written, the value of the local port is copied to the communicator. Release tasks are dispatched for execution by an EDF scheduler; the scheduler is external to the E Machine.

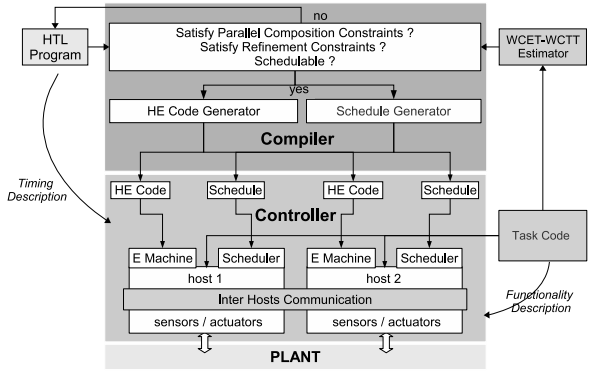

Figure 8: Structure of compiler and runtime system

The compiler generates code for program, module and mode by invoking Alg. 1, Alg. 2 and Alg. 3 respectively. The compiler uses symbolic addresses to refer to different parts of the code. For each program P, $program\_init\_address[\mathtt{P}]$ and $program\_start\_address[\mathtt{P}]$ denotes the address of the HE code block that initializes and executes P respectively. For each module M, $module\_init\_address[\mathtt{M}]$ and $module\_start\_address[\mathtt{M}]$ denotes the address of the HE code block that initializes and executes M respectively. For each mode m, $mode\_start\_address[\mathtt{m}]$ is the address of the HE code block that starts m and $target\_mode\_address[\mathtt{m}]$ is the address of HE code block that will be executed when another mode switches to m. Each mode m is divided in uniform units corresponding to the smallest period between two time events (i.e., write of a communicator or read of a communicator) in m. Given a mode m, the duration of an unit $\gamma[\mathtt{m}]$ is the gcd of all access periods of all communicators accessed (i.e. read or written) in m and the total number of units is $\pi[\mathtt{m}]/\gamma[\mathtt{m}]$, where $\pi[\mathtt{m}]$ is the period of m. For each unit i of every mode m the compiler generates separate code blocks for updating communicators, checking switches (and related actions) and reading communicators (and releasing tasks): the address of the HE code block that writes communicators is $mode\_unit\_write[\mathtt{m}, i]$, the address of the HE code block that checks switch condition is $mode\_unit\_switch[\mathtt{m}, i]$, and the address of the HE code block that reads communicators is $mode\_unit\_read[\mathtt{m}, i]$. HTL semantics constraints that at any instance, communicator writes, mode switch checks, communicator reads and task releases should be done in the above order to maintain consistency of communicator values across all modules. The address of the HE code block that sets up the execution order of communicator writes, switch checks and communicator reads (and task releases) is $mode\_body\_address[\mathtt{m}]$. Instructions may forward reference to any of the above symbolic addresses and therefore need fix up during compilation.

Alg. 1 generates code for a program P on a host h. The code at address $program\_init\_address[\mathtt{P}]$ initializes all communicators declared in P by calling respective initialization drivers ($\mathtt{init}(\cdot)$ denotes the initialization driver for a communicator or a port) and then calls initialization subroutines for each of the modules. Code at address $program\_start\_address[\mathtt{P}]$ calls the start subroutine for each module M in P.

Algorithm 1 GenerateECodeForProgramOnHost(P, h)

set $program\_init\_address[\mathtt{P}]$ to PC and fix up
// initialize communicators
$\forall \mathtt{c} \in \mathtt{communicators}(\mathtt{P})$:$emit(call(\mathtt{init}(\mathtt{c})))$
// initialize all the modules in P
$\forall \mathtt{M} \in \mathtt{modules}(\mathtt{P})$:
  $emit(jumpSubroutine(module\_init\_address[\mathtt{M}]))$
// return from initialization subroutine of P
$emit(return)$
set $program\_start\_address[\mathtt{P}]$ to PC and fix up
// start all the modules in P
$\forall \mathtt{M} \in \mathtt{modules}(\mathtt{P})$:
  $emit(jumpSubroutine(module\_start\_address[\mathtt{M}]))$
// return from start subroutine of P
$emit(return)$

Alg. 2 generates code for a module M on host h. Code at address $module\_init\_address[\mathtt{M}]$ initializes all task ports (denoted by $\mathtt{taskPorts}(\mathtt{M})$) of the tasks in M by calling respective initialization drivers. All tasks maintain two sets of local ports, called *task input ports* and *task output ports*, which are not accessible by other tasks. At release, the tasks reads communicators and ports to task input ports and execute on the value of the task input ports. At completion, the task output ports are updated. The communicators and ports are written from the task output ports when the writing is due. Code at $module\_start\_address[\mathtt{M}]$ calls the execution code for the start mode, $\mathtt{start}[\mathtt{M}]$, for the module M.

Algorithm 2 GenerateECodeForModuleOnHost(M, h)

set $module\_init\_address[\mathtt{M}]$ to PC and fix up
// initialize task ports
$\forall \mathtt{p} \in \mathtt{taskPorts}(\mathtt{M})$:$emit(call(\mathtt{init}(\mathtt{p})))$
// return from initialization subroutine of M
$emit(return)$
set $module\_start\_address[\mathtt{M}]$ to PC and fix up
//start the start mode of M
$emit(jumpSubroutine(mode\_start\_address[\mathtt{start}[\mathtt{M}]]))$
// return from start subroutine of M
$emit(return)$

We will use the following auxiliary operators for Alg. 3. The set $\mathtt{readDrivers}(\mathtt{m}, i)$ contains the drivers that load the tasks in mode m with values of the communicators that are read by these tasks at unit i. The set $\mathtt{writeDrivers}(\mathtt{m}, i)$ contains the drivers that load the communicators with the output of the tasks in mode m that write to these communicators at unit i. The set $\mathtt{portDrivers}(\mathtt{t})$ contains the drivers that load task input ports of task t with the values of the ports on which t depends. The set $\mathtt{complete}(\mathtt{t})$ contains the events that signal the completion of the tasks on which task t depends, and that signal the read time of the task t. The set $\mathtt{releasedTasks}(\mathtt{m}, i)$ contains the tasks in mode m, with no precedences, that are released at unit i. The set $\mathtt{precedenceTasks}(\mathtt{m})$ contains the tasks in mode m that depend on other tasks.

Alg. 3 first emits code (at address $mode\_start\_address[\mathtt{m}]$) for checking all the mode switches (lines **1 - 3**) in a mode `m`, so that they are tested first time `m` is invoked. Next, code is generated (at address $target\_mode\_address[\mathtt{m}]$) to handle the case when no switch is enabled: a call to code at $mode\_body\_address[\mathtt{m}]$, followed by a call to the refinement program (if any). This sets the execution of a mode before the execution of the refinement program. Code at $mode\_body\_address[\mathtt{m}]$ (lines **40 - 49**) sequences the execution order of communicator writes, switch checks and communication reads (and subsequent task release), for unit zero of mode `m`. This is done by emitting a future instruction (line **41**) for $mode\_unit\_write[\mathtt{m}, 0]$ (trigger added to $writeQ$), a future instruction (line **42**) for $mode\_unit\_switch[\mathtt{m}, 0]$ (trigger added to $switchQ$) and a future instruction (line **49**) for $mode\_unit\_read[\mathtt{m}, 0]$ (trigger added to $readQ$). Whenever a trigger is created and added to a queue, the relevant trigger pointer is stored in register $R1$. Once a trigger is added in the switch queue, the hierarchy information has to be updated (lines **43 - 48**). There are two scenarios: one, the code is invoked by handling an enabled trigger in the switch queue i.e. a mode switch has occurred or a mode is being reinvoked (lines **28 - 39**) and two, the code is invoked when a mode is executed for the first time (line **5**). In both the scenarios register $R0$ records the relevant hierarchy information. In the first scenario it stores the name of the last trigger in the switch queue that was handled (by semantics, if any trigger is handled the name is stored in $R0$). In the second scenario, it stores the name of the last trigger in the switch queue that was created. Code in lines **43 - 47** redirects the parent and children of $R0$ to $R1$. A copy of $R1$ needs to be stored in $R2$ (line **48**), as a new trigger for the read queue may remove the information of the last trigger added to the switch queue from $R1$.

Code emission at lines **6 - 17** checks whether a refinement program exists and subsequently updates the hierarchy information if there is one. Before the code generation for refinement program (line **12**), the hierarchy is updated (lines **7 - 11**) as refinement adds one level of hierarchy; once the code generation of the refinement program completes the level is restored (lines **13 - 16**). The hierarchy is updated through register $R0$. The parent of $R0$ is pushed onto the stack (lines **8 - 9**); the parent of the trigger pointed by $R0$ is changed to the trigger name in $R2$ (which contains a pointer to the last trigger added to the switch queue) and children list is reset (code for refinement program has yet to be generated and thus there is no children information). In effect, for the code generation of the refinement program, parent of $R0$ points to the parent trigger of all the triggers to be added in the switch queue for that program. To restore the hierarchy level, the parent of $R0$ is updated by popping the parent stack and is used by modes of parallel modules.

The code at $mode\_unit\_write[\mathtt{m}, i]$ (lines **23 - 27**) calls the driver for each communicator being written at the unit i of mode `m`. The code at $mode\_unit\_switch[\mathtt{m}, i]$ (lines **29 - 39**) checks the mode switches. In HTL, modes can switch only at period boundaries; so the switches are checked only for unit zero (line **28**). If no mode switch occurs (line **33**) the code jumps to $mode\_body\_address[\mathtt{m}]$. If a mode switch occurs, then all children of the last enabled trigger in the switch queue (the name is stored in register $R0$) are removed (lines **34 - 37**). The removal of children is recursive, thus all children of subsequent children are also removed. Once the children are removed, the code jumps (lines **38 - 39**) to the target address of the destination mode $target\_mode\_address[\mathtt{m}']$, where $\mathtt{m}'$ is the destination mode. The code at $mode\_unit\_read[\mathtt{m}, i]$ (lines **52 - 71**) reads all communicators (by calling drivers that copy from communicators into task input ports) that are to be read at unit i, and releases all tasks (with no precedences), that should be released at unit i. For unit zero (line **58**), code is generated to release precedence tasks (lines **59 - 69**). For each task `t` with precedences, a trigger is added to $readQ$: the trigger is activated at the completion of preceding tasks of `t`; and the subsequent code writes input ports of `t` and releases `t`. Lines (**72 - 76**) emit code to jump from one unit to the next; the codes add triggers to the write and the read queue only as switches are not possible in the middle of HTL modes.

The code generation algorithm for a program/ module/ mode accesses other programs, modules or modes through symbolic addresses and does not influence the code generation of other programs, modules and modes. Thus parts of HTL programs can be compiled in any order separately.

8. COMPARISON AND RELATED WORK

E code vs. HE Code. The E code and the HE code are compared in two ways: runtime overhead and code size generated by the HTL compiler. We measured the time spent in interpreting E code and HE code for the 3TS case study HTL program; the delay introduced by code interpretation is below 1% for both E code and HE code.

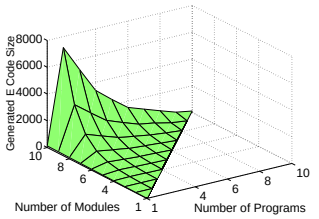

Figure 9: Number of E code instructions

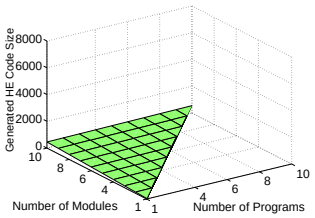

Figure 10: Number of HE code instructions

The code size is compared for HTL descriptions with m programs (i.e. one top-level program and $m-1$ refinement programs) and n modules ($m \leq n$ i.e. we ruled out empty programs) where each module has two modes switching between themselves. For each such scenario there are a number of possible HTL descriptions. Consider the case when $m = 2$ and $n = 3$; with the above restrictions there is one top-level program and one refinement program (refining one of the modes of the top-level program). There are two possible HTL descriptions: top-level program with two modules (i.e. refinement program with one module) and refinement program with two modules (i.e. top-level program with one module). For each m and n, the worst-case code size for E code and HE code are compared. The number of HE code instructions depends upon the number of programs and modules and is thus fixed for any description for given m and n. The number of E code instructions depends upon the flattening and thus widely varies across the different descriptions for given m and n. Fig. 9 and Fig. 10 compares the code size for the E code and the HE code respectively for $1 \leq m \leq 10$ and $1 \leq n \leq 10$. The worst case E program (7177 E code instructions) is an order of magnitude larger than that of the HE program (555 HE code instructions).

Code Generation for Timed Languages. Timed languages have been pioneered by Giotto [8]. In Section 1 we discussed the difference in code generation for a flat structure like Giotto and our proposed approach for HTL. Other LET based languages include TDL [5] and Timed-Multitasking (TM) [13]. Like Giotto, TDL is restricted to one level of periodic tasks and the code generation technique does not address hierarchical programs. TM, an actor based language, uses an event-triggered approach by expressing LET through deadlines. TM can express hierarchy by having actors defined in other actors; however the code generation does not explicitly addresses the hierarchical structure.

Code Generation for Synchronous Languages. Synchronous languages (e.g. Esterel [3] and Lustre [7]) theoretically subsume

Algorithm 3: GenerateECodeForModeOnHost(m, h)

0 set $mode\_start\_address$[m] to PC and fix up
1 // check mode switches
2 $\forall(cnd, \mathtt{m}') \in \mathtt{switches(m)}$:
3 $emit(jumpIf(cnd, target\_mode\_address[\mathtt{m}']))$
4 set $target\_mode\_address$[m] to PC and fix up
5 $emit(jumpSubroutine(mode\_body\_address[\mathtt{m}]))$
6 **if** (program P refines m)
7 //increment the level
8 $emit(getParent(R0, R3))$
9 $emit(pushRegister(R3))$
10 $emit(setParent(R0, R2))$
11 $emit(cleanChildren(R0))$
12 $emit(jumpSubroutine(program\_start\_address[\mathtt{program[m]}]))$
13 //decrement the level
14 $emit(popRegister(R3))$
15 $emit(setParent(R0, R3))$
16 $emit(cleanChildren(R0))$
17 **end if**
18 // return from start subroutine of m
19 // OR wait for other triggers to become enabled
20 $emit(return)$
21 $i := 0$
22 **while** $i < \pi[\mathtt{m}]/\gamma[\mathtt{m}]$ **do**
23 set $mode\_unit\_write$[m, i] to PC and fix up
24 // write communicators from task output ports
25 $\forall d \in \mathtt{writeDrivers(m}, i)$:$emit(call(d))$
26 // wait for other triggers to become enabled
27 $emit(return)$
28 **if** $(i = 0)$
29 set $mode\_unit\_switch[\mathtt{m}, 0]$ to PC and fix up
30 // check mode switches
31 $\forall(cnd, \mathtt{m}') \in \mathtt{switches(m)}$:
32 $emit(jumpIf(cnd, PC + 2))$
33 $emit(jumpAbsolute(PC + 4))$
34 // cancel all triggers related to the refining
35 // program of m, and its subprograms
36 $emit(deleteChildren(R0))$
37 $emit(cleanChildren(R0))$
38 // switch to mode m'
39 $emit(jumpAbsolute(target\_mode\_address[\mathtt{m}']))$
40 set $mode\_body\_address$[m] to PC and fix up
41 $emit(writeFuture(\pi[\mathtt{m}], mode\_unit\_write[\mathtt{m}, 0]))$
42 $emit(switchFuture(\pi[\mathtt{m}], mode\_unit\_switch[\mathtt{m}, 0]))$
43 $emit(getParent(R0, R3))$
44 $emit(replaceChild(R3, R0, R1))$
45 $emit(setParentOfChildren(R0, R1))$
46 $emit(setParent(R1, R3))$
47 $emit(copyChildren(R1, R0))$
48 $emit(copyRegister(R1, R2))$
49 $emit(readFuture(0, mode\_unit\_read[\mathtt{m}, 0]))$
50 $emit(return)$
51 **end if**
52 set $mode\_unit\_read$[m, i] to PC and fix up
53 **if** (mode m is contained in a module on host h)
54 // read communicators into task input ports
55 $\forall d \in \mathtt{readDrivers(m}, i)$:$emit(call(d))$
56 // release tasks with no precedences
57 $\forall \mathtt{t} \in \mathtt{releasedTasks(m}, i)$:$emit(release(\mathtt{t}))$
58 **if** $(i = 0)$
59 // release tasks with precedences
60 $\forall \mathtt{t} \in \mathtt{precedenceTasks(m)}$:
61 // wait for tasks on which t depends to complete
62 $emit(readFuture(\mathtt{complete(t)}, PC + 2))$
63 $emit(jumpAbsolute(PC + 3 + |\mathtt{portDrivers(t)}|))$
64 // read ports of tasks on which t depends,
65 // then release t
66 $\forall d \in \mathtt{portDrivers(t)}$:$emit(call(d))$
67 $emit(release(\mathtt{t}))$
68 // wait for other triggers to become enabled
69 $emit(return)$
70 **end if**
71 **end if**
72 **if**$(i < \pi[\mathtt{m}]/\gamma[\mathtt{m}] - 1)$
73 // jump to the next unit of mode m
74 $emit(writeFuture(\gamma[\mathtt{m}], mode\_unit\_write[\mathtt{m}, i + 1]))$
75 $emit(readFuture(\gamma[\mathtt{m}], mode\_unit\_read[\mathtt{m}, i + 1]))$
76 **end if**
77 // wait for other triggers to become enabled
78 // OR return from body subroutine of m
79 $emit(return)$
80 $i := i + 1$
81 **end while**

HTL; however HTL offers an explicit hierarchical program structure that supports refinement of tasks into task groups with precedences. Simulink-to-SCADE/Lustre-to-TTA [4] is a tool chain that accepts discrete time models written in Simulink, translates to Lustre models, verifies system properties (e.g. schedulability) and generates code for a target time-triggered architecture. Taxys [2], a tool chain that combines Esterel and model checker Kronos, generates an application specific scheduler that ensures timing commitment of tasks. Our code generation technique differs from the above two approaches in accounting for the hierarchical structure (e.g. Simulink models are hierarchical but Lustre is not which necessitates the code generator to flatten the structure) and in generating code for a virtual machine (both the above tool chains generate code for specific target) which makes the generated code portable across implementations.

9. CONCLUSION

Previously we presented an implementation of HTL, a hierarchical coordination language for distributed hard real-time applications on E Machine, a virtual machine. However, HTL programs must be flattened because of the limitations of the E Machine. This paper presents a modified E Machine to enable separate and linear-space-bounded compilation of HTL. We introduced the semantics of the modified E Machine and the changes in compile-time and runtime infrastructure. In the future, we plan to use the modified E Machine for high-performance, 50-100Hz helicopter flight control [1].

10. REFERENCES

[1] J. Auerbach, D.F. Bacon, D.T. Iercan, C.M. Kirsch, V.T.'Rajan, H. Röck, and R. Trummer. Java takes flight: Time-portable real-time programming with exotasks. In *LCTES*, 2007. ACM.

[2] V. Bertin, E. Closse, M. Poize, J. Pulou, J. Sifakis, P. Venier, D. Weil, and S. Yovine. Taxys = Esterel + Kronos. A tool for verifying real-time properties of embedded systems. In *Conference on Decision and Control*, 2001. IEEE.

[3] F. Boussinot and R. de Simone. The ESTEREL language. *Proceedings of the IEEE*, 79(9). 1991.

[4] P. Caspi, A. Curic, A. Maignan, C. Sofronis, S. Tripakis, and P. Niebert. From Simulink to SCADE/Lustre to TTA: a layered approach for distributed embedded applications. In *LCTES*, 2003. ACM.

[5] E. Farcas, C. Farcas, W. Pree, and J. Templ. Transparent distribution of real-time components based on logical execution time. In *LCTES*, 2005. ACM.

[6] A. Ghosal, D. Iercan, T. A. Henzinger, C. M. Kirsch, and A. Sangiovanni-Vincentelli. A hierarchical coordination language for interacting real-time tasks. In *EMSOFT* 2006. ACM.

[7] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9). 1991.

[8] T. A. Henzinger, B. Horowitz, and C. M. Kirsch. GIOTTO: A time-triggered language for embedded programming. *Proceedings of the IEEE*, 91. 2003.

[9] T. A. Henzinger and C. M. Kirsch. The Embedded Machine: Predictable, portable real-time code. In *PLDI*, 2002. ACM.

[10] D. T. Iercan. Tsl compiler. Technical report, 'Politehnica' University of Timisoara, 2005.

[11] C.M. Kirsch, M.A.A. Sanvido, and T.A. Henzinger. A programmable microkernel for real-time systems. In *USENIX VEE*, 2005. ACM.

[12] C.M. Kirsch, M.A.A. Sanvido, T.A. Henzinger, and W. Pree. A Giotto-based helicopter control system. In *EMSOFT*, 2002. LNCS 2491. Srpinger.

[13] J. Liu and E. A. Lee. Timed multitasking for real-time embedded software. *IEEE Control Systems Magazine*, 23(1). 2003.