A Hierarchical Coordination Language for Interacting Real-Time Tasks[*]

Arkadeb Ghosal
UC Berkeley
arkadeb@eecs.berkeley.edu

Thomas A. Henzinger
EPFL
tah@epfl.ch

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

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

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

ABSTRACT

We designed and implemented a new programming language called Hierarchical Timing Language (HTL) for hard real-time systems. Critical timing constraints are specified within the language, and ensured by the compiler. Programs in HTL are extensible in two dimensions without changing their timing behavior: new program modules can be added, and individual program tasks can be refined. The mechanism supporting time invariance under parallel composition is that different program modules communicate at specified instances of time. Time invariance under refinement is achieved by conservative scheduling of the top level. HTL is a coordination language, in that individual tasks can be implemented in "foreign" languages. As a case study, we present a distributed HTL implementation of an automotive steer-by-wire controller.

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

General Terms: Languages.

Keywords: Real Time; Automotive Systems.

1. INTRODUCTION

Much current real-time programming proceeds by trial and error: if during a program test some task misses its deadline, then the task priorities are reassigned, and new tests are performed. In rare cases can the timing of a program be *proved* correct using scheduling theory and/or formal verification. Scheduling analysis becomes difficult when the program structure is irregular, with branches, exceptions, and dynamic task creation. Formal techniques are difficult due to state space explosion.

Part of the problem is that design practice deals with timing in an indirect way, often through low-level constructs such as priorities. In this paper, we present a new high-level coordination language for interacting hard real-time tasks called Hierarchical Timing Language (HTL). Like Giotto [1], our language refers directly to real-time instances, but it is more general than Giotto, in that it offers hierarchical layers of abstraction. Besides adding program structure, a main benefit of the abstraction hierarchy is that feasible schedules for lower layers can be efficiently constructed from feasible schedules for higher layers.

HTL permits composition and refinement of programs without changing their real-time behavior. Parallel program modules communicate with each other and with the environment through so-called *communicators*, of which sensors and actuators are special cases. A communicator defines a sequence of real-time instances of a static variable. Task *reads* and *writes* specify communicator instances. As the read and written time instances of communicators are fixed by a program, they remain unchanged when the context of the program is modified. In other words, the communicator instances specify a *logical execution time* (LET) [1] slot for each task; the actual physical execution of the task must fall within this slot. As long as physical task execution falls within the LET interval, the functional and timing semantics of a program is deterministic, independent of the actual task schedule. In particular, individual program modules can be reused in different contexts without changing their timing behavior, or upgraded without affecting the timing of rest of the system.

In HTL, tasks can be refined iteratively by groups of tasks with precedence relations. Each task refinement is constrained in such a way that if the task is schedulable, then the more detailed replacement group of tasks is schedulable as well. As a consequence, schedulability needs to be checked only for the top level of an HTL program. This property avoids a combinatorial explosion, and permits scheduling to be performed by the HTL compiler. The compiler rejects a program if it cannot guarantee that its timing specification is satisfied on a given platform (which is specified through worst-case execution times for all tasks).

In addition to module composition (concurrency) and task

[*]This work was supported in part by the GSRC grant 2003-DT-660, by the NSF grant CCR-0208875, and by 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.

refinement (hierarchy), HTL supports the collection of tasks into *modes*, which can be composed sequentially. We provide an operational semantics and a compiler for HTL. The compiler performs a schedulability analysis and translates HTL programs into code for the Embedded Machine (E code) [2]. The compiler can generate E code for a distributed HTL implementation; we assume that the Embedded Machine has been implemented on each host over which the HTL implementation is distributed. The semantics of an HTL program remains independent of the number of hosts, but the analysis and code generation takes into account the distribution.

To demonstrate the expressive power and the feature of HTL, we present an automotive steer-by-wire controller as a case study. A steer-by-wire system removes the mechanical linkage between steering wheel and car with a set of sensors, actuators, and a controller distributed over several processors. Typically the sensors and actuators are spread over four processors for each of the wheels; the controller implementation (along with different functionalities like fault detection, supervisory control, and power coordination) is distributed on more than one processor. The example shows the use of communicators and task refinement, while illustrating the need for horizontal and vertical extensions of the software. Horizontally, parallel modules can be appended to the implementation without changing the timing behavior of the implementation. Vertically, the refinement concept can be used to provide (temporal) space for future extensions.

2. LANGUAGE OVERVIEW

Tasks. The computational units of HTL are LET *tasks*. The LET model (Fig. 1) decouples the times when a task reads input and writes output from the time when the task executes. A LET task is sequential code with its own memory space (which cannot be accessed by other tasks) and without internal synchronization points. *Release* and *termination* events, which are triggered by clock ticks or sensor interrupts, determine the LET of the task; the task is *active* between these two events. The input of the task is written into its memory when the release event occurs, not when the task actually *starts* executing. Similarly, the output is made available to other tasks or actuators at the termination event, even if the task *completes* physical execution earlier. Between the start and completion of execution the task may be preempted and resumed any number of times. A LET task is *time-safe* on some given hardware if the task completes execution on that hardware before the termination event occurs. Time-safe LET tasks are time and value deterministic, portable and composable [2].

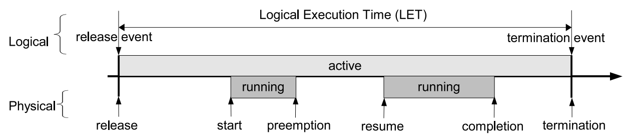

Figure 1: A LET (logical execution time) task

Communicators. The communication model for HTL is centered around *communicators*. A communicator is a typed variable that can be accessed (read from or written to) only at specific time instances. These time instances are periodic and specified through a communicator period. Sensors and actuators are communicators, but communicators can also be used to exchange data between tasks. A task reads from certain instances of some communicators, computes a function, and updates certain instances of the same or other communicators. Fig. 2 shows the interaction between four communicators ($\mathtt{c}_1$, $\mathtt{c}_2$, $\mathtt{c}_3$, and $\mathtt{c}_4$, with periods 2, 3, 4, and 3, respectively) and two tasks ($\mathtt{t}_1$ and $\mathtt{t}_2$). Task $\mathtt{t}_1$ reads the second instances of $\mathtt{c}_1$ and $\mathtt{c}_4$ and updates the fourth instance of $\mathtt{c}_2$. Task $\mathtt{t}_2$ reads the second instance of $\mathtt{c}_3$ and updates the sixth instance of $\mathtt{c}_1$ and the fifth instance of $\mathtt{c}_2$. The *read* and *write* instances specify the LET for the tasks: the latest *read* instance determines the release time, and the earliest *write* instance determines the termination time. Thus, the LET of $\mathtt{t}_1$ spans from time 3 to time 9, and the LET of $\mathtt{t}_2$ from 4 to 10.

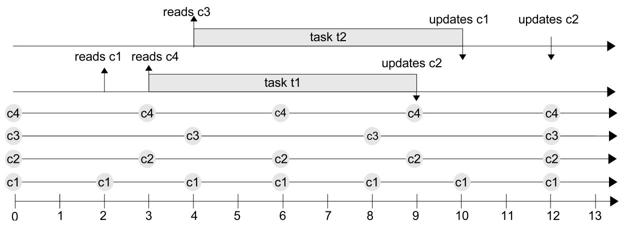

Figure 2: Communicators and tasks

Communicators are the key to compose HTL programs, because they exhibit deterministic behavior: given sufficient CPU speed for time-safety, the real-time behavior of a program is determined by the input (i.e., the values of all sensor instances), independent of the CPU speed and utilization. The determinism is ensured by prohibiting update races on communicators (different tasks cannot write to the same instance of a communicator), and by ensuring that every communicator instance is updated before it is read.

Modes. HTL generalizes the LET model from tasks to task groups. While tasks with different frequencies can communicate only through communicators, HTL allows direct communication between tasks with identical frequencies. A set of interacting tasks with the same frequency form a *mode* with a specified mode period. The communication flow between tasks within a mode determines an acyclic *precedence* relation on the tasks in the mode.

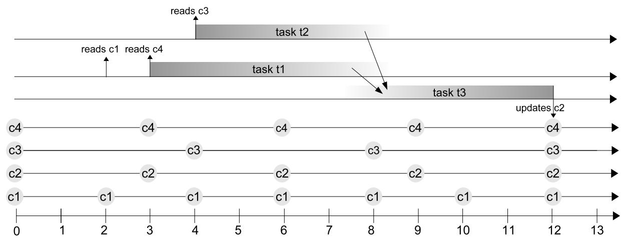

Figure 3: A mode with task precedences

Fig. 3 shows three tasks $\mathtt{t}_1$, $\mathtt{t}_2$, and $\mathtt{t}_3$ within a mode. Task $\mathtt{t}_1$ reads the second instances of $\mathtt{c}_1$ and $\mathtt{c}_4$, and task $\mathtt{t}_2$ reads the second instance of $\mathtt{c}_3$. No termination events are specified for $\mathtt{t}_1$ and $\mathtt{t}_2$; instead, task $\mathtt{t}_3$ reads the outputs of $\mathtt{t}_1$ and $\mathtt{t}_2$, and updates the fifth instance of $\mathtt{c}_2$. The communication between $\mathtt{t}_1$ (and $\mathtt{t}_2$) and $\mathtt{t}_3$ occurs through *ports*. A port is a typed variable, but unlike a communicator, it is not bound to time instances (i.e., as soon as $\mathtt{t}_1$ completes execution, $\mathtt{t}_3$ can read the output port of $\mathtt{t}_1$ and start execution). Thus, within the mode, the two tasks $\mathtt{t}_1$ and $\mathtt{t}_2$ precede the third task $\mathtt{t}_3$.

The tasks within a mode interact through ports and communicators; tasks from different modes interact only through communicators. Fig. 4 shows two modes $\mathtt{m}_1$ and $\mathtt{m}_2$ with periods 6 and 12, respectively.

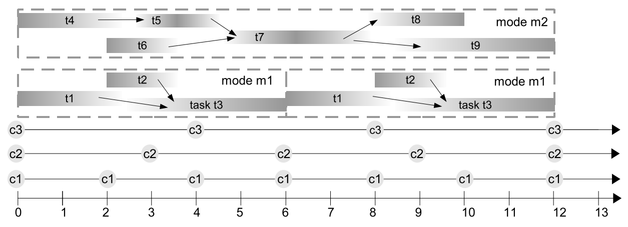

Figure 4: Modes

Modules. In real-time applications, a group of tasks may have to be replaced by an alternate group depending on some specific condition (e.g., a certain sensor reading). HTL accommodates this by allowing *mode switches* at the end of mode periods, which are triggered by conditions on communicator and port values. A network of modes and mode switches is called a *module* (Fig. 5). An HTL *program* is a set of modules and a set of communicators. While the modes within a module are composed sequentially (i.e., at any time, the tasks of at most one mode of a module are active), the modes from different modules are composed in parallel and may interact through communicators. Communicators thus serve several purposes: to exchange data between tasks from different parallel modules, and to exchange data from one task within a module to a later task within the same module (but possibly in a different mode). One mode in each module is specified as the *start mode*.

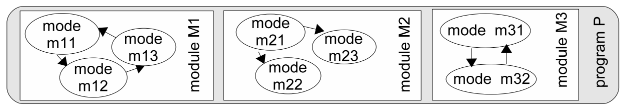

Figure 5: Modules

Refinement. Specifying all behaviors through mode switching is cumbersome. For a structured and concise specification, we introduce the concept of *mode refinement*: each mode in a program can be specified by an HTL program. This does not add expressiveness to the model; in fact, an HTL program with multiple levels of refinement can be translated into an equivalent "flat" program without refinement. Refinement is useful not only for compact representation, but also for conservatively simplifying the program analysis: we check schedulability only for the top-level program, and constrain refinements so that they preserve schedulability. Fig. 6 shows a mode $\mathtt{m}$, and a mode $\mathtt{m}'$ from a program which refines $\mathtt{m}$. HTL imposes certain restrictions on $\mathtt{m}'$ to preserve schedulability. First, the period of mode $\mathtt{m}'$ is identical to that of $\mathtt{m}$. This ensures that when $\mathtt{m}$ switches (which is only possible at the end of its period), then all tasks in the modes refining $\mathtt{m}$ have terminated execution. Second, every task in $\mathtt{m}'$ refines an unique task in $\mathtt{m}$ (shown by dashed arrows); e.g., $\mathtt{t}'_5$ (child) refines $\mathtt{t}_5$ (parent). HTL considers $\mathtt{t}_5$ as a placeholder (an *abstract* task) for $\mathtt{t}'_5$ (the *concrete* task): the abstract task $\mathtt{t}_5$ does not execute at run-time but ensures that $\mathtt{t}'_5$ is accounted for during the schedulability analysis of the top-level program. Therefore, (1) the latest (resp. earliest) communicator read (resp. write) of $\mathtt{t}'_5$ must be equal to or earlier (resp. later) than that of $\mathtt{t}_5$; (2) every task that precedes $\mathtt{t}'_5$ must refine a task that precedes $\mathtt{t}_5$; and (3) the WCET of $\mathtt{t}'_5$ must be less than or equal to the WCET of $\mathtt{t}_5$. These three constraints ensure that if $\mathtt{t}_5$ can be scheduled in the top-level program, then $\mathtt{t}'_5$ can be scheduled in the complete program.

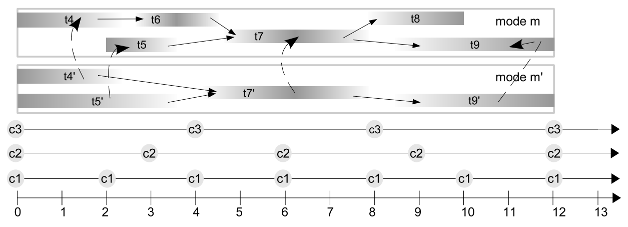

Figure 6: Refinement

Refinement can represent both *choice* and *change* of behavior. Choice is expressed when an abstract task $\mathtt{t}$ in a mode $\mathtt{m}$ is the parent of different (concrete or abstract) tasks in several modes of a program that refines $\mathtt{m}$. Change is expressed by having a (concrete or abstract) task that refines $\mathtt{t}$ reading from and writing to different communicators than $\mathtt{t}$ does (within the constraints listed above). Fig. 7 shows a diagrammatic view of an HTL program with refinement. The modes contain tasks with precedences.

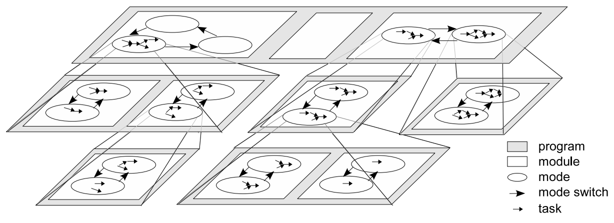

Figure 7: An HTL program

Distribution. Many embedded applications are distributed: the tasks are distributed on several hosts and interact with each other through communication channels. In HTL, distribution is specified through a mapping of top-level modules to hosts (all refinements are bound to the same hosts). The distribution is implemented by replicating shared communicators on all hosts, and then have the tasks that write to shared communicators broadcast the outputs. The semantics (i.e., the real-time behavior) of an HTL program is independent of the number of hosts, but code generation and schedulability analysis must take the distribution into account. For this purpose, the LET model (Fig. 8) is extended to include both WCETs as well as worst-case output transmission times (WCTTs).

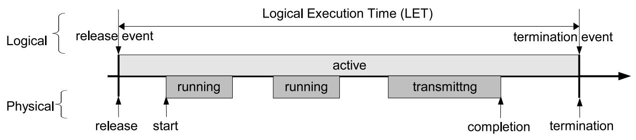

Figure 8: LET model with transmission times

3. STEER-BY-WIRE

A *steer-by-wire* (SBW) control system replaces the mechanical linkage between steering wheel and car wheels by a set of electric motors that control the wheel angle, and a controller that computes the required wheel motor actuation. To maintain a realistic road condition feel for the driver, a force feedback actuator is placed on the steering wheel. The specific architecture that has been used here is a simplified SBW model used by General Motors for their prototype hydrogen fuel-cell car FX-3. This example is an imitation of the concerns and requirements of automotive design, and does not represent a real set of control algorithms for an actual product or prototype.

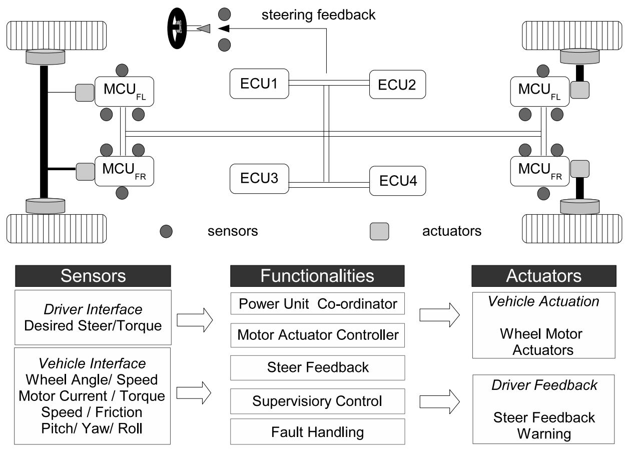

Figure 9: Steer-by-Wire: schematic view

An SBW architecture (Fig. 9) consists of eight hosts (CPUs): four motor control units (MCUs) and four electronic control units (ECUs). The MCUs are placed near the wheels; they detect sensor values related to wheels (wheel angle, motor current, speed, friction, power, pitch, and yaw) and send signals to motor actuators. The ECUs implement functionalities such as computation of wheel motor actuation, steer feedback, supervisory control, fault handling, and power coordination. Supervisory control coordinates between steering, braking, and suspension. The supervisor typically runs in triple-redundant mode (three copies are executed in three different hosts). The fault handling system detects, isolates, and mitigates faults, and warns the driver in case of a fault. The power coordinator handles the coordination of motor current computed by the controller with rest of the power grid. All hosts are connected through a communication link that allows broadcast from any host.

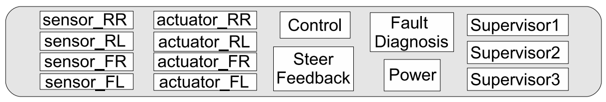

Figure 10: SBW modules

Steer-by-Wire in HTL. The HTL implementation of the SBW functionality consists of fifteen modules (Fig. 10[1]): sensor and actuator units for each of the four wheels, and computational units for control, steer feedback, fault diagnosis, power, and supervision (three copies). Fig. 11 shows the modal structure for each of the modules, and the periods of the modes (in milliseconds). The modes capture changes in the computation policy under different environment condition; e.g., wheel actuation needs to be done faster above a critical speed. A specific scenario(Fig. 12) of inter-mode communication is discussed next; refer to http://htl.cs.uni-salzburg.at for a full specification.

[1]In the figures modules and modes are denoted by rectangles and ellipses, respectively.

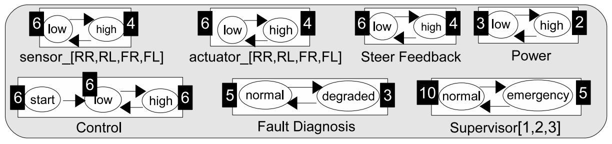

Figure 11: SBW modes

The angle of the rear-left wheel (while the car is moving at a high speed) is measured by three sensors. The sensor values are read by a task that mediates on the value to be considered and then passes it on to the control task. This necessitates communication between mode `high` (module `sensor_RL`), mode `high` (module `steer feedback`), and mode `high` (module `control`). The modes and communicators have periods of 4,000 and 5,00 microseconds, respectively. Communicators instances are referred relative to the mode period (the 0-th instance corresponds to the start of the mode). Tasks `tA3`, `tA4`, and `tA5` (mode `high` of module `sensor_RL`) read three sensors `sA3`, `sA4`, and `sA5` respectively at the start of the period and update the first instances of the three communicators `cA3`, `cA4` and `cA5`. The task `MEDAngleRL` (mode `high` of module `steer feedback`) reads the first instance of the above communicators and writes to the second instance of `cAngleRL`. The task `cntrlFUN` reads the second instance of `cAngleRL` (among other communicators) and computes the value of the wheel actuation signal. The task `RackPinAct` reads the output of `cntrlFUN` and computes the power requirement for the motors.

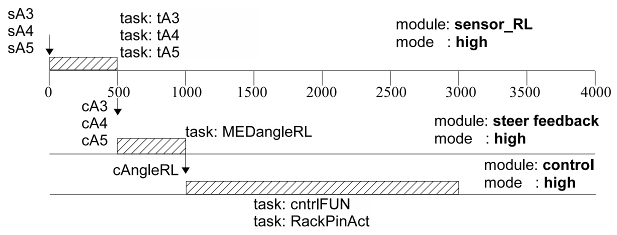

Figure 12: Communication in SBW implementation

Fig. 13 shows the refinement of some of the modes discussed above. Refinement allows a concise representation of changes in computation policies. For example, mode `low` (module `control`) is refined by a program that has one module with two modes, `idle` and `motion`. Without hierarchical refinement, module `control` would have 6 modes and 17 mode switches; this is not only inefficient but also error prone. The modules of the root program are distributed over eight hosts: the sensor and actuator modules for each wheel share one host; the modules `control`, `steer feedback`, and `fault diagnosis` are distributed over three hosts along with one supervisor module on each host; the module `power` is assigned its own host.

Implementation. We implemented the SBW controller on eight AMD Duron 1.4GHz machines with 256MB RAM connected by a 100Mbps Ethernet network. The case study is written in 873 lines of HTL code and compiled to around 1,800 E machine instructions per host. The E machine [2] is written in C and executes the generated code with an overhead between 60 and 300 microseconds per time instant for which it is invoked. The tasks are written in C. Our implementation simulates the case study in real time but at a frequency of 2Hz, which is 1,000 times slower than the actual system.

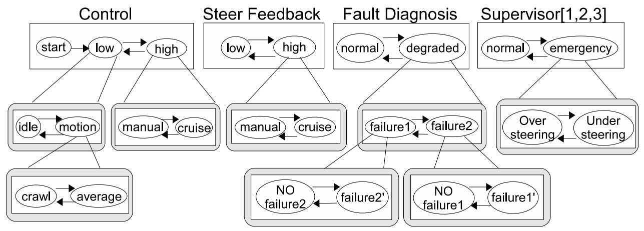

Figure 13: SBW program with all refinements

4. ABSTRACT SYNTAX

We provide the main components of an HTL program in an abstract way. A concrete syntax [3] can be defined for this abstract syntax. An *HTL program* $\mathtt{P}$ consists of the following:

1 A set of *communicator declarations* $\mathtt{commdecl}$. A communicator declaration $(\mathtt{c}, \mathtt{type}, \mathtt{init}, \pi_\mathtt{c})$ consists of a communicator name $\mathtt{c}$, a data type $\mathtt{type}$, an initial value $\mathtt{init}$, and a communicator period $\pi_\mathtt{c} \in \mathbb{N}_{>0}$. The communicator names are unique. The set of declared communicator names for a program $\mathtt{P}$ is $\mathtt{comms}(\mathtt{P})$. Given a communicator $\mathtt{c} \in \mathtt{comms}(\mathtt{P})$, we write $\mathtt{type}[\mathtt{c}]$ for the range of values that $\mathtt{c}$ can evaluate to, and $\mathtt{init}[\mathtt{c}]$ for the initial value of $\mathtt{c}$.

2 A set of *module declarations* $\mathtt{moduledecl}$. A module declaration $(\mathtt{M}, \mathtt{portdecl}, \mathtt{taskdecl}, \mathtt{modedecl}, \mathtt{start})$ consists of a module name $\mathtt{M}$, a set of port declarations $\mathtt{portdecl}$, a set of task declarations $\mathtt{taskdecl}$, a set of mode declarations $\mathtt{modedecl}$, and a start mode $\mathtt{start}$. The module names are unique. The set of declared module names for a program $\mathtt{P}$ is $\mathtt{modules}(\mathtt{P})$. Given a module $\mathtt{M} \in \mathtt{modules}(\mathtt{P})$, we write $\mathtt{start}[\mathtt{M}]$ for the start mode.

2.1 A *port declaration* $(\mathtt{p}, \mathtt{type}, \mathtt{init})$ consists of a port name $\mathtt{p}$, a data type $\mathtt{type}$, and an initial value $\mathtt{init}$. The port names are unique for each module. The set of declared port names for a module $\mathtt{M}$ is $\mathtt{ports}(\mathtt{M})$. Given a port $\mathtt{p} \in \mathtt{ports}(\mathtt{M})$, we write $\mathtt{type}[\mathtt{p}]$ for the range of values that $\mathtt{p}$ can evaluate to, and $\mathtt{init}[\mathtt{p}]$ for the initial value of $\mathtt{p}$.

2.2 A *task declaration* $(\mathtt{t}, \mathtt{filist}, \mathtt{folist}, \mathtt{fn})$ consists of a task name $\mathtt{t}$, a list of formal input parameters $\mathtt{filist}$, a list of formal output parameters $\mathtt{folist}$, and an optional task function $\mathtt{fn}$. An element of the lists of formal input and output parameters is a data type $\mathtt{type}$. If $\upsilon[\mathtt{type}]$ denotes the range of values for $\mathtt{type}$, the length of $\mathtt{filist}$ is m, and the length of $\mathtt{folist}$ is n, then $\mathtt{fn}$ is a function from $\Pi_{1 \leq i \leq m} \upsilon[\mathtt{filist}(i)]$ to $\Pi_{1 \leq i \leq n} \upsilon[\mathtt{folist}(i)]$. The function $\mathtt{fn}$ is absent for *abstract* tasks; if $\mathtt{fn}$ is present, then $\mathtt{t}$ is *concrete*. The task names are unique for each module. The set of declared task names for a module $\mathtt{M}$ is $\mathtt{taskset}(\mathtt{M})$.

2.3 A *mode declaration* $(\mathtt{m}, \pi_\mathtt{m}, \mathtt{invocs}, \mathtt{switches}, \mathtt{refp})$ consists of a mode name $\mathtt{m}$, a mode period $\pi_\mathtt{m} \in \mathbb{N}_{>0}$, a set of task invocations $\mathtt{invocs}$, a set of mode switches $\mathtt{switches}$, and an optional program name $\mathtt{refp}$. The program $\mathtt{refp}$ is absent for modes that are not refined further; otherwise $\mathtt{refp}$ defines the *refining program.* The mode names are unique for each module. The set of declared mode names for a module $\mathtt{M}$ is $\mathtt{modes}(\mathtt{M})$, and $\mathtt{start}[\mathtt{M}] \in \mathtt{modes}(\mathtt{M})$.

2.3.1 A *task invocation* $(\mathtt{t}, \mathtt{ailist}, \mathtt{aolist}, \mathtt{ptask})$ consists of a task name $\mathtt{t} \in \mathtt{taskset}(\mathtt{M})$, a list of actual input parameters $\mathtt{ailist}$, a list of actual output parameters $\mathtt{aolist}$, and an optional task name $\mathtt{ptask}$. An element of the lists of actual input and output parameters is either a port $\mathtt{p} \in \mathtt{ports}(\mathtt{M})$, or a pair $(\mathtt{c}, \mathtt{i})$ consisting of a communicator name $\mathtt{c} \in \mathtt{comms}(\mathtt{P})$ and a communicator instance number $\mathtt{i} \in \mathbb{N}$, where $0 \leq \mathtt{i} \leq \frac{\pi_\mathtt{m}}{\pi_\mathtt{c}}$. If $\mathtt{P}$ refines a mode of another program $\mathtt{P}'$, then the communicator $\mathtt{c}$ may be declared in $\mathtt{P}'$; see below. The task name $\mathtt{ptask}$ is absent if $\mathtt{M}$ is a top-level module; otherwise $\mathtt{ptask}$ defines the *parent task* and the invocation associated with the parent task defines the *parent task invocation.* The task names of the invocations are unique for each mode.

2.3.2 A *mode switch* $(\mathtt{cnd}, \mathtt{m}')$ consists of a condition $\mathtt{cnd}$ which is expressed as a predicate on the ports in $\mathtt{ports}(\mathtt{M})$ and communicators in $\mathtt{comms}(\mathtt{P})$ (and communicators declared in super-programs of $\mathtt{P}$; see below), and a destination mode name $\mathtt{m}' \in \mathtt{modes}(\mathtt{M})$. The mode switches are deterministic for each mode, i.e., for port and communicator values, at most one of the mode switch conditions is true.

Hierarchical program structure. A module $\mathtt{M}_n$ is a *sub-module* of a module $\mathtt{M}_1$ if $\mathtt{M}_n = \mathtt{M}_1$, or there exists n modules $\mathtt{M}_1, \mathtt{M}_2, \ldots, \mathtt{M}_n$ such that for every pair $\mathtt{M}_j, \mathtt{M}_{j+1}$ there exists a mode declaration $(\mathtt{m}, \cdot, \cdot, \cdot, \mathtt{P}) \in \mathtt{modedecl}(\mathtt{M}_j)$ with $\mathtt{M}_{j+1} \in \mathtt{modules}(\mathtt{P})$ for $1 \leq j < n$. If $\mathtt{M}' \in \mathtt{modules}(\mathtt{P}')$, $\mathtt{M} \in \mathtt{modules}(\mathtt{P})$, and $\mathtt{M}'$ is a sub-module of $\mathtt{M}$, then $\mathtt{P}'$ is a *sub-program* of $\mathtt{P}$, and $\mathtt{P}$ is a *super-program* of $\mathtt{P}'$. A program $\mathtt{P}$ is both a sub-program and a super-program of itself. A *top (leaf) level program* is one with no super (sub) programs other than itself. A *flat program* is one which is both a top-level and a leaf-level program. The *abstract program* $\mathtt{abs}(\mathtt{P})$ for program $\mathtt{P}$ is the top-level program with all refining programs removed from the mode declarations of $\mathtt{P}$.

Given a mode declaration $(\mathtt{m}, \cdot, \cdot, \cdot, \mathtt{P})$, mode $\mathtt{m}$ is *parent* of mode $\mathtt{m}'$ if $\mathtt{m}'$ is a mode in some module of $\mathtt{P}$; $\mathtt{m}'$ is *child* of $\mathtt{m}$. The set of *ancestors* of a mode $\mathtt{m}$, denoted $\mathtt{ancestors}(\mathtt{m})$, is the smallest set of modes that includes the parent of $\mathtt{m}$ and the ancestors of the parent. The *start set* of $\mathtt{m}$, denoted $\mathtt{starts}(\mathtt{m})$, is the smallest set that includes $\mathtt{m}$ and contains the start sets of all start modes of modules in $\mathtt{P}$.

The *accessible communicator set* for module $\mathtt{M}$ in program $\mathtt{P}$ is the set of communicators declared by the super-programs of $\mathtt{P}$. The *write-set* for $\mathtt{M}$ is the set of communicators occurring as actual output parameters of task invocations of modes in $\mathtt{M}$. The *hierarchical write-set* for $\mathtt{M}$ is the set of communicators that belong to the accessible communicator set of $\mathtt{M}$ and to the write-sets of all sub-modules of $\mathtt{M}$.

A communicator not written to by any task invocation is called an *input communicator.* Input communicators are written by the physical environment (sensors), or by other programs. *Input communicator set* $\mathtt{icset}(\mathtt{P})$ for program $\mathtt{P}$ is the set of input communicators for all sub-programs of $\mathtt{P}$.

Task precedences. A task invocation $\mathtt{inv}$ *precedes* another invocation $\mathtt{inv}'$ if $\mathtt{inv}'$ reads any port written by $\mathtt{inv}$; in this case, $\mathtt{inv}'$ *follows* $\mathtt{inv}$. The *preceding invocation set* of a task invocation $\mathtt{inv}$ in mode $\mathtt{m}$, denoted $\mathtt{prec}(\mathtt{inv}, \mathtt{m})$, is the set of task invocations that precede $\mathtt{inv}$. The *read time* (resp. *write time*), $\mathtt{rtime}(\mathtt{inv}, \mathtt{m})$ (resp. $\mathtt{ttime}(\mathtt{inv}, \mathtt{m})$) of a task invocation $\mathtt{inv}$ in mode $\mathtt{m}$ is the largest (resp. smallest) interval from the start of mode $\mathtt{m}$ at which a communicator instance is read (resp. written) by $\mathtt{inv}$ or any of its preceding (resp. following) invocations.

Each task invocation has an input (resp. output) port associated with each actual input (resp. output) parameter; we will refer to these ports as *task ports.* The set of input (resp. output) task ports for task invocation $\mathtt{inv}$ is $\mathtt{tips}(\mathtt{inv})$ (resp. $\mathtt{tops}(\mathtt{inv})$). A task port has the same type and initial value as the communicator (or module port) corresponding to the actual parameter denoted by the task port. The input (resp. output) task port that reads a port $\mathtt{p}$ of a module is $\mathtt{tip}^{\mathtt{p}}_{\mathtt{inv}}$ (resp. $\mathtt{top}^{\mathtt{p}}_{\mathtt{inv}}$). The input (resp. output) task port that reads from (resp. writes to) $\mathtt{i}$-th instance of a communicator $\mathtt{c}$ is $\mathtt{tip}^{\mathtt{c},\mathtt{i}}_{\mathtt{inv}}$ (resp. $\mathtt{top}^{\mathtt{c},\mathtt{i}}_{\mathtt{inv}}$). The set of task ports (both input and output) for a module $\mathtt{M}$ is $\mathtt{tpset}(\mathtt{M})$. The set of (module) ports read (resp. written) by a task invocation $\mathtt{inv}$ is $\mathtt{rdset}(\mathtt{inv})$ (resp. $\mathtt{wrset}(\mathtt{inv})$).

Well-formed programs. A program is *well-formed* if it conforms to the following syntactic restrictions (refer [3] for formal definitions):

Constraints on programs: (1) There is only one top-level program; and (2) each mode (other than those of the top-level program) has an unique parent mode.

Constraints on communicators: (1) if a communicator is declared in program $\mathtt{P}$, then it is not redeclared in any other sub-program of $\mathtt{P}$; (2) if a communicator $\mathtt{c}$ is accessed (read or written) by a task invocation or a switch in a mode of module $\mathtt{M}$ in program $\mathtt{P}$, then $\mathtt{c}$ is declared in one of the super-programs of $\mathtt{P}$; and (3) if a communicator $\mathtt{c}$ belongs to the hierarchical write-set of a module $\mathtt{M}$, then $\mathtt{c}$ does not belong to hierarchical write-set of any sibling module of $\mathtt{M}$.

Constraints on task invocations: (1) for every task invocation, the read time is earlier than the write time; (2) the precedence relation on task invocations is acyclic; (3) two task invocations in a mode cannot write to the same port or to the same instance of a communicator; (4) if a task invocation reads from or writes to a communicator $\mathtt{c}$ (resp. port $\mathtt{p}$), then the type of $\mathtt{c}$ (resp. $\mathtt{p}$) complies to the type of the corresponding formal parameter in task declaration; (5) if a task invocation in a mode $\mathtt{m}$ reads from or writes to a communicator $\mathtt{c}$, then period $\pi_{\mathtt{m}}$ is a multiple of the communicator period $\pi_{\mathtt{c}}$; and (6) every task invocation reads from a communicator or is transitively preceded by a task invocation that reads from a communicator, and writes to a communicator or is transitively followed by a task invocation that writes to a communicator.

Constraints on refinement: (1) if program $\mathtt{P}$ refines a mode $\mathtt{m}$, then the period of all modes in $\mathtt{P}$ is equal to $\pi_{\mathtt{m}}$ (this ensures that when there is a mode switch, there is no unsafe termination of tasks in lower-level modes); (2) every task invocation of a mode $\mathtt{m}$ that does not belong to the top-level program has an abstract parent task invocation in the parent of $\mathtt{m}$ (this ensures that the parent task acts as a placeholder for its children during schedulability analysis); (3) any two distinct task invocations in two modes of two (possibly identical) sibling modules have distinct parent task invocations (this ensures that all tasks that can potentially execute in parallel have unique parents); and (4) if $\mathtt{inv}'$ is the parent task invocation of $\mathtt{inv}$, then the read time of $\mathtt{inv}$ is not later than that of $\mathtt{inv}'$, the write time of $\mathtt{inv}$ is not earlier than that of $\mathtt{inv}'$, and every invocation that precedes $\mathtt{inv}$ refines a task invocation that precedes $\mathtt{inv}'$ (this ensures that the parent invocation is more constrained in time than the child task invocation, which is used in the schedulability analysis).

Well-timed programs. The *host set* $\mathtt{hset}$ for a program $\mathtt{P}$ is the set of hosts over which the program is distributed. A *host mapping* $\mathtt{hmap}$ is a function that assigns each module in top-level program of $\mathtt{P}$ to a host $\mathtt{h} \in \mathtt{hset}$. For a given host set $\mathtt{hset}$ and host mapping $\mathtt{hmap}$, the *worst-case execution time* (WCET) and the *worst-case transmission time* (WCTT) maps, $\mathtt{wemap}$ and $\mathtt{wtmap}$, assign two natural numbers to each task of $\mathtt{P}$: the WCET $\mathtt{wemap}(\mathtt{t})$ bounds the execution time of task $\mathtt{t}$ on host $\mathtt{hmap}(\mathtt{t})$; the WCTT $\mathtt{wtmap}(\mathtt{t})$ bounds the broadcasting of the outputs of task $\mathtt{t}$ to all hosts. Given a well-formed HTL program $\mathtt{P}$ together with an execution platform specified in terms of $\mathtt{hset}$, $\mathtt{hmap}$, $\mathtt{wemap}$, and $\mathtt{wtmap}$, the program $\mathtt{P}$ is *well-timed* if for all task invocations $\mathtt{inv}_1 = (\mathtt{t}_1, \cdot, \cdot, \cdot)$, $\mathtt{inv}_2 = (\mathtt{t}_1, \cdot, \cdot, \cdot)$: if $\mathtt{inv}_1$ is the parent invocation of $\mathtt{inv}_2$, then $\mathtt{wcet}(\mathtt{t}_1) \geq \mathtt{wcet}(\mathtt{t}_2)$ and $\mathtt{wctt}(\mathtt{t}_1) \geq \mathtt{wctt}(\mathtt{t}_2)$. Note that while well-formedness is independent of the execution platform, well-timedness is not.

5. OPERATIONAL SEMANTICS

The semantics (i.e., set of traces) of an HTL program is independent of the execution platform, which will be taken into account later, for code generation and schedulability analysis. The execution of an HTL program yields a (possibly infinite) sequence of configurations, called trace. Each configuration consists of values for all program variables (ports and communicators), a set of triggers, and a set of released (but not yet completed) tasks. A trigger defines an action to be taken at a future event, which is specified by an integer $\mathtt{n}$ and a set $\mathtt{cmps}$ of tasks: the trigger becomes enabled as soon as $\mathtt{n}$ time ticks have passed *and* all tasks in $\mathtt{cmps}$ have completed execution. In practice, time ticks and task completion event are raised by the execution platform through interrupts; the time unit is required to be a harmonic fraction of all communicator and mode periods. When a trigger becomes enabled, the associated action is carried out; this may be a communicator write (handled by write triggers), mode switch (handled by switch triggers), communicator read (handled by read triggers), or task release (handled by release triggers). Enabled write, switch, read, and release triggers are handled in this order. If no trigger is enabled, then the next time tick is awaited, and any number of released tasks may complete their execution.

Formally, a *trace* of an HTL program is a sequence of configurations $\mathtt{u}_0, \mathtt{u}_1, \ldots$, where $\mathtt{u}_0$ is the initial configuration, and for all $i > 0$, configuration $\mathtt{u}_i$ is a time event, write, switch, read, or release successor of $\mathtt{u}_{i-1}$. Each *configuration* $\mathtt{u}$ is a triple $(\mathtt{state}, \mathtt{trgs}, \mathtt{tasks})$, where $\mathtt{state}$ is variable state, $\mathtt{trgs}$ is a set of triggers, and $\mathtt{tasks}$ is a set of tasks. The *variable state* is a valuation of all communicators, module ports, and task ports. Without loss of generality, we assume that all communicator names and port names (both module and task ports) across the refinement hierarchy are unique; we furthermore assume that each task name uniquely identifies a particular task invocation (this can be achieved by duplication and renaming of task declarations). At a configuration $\mathtt{u}$, $\mathtt{c}_{\mathtt{u}}$ (resp. $\mathtt{p}_{\mathtt{u}}$) denotes the value of a communicator $\mathtt{c}$ (resp. port $\mathtt{p}$) and $\mathtt{cnd}_{\mathtt{u}}$ denotes the boolean value of a mode switch condition $\mathtt{cnd}$.

A *trigger* $\mathtt{g}$ is a triple $(\tau, \mathtt{e}, \mathtt{a})$, where $\tau \in \{\mathtt{w}, \mathtt{s}, \mathtt{d}, \mathtt{r}\}$ is a tag that identifies write, switch, read, and release triggers, respectively; $\mathtt{e}$ is an event instance; and $\mathtt{a}$ is action to be carried out when the trigger is handled. An *event instance* is a pair $(\mathtt{n}, \mathtt{cmps})$, where $\mathtt{n} \in \mathbb{N}_{\geq 0}$ and $\mathtt{cmps}$ is a set of task names. The trigger is *enabled* if $\mathtt{n} = 0$ and $\mathtt{cmps} = \emptyset$. A configuration is *waiting* if none of its triggers is enabled. For write triggers ($\tau = \mathtt{w}$), the action $\mathtt{a} = (\mathtt{c}, \mathtt{i}, \mathtt{t})$ specifies that the $\mathtt{i}$-th instance of communicator $\mathtt{c}$ will be updated to the value of the corresponding output task port of task $\mathtt{t}$. For switch triggers ($\tau = \mathtt{s}$), the action $\mathtt{a} = (\mathtt{sw}, \mathtt{m})$ specifies that the condition of mode switch $\mathtt{sw}$ of mode $\mathtt{m}$ will be checked, causing a possible mode switch. For read trigger ($\tau = \mathtt{d}$), the action $\mathtt{a} = (\mathtt{t}, \mathtt{c}, \mathtt{i})$ specifies that the $\mathtt{i}$-th instance of communicator $\mathtt{c}$ will be read into the corresponding input task port of task $\mathtt{t}$. For release triggers ($\tau = \mathtt{r}$), the action $\mathtt{a} = \mathtt{t}$ specifies that the (unique) invocation of task $\mathtt{t}$ will be released.

The five successor relations on configurations are defined formally in Fig. 14. Configuration $\mathtt{u}$ has a *time-event successor* if $\mathtt{u}$ is waiting; in this case, a time tick event and possibly some task completion events occur. The output task ports of the completed tasks are updated. The completed tasks are removed from the set of released tasks, and from all task sets of the triggers in $\mathtt{u}$. The positive time tick counts of

successor[†]	the following conditions hold on $\mathtt{u} = (\mathtt{state}, \mathtt{trgs}, \mathtt{tasks})$	the following conditions hold on $\mathtt{u}' = (\mathtt{state}', \mathtt{trgs}', \mathtt{tasks}')$
time event	no enabled trigger in $\mathtt{trgs}$	$\mathtt{tasks}' \subseteq \mathtt{tasks}$, if $(\cdot, (\mathtt{n}, \mathtt{cmps}), \cdot) \in \mathtt{trgs}$ then $(\cdot, (\mathtt{n} \ominus 1, \mathtt{cmps} \setminus (\mathtt{tasks} \setminus \mathtt{tasks}')), \cdot) \in \mathtt{trgs}'$, $\forall \mathtt{t} \in \mathtt{tasks} \setminus \mathtt{tasks}'$: $\forall \mathtt{p} \in \mathtt{tops}(\mathtt{t}) : \mathtt{p}_{\mathtt{u}'} = \mathtt{fn}[\Pi_{\mathtt{p}' \in \mathtt{tips}(\mathtt{t})} \mathtt{p}'_{\mathtt{u}'}]$, $\forall \mathtt{p} \in \mathtt{wrset}(\mathtt{t}) : \mathtt{p}_{\mathtt{u}'} = \mathtt{tops}^{\mathtt{p}}_{\mathtt{t},\mathtt{u}'}$ $\forall \mathtt{c} \in \mathtt{icset}(\mathtt{P}) : \mathtt{c}_{\mathtt{u}'} = \vartheta(\mathtt{type}_{\mathtt{c}})^{\star}$
write	$\exists \mathtt{g} = (\mathtt{w}, (0, \emptyset), (\mathtt{c}, \mathtt{i}, \mathtt{t})) \in \mathtt{trgs}$	$\mathtt{c}_{\mathtt{u}'} = \mathtt{top}^{\mathtt{c},\mathtt{i}}_{\mathtt{t},\mathtt{u}'}$, $\mathtt{trgs}' = \mathtt{trgs} \setminus \{\mathtt{g}\}$, $\mathtt{tasks}' = \mathtt{tasks}$

switch	no enabled write trigger in $\mathtt{trgs}$,
	$\exists \mathtt{g} = (\mathtt{s}, (0, \emptyset), (\mathtt{sw}, \mathtt{m})) \in \mathtt{trgs}$:
	$(\mathtt{sw} = (\mathtt{cnd}, \mathtt{m}_1) \wedge$
	$\forall (\mathtt{s}, (0, \emptyset), (\cdot, \mathtt{m}_2)) \in \mathtt{trgs} \setminus \{\mathtt{g}\}$: $\mathtt{m}_2 \notin \mathtt{ancestors}(\mathtt{m}))$
		if $\neg \mathtt{cnd}_{\mathtt{u}}$ and exists other enabled switch trigger for $\mathtt{m}$ in $\mathtt{trgs}$:
		$\mathtt{trgs}' = \mathtt{trgs} \setminus \{\mathtt{g}\}$, $\mathtt{tasks}' = \mathtt{tasks}$
		
		if $\neg \mathtt{cnd}_{\mathtt{u}}$ and no other enabled switch trigger for $\mathtt{m}$ in $\mathtt{trgs}$:
		$\mathtt{trgs}' = \mathtt{gset}_i(\mathtt{m}) \cup \mathtt{trgs} \setminus \{\mathtt{g}\}$, $\mathtt{tasks}' = \mathtt{tasks}$
		
		if $\mathtt{cnd}_{\mathtt{u}}$:
		$\mathtt{trgs}' = \cup_{\mathtt{m}_3 \in \mathtt{starts}(\mathtt{m}_1)} \mathtt{gset}_i(\mathtt{m}_3) \cup \mathtt{trgs} \setminus \mathtt{gset}_r(\mathtt{m}, \mathtt{u})$,
		$\mathtt{tasks}' = \mathtt{tasks}$

read	no enabled write or switch triggers in $\mathtt{trgs}$, $\exists \mathtt{g} = (\mathtt{d}, (0, \emptyset), (\mathtt{t}, \mathtt{c}, \mathtt{i})) \in \mathtt{trgs}$	$\mathtt{tip}^{\mathtt{c},\mathtt{i}}_{\mathtt{t},\mathtt{u}'} = \mathtt{c}_{\mathtt{u}'}$, $\mathtt{trgs}' = \mathtt{trgs} \setminus \{\mathtt{g}\}$, $\mathtt{tasks}' = \mathtt{tasks}$
release	no enabled write or switch or read triggers in $\mathtt{trgs}$, $\exists \mathtt{g} = (\mathtt{r}, (0, \emptyset), \mathtt{t}) \in \mathtt{trgs}$	$\forall \mathtt{p} \in \mathtt{rdset}(\mathtt{t}) : \mathtt{tip}^{\mathtt{p}}_{\mathtt{t},\mathtt{u}'} = \mathtt{p}_{\mathtt{u}'}$, $\mathtt{trgs}' = \mathtt{trgs} \setminus \{\mathtt{g}\}$, $\mathtt{tasks}' = \mathtt{tasks} \cup \{\mathtt{t}\}$

[†] *Values of variables remain unchanged from* $\mathtt{u}$ *to* $\mathtt{u}'$ *unless noted.*
* *ϑ non-deterministically assigns a value from type* $\mathtt{type}_{\mathtt{c}}$ *of communicator* $\mathtt{c}$.

$$\begin{aligned} \mathtt{n} \ominus 1 &= n - 1, \text{ if } n > 0 \\ &= n, \text{ otherwise} \end{aligned} \qquad \begin{aligned} &\mathtt{gset}_i(\mathtt{m}) = \text{Procedure\_Invoke\_Mode}(\mathtt{m}) \\ &\mathtt{gset}_r(\mathtt{m}, \mathtt{u}) = \{(\mathtt{s}, (0, \emptyset), (\cdot, \mathtt{m}_4)) \in \mathtt{trgs}(\mathtt{u}) : (\mathtt{m}_4 = \mathtt{m}) \vee (\mathtt{m} \in \mathtt{ancestors}(\mathtt{m}_4))\} \end{aligned}$$

Figure 14: Definition of successor configurations

all triggers in $\mathtt{u}$ are reduced by one. All input communicators are non-deterministically assigned a value from their type; for simplicity we assume all input communicators have period one.

Configuration $\mathtt{u}$ has a *write successor* if $\mathtt{u}$ contains an enabled write trigger. The write trigger is removed and a communicator is updated from an output task port. Configuration $\mathtt{u}$ has a *switch successor* if in $\mathtt{u}$ no write trigger is enabled, a switch trigger (say, for mode $\mathtt{m}$) is enabled, and there are no enabled switch triggers for any ancestors of $\mathtt{m}$. There are three different scenarios: (1) if the switch condition evaluates to false and there exists another enabled switch trigger from $\mathtt{m}$, then the switch trigger is removed; (2) the switch condition evaluates to false and there exists no other enabled switch trigger from $\mathtt{m}$, then the switch trigger is removed and $\mathtt{m}$ is invoked; and (3) if the switch condition (say destination mode is $\mathtt{m}'$) evaluates to true, then all enabled switch triggers of $\mathtt{m}$ and of the descendants of $\mathtt{m}$ are removed, and all modes in $\mathtt{starts}(\mathtt{m}')$ are invoked. An *invocation* of mode $\mathtt{m}$ (see Alg. 1) involves the following steps: (1) for each task invocation in $\mathtt{m}$, a read trigger is added for each communicator input; (2) for each task invocation in $\mathtt{m}$, a write trigger is added for each communicator output; (3) for each task invocation, a release trigger is added; and (4) for each mode switch, a switch trigger is added.

Configuration $\mathtt{u}$ has a *read successor* if in $\mathtt{u}$ no write or switch trigger is enabled, but some read trigger is enabled. The read trigger is removed and a communicator is read into an input task port. Configuration $\mathtt{u}$ has a *release successor* if in $\mathtt{u}$ no write or switch or read trigger is enabled, but a release trigger is enabled. The release trigger is removed, the input ports of a task are loaded with output port values from preceding tasks, and the task is added to the set of released tasks.

Algorithm 1 Procedure_Invoke_Mode($\mathtt{m}$)

```
  $\mathtt{gset} = \emptyset$;
  $\forall \mathtt{inv} = (\mathtt{t}, \mathtt{ailist}, \mathtt{aolist}, \cdot) \in \mathtt{invocs}(\mathtt{m})$
    $\forall k \in \mathbb{N}$ s.t. $\mathtt{ailist}[k] = (\mathtt{c}, \mathtt{i})$
      add trigger $(\mathtt{d}, (\mathtt{i} \cdot \pi_{\mathtt{c}}, \emptyset), (\mathtt{t}, \mathtt{c}, \mathtt{i}))$ to $\mathtt{gset}$;
    $\forall j \in \mathbb{N}$ s.t. $\mathtt{aolist}[j] = (\mathtt{c}, \mathtt{i})$
      add trigger $(\mathtt{w}, (\mathtt{i} \cdot \pi_{\mathtt{c}}, \emptyset), (\mathtt{c}, \mathtt{i}, \mathtt{t}))$ to $\mathtt{gset}$;
    add trigger $(\mathtt{r}, (\mathtt{n}, \mathtt{cmps}), \mathtt{t})$ to $\mathtt{gset}$ for
      $\mathtt{n} = \mathtt{rtime}(\mathtt{t})$ and $\mathtt{cmps} = \{\mathtt{t}' : (\mathtt{t}', \cdot, \cdot, \cdot) \in \mathtt{prec}(\mathtt{t}, \mathtt{m})\}$;
  $\forall \mathtt{sw} \in \mathtt{switches}[\mathtt{m}]$
    add trigger $(\mathtt{s}, (\pi[\mathtt{m}], \emptyset), (\mathtt{sw}, \mathtt{m}))$ to $\mathtt{gset}$;
  return $\mathtt{gset}$.
```

The *initial* configuration of a program is defined as follows: the variable state assigns the initial values to all module ports, task input ports and communicators and default values (specified by the types) to task output ports; the trigger set consists of triggers by invoking modes in start set of $\mathtt{start}(\mathtt{M})$ for each module $\mathtt{M}$ in top-level program; and an empty task set.

6. HTL COMPILER

We have designed and implemented a compiler for full, distributed HTL in Java. The compiler checks well-formedness, well-timedness, and schedulability of a given HTL program, flattens the program into a semantically equivalent HTL program with only top-level modules, and then generates so-called *E code* for the flattened program targeting the *E(mbedded) Machine* [2]. In our experiments, we have used an existing implementation of the E machine written in C running on Linux. E code specifies the exact real-time instants when port and communicator values are exchanged,

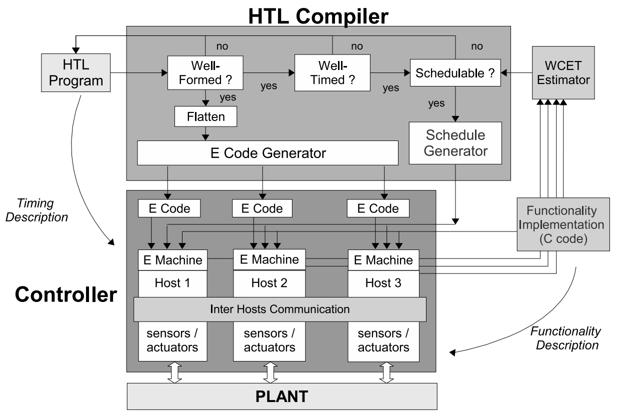

Figure 15: Structure of compiler and target

and when tasks are released and terminated. E code neither implements the actual tasks' functionality nor specifies when released tasks actually execute. Task functionality must be implemented in some other programming language and compiled separately using an appropriate compiler. Here, we have chosen C for implementing tasks, since our version of the E machine is implemented in C. Released tasks are dispatched for execution by an EDF scheduler that is external to the E machine and also implemented in C.

Fig. 15 depicts the structure of the compiler and the target architecture. Each host runs its own E machine and maintains its own copies of all task ports and communicators of an HTL program, even if some task ports and communicators are never accessed by tasks on that host. The compiler generates E code for each host separately. The idea is to compile repeatedly the whole HTL program for each host and to generate E code that implements the whole program on each host, except that the tasks of the modules not mapped to a host are not invoked on that host. Thus the generated E code is identical across all hosts except for the instructions that invoke tasks. Each task invocation involves broadcasting the task's output port values and storing the values in the respective task output ports on all receiving hosts. As a result, each host maintains a complete image of all port and communicator values of an HTL program. Note that all host-to-host transmission is done by the tasks, not by the E code. In our prototype implementation, tasks broadcast via sockets and standard Ethernet. Memory consumption as well as transmission load may be minimized if necessary using, e.g., data-flow analysis, which is, however, future work. In the following, we explain each phase of the compiler.

Checking well-formedness, well-timedness and schedulability. Before flattening the input program, the compiler checks the well-formedness and well-timedness of the program. In particular, the compiler verifies that any concrete task indeed refines its parent task to make sure that the subsequent top-level scheduling test guarantees overall schedulability. The compiler performs an EDF-scheduling test on the abstract, top-level portion of the input program only. If the test succeeds, according to our result in Section 7, the whole input program is schedulable. This result also applies to distributed HTL programs as long as the WCTT for broadcasting the output port values of each task is added to the WCET of the task, and the WCTT includes the time it takes to resolve any collisions even when all hosts try to broadcast at the same time. In our current implementation, output port values of a task are always broadcast as soon as an invocation of the task completes. Transmission and scheduling techniques that may utilize the network more efficiently, e.g. [4], can also be used but have not been implemented.

Flattening. The HTL compiler flattens a given well-formed and well-timed HTL program into a semantically equivalent flat HTL program. A module of a flat HTL program may only contain modes that do not contain a refining program. Flattening works by essentially computing the product of all modes in the refinement of each top-level module of the original program. This is easy because these modes all have the same period. Only modes in different top-level modules may have different periods. In order to maintain semantical equivalence, flattening needs to prioritize mode switch checking, i.e., mode switches in more abstract modules need to be checked before mode switches in more concrete modules [3]. Flattening an HTL program may in theory result in generated code that is exponentially larger than the size of the input program. However, execution of the generated code is very efficient and is readily supported by existing versions of the E machine. An HTL compiler that may shift the trade-off between code size and execution efficiency more towards smaller code size by generating code directly from the unflattened input program is future work. Note that for such a compiler the design of the E machine may have to be modified as well.

Target machine. The HTL compiler generates E code, which has a semantics that is designed to simplify code generation and can be executed very efficiently [5]. Besides releasing tasks, E code also controls when port and communicator values are copied (or initialized) using so-called *drivers*, which are implemented in C, one for each data type. E code consists of the following instructions: a $call(d)$ instruction executes the driver d; a $release(t)$ instruction releases the task t for execution by the EDF scheduler; a $future(g, a)$ instruction marks the E code at the address a for future execution when the predicate g evaluates to true, i.e., when g is *enabled*. We call g a *trigger*, which observes events such as time ticks and the completion of tasks. Here, we only use triggers that are enabled when all observed events have occurred. The E machine maintains a FIFO queue of trigger-address pairs. 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. An $if(\texttt{cnd}, a)$ instruction branches to the E code at address a if predicate `cnd` evaluates to true. We call `cnd` a *condition*, which observes port or communicator states. A $jump(a)$ instruction is an absolute jump to E code address a, and a *return* instruction completes the execution of an E code sequence.

E code generation. Given a well-formed, well-timed, schedulable, and flattened HTL program and a mapping of its top-level modules to hosts, the HTL compiler generates E code for the program and mapping by invoking Algorithm 2 for each host, which invokes Algorithm 3 to generate E code for each module of the program, which finally invokes Algorithm 4 to generate E code for each mode of each module. The compiler conceptually divides each mode into uniform temporal segments called units. The *unit* of a mode is the smallest time interval at which any two consecutive communicator instances are accessed in that mode. Given a mode `m`, we denote the duration of its unit by $\gamma[\texttt{m}]$, which is the gcd of all access periods of all communicators accessed in `m`. The total number of units of `m` is $\pi[\texttt{m}]/\gamma[\texttt{m}]$, where $\pi[\texttt{m}]$ is the period of `m`. The compiler generates separate E code blocks for each unit of a mode. The address of an E code block corresponding to unit i of a mode m is denoted by *unit_address*[m, i]. This is a symbolic address to which instructions may forward reference and therefore may need fix up during compilation. We use similar notation for other symbolic addresses.

We also use the following auxiliary operators. The driver init(x) initializes the communicator or task port x. The set readDrivers(m, u) contains the drivers that load the tasks in mode m with values of the communicators that are read by these tasks at unit u. The set writeDrivers(m, u) contains the drivers that load the communicators with the output of the tasks in mode m that write to these communicators at unit u. The set portDrivers(t) contains the drivers that load input ports of task t with the values of module ports on which t depends. The set complete(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 releasedTasks(m, u) contains the tasks in mode m with no precedences that are released at unit u. The set precedenceTasks(m) contains the tasks in mode m with precedences.

Algorithm 2 generates instructions to initialize all communicators and modules. Here, we use instructions of the form *future*(0, *module_address*[M]), which effectively execute the E code at the address *module_address*[M] similar to a jump-to-subroutine instruction. However, the actual mechanism is more complicated: for a *future*(0, a) instruction the E machine appends the already enabled trigger-address pair (true, a) to the trigger queue and then proceeds to the next instruction. Only when the E machine reaches a *return* instruction, the machine checks the trigger queue again and eventually removes the pair (true, a) from the trigger queue and executes the E code at the address a but not before it executed the E code of all other enabled trigger-address pairs occurring before (true, a) in the queue.

Algorithm 2 GenerateECodeForProgramOnHost(P, h)

```
// initialize communicators
$\forall$c $\in$ comms(P): emit(call(init(c)))
// initialize and start each module
$\forall$M $\in$ modules(P): emit(future(0, module_address[M]))
// end initialization phase
emit(return)
// generate code for each module
$\forall$M $\in$ modules(P): GenerateECodeForModuleOnHost(M, h)
```

Algorithm 3 generates instructions to initialize all task ports in a module, and to start the execution of the module by jumping to the E code of the first unit of the start mode of the module. *PC* denotes the compiler's program counter.

Algorithm 3 GenerateECodeForModuleOnHost(M, h)

```
set module_address[M] to PC and fix up
// initialize task ports
$\forall$p $\in$ tpset(M): emit(call(init(p)))
// jump to the start mode at unit 0
emit(jump, unit_address[start[M], 0])
// generate code for each mode
$\forall$m $\in$ modes(M): GenerateECodeForModeOnHost(m, h)
```

Algorithm 4 generates the E code for all units of a mode. Only unit 0 contains instructions to check mode switching because mode switching may only occur at the beginning of a mode. When a mode switch occurs, E code execution continues at the *mode_address*[m′] of the target mode m′, not the *unit_address*[m′, 0], since only at most one mode switch per time instant may occur. At each time instant, the generated E code uses *future*(0, a) instructions to write communicators always before any communicator is read making sure that the latest communicator values are used across all modules. Communicator and port values need not be buffered since tasks are invoked at most once per mode period and communicator-to-port transactions are done as soon as possible while port-to-communicator transactions are done as late as possible. It is therefore sufficient to have a single copy of each communicator and task port on each host. Additional memory is not required.

Algorithm 4 GenerateECodeForModeOnHost(m, h)

```
u := 0
while u < $\pi$[m]/$\gamma$[m] do
  set unit_address[m, u] to PC and fix up
  // update communicators with task output
  $\forall$d $\in$ writeDrivers(m, u): emit(call(d))
  // continue after other modules updated communicators
  emit(future(0, PC + 2))
  emit(return)
  if (u = 0)
    // check mode switches
    $\forall$(cnd, m′) $\in$ switches(m): emit(if(cnd, mode_address[m′]))
    set mode_address[m] to PC and fix up
  end if
  if (mode m is contained in a module on host h)
    // read communicators into tasks
    $\forall$d $\in$ readDrivers(m, u): emit(call(d))
    // release tasks with no precedences
    $\forall$t $\in$ releasedTasks(m, u): emit(release(t))
    if (u = 0)
      // release tasks with precedences
      $\forall$t $\in$ precedenceTasks(m):
        // wait for tasks on which t depends to complete
        emit(future(complete(t), PC + 2))
        emit(jump(PC + 5 + |portDrivers(t)|))
        // release t after other modules updated communicators
        emit(future(0, PC + 2))
        emit(return)
        // read ports of tasks on which t depends, then release t
        $\forall$d $\in$ portDrivers(t): emit(call(d))
        emit(release(t))
        emit(return)
    end if
  end if
  // continue mode after $\gamma$[m] time
  emit(future($\gamma$[m], unit_address[m, u + 1 mod $\pi$[m]/$\gamma$[m]]))
  emit(return)
  u := u + 1
end while
```

7. SCHEDULABILITY

The schedulability analysis for HTL is given a program P, a set hset of hosts, a host map hmap (an assignment from modules to hosts), a WCET map wemap (an assignment from tasks to positive integers), and a WCTT map wtmap (also an assignment from tasks to positive integers). For scheduling, we extend program configurations with a map tmap that assigns to every task t in the task set, a nonnegative integer tmap(t) that indicates the number of time units that the currently active invocation of t has been executed. An *extended program trace* is a sequence of such extended configurations. A *scheduler* is a function that maps every finite extended program trace (representing the past execution of the program) which ends in a waiting configuration, to a scheduling decision for every time unit and every host: it may decide to keep the host idle (no task is chosen from the task set), execute a task (from the task set), or transmit the output of a task (from the task set). The scheduler is assumed to be discrete time; i.e., it makes decisions at a clock tick. The clock tick is synchronized over all hosts and is a harmonic fraction of the program clock (which is the minimum interval between communicator accesses or mode switches). The WCETs and the WCTTs are specified as multiples of clock ticks.

The scheduler is *time- and transmission-safe* if every infinite extended program trace produced by the scheduler has the following two properties: (1) a task writing to a communicator must have completed execution and output transmission before the communicator update; and (2) if a host is transmitting, then all other hosts are listening (i.e., neither executing nor transmitting). The *schedulability problem* asks, given `P`, `hset`, `hmap`, `wemap`, and `wtmap`, if there exists a time- and transmission-safe scheduler. It can be solved in time linear in the number of extended program configurations [6], but this is often too expensive.

For efficiency reasons, we solve the schedulability problem only for flat HTL programs, namely, for the top-level program `abs(P)`. Then, due to the HTL constraints on refinement, we can construct a scheduler for the complete program `P` from a scheduler for `abs(P)`. This is achieved by iteratively scheduling every task execution during time slots in which the parent task is executed; see [3]. In other words, HTL guarantees that top-level schedulability is a sufficient condition for schedulability. For top-level schedulability on a single host, we use an EDF scheduling algorithm for tasks with precedences [7]. For top-level schedulability on multiple hosts, one needs to account for transmission times, and we can use the techniques of [4].

8. RELATED WORK

Timed languages. HTL builds on the LET concept pioneered by the Giotto language [1]. LET-based languages include TDL [8], which like Giotto is restricted to one level of periodic tasks; Timed Multitasking (TM) [9], which defines LET properties through deadlines; and xGiotto [10], an event-triggered LET language. HTL differs from Giotto in that logical execution times are defined through the reading and writing of communicator instances. This adds considerable flexibility, and naturally supports task precedences and hierarchical refinement. At the other extreme, in fully event-triggered timed languages, such as xGiotto, scheduling quickly becomes intractable.

Synchronous languages. Esterel [11], Lustre [12], and Signal [13] are based on the synchrony assumption that the execution platform is sufficiently fast as to complete execution before the arrival of the next environment event. As with timed languages, the behavior of synchronous programs is deterministic. While the synchronous languages theoretically subsume timed languages, HTL offers explicitly a program structure that supports the refinement of tasks into task groups with precedences.

Other real-time languages. Most real-time modeling languages such as Simulink [14] rely on simulators and code generators (e.g., Real-Time Workshop [15]) to define the semantics. There are also soft real-time languages for specialized domains: nesC [16] is targeted towards sensor networks; Erlang [17] towards telecommunication; Flex [18] offers flexible trade-offs between time, resources, and precision. Program execution in any of these languages is not schedule independent.

9. CONCLUSION

We presented HTL, a hierarchical coordination language for safety critical hard real-time applications. HTL is built upon the Logical Execution Time model of task execution and allows parallel composition of modules and horizontal refinement of tasks without modifying the timing behavior. The hierarchical layers of abstraction allows efficient and concise specification without overloading program analysis. We introduced the restrictions on a general HTL program to guarantee schedulability of lower levels if higher levels of abstraction are schedulable. We also presented the operational semantics for HTL and the implementation of an HTL compiler. The compiler checks well-formedness, well-timedness, and schedulability of a given HTL program, flattens the program into a semantically equivalent HTL program with only top-level modules, and then generates code for the Embedded Machine. A steer-by-wire system was used to illustrate the use and the features of HTL.

10. REFERENCES

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

[2] 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

[3] Ghosal, A., Henzinger, T.A., Iercan, D., Kirsch, C., Sangiovanni-Vincentelli, A.: Hierarchical timing language. Technical report, UC Berkeley (2006)

[4] Tindel, K., Clark, J.: Holistic schedulability for distributed hard real-time systems. Microprocessing and Microprogramming - Euromicro Journal **40** (1994) 117–134

[5] Kirsch, C., Sanvido, M., Henzinger, T.: A programmable microkernel for real-time systems. In: Proc. ACM/USENIX Conference on Virtual Execution Environments, ACM Press (2005) 35–45

[6] 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

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

[8] Farcas, E., Farcas, C., Pree, W., Templ, J.: Transparent distribution of real-time components based on logical execution time. SIGPLAN Not. **40** (2005) 31–39

[9] Liu, J., Lee, E.A.: Timed multitasking for real-time embedded software. IEEE Control Systems Magazine **23** (2003) 65–75

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

[11] Boussinot, F., de Simone, R.: The ESTEREL language. Proceedings of the IEEE **79** (1991) 1293–1304

[12] Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous data-flow programming language LUSTRE. Proceedings of the IEEE **79** (1991) 1305–1320

[13] Guernic, P.L., Borgne, M.L., Gauthier, T., Maire, C.L.: Programming real time applications with SIGNAL. Proceedings of the IEEE **79** (1991) 1321–1336

[14] Simulink: (http://www.mathworks.com/products/simulink/)

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

[16] Gay, D., Levis, P., von Behren, R., Welsh, M., Brewer, E., Culler, D.: The nesC language: A holistic approach to networked embedded systems. In: Proceedings of Programming Languages Design and Implementation, ACM Press (2003) 1–11

[17] Armstrong, J., Virding, R., Wikström, C., Williams, M.: Concurrent Programming in Erlang. Second edn. Prentice-Hall (1996)

[18] Kenny, K., Lin, K.J.: Building flexible real-time systems using the FLEX language. IEEE Computer **24** (1991) 70–78