Hierarchical Timing Language

Arkadeb Ghosal
Thomas A. Henzinger
Daniel Iercan
Christoph Kirsch
Alberto L. Sangiovanni-Vincentelli

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2006-79
http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-79.html

May 23, 2006

Copyright © 2006, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Acknowledgement

This work was supported in part by GSRC grant 2003-DT-660 and in part 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.

Hierarchical Timing Language*

Arkadeb Ghosal	Thomas A. Henzinger	Daniel Iercan
UC Berkeley	EPFL	”Politehnica” U. of Timisoara
arkadeb@eecs.berkeley.edu	tah@epfl.ch	daniel.iercan@aut.upt.ro

Christoph Kirsch	Alberto Sangiovanni-Vincentelli
University of Salzburg	UC Berkeley
ck@cs.uni-salzburg.at	alberto@eecs.berkeley.edu

May 23, 2006

Abstract

We have designed and implemented a new programming language for hard real-time systems. Critical timing constraints are specified within the language, and ensured by the compiler. The main novel feature of the language is that programs are extensible in two dimensions without changing their timing behavior: new program modules can be added, and individual program task can be refined. The mechanism that supports 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. The language, which assembles real-time tasks within a hierarchical module structure with timing constraints, is called Hierarchical Timing Language (HTL). It is a coordination language, in that individual tasks can be implemented in other languages. We present a distributed HTL implementation of an automotive steer-by-wire controller as a case study.

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, by scheduling theory 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 refers to time in an indirect way, often through low-level constructs such as priorities. One of the main challenges in real-time programming, therefore, is lifting the level of abstraction. In

*This work was supported in part by GSRC grant 2003-DT-660 and in part 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.

this report, 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 the 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 the rest of the system.

In HTL, tasks can be refined, in multiple levels, 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 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 refinement (hierarchy), HTL supports the collection of tasks into *modes*, which can be composed sequentially. We provide an operational semantics for HTL and define a (simple) compiler to generate E code (code used by the Embedded Machine). The compiler performs schedulability analysis and translates HTL programs into code for the Embedded Machine [2]. The HTL compiler can generate E code for an 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.

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, and the controller (along with different functionalities like fault detection, supervisory control, and power coordinator) is implemented on more than one processor. While the example is particularly useful to show the use of communicators and task refinement, it also illustrates the need of 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.

Overview. Section 2 presents a brief overview on communication and computation model of HTL and basic program structures. Section 3 discusses an implementation of the steer-by-wire controller in HTL. Section 4 presents the abstract syntax for HTL and the restrictions required to ensure race-free execution and schedulable programs. Section 5 presents the operational semantics of the language. Section 6 discusses the compiler for HTL to generate E code.

Section 7 presents an overview of schedulability check for HTL. Section 8 compares HTL with related works. Section 9 concludes the report.

2 Overview

Logical execution time. LET model(Figure 1) decouples the time when a software task reads input and writes output from the time when the task executes. An LET task is a sequential code operating on memory that has been assigned to the task upon release (and is not accessible to any other tasks). The *release* and *termination* decides the LET for the task; the task is *active* between the release and termination events which are triggered clock ticks or sensor interrupts. The input of an LET task is written into its assigned memory when the release event occurs and not when the task actually *starts* executing. Similarly the output is available to other tasks or actuators at the termination event, even if the task completes physical execution earlier. In between the start and completion of execution the task may be preempted and then resume the execution. LET tasks are time and value deterministic, portable and composable. An LET task is *time safe* on some given hardware if the task *completes* execution on that hardware before the termination event occurs.

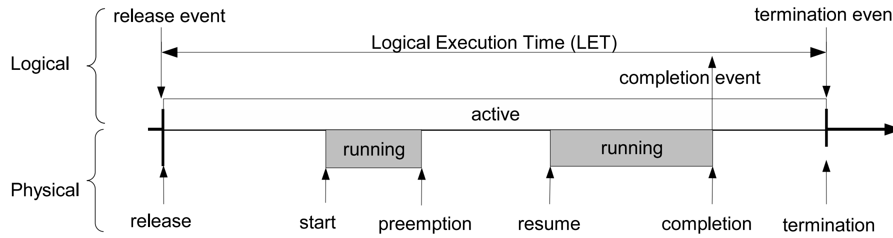


Figure 1: Logical Execution Time (LET) Model

Communication and computation model. The communication model for HTL is centered around *communicators*. A communicator is a variable (with a structured data type and has values from a set that complies to the data type) which can be accessed (i.e. read from or write to) only at specific time instances. We specify the time instance through a communicator period. The computation model for HTL comprises of software tasks; a *task* is sequential code without any internal synchronization point. A task reads from certain instances of some communicators, evaluates a pre-defined function based on the input and updates certain instances of the same or other communicators.

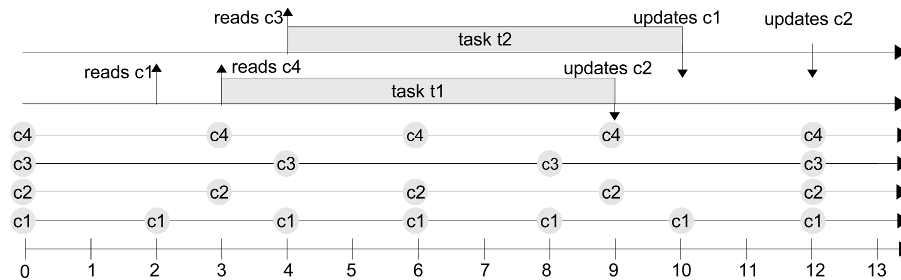


Figure 2: Communicators and Tasks

Figure 2 shows the interaction between four communicators and two tasks. The four communicators, c_1, c_2, c_3 and

c_4 have periods 2, 3, 4 and 3 respectively; the instances of access have been shown along the time line. Task t_1 reads second instances of c_1 and c_4 and updates fourth instance of c_2. Task t_2 reads second instance of c_3 and updates sixth instance of c_1 and fifth instance of c_2. The read and write instances implicitly specify the LET for the tasks; thus LET of t_1 span from time unit 3 to time unit 9 and LET of t_2 span from time unit 4 to time unit 10.

Communicators determine the interaction between tasks. While communicators are the key to write HTL programs in a compositional way, they also ensure deterministic behavior. Determinism implies that given sufficient CPU speed, the real-time behavior of the program is determined by the input, independent of the host speed and utilization and is maintained by ensuring no race on communicators and that a communicator is updated before it is read.

HTL allows direct communication between tasks with identical frequencies (tasks with different frequencies can only communicate via communicators); communication through ports ensure zero latency. Figure 3 shows three tasks t_1, t_2 and t_3. Task t_1 reads second instances of c_1 and c_4 and task t_2 reads second instance of c_3. Completion of t_1 and t_2 are not specified; instead a third task t_3 reads the evaluation of t_1 and t_2 and updates the fifth instance of c_2. The communication between t_1 (and t_2) and t_3 occurs through ports. A *port* is variable with fixed data type but is not bound to time instances i.e. as soon as the evaluation of t_1 is complete, task t_2 can read the output.

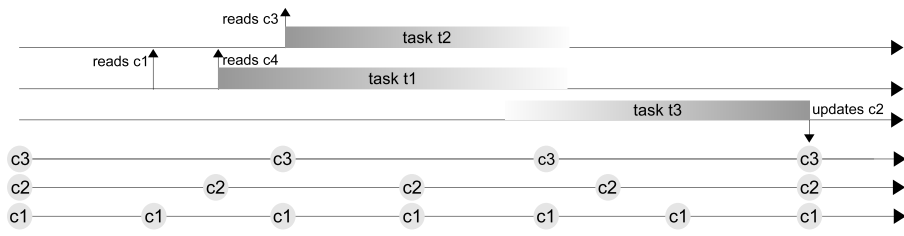


Figure 3: Communicators, Ports and Tasks

Program. An HTL *program* is a set of communicators and a set of modes. A *mode* is a group of task with identical frequency (expressed as mode period). Tasks within a mode may interact through ports; however tasks in different modes can only communicate through communicators. The ports in a mode express the precedence relation between tasks; if a task *precedes* another task then the second task must read a port updated by another task. Figure 4 shows two modes m_1 and m_2 with periods 6 and 12 respectively.

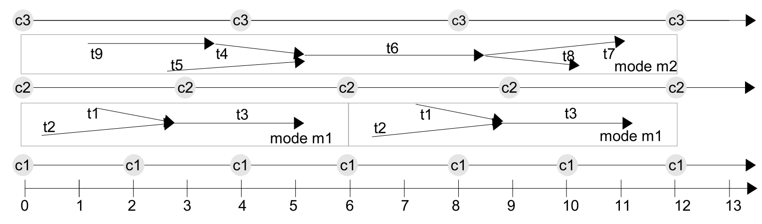


Figure 4: Modes, Tasks and Communicators

In real-time applications a group of tasks get replaced by an alternate group depending on some pre-defined condition (e.g. change in temperature readings). HTL accommodates this by allowing switching of modes (possible only at end of mode periods) based on some condition specified as a predicate on communicators and ports. A network of modes switching between themselves is referred as a *module*(Figure 5). In the modified specification model, an HTL program is a set of modules and a set of communicators. The modules are composed in parallel while modes in a module are composed sequentially. At any instance, tasks of at most one mode of a module may be executing. One mode in each

module is specified as the start mode and starts executing (before any other modes of the module) when the module is executed.

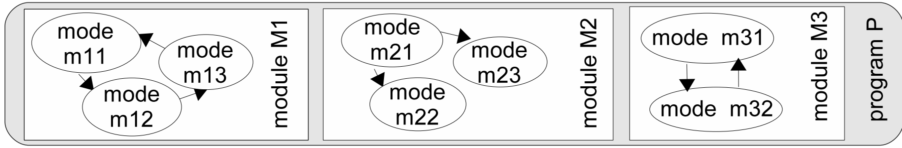


Figure 5: Modules and Modes

Refinement. Specifying all behaviors through mode switching is cumbersome. To have an efficient and concise specification we introduce the concept of *mode refinement*; a mode in a program can be replaced by an HTL program. This does not add expressiveness of the model; in fact an HTL program with arbitrary levels of refinement can be translated into one with no refinement. However the feature allows a compact representation without overloading analysis; e.g. for schedulability analysis one does not need to consider all possible combinations of refinement modes and this save subsequent computation effort. In fact for an HTL program (with certain restrictions) schedulability is ensured if the top-level program (without considering any refinement) is schedulable.

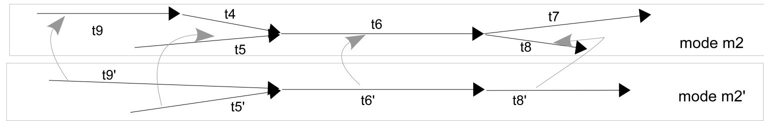


Figure 6: Refinement

Figure 6 shows a mode $\mathtt{m}$ and a mode $\mathtt{m}'$ (from a program which refines $\mathtt{m}$). The tasks in $\mathtt{m}$ execute in parallel to that of $\mathtt{m}'$. HTL imposes certain restrictions on $\mathtt{m}'$ to ensure efficient analysis. First, 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 period), all tasks in the modes refining $\mathtt{m}$ has terminated execution. Second, every task in $\mathtt{m}'$ maps to an unique task in $\mathtt{m}$ (shown by grey arrows) e.g. $\mathtt{t}_5'$ (child) to $\mathtt{t}_5$ (parent). HTL considers $\mathtt{t}_5$ as a placeholder (or an *abstract* task) to that of $\mathtt{t}_5'$ (or an *concrete* task); in other words $\mathtt{t}_5$ does not execute at run-time but ensures that $\mathtt{t}_5'$ is accounted for during schedulability analysis of the top program. Also, the latest (earliest) communicator read (write) of $\mathtt{t}_5'$ should be equal to or earlier (later) than that of $\mathtt{t}_5$, dependencies of $\mathtt{t}_5'$ should be a subset of dependencies of $\mathtt{t}_5$ and worst-case-execution-time of $\mathtt{t}_5'$ should be less than equal to that of $\mathtt{t}_5$. The constraint ensures that if $\mathtt{t}_5$ can be scheduled in the top program, $\mathtt{t}_5'$ can be scheduled for the whole program (refer Section 7 for complete analysis).

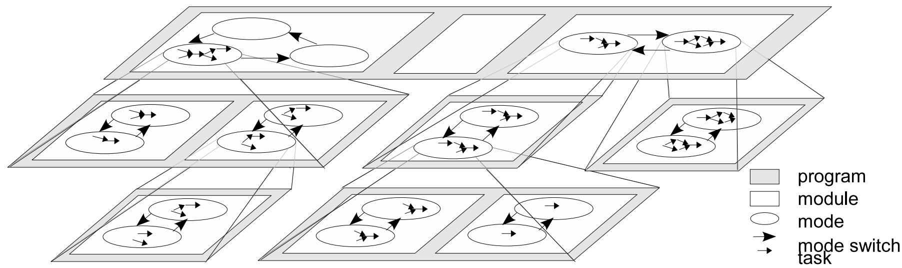


Figure 7: Program and Refinement

Refinement allows efficient specification by allowing *choice* and *change of behavior*. The choice is expressed when an abstract task in a mode can be parent of several concrete or abstract task in different modes of a refinement program.

By using the hierarchy, the choices can be used in a structured manner. The change of behavior is exploited by having the child task reading from/ writing to different communicators within the constraints discussed above. Figure 7 shows a diagrammatic view of an HTL program with refinement. The modes shows the tasks with precedences and the refinement modes show possible orientation of children concrete/ abstract tasks.

Distribution. Many embedded applications are distributed; tasks are distributed on several hosts and interact with each other through communication channels. In HTL, distribution is specified through a mapping of modules to hosts; the distribution is implemented by replicating shared communicators on all hosts and then have the tasks (writing to the shared communicators) broadcast the outputs. The semantics remain the same as if they were running on a single host; however code generation and analysis take the distribution into account. The LET model is extended to include both execution and transmission of output (Figure 8).

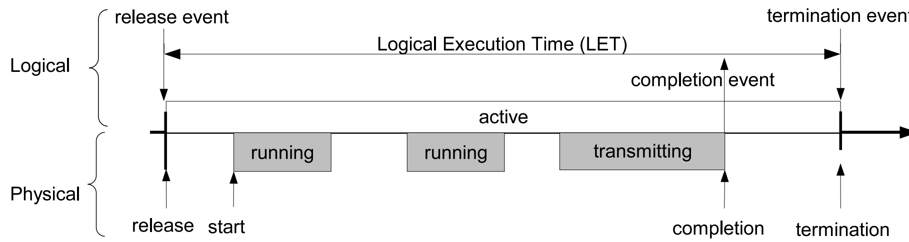


Figure 8: Task execution and transmission

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 steering wheel angle sensors, 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 steer-by-wire model used by General Motors for their prototype hydrogen fuel-cell car FX-3. The example is an imitation of the concerns and requirements and does not represent a real set of control algorithms for an actual product or prototype.

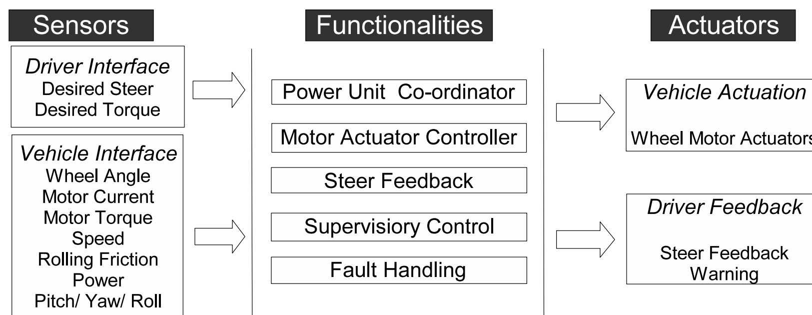


Figure 9: Data Flow and Functional Blocks

The sensors (Figure 9) read desired steer/ torque from driver and current vehicle state (wheel angle, motor current, speed, friction, power, pitch, yaw etc). The system functionality is divided into five parts: computation of wheel

motor actuation and steer feedback, supervisory control, fault handling and power coordinator. Supervisory control coordinates between steering, braking and suspension; for simplicity we are not implementing the braking and suspension and assume that the interface is being provided as a set of sensor values. The supervisor typically runs in triple-redundant mode (three copies are executed in three different processors). The fault handling system detects, isolates and mitigates fault and warns the driver in case of fault. Power coordinator handles the coordination of motor current computed by the controller with rest of the power grid.

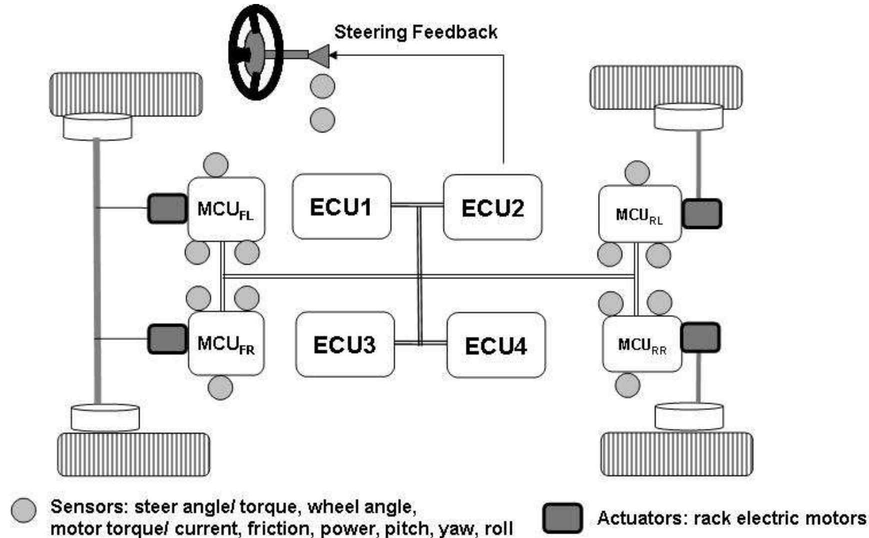


Figure 10: Implementation of SBW system

An architecture for SBW (Figure 10) consists of eight hosts (or processors): four motor control units (MCUs) and four electronic control units (ECUs). The MCUs are placed near the wheels and detect sensor values related to wheels and send signals to motor actuator. The ECUs implement rest of the functionalities. All hosts are connected through a communication link that allows broadcast from any host.

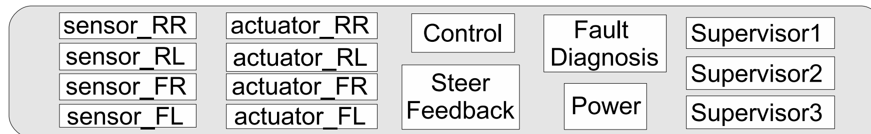


Figure 11: Modules for the SBW implementation

Steer-by-wire in HTL. Each of the functional units of the SBW system is represented by a module in the HTL implementation (Figure 11[1]). There are nine modules: sensor units for each wheel (rear-left: RL, rear-right: RR, front-left: FL, and front-right: FR), actuator units for each of the four wheels (RL, RR, FL, and FR), and computational units for control, steer feedback, fault diagnosis, power and supervisor (there are three copies of supervisor).

Each of the above units behave differently under varied conditions. For example, the wheel actuation need to be done faster over a critical speed; computation of actuation signal at the time of engine start is different from that at

[1]In the figures program, module and mode will be denoted by an oval box, rectangle and an ellipse respectively.

high speed; supervisor functions differently when the car is running under emergency condition; fault diagnosis uses a different set of computations at normal driving conditions than when a fault is detected etc. The change in the behavior is captured by mode switches; Figure 12 shows the different modes of the modules (period of the modes are in millisecond and are shown in the shaded box). Due to space constraints, we will not present the details of the modes; refer http://htl.cs.uni-salzburg.at for full specification. Next we will present a specific scenario of inter-mode communication.

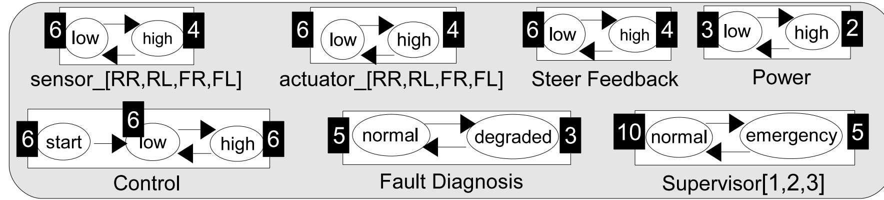


Figure 12: Modes for the modules

Angle for rear-left wheel (car is moving at a high speed) is measured by three sensors; the sensor values (after appropriate modification) are read by a task that mediates on the value to be considered and then pass it onto 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 period 4000 and 500 microseconds respectively. Instances of communicators are referred relative to mode period (0-th instance corresponds to the start of the mode). Tasks `SENWheelA3, SENWheelA4, SENWheelA5` (in mode `high` of module `sensor_RL`) reads three sensors A3, A4, A5 (connected to RL wheel) at the start of the period and update first instance of three communicators `cA3`, `cA4` and `cA5`. Task `MEDAngleRL` (in mode `high` of module `steer feedback`) reads the first instance of the above communicators. The program semantics ensure that write precedes read. Task `MEDAngleRL` write to the second instance of communicator `cAngleRL`. Task `cntrlFUN` reads the second instance of `cAngleRL` (among other communicators) and computes value of wheel actuation signal. Task `RackPinAct` reads the output port of `cntrlFUN` and computes the power requirement for the motors.

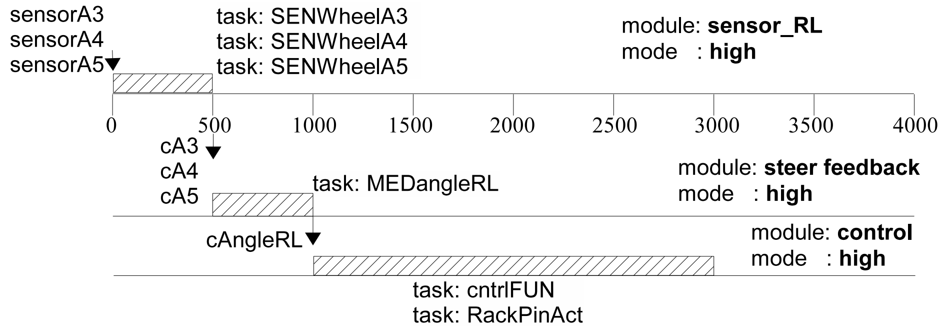


Figure 13: Communication in SBW implementation

Scenario-specific functionalities can be further differentiated. For example, at high speeds control law for computing the actuation signal differs on the basis of whether the car is driven manually or under cruise. Similarly the tasks executed during emergency by the supervisory control are different in the case of under-steering from that of oversteering. Fault handling functionality depends on whether there is a communication fault or processor fault. If all of these are expressed in one module, the module size becomes large and unmanageable - refinement allows an efficient solution. In the SBW program (Figure 14), the mode `low` (of module `control`) is refined by a program which has one module

with two modes `idle` and `motion`; the mode `motion` is further refined by a program with a module with two modes `crawl` and `average`. If the modes were not refined then the module `control` would have 6 modes and 17 mode switches; this is not only inefficient but error prone.

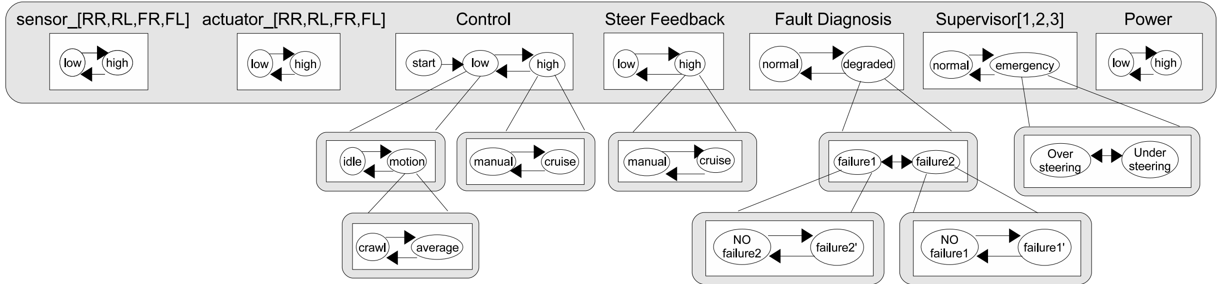


Figure 14: HTL program for SBW system with all the refinements

Distribution of an HTL program is specified by a mapping of the modules of the top-level program (a program which does not refine any mode) to hosts; the modes of the module and the corresponding refinements are bound to execute on the same host. The SBW system is distributed over 8 hosts: sensor and actuator modules for each wheel share one host; modules `control`, `steer feedback` and `fault diagnosis` are distributed on three hosts along with one supervisor module; module `power` is assigned to one host.

Implementation. We have implemented the case study 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 1800 virtual machine instructions per host. The virtual 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. See Section 6 for more details on the HTL compiler and runtime system. The tasks are written in C but do not actually implement any functionality, only bounded empty loops. Our implementation simulates the case study in real time but at a frequency of 2Hz, which is 1000 times slower than the actual system, and therefore only demonstrates the correctness of the code generated for the HTL program of the case study.

4 Abstract Syntax

We provide the main components of the HTL language in an abstract way. In practise a concrete syntax can be written from this abstract syntax (refer Appendix A).

An *HTL program* $\mathtt{P}$ consists of the following components:

- 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 structured data type[2] $\mathtt{type}$, an initial value $\mathtt{init}$ (if different from the default value of $\mathtt{type}$), and a period of access $\pi_{\mathtt{c}} \in \mathbb{N}_{>0}$. If $(\mathtt{c}, ., ., .)$ and $(\mathtt{c}', ., ., .)$ are two distinct communicator declarations then $\mathtt{c} \neq \mathtt{c}'$. The set of declared communicator names for a program $\mathtt{P}$ be $\mathtt{comms}(\mathtt{P})$ i.e. $\mathtt{comms}(\mathtt{P}) = \{\mathtt{c} | (\mathtt{c}, ., ., ., .) \in \mathtt{commdecl}(\mathtt{P})\}$. Given a communicator $\mathtt{c} \in \mathtt{comms}(\mathtt{P})$, the type $\mathtt{type}[\mathtt{c}]$ denotes the range of values the communicator can evaluate to and $\mathtt{init}[\mathtt{c}]$ denotes the initial value of the communicator. The evaluation of a communicator $\mathtt{val}[\mathtt{c}]$ is a function that maps $\mathtt{c}$ to a value in $\mathtt{type}[\mathtt{c}]$.
- a set of *module declarations* $\mathtt{moduledecl}$. A module declaration $(\mathtt{M}, \mathtt{portdecl}, \mathtt{taskdecl}, \mathtt{modedecl}, \mathtt{smode})$ 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 mode name $\mathtt{smode}$. If $(\mathtt{M}, ., ., ., .)$ and $(\mathtt{M}', ., ., ., .)$ are two distinct module declarations then $\mathtt{M} \neq \mathtt{M}'$. The set of declared module names for a program $\mathtt{P}$ be $\mathtt{modules}(\mathtt{P})$ i.e. $\mathtt{modules}(\mathtt{P}) = \{\mathtt{M} | (\mathtt{M}, ., ., ., .) \in \mathtt{moduledecl}(\mathtt{P})\}$.
  - a *port declaration* $(\mathtt{p}, \mathtt{type}, \mathtt{init})$ consists of a port name $\mathtt{p}$, a structured data type $\mathtt{type}$, and an initial value $\mathtt{init}$ (if different from the default value of $\mathtt{type}$). If $(\mathtt{p}, ., .)$ and $(\mathtt{p}', ., .)$ are two distinct port declarations then $\mathtt{p} \neq \mathtt{p}'$. The set of declared port names for a module $\mathtt{M}$ be $\mathtt{ports}(\mathtt{M})$ i.e. $\mathtt{ports}(\mathtt{M}) = \{\mathtt{p} | (\mathtt{p}, ., .) \in \mathtt{portdecl}(\mathtt{M})$. Given a port $\mathtt{p} \in \mathtt{ports}(\mathtt{M})$, the type $\mathtt{type}[\mathtt{p}]$ denotes the range of values the port can evaluate to and $\mathtt{init}[\mathtt{p}]$ denotes the initial value of the port. The evaluation of a port $\mathtt{val}[\mathtt{p}]$ is a function that maps $\mathtt{p}$ to a value in $\mathtt{type}[\mathtt{p}]$.
  - 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 list of formal input parameters is a data type; i.e. for all $1 \leq j \leq |\mathtt{filist}|$, $\mathtt{filist}(j) = \mathtt{type}$. Similarly an element of the list of formal output parameters is a data type; i.e. for all $1 \leq k \leq |\mathtt{folist}|$, $\mathtt{folist}(k) = \mathtt{type}$. If $(\mathtt{t}, ., ., .)$ and $(\mathtt{t}', ., ., .)$ are two distinct task declarations then $\mathtt{t} \neq \mathtt{t}'$. Let $\mathtt{val}[\mathtt{type}]$ denote the range of values for a particular data type $\mathtt{type}$. The function $\mathtt{fn}$ is defined as $\mathtt{fn} : \cup_i \mathtt{val}[\mathtt{filist}(i)] \rightarrow \cup_j \mathtt{val}[\mathtt{folist}(j)]$.
  - a *mode declaration* $(\mathtt{m}, \pi_{\mathtt{m}}, \mathtt{invocs}, \mathtt{switches}, \mathtt{refprog})$ 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 $\mathtt{refprog}$. If $(\mathtt{m}, ., ., ., .)$ and $(\mathtt{m}', ., ., ., .)$ are two distinct mode declarations then $\mathtt{m} \neq \mathtt{m}'$. The set of declared mode names for a module $\mathtt{M}$ is $\mathtt{modes}(\mathtt{M})$ i.e. $\mathtt{modes}(\mathtt{M}) = \{\mathtt{m} | (\mathtt{m}, ., ., .) \in \mathtt{modedecl}(\mathtt{M})$.
    - a *task invocation* $(\mathtt{t}, \mathtt{ailist}, \mathtt{aolist}, \mathtt{ptask})$ consists of a task name $\mathtt{t}$, 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 $\mathtt{ailist}$ (or $\mathtt{aolist}$) is either a port $\mathtt{p}$ or a pair $(\mathtt{c}, \mathtt{i})$ with a communicator name $\mathtt{c}$ and an instance

[2] Data type indicates common types like integer, float and boolean. More complex data types like arrays can be defined; however types are not an integral part of the program definition and a detailed discussion has been left out.

number $\mathtt{i} \in \mathbb{N}$. Task names of the invocations are unique i.e. if $(\mathtt{t},.,.,.)$ and $(\mathtt{t}',.,.,.)$ are two different task invocations then $\mathtt{t} \neq \mathtt{t}'$.

* a *mode switch* $(\mathtt{cnd},\mathtt{m})$ consists of a condition $\mathtt{cnd}$ (expressed as a predicate on ports and communicators) and a destination mode name $\mathtt{m} \in \mathtt{modes}(\mathtt{M})$. If $(\mathtt{cnd},.)$ and $(\mathtt{cnd}',.)$ are two distinct mode switches, then for all valuations of ports and communicators, either $\mathtt{cnd}$ evaluates to $\mathtt{false}$ or $\mathtt{cnd}'$ evaluates to $\mathtt{false}$ i.e. mode switches are deterministic. The set of destination mode names from a mode $\mathtt{m}$ be $\mathtt{destmodes}(\mathtt{m})$ i.e. $\mathtt{destmodes}(\mathtt{m}) = \{\mathtt{m} | (.,\mathtt{m}') \in \mathtt{switches}(\mathtt{m})\}$.

4.1 Definitions based on hierarchy

In the section we will relate components e.g. modules or programs or communicator accesses across levels of hierarchy in a HTL program.

Module types. A module $\mathtt{M}_n$ is a *sub-module* of a module $\mathtt{M}_1$ if there exists $n \in \mathbb{N}_{>1}$ modules $\mathtt{M}_1,\mathtt{M}_2,..,\mathtt{M}_n$ such that for every pair $\mathtt{M}_j,\mathtt{M}_{j+1}$ there exists a mode declaration $(\mathtt{m},.,.,.,\mathtt{P}) \in \mathtt{modedecl}(\mathtt{M}_j)$ and $\mathtt{M}_{j+1} \in \mathtt{modules}(\mathtt{P})$ for $1 \leq j < n$. The module $\mathtt{M}_1$ is a *super-module* of $\mathtt{M}_n$. A module is a sub-module (and a super-module) of itself. A *top-level module* is one with no super-module other than itself; a *leaf module* is one with no sub-module other than itself. A module $\mathtt{M}_2$ is an *immediate sub-module* of a module $\mathtt{M}_1$ if there exists a mode declaration $(\mathtt{m},.,.,.,\mathtt{P}) \in \mathtt{modedecl}(\mathtt{M}_1)$ and $\mathtt{M}_2 \in \mathtt{modules}(\mathtt{P})$. The module $\mathtt{M}_1$ is an *immediate super-module* of $\mathtt{M}_2$. An immediate sub (super) module is also a sub (super) module. The set of all the sub-modules of a module $\mathtt{M}$ is $\mathtt{submdl}(\mathtt{M})$. Module $\mathtt{M}$ is a *sibling* of module $\mathtt{M}'$ if $\mathtt{M},\mathtt{M}' \in \mathtt{modules}(\mathtt{P})$ for a program $\mathtt{P}$. The *sibling set* for module $\mathtt{M}$ is $\mathtt{sibset}(\mathtt{M}) = \mathtt{modules}(\mathtt{P}) \setminus \mathtt{M}$.

Program types. If $\mathtt{M}' \in \mathtt{modules}(\mathtt{P}')$, $\mathtt{M} \in \mathtt{modules}(\mathtt{P})$ and $\mathtt{M}'$ is a (immediate) sub-module of $\mathtt{M}$, then $\mathtt{P}'$ is a *(immediate) sub-program* of $\mathtt{P}$ and $\mathtt{P}$ is a *(immediate) super-program* of $\mathtt{P}'$. A program $\mathtt{P}$ is both a sub-program and a super-program to itself. A *top-level program* is one with no super-program than itself. A *leaf-level program* is one with no sub-program than itself. A *flat program* is one which is both a top-level and leaf-level program.
Abstract program $\mathtt{abs}(\mathtt{P})$ for program $\mathtt{P}$ is the top-level program with all immediate sub-programs removed, i.e. if $\mathtt{P} = (\mathtt{commdecl},\mathtt{moduledecl})$ then $\mathtt{abs}(\mathtt{P}) = (\mathtt{commdecl},\mathtt{moduledecl}')$ where $(\mathtt{M},\mathtt{portdecl},\mathtt{taskdecl},\mathtt{modedecl}', \mathtt{smode}) \in \mathtt{moduledecl}'$ if $(\mathtt{M},\mathtt{portdecl},\mathtt{taskdecl},\mathtt{modedecl},\mathtt{smode}) \in \mathtt{moduledecl}$ where mode declaration $(\mathtt{m},\mathtt{invocs},\mathtt{switches}) \in \mathtt{modedecl}'$ for every mode declaration $(\mathtt{m},\mathtt{invocs},\mathtt{switches},.) \in \mathtt{modedecl}$. An abstract program is always flat.

Mode types. Given a mode declaration $(\mathtt{m},.,.,\mathtt{P})$, mode $\mathtt{m}$ is *parent* of mode $\mathtt{m}'$ where $\mathtt{m}'$ is any mode in $\mathtt{P}$. Mode $\mathtt{m}_n$ is *transitive parent* of mode $\mathtt{m}_1$ if there exists $n \in \mathbb{N}_{>1}$ modes such that for every pair $\mathtt{m}_i,\mathtt{m}_{i+1}$, where $1 \leq i < n$, $\mathtt{m}_{i+1}$ is a parent of $\mathtt{m}_i$. A (transitive) parent mode $\mathtt{m}$ is a *top-level parent* if $\mathtt{m}$ is declared in a top-level program. *Ancestors* $\mathtt{ancestors}(\mathtt{m})$ of mode $\mathtt{m}$ is a set of modes that includes the parent modes of $\mathtt{m}$ and the ancestors of the parent modes; ancestors for modes of top level program is empty. The *start mode* $\mathtt{start}[\mathtt{M}]$ of a module $\mathtt{M}$ is the mode name in the module declaration i.e. $\mathtt{start}[\mathtt{M}] = \mathtt{smode}$ if $(\mathtt{M},.,.,.,\mathtt{smode})$ is the corresponding module declaration. The *start set* $\mathtt{startSet}(\mathtt{m})$ of a mode $\mathtt{m}$ is a set that includes mode $\mathtt{m}$ and start sets of the start modes of all modules in $\mathtt{P}$ i.e. $\mathtt{startSet}(\mathtt{m}) = \{\mathtt{m}\} \cup \{ \bigcup_{\mathtt{M}' \in \mathtt{modules}(\mathtt{P})} \mathtt{startSet}(\mathtt{start}(\mathtt{M}'))\}$ where $\mathtt{P}$ refines $\mathtt{m}$.

4.2 Communicators and ports access

Accessed by task invocations. *Input communicator set* $\texttt{icoms}(\texttt{inv},\texttt{m})$ for a task invocation $\texttt{inv} \in \texttt{invocs}(\texttt{m})$ is the set of communicators read by inv, *output communicator set* $\texttt{ocoms}(\texttt{inv},\texttt{m})$ is the set of communicators written by inv, *input port set* $\texttt{iprts}(\texttt{inv},\texttt{m})$ is the set of ports read by inv and *output port set* $\texttt{oprts}(\texttt{inv},\texttt{m})$ is the set of ports updated by inv. Formally,

- Input communicator set, $\texttt{icoms}(\texttt{inv},\texttt{m}) = \{\texttt{c} | \exists j \text{ s.t. } \texttt{ailist}[j] = (\texttt{c},.)\}$.
- Output communicator set, $\texttt{ocoms}(\texttt{inv},\texttt{m}) = \{\texttt{c} | \exists j \text{ s.t. } \texttt{aolist}[j] = (\texttt{c},.)\}$.
- Input port set $\texttt{iprts}(\texttt{inv},\texttt{m}) = \{\texttt{p} | \exists j \text{ s.t. } \texttt{ailist}[j] = \texttt{p}\}$.
- Output port set $\texttt{oprts}(\texttt{inv},\texttt{m}) = \{\texttt{p} | \exists j \text{ s.t. } \texttt{aolist}[j] = \texttt{p}\}$.

Accessed by switches. *Switch communicator set* $\texttt{scoms}(\texttt{sw},\texttt{m})$ for a switch $\texttt{sw} = (\texttt{cnd},.) \in \texttt{switches}(\texttt{m})$ is the set of communicators in predicate of switch condition cnd and *switch port set* $\texttt{sprts}(\texttt{sw},\texttt{m})$ is the set of ports in cnd. Formally,

- Switch communicator set $\texttt{scoms}(\texttt{sw},\texttt{m}) = \{\texttt{c}\}$ if there exists communicator c in the condition cnd.
- Switch port set $\texttt{sprts}(\texttt{sw},\texttt{m}) = \{\texttt{p}\}$ if there exists port p in the condition cnd.

Accessed by modules. An *accessible communicator set* acccommset for a module $\texttt{M} \in \texttt{modules}(\texttt{P})$ is the set of communicators declared by the super-programs of P. A *read set* $\texttt{readset}(\texttt{M})$ for a module M is the set of communicators read by task invocations and used by mode switches of modes in M. A *write set* $\texttt{writeset}(\texttt{M})$ for a module M is the set of communicators updated by task invocations of modes in M. A *hierarchical read set* ($\texttt{hierreadset}(\texttt{M})$) for a module M is the set of communicators that belongs both to the read-set and accessible communicator set of any sub-module of M. A *hierarchical write set* ($\texttt{hierwriteset}(\texttt{M})$) for a module M is the set of communicators that belongs both to the write set and accessible communicator set of any sub-module of M. Formally,

- Accessible communicator set $\texttt{acccommset}(\texttt{M}) = \{\texttt{c} | (\texttt{c},.,.,.) \in \texttt{commdecl}(\texttt{P}')\}$ where $\texttt{P}'$ is a super-program of P.
- Read set $\texttt{readset}(\texttt{M})$, $\{\texttt{c} | (\texttt{c} \in \texttt{icoms}(\texttt{inv},\texttt{m}) \text{ or } \texttt{c} \in \texttt{scoms}(\texttt{sw},\texttt{m})) \text{ and } \texttt{m} \in \texttt{modes}(\texttt{M})\}$.
- Write set $\texttt{writeset}(\texttt{M})$, $\{\texttt{c} | \texttt{c} \in \texttt{ocoms}(\texttt{inv},\texttt{m}) \text{ and } \texttt{m} \in \texttt{modes}(\texttt{M})\}$.
- Hierarchical read set, $\texttt{hierreadset}(\texttt{M}) = \bigcup_{\texttt{M}' \in \texttt{submdl}(\texttt{M})} (\texttt{readset}(\texttt{M}') \cap \texttt{acccommset}(\texttt{M}'))$.
- Hierarchical write set, $\texttt{hierwriteset}(\texttt{M}) = \bigcup_{\texttt{M}' \in \texttt{submdl}(\texttt{M})} (\texttt{writeset}(\texttt{M}') \cap \texttt{acccommset}(\texttt{M}'))$.

4.3 Definitions related to tasks

Declaration types. An *abstract task declaration* for a task t is a task declaration with no function definition i.e. of the form $(\texttt{t},.,.)$. A *concrete task declaration* is one with function definition i.e. of the form $(\texttt{t},.,.,\texttt{fn})$.

Relating dependencies. A binary relation $\texttt{prec}(\texttt{m})$ for mode m contains the dependency information of the tasks. A task invocation $\texttt{inv}_1$ *precedes* another task invocation $\texttt{inv}_n$ (or $(\texttt{inv}_1,\texttt{inv}_n) \in \texttt{prec}(\texttt{m})$) if there exists $n \in \mathbb{N}_{>1}$ different task invocations $\texttt{inv}_1,...,\texttt{inv}_n$ such that for each pair $\texttt{inv}_j$ and $\texttt{inv}_{j+1}$, $\texttt{oprts}(\texttt{inv}_j) \cap \texttt{iprts}(\texttt{inv}_{j+1}) \neq \phi$ where $1 \leq j < n$. The *preceding invocation set* $\texttt{prec}(\texttt{inv},\texttt{m})$ is the set of task invocations preceding inv in mode m i.e. $\texttt{prec}(\texttt{inv},\texttt{m}) = \{\texttt{inv}' | (\texttt{inv}',\texttt{inv}) \in \texttt{prec}(\texttt{m})$. The *following invocation set* $\texttt{foll}(\texttt{inv},\texttt{m})$ is the set of task invocations following inv in mode m i.e. $\texttt{foll}(\texttt{inv},\texttt{m}) = \{\texttt{inv}' | (\texttt{inv},\texttt{inv}') \in \texttt{prec}(\texttt{m})$. A task invocation $\texttt{inv}'$ *immediately precedes* an invocation inv if $\texttt{oprts}(\texttt{inv}',\texttt{m}) \cap \texttt{iprts}(\texttt{inv},\texttt{m}) \neq \phi$. The *immediately preceding set* $\texttt{immprec}(\texttt{inv},\texttt{m})$

is the set of task invocation that immediately precedes $\mathtt{inv}$.

Read/ write time. *Read time* $\mathtt{r}(\mathtt{inv},\mathtt{m})$ of a task invocation $\mathtt{inv} = (.,\mathtt{ailist},\mathtt{aolist},.)$ in a mode $\mathtt{m}$ is the latest communicator instance it reads from. *Write time* $\tau(\mathtt{inv},\mathtt{m})$ for a task invocation $\mathtt{inv}$ in mode $\mathtt{m}$ is the earliest communicator instance to which it writes to. Formally,
- Read time $\mathtt{r}(\mathtt{inv},\mathtt{m}) = \max_j(\pi_\mathtt{c} \cdot \mathtt{i})$ where $\mathtt{ailist}[j] = (\mathtt{c},\mathtt{i})$ and $(\mathtt{c},.,.,\pi_\mathtt{c})$ is the communicator declaration.
- Write time $\tau(\mathtt{inv},\mathtt{m}) = \min_k(\pi_\mathtt{c} \cdot \mathtt{i})$ where $\mathtt{aolist}[k] = (\mathtt{c},\mathtt{i})$ and $(\mathtt{c},.,.,\pi_\mathtt{c})$ is the communicator declaration.
For an invocation which does not read any communicator $\mathtt{r} = 0$ i.e. start of the mode period; for an invocation which does not write to any communicator $\tau = \pi[\mathtt{m}]$ i.e. end of the mode period.
Transitive read time $\mathtt{r}^*(\mathtt{inv},\mathtt{m})$ of a task invocation in a mode $\mathtt{m}$ is the latest communicator instance that the invocation or any of its preceding invocation reads from. *Transitive write time* $\tau^*(\mathtt{inv},\mathtt{m})$ of a task invocation is the earliest communicator instance that the invocation or any of its following invocation writes to.
- Transitive read time $\mathtt{r}^*(\mathtt{inv},\mathtt{m}) = \max(\mathtt{r}(\mathtt{inv},\mathtt{m}), \max_{\mathtt{inv}'}(\mathtt{r}^*(\mathtt{inv}',\mathtt{m})))$ where $(\mathtt{inv}',\mathtt{inv}) \in \mathtt{prec}(\mathtt{inv},\mathtt{m})$.
- Transitive write time $\tau^*(\mathtt{inv},\mathtt{m}) = \min(\tau(\mathtt{inv},\mathtt{m}), \min_{\mathtt{inv}'}(\tau^*(\mathtt{inv}',\mathtt{m})))$ where $(\mathtt{inv},\mathtt{inv}') \in \mathtt{prec}(\mathtt{m})$.
For an invocation with no preceding invocation, $\mathtt{r}^* = \mathtt{r}$. For an invocation with no following invocation, $\tau^* = \tau$.

Parent task. Task $\mathtt{t}_2$ is *parent* of task invocation $\mathtt{inv}_1 = (\mathtt{t}_1,.,.,\mathtt{t}_2) \in \mathtt{invocs}(\mathtt{m}_1)$ where $\mathtt{inv}_2 = (\mathtt{t}_2,.,.,.) \in \mathtt{invocs}(\mathtt{m}_2)$. Invocation $\mathtt{inv}_2$ is the parent invocation of $\mathtt{inv}_1$. A task $\mathtt{t}_{n+1}$ is an *n-th transitive parent* of task invocation $\mathtt{inv}_1 = (\mathtt{t}_1,.,.,\mathtt{t}_2) \in \mathtt{invocs}(\mathtt{m}_1)$ if there exists $n \in \mathbb{N}_{>1}$ modes $\mathtt{m}_1,..,\mathtt{m}_n$ such that for any two modes $\mathtt{m}_j,\mathtt{m}_{j+1}$, $(\mathtt{t}_j,.,.,\mathtt{t}_{j+1}) \in \mathtt{invocs}(\mathtt{m}_j)$ and $(\mathtt{t}_{j+1},.,.,.) \in \mathtt{invocs}(\mathtt{m}_{j+1})$, for all $1 \leq j < n$. The task invocation associated with $\mathtt{t}_{n+1}$ is the n-th transitive parent invocation of $\mathtt{inv}_1$. A parent task is also a 1-st transitive parent. A task $\mathtt{ptask}$ is a *top-level parent* of a task invocation $(\mathtt{t},.,.,\mathtt{ptask})$ if $(\mathtt{ptask},.,.) \in \mathtt{invocs}(\mathtt{m})$ where $\mathtt{m}$ is one of the modes of top-level modules and $\mathtt{ptask}$ is m-th transitive parent of $\mathtt{t}$ for some $m \in \mathbb{N}$. Transitive parent and top-level parent has no definition for task invocations in modes of top-level modules.

Task interface. For each task $\mathtt{t}$ we will consider a set of local input (output) variables, each with a data type implying the range of values it can store. At termination the local output variables are updated with the evaluation of task (specified by the function in task declaration) on the values of local input variable (at the instance of task release). The local input (output) variable for a port $\mathtt{p}$ being read (written) is denoted as $\mathtt{liv}^\mathtt{t}(\mathtt{p})$ ($\mathtt{lov}^\mathtt{t}(\mathtt{p})$). The local input (output) variable for $\mathtt{i}$-th instance of a $\mathtt{c}$ being read (written) is $\mathtt{liv}^\mathtt{t}(\mathtt{c},\mathtt{i})$ ($\mathtt{lov}^\mathtt{t}(\mathtt{c},\mathtt{i})$). The value of a local input (output) variable is denoted by $\mathtt{val}(\cdot)$.

4.4 Definitions related to distribution

Host map. Given an HTL program $\mathtt{P}$ and a set of hosts $\mathtt{hset}$, *host map* is a map from hosts in $\mathtt{hset}$ to top-level modules of $\mathtt{P}$ and denotes the distribution of $\mathtt{P}$; $\mathtt{hmap}(\mathtt{h})$ is the set of top-level modules mapped to host $\mathtt{h}$.

Partial program. Given a top-level program $\mathtt{P} = (\mathtt{commdecl},\mathtt{moduledecl})$ and a host map, *partial program* $\mathtt{P}_\mathtt{h}$ for a host $\mathtt{h}$ is the set of communicators and top-level modules of $\mathtt{P}$ mapped to $\mathtt{h}$; formally $\mathtt{P}_\mathtt{h} = (\mathtt{commdecl},\mathtt{moduledecl}')$ where for every $(\mathtt{M},\mathtt{moduledecl},\mathtt{portdecl},\mathtt{modedecl},\mathtt{smode}) \in \mathtt{moduledecl}'$, there is a module declaration $(\mathtt{M},\mathtt{moduledecl},\mathtt{portdecl},\mathtt{modedecl},\mathtt{smode}) \in \mathtt{moduledecl}$ and $\mathtt{M} \in \mathtt{hmap}(\mathtt{h})$.

Worst case mapping. *WCET (WCTT) map*, $\mathtt{wemap}_\mathtt{h}$ ($\mathtt{wtmap}_\mathtt{h}$), is a mapping from tasks to worst-case-execution

(transmission) times (relative to host $\mathtt{h}$); $\mathtt{wemap_h(t)}$ ($\mathtt{wtmap_h(t)}$) denotes the wcet (wctt) for task $\mathtt{t}$ on host $\mathtt{h}$. For a task invocation $\mathtt{inv} = (\mathtt{t},.,.,.)$, $\mathtt{wcet(inv)} = \mathtt{wemap_h(t)}$ and $\mathtt{wctt(inv)} = \mathtt{wtmap_h(t)}$.

Task interface. Communicators of top-level program are shared across all hosts. Each host maintain a local output variable for tasks writing to these communicators. If task $\mathtt{t}$ (executing on host $\mathtt{h}_i$) writes to a communicator $\mathtt{c}$ of the top-level program, then a local output variable $\mathtt{liv^t_{h_j}[c,i]}$ is maintained for the task on all hosts $\mathtt{h}_j$ other than host $\mathtt{h}_i$. On completion of execution of $\mathtt{t}$ on $\mathtt{h}_i$, the output is transmitted to host $\mathtt{h}_j$ and stored in $\mathtt{liv^t_{h_j}[c,i]}$. When communicator write is due, $\mathtt{c}$ is updated from the local variable.

4.5 Well-formed HTL program.

A HTL program is *well-formed* if it conforms to the following restrictions on program, communicators, task invocations and refinements.

Constraints on programs.

C1.1 There is only one top-level program.

C1.2 For each program (other than top-level program) there is only one immediate super-program.

C1.3 For each module (other than top-level module) there is only one immediate super-module.

C1.4 A program cannot refine more than one mode of a module i.e. if there exists two mode declarations $(\mathtt{m}_1,.,.,\mathtt{P}_1)$ and $(\mathtt{m}_2,.,.,\mathtt{P}_2)$ where $\mathtt{m}_1,\mathtt{m}_2 \in \mathtt{modes(M)}$ then $\mathtt{P}_1 \neq \mathtt{P}_2$.

C1.5 The start mode of a module should belong to the mode set of a module i.e. for a module declaration $(\mathtt{M},.,.,.,\mathtt{smode})$, $\mathtt{smode} \in \mathtt{modes(M)}$.

C1.6 The set of destination modes from mode switches should be from the set of modes of the corresponding module i.e. if $\mathtt{m} \in \mathtt{modes(M)}$ then $\mathtt{destmodes(m)} \in \mathtt{modes(M)}$.

Constraints on communicators.

C2.1 If a communicator has been declared in program $\mathtt{P}$ then it cannot be redeclared in any sub-program other than $\mathtt{P}$ i.e. if $(\mathtt{c},.,.,.) \in \mathtt{commdecl(P)}$ then $(\mathtt{c},.,.,.) \notin \mathtt{commdecl(P')}$ for all sub-program $\mathtt{P'}$ of $\mathtt{P}$ other than $\mathtt{P}$ itself.

C2.2 If a communicator is accessed by a task invocation or switch in mode of $\mathtt{M}$ (in program $\mathtt{P}$) then the communicator must be declared in one of the super-programs of $\mathtt{P}$; i.e. read and write set should be subset of accessible communicator set ($\mathtt{readset(M)} \subseteq \mathtt{acccommset(M)}$ and $\mathtt{writeset(M)} \subseteq \mathtt{acccommset(M)}$).

C2.3 If a communicator $\mathtt{c}$ is (hierarchically) written by a module $\mathtt{M}$ then none of the sibling modules can (hierarchically) write to $\mathtt{c}$, i.e. if $\mathtt{c} \in \mathtt{hierwriteset(M)}$, then for all modules $\mathtt{M'} \in \mathtt{sibset(M)}$, $\mathtt{c} \notin \mathtt{hierwriteset(M')}$.

Constraints on task invocations.

C3.1 For a task invocation $\texttt{inv}$ in mode $\texttt{m}$ read time should be earlier than write time, $\texttt{r}(\texttt{inv},\texttt{m}) < \tau(\texttt{inv},\texttt{m})$.

C3.2 For a task invocation $\texttt{inv}$ in mode $\texttt{m}$ transitive read time should be earlier than transitive write time, $\texttt{r}^*(\texttt{inv},\texttt{m}) < \tau^*(\texttt{inv},\texttt{m})$.

C3.3 Precedences between tasks should be acyclic $(\texttt{inv}_i,\texttt{inv}_j) \in \texttt{prec}(\texttt{m})$, $\texttt{inv}_i \neq \texttt{inv}_j$.

C3.4 If a task invocation $\texttt{inv}$ (in a mode $\texttt{m}$) reads or writes a port, the port must be declared in module $\texttt{M}$, where $\texttt{m} \in \texttt{modes}(\texttt{M})$; formally, if $\texttt{p} \in \texttt{iprts}(\texttt{inv})$ (or in $\texttt{oprts}(\texttt{inv})$) there must be declaration $(\texttt{p},.,.) \in \texttt{portdecl}(\texttt{M})$.

C3.5 Two task invocations $\texttt{inv} = (.,.,\texttt{aolist},.)$ and $\texttt{inv}' = (.,.,\texttt{aolist}',.)$ of a mode $\texttt{m}$ cannot write to the same port or to same instance of a communicator; i.e. $\texttt{oprts}(\texttt{inv},\texttt{m}) \cap \texttt{oprts}(\texttt{inv}',\texttt{m}) = \phi$ and if $(\texttt{c},\texttt{i}) \in \texttt{aolist}$ then $(\texttt{c},\texttt{i}) \notin \texttt{aolist}'$.

C3.6 A task can be invoked in a mode if it has a declaration in the corresponding module and the size of the input (output) parameter list for the invocation is of the same size as that of the declaration; if $(\texttt{t},\texttt{ailist},\texttt{aolist},.) \in \texttt{invocs}(\texttt{m})$ and $\texttt{m} \in \texttt{modes}(\texttt{M})$ then $(\texttt{t},\texttt{filist},\texttt{folist},.) \in \texttt{taskdecl}(\texttt{M})$ with $|\texttt{ailist}| = |\texttt{filist}|$ and $|\texttt{aolist}| = |\texttt{folist}|$.

If the j-th element of the input (output) list $\texttt{ailist}$ ($\texttt{aolist}$) is a communicator-instance pair $(\texttt{c},\texttt{i})$ and the corresponding communicator declaration is $(\texttt{c},\texttt{type},.,\pi_\texttt{c})$ then the following should hold: (1) mode period is multiple of communicator access period i.e. $\text{mod}(\frac{\pi_\texttt{m}}{\pi_\texttt{c}}) = 0$, (2) task invocation cannot read from an instance corresponding to the end of the period i.e. $0 \leq \texttt{i} < \frac{\pi_\texttt{m}}{\pi_\texttt{c}}$ (similarly it cannot write to an communicator instance at the start of the period i.e. $0 < \texttt{i} \leq \frac{\pi_\texttt{m}}{\pi_\texttt{c}}$) and (3) type of the communicator should match the corresponding element of the formal input (output) list i.e. $\texttt{filist}[j] = \texttt{type}$ ($\texttt{folist}[j] = \texttt{type}$). A task invocation cannot write to the same instance of a communicator more than once i.e. $\nexists i,k$ s.t. $\texttt{aolist}[i] = \texttt{aolist}[k] = (\texttt{c},\texttt{i})$.

If the j-th element of input (output) list of a task invocation is a port then the j-th element of the input (output) list of the corresponding task declaration should be the same type as the port; i.e. if $\texttt{ailist}[j] = \texttt{p}$ ($\texttt{aolist}[j] = \texttt{p}$) then $\texttt{filist}[j] = \texttt{type}$ ($\texttt{folist}[j] = \texttt{type}$) where $(\texttt{p},\texttt{type},.)$ is the corresponding port declaration. A task invocation cannot write to the same port more than once i.e. $\nexists i,k$ such that $\texttt{aolist}[i] = \texttt{aolist}[k] = \texttt{p}$.

Constraints on refinement.

C4.1 Period of mode $\texttt{m}$ and all modes in program $\texttt{P}$ refining $\texttt{m}$ should be identical; formally if there is a mode declaration $(\texttt{m},\pi_\texttt{m},.,.,\texttt{P})$ then for all mode declarations $(\texttt{m}',\pi_\texttt{m}{}',.,.,.) \in \texttt{modedecl}(\texttt{M})$, $\pi_\texttt{m}{}' = \pi_\texttt{m}$ where $\texttt{M} \in \texttt{modules}(\texttt{P})$. Mode switches of a program being checked in top-down way, the constraint ensures that there is no unsafe termination of tasks in refinement modes.

C4.2 Every task invocation of a mode $\texttt{m}$ in a module $\texttt{M}$ other than the top-level modules should have a parent task; the parent task should have an abstract declaration in the immediate super-module and should be invoked in the parent of $\texttt{m}$. Formally, a task invocation should be of the form $\texttt{inv} = (\texttt{t},.,.,\texttt{ptask}) \in \texttt{invocs}(\texttt{m})$ where $\texttt{m} \in \texttt{modes}(\texttt{M})$ (and $\texttt{M}$ is not top-level module). The parent task should have an abstract declaration i.e. $(\texttt{ptask},.,.) \in \texttt{taskdecl}(\texttt{M}')$

and $\texttt{inv}_p = (\texttt{ptask}, ., ., .) \in \texttt{invocs}(\texttt{m}')$ where $\texttt{m}'$ is parent mode of $\texttt{m}$ and $\texttt{M}'$ is immediate super module of $\texttt{M}$. The constraint ensures that the parent task is not executed during the execution of the program but acts as placeholder for the children during program analysis.

C4.3 A task invocation (in a mode of a module $\texttt{M}$) should have an unique parent task relative to all task invocations in the same mode and to task invocations in modes of sibling modules of $\texttt{M}$. Formally if $\texttt{t}_p$ (in $\texttt{m}_p$) is the parent task invocation for $\texttt{inv}$ (in $\texttt{m}$ of module $\texttt{M}$) then no other task invocation of mode $\texttt{m}$ or any mode $\texttt{m}'$ (where $\texttt{m}' \in \texttt{modes}(\texttt{M}')$ and $\texttt{M}'$ is a sibling module of $\texttt{M}$) should have $\texttt{t}_p$ as parent. The constraint ensures that all tasks that can potentially execute in parallel have an unique top-level parent.

C4.4 If $\texttt{inv}'$ is the parent task invocation of $\texttt{inv}$ then the read time of $\texttt{inv}$ should be no later than that of $\texttt{inv}'$ and the write time of $\texttt{inv}$ should be no earlier than that of $\texttt{inv}'$ i.e. $\texttt{r}(\texttt{inv},\texttt{m}) \leq \texttt{r}(\texttt{inv}',\texttt{m}')$ and $\tau(\texttt{inv},\texttt{m}) \geq \tau(\texttt{inv}',\texttt{m}')$ where $\texttt{inv} \in \texttt{invocs}(\texttt{m})$ and $\texttt{m}'$ is the parent mode of $\texttt{m}$. The constraint ensures that the parent invocation is more constraint in time than child task.

C4.5 Every relation in precedence set of a mode $\texttt{m}$ should be preserved in the parent mode $\texttt{m}'$; i.e. for all pairs of task invocations $(\texttt{inv}_1, \texttt{inv}_2) \in \texttt{prec}(\texttt{m})$, there should be $(\texttt{inv}'_1, \texttt{inv}'_2) \in \texttt{prec}(\texttt{m}')$ where $\texttt{inv}'_1$ and $\texttt{inv}'_2$ are parent task invocations of $\texttt{inv}_1$ and $\texttt{inv}_2$. The constraint ensures that the parent task invocation is more constrained in dependencies than child invocation.

4.6 Well-timed HTL program

The notion of well-formedness of a program is independent of the run-time system. To ensure that schedulability analysis can be performed only on the top-level program, a task (in refinement) should use less resources than its parent task. An *well-formed* HTL program is *well-timed* if wcet and wctt of task invocation is not greater than the wcet and wctt of the parent task invocation i.e. $\texttt{wemap}_\texttt{h}(\texttt{inv}) \leq \texttt{wemap}_\texttt{h}(\texttt{inv}')$ and $\texttt{wtmap}_\texttt{h}(\texttt{inv}) \leq \texttt{wtmap}_\texttt{h}(\texttt{inv}')$ where $\texttt{inv}'$ is the parent invocation of $\texttt{inv}$ and they run on host $\texttt{h}$. The constrained ensures that resources used by a parent invocation is at least same as that of the child invocation.

4.7 Claims on well-formed HTL program

Claim 1 *Parent mode of a mode* $\texttt{m}$ *is unique.* The claim can be proved by contradiction. Consider a mode $\texttt{m} \in \texttt{modes}(\texttt{M})$ where $\texttt{M} \in \texttt{modules}(\texttt{P})$. Assume there are two mode declarations $(\texttt{m}_1, ., ., \texttt{P})$ and $(\texttt{m}_2, ., ., \texttt{P})$ with $\texttt{m}_1 \neq \texttt{m}_2$ i.e. program $\texttt{P}$ refines both $\texttt{m}_1$ and $\texttt{m}_2$. Modes $\texttt{m}_1$ and $\texttt{m}_2$ cannot be in different programs as then $\texttt{P}$ be would have more than one immediate super program (constraint C1.2). Modes $\texttt{m}_1$ and $\texttt{m}_2$ cannot be in different modules of same program as then $\texttt{M}$ would have more than one immediate super module (constraint C1.3). If modes $\texttt{m}_1$ and $\texttt{m}_2$ are in the same module then $\texttt{P}$ cannot refine both the modes (constraint C1.3). Hence the initial assumption cannot hold. It can be similarly proved that top-level parent of a mode is unique.

Claim 2 *Every task invocation other than in the top-level program has a top-level parent.* Consider a task invocation $\texttt{inv} \in \texttt{invocs}(\texttt{m})$; if $\texttt{m}$ is a mode in a refinement program then $\texttt{inv}$ has a parent in parent mode of $\texttt{m}$ (constraint C4.2).

If parent of $\mathtt{m}$, $\mathtt{m}_p$ is in top-level module then the claim holds. Otherwise the claim can be proved by induction. Let j-th ($j \in \mathbb{N}_{>0}$) transitive parent $\mathtt{inv}'$ of $\mathtt{inv}$ is in $\mathtt{m}'$; $\mathtt{inv}'$ must have a top-level parent (inductive assumption) which is also a top-level parent for $\mathtt{inv}$ (definition).

Claim 3 *j-th transitive parents for all task invocations in a mode* $\mathtt{m}_1$ *belongs to the same mode* $\mathtt{m}_n$ *for some* $j \in \mathbb{N}_{>0}$. The proof is by induction. Parent mode $\mathtt{m}_2$ of a mode $\mathtt{m}_1$ is unique (claim 1); hence (1-st transitive) parent of all task invocations belong to $\mathtt{m}_2$ (constraint C4.2). This is the base case. Consider m-th transitive parents of task invocations belongs to $\mathtt{m}_j$ (inductive assumption). The parent mode of $\mathtt{m}_j$ is unique i.e. parents of all task invocation in $\mathtt{m}_j$ belongs to the same mode; these are the $j+1$-th transitive parents of invocations in $\mathtt{m}_1$ (definition). Hence $j+1$-th transitive parents of task invocations in $\mathtt{m}_1$ belong to an unique mode.

Claim 4 *Top-level parents for all task invocations in a mode* $\mathtt{m}_1$ *belongs to the same mode* $\mathtt{m}_n$ *in top-level program.* From claim 2, every task invocation in a refinement program has a top-level parent which is also j-th transitive parent for some $j \in \mathbb{N}_{>0}$ (definition) and j-th transitive parents for all task invocations in a mode $\mathtt{m}_1$ belongs to the same mode (claim 3).

Claim 5 *Every task invocation has a unique top-level parent relative to all task invocations that can be invoked in parallel.* Let there be a task invocation $\mathtt{inv} = (\mathtt{t}, ., ., \mathtt{ptask}) \in \mathtt{invocs}(\mathtt{m})$ where $\mathtt{m} \in \mathtt{modes}(\mathtt{M})$ ($\mathtt{M}$ is not a top-level module) and the top-level parent task be $\mathtt{inv}' = (\mathtt{t}', ., .) \in \mathtt{invocs}(\mathtt{m}')$ where $\mathtt{m}' \in \mathtt{modes}(\mathtt{M}')$ and $\mathtt{M}'$ is a top-level module. Let $\mathtt{P}'$ refines $\mathtt{m}'$. Consider a task invocation $\mathtt{inv}'' = (\mathtt{t}'', ., ., \mathtt{ptask}'') \in \mathtt{invocs}(\mathtt{m}'')$ where $\mathtt{m}'' \in \mathtt{modes}(\mathtt{M}'')$ and $\mathtt{M}''$ is not a top-level module. We will prove $\mathtt{inv}$ and $\mathtt{inv}''$ have either different top-level parents or they do not execute in parallel (or they are identical).

Both $\mathtt{M}$ and $\mathtt{M}''$ has to be sub-modules of $\mathtt{M}'$; otherwise $\mathtt{t}'$ cannot be parent for $\mathtt{t}''$ (from program structure). Both $\mathtt{M}$ and $\mathtt{M}''$ should be sub-modules of modules in $\mathtt{P}'$; otherwise $\mathtt{M}''$ cannot execute in parallel.

If $\mathtt{M}, \mathtt{M}'' \in \mathtt{modules}(\mathtt{P}')$ we have the following cases:
(i) $\mathtt{M} \neq \mathtt{M}''$ then $\mathtt{inv}$ and $\mathtt{inv}''$ have different parent task (from constraint 4.3) which are also the top-level parents.
(ii) $\mathtt{M} = \mathtt{M}''$ but $\mathtt{m} \neq \mathtt{m}''$ then $\mathtt{inv}$ and $\mathtt{inv}''$ cannot be invoked in parallel.
(iii) $\mathtt{M} = \mathtt{M}''$ and $\mathtt{m} = \mathtt{m}'$ but $\mathtt{t} \neq \mathtt{t}''$. Then task invocations $\mathtt{inv}$ and $\mathtt{inv}''$ should have different parent tasks in $\mathtt{m}'$ (from constraint 4.3) which are also the top-level parents.
(iv) $\mathtt{M} = \mathtt{M}''$, $\mathtt{m} = \mathtt{m}'$ and $\mathtt{t} = \mathtt{t}''$. then the invocations are identical (there cannot be two invocations with identical tasks).
If $\mathtt{P}'$ is leaf-level program then no further analysis is required.

If
- $\mathtt{M} \in \mathtt{modules}(\mathtt{P}')$ and $\mathtt{M}''$ is a sub-program for any sibling module $\mathtt{M}^*$ of $\mathtt{M}$, then there exists some integer n such that n-th transitive parent of $\mathtt{inv}''$ is invoked in some mode of $\mathtt{M}^*$
- $\mathtt{M}'' \in \mathtt{modules}(\mathtt{P}')$ and $\mathtt{M}$ is a sub-module for any sibling module $\mathtt{M}^*$ of $\mathtt{M}''$ then there exists some integer m such that m-th transitive parent of $\mathtt{inv}$ is invoked in some mode of $\mathtt{M}^*$
- $\mathtt{M}$ and $\mathtt{M}''$ are sub-modules of different modules of $\mathtt{P}'$, then there exists some integers m, n such that m-th and n-th transitive parents of $\mathtt{inv}$ and $\mathtt{inv}''$ respectively are invoked in different modules of $\mathtt{P}'$.

All of the above three situations are special instances of case (i) analyzed earlier and thus the top-level parent must be different for the two task invocations.

If $\mathtt{M} \in \mathtt{modules}(\mathtt{P}')$ and $\mathtt{M}''$ is a sub-module of $\mathtt{M}$ (other than $\mathtt{M}$) we have:
- there exists integer m such that $\mathtt{inv}$ is m-th transitive parent of $\mathtt{inv}''$
- there exists integer m such that $\mathtt{inv}_i \in \mathtt{invocs}(\mathtt{m})$ is the m-th transitive parent of $\mathtt{inv}''$.
In first case, $\mathtt{t}$ should have an abstract declaration and does not get executed. In second case, $\mathtt{inv}$ and $\mathtt{inv}_i$ should have different parents in $\mathtt{m}'$ (which are also their top-level parents). This in turn implies different top-level parent of $\mathtt{inv}$ and $\mathtt{inv}''$.

The case where $\mathtt{M}'' \in \mathtt{modules}(\mathtt{P}')$ and $\mathtt{M}$ is a sub-module of $\mathtt{M}''$ (other than $\mathtt{M}''$) has a symmetric analysis to the last one (by interchanging $\mathtt{M}$ and $\mathtt{M}''$).

The last analysis deals with both $\mathtt{M}$ and $\mathtt{M}''$ being sub-module of a module $\mathtt{M}_i$ in $\mathtt{P}'$. Both the modules should belong to refinement program $\mathtt{P}_i$ of a mode $\mathtt{m}_i$ in $\mathtt{M}_i$ (otherwise the tasks cannot be invoked in parallel). The subsequent analysis can be done in a similar way we did for $\mathtt{P}'$ (by replacing $\mathtt{P}'$ with $\mathtt{P}_i$).

Claim 6 *If* $\mathtt{inv}'$ *is the parent task invocation of* $\mathtt{inv}$ *then the transitive read time of* $\mathtt{inv}$ *should be no later than that of* $\mathtt{inv}'$ *and the transitive write time of* $\mathtt{inv}$ *should be no earlier than that of* $\mathtt{inv}'$ *i.e.* $\mathtt{r}^*(\mathtt{inv},\mathtt{m}) \leq \mathtt{r}^*(\mathtt{inv}',\mathtt{m}')$ *and* $\tau^*(\mathtt{inv},\mathtt{m}) \geq \tau^*(\mathtt{inv}',\mathtt{m}')$ *where* $\mathtt{inv} \in \mathtt{invocs}(\mathtt{m})$ *and* $\mathtt{m}'$ *is the parent mode of* $\mathtt{m}$*.*

From well-formedness criterion, period of $\mathtt{m}$ and $\mathtt{m}'$ are identical and modes switches at period boundaries; hence matching a period of $\mathtt{m}$ with that of $\mathtt{m}'$ is sufficient for the claim. The read time, transitive read time, write time and transitive write time for $\mathtt{inv}$ be $\mathtt{r}$, $\mathtt{r}^*$, τ and τ^* respectively. The read time, transitive read time, write time and transitive write time for $\mathtt{inv}'$ be $\mathtt{r}'$, $\mathtt{r}'^*$, τ' and τ'^* respectively. By induction we will show $\mathtt{r}^* \leq \mathtt{r}'^*$ and $\tau^* \geq \tau'^*$.

Release time: From definitions, $\mathtt{r}^* = \max(\mathtt{r}, \max_i(\mathtt{r}_i^*))$ where $\mathtt{inv}_i \in \mathtt{prec}(\mathtt{inv},\mathtt{m})$ and $\mathtt{r}_i^*$ is the transitive release time of $\mathtt{inv}_i$. Again, $\mathtt{r}'^* = \max(\mathtt{r}', \max_k(\mathtt{r}_k'^*))$ where $\mathtt{r}_k'^*$ is the transitive release time of $\mathtt{inv}_k'$ and $\mathtt{inv}_k' \in \mathtt{prec}(\mathtt{inv}',\mathtt{m}')$. From well-formedness constraints precedences of $\mathtt{m}$ are contained in $\mathtt{m}'$. So parents of the set $\mathtt{prec}(\mathtt{inv},\mathtt{m})$ should be a subset of $\mathtt{prec}(\mathtt{inv}',\mathtt{m}')$. If $\mathtt{inv}_i'$ (in the set $\mathtt{prec}(\mathtt{inv}',\mathtt{m}')$) is the parent of $\mathtt{inv}_i$ (in the set $\mathtt{prec}(\mathtt{inv},\mathtt{m})$) then from inductive assumption $\mathtt{r}_i^* \leq \mathtt{r}_i'^*$. This implies $\max_i(\mathtt{r}_i^*) \leq \max_k(\mathtt{r}_k'^*)$ in the above definitions. We have $\mathtt{r} \leq \mathtt{r}'$ from well-formedness constraints. From the last two conditions we have $\mathtt{r}^* \leq \mathtt{r}'^*$.

Termination time: From definitions, $\tau^* = \min(\tau, \min_i(\tau_i^*))$ where $\mathtt{inv}_i \in \mathtt{foll}(\mathtt{inv},\mathtt{m})$ and τ_i^* is the transitive termination time of $\mathtt{inv}_i$. Again, $\tau'^* = \min(\tau', \min_k(\tau_k'^*))$ where $\tau_k'^*$ is the transitive termination time of $\mathtt{inv}_k'$ and $\mathtt{inv}_k' \in \mathtt{foll}(\mathtt{inv}',\mathtt{m}')$. From well-formedness constraints, precedences of $\mathtt{m}$ are contained in $\mathtt{m}'$. So parents of the set $\mathtt{foll}(\mathtt{inv},\mathtt{m})$ should be a subset of $\mathtt{foll}(\mathtt{inv}',\mathtt{m}')$. If $\mathtt{inv}_i'$ (in the set $\mathtt{prec}(\mathtt{inv}',\mathtt{m}')$) is the parent of $\mathtt{inv}_i$ (in the set $\mathtt{prec}(\mathtt{inv},\mathtt{m})$) then from inductive assumption $\tau_i^* \geq \tau_i'^*$. This implies $\min_i(\tau_i^*) \geq \min_k(\tau_k'^*)$ in the above definitions. We have $\tau \geq \tau'$ from well-formedness constraints. From the last two conditions we have $\tau^* \geq \tau'^*$.

Base Case: If task invocation $\mathtt{inv}$ does not follow any task $\mathtt{r}^* = \mathtt{r}$. For parent task invocation $\mathtt{inv}'$: $\mathtt{r}'^* = \max(\mathtt{r}', \cdot)$. From constraints $\mathtt{r} \leq \mathtt{r}'$. Hence $\mathtt{r}^* \leq \mathtt{r}'^*$. If $\mathtt{inv}$ does not precede any task $\tau^* = \tau$. For parent task invocation $\mathtt{inv}'$: $\tau'^* = \min(\tau', \cdot)$. From constraints $\tau \geq \tau'$. Hence $\tau^* \geq \tau'^*$.

5 Operational Semantics

The execution of a TSL program yields a (possibly infinite) sequence of configurations. Each configuration consists of values of all program variables (ports and communicators), a set of triggers (where triggers are of type write, switch, read or release), and a set of released tasks. A trigger defines an action to be taken at an event which is specified as a combination of time ticks and a set of completion events (of tasks). An event is enabled when the number of time ticks to wait is zero and all the completion events have occurred. When a trigger is handled, action associated with the trigger is carried out. An action may be communicator write (handled by write triggers), communicator read (handled by read triggers), task release (handled by release triggers) or mode switch check and subsequent invocation of modes (handled by switch triggers). A configuration is waiting if there are no enabled triggers in trigger set; any other configuration is non-waiting. A time tick or a completion event is handled only if a configuration is non-waiting. For a non-waiting configuration there are four possible transitions: write, switch, read and release transition. A write transition occurs if there is at least one enabled write trigger. A switch transition occurs if no write trigger is enabled and at least one switch trigger is enabled. A read transition occurs if no write or switch triggers are enabled and at least one read trigger is enabled. A release transition occurs if no write, switch or read triggers are enabled and at least one release trigger is enabled.

Configuration. Let P[3] be an HTL program distributed on host set $\mathtt{hset}$. The execution trace of P is a (possibly infinite) sequence of configuration. A *configuration* u is a tuple $(\mathtt{statecol}, \mathtt{trgscol}, \mathtt{taskscol})$ where variable state collection $\mathtt{statecol}$ is a set of variable states for each host, trigger set collection $\mathtt{trgscol}$ is a set of trigger sets for each host and task set collection $\mathtt{taskscol}$ is a set of task sets for each host. Formally, $\mathtt{statecol} = \{\mathtt{state}_1, ., \mathtt{state}_i, .\mathtt{state}_{|\mathtt{hset}|}\}$ where $\mathtt{state}_i$ is a function from communicators and ports to values for host $\mathtt{h}_i$; $\mathtt{trgscol} = \{\mathtt{trgs}_1, ., \mathtt{trgs}_i, .\mathtt{trgs}_{|\mathtt{hset}|}\}$ where $\mathtt{trgs}_i$ is set of triggers for host $\mathtt{h}_i$ and $\mathtt{taskscol} = \{\mathtt{tasks}_1, ., \mathtt{tasks}_i, .\mathtt{tasks}_{|\mathtt{hset}|}\}$ where $\mathtt{tasks}_i$ is a set of released tasks for host $\mathtt{h}_i$. Given a configuration u and a host $\mathtt{h}_i$, the variable state, trigger set and task set for $\mathtt{h}_i$ is denoted as $\mathtt{state}_i(\mathtt{u})$, $\mathtt{trgs}_i(\mathtt{u})$ and $\mathtt{tasks}_i(\mathtt{u})$.

Variable state. A *variable state* $\mathtt{state}_i$ (for host $\mathtt{h}_i$) is a valuation of communicators and ports (accessed by the partial program $\mathtt{P}_{\mathtt{h}_i}$ on host $\mathtt{h}_i$) to values. The set of communicators consists of those accessed by the sub-modules of the top-level modules (mapped to h) i.e. $\bigcup_{\mathtt{M} \in \mathtt{hmap(h)}} (\mathtt{hierreadset(M)} \cup \mathtt{hierwriteset(M)})$. The set of ports consists of those used by sub-modules of top-level modules (mapped to h) i.e. $\bigcup_{\mathtt{M} \in \mathtt{hmap(h)}} \bigcup_{\mathtt{M}' \in \mathtt{submdl(M)}} \mathtt{ports}(\mathtt{M}')$.

Events. An *event* is an interrupt raised by the host on which the program executes. We will consider two type of events: time tick event and task completion event. The *time tick event* is bound to the system clock. The resolution of the clock is assumed to be the highest common factor of all communicator and mode periods. The clocks of all hosts are assumed to be in synch. The *task completion event* is generated internally by the program and are similar to software interrupts. To generate a completion event the program bind it to a specific task invocation at run-time. When the task terminates, a completion event is raised. For a task t we will denote the completion event as *compl*(t). A completion event can occur simultaneously with a time tick event; however for any host only one completion event can occur at any instance. Formally, an event e is a pair (τ, υ) where τ is a tag and υ is a value assigned to the event. For time tick event, $\tau \in \mathbb{N}$ and the value is $\mathtt{true}$ (implying that the clock is consistent and present at uniformly separated points). For completion event $\tau \in \mathbb{R}$ and υ is a set of task names $\mathtt{t}_i$ (implying the completion of task $\mathtt{t}_i$) for host $\mathtt{h}_i$.

[3] We will assume all task invocations have unique task names i.e. a task name uniquely identifies an invocation; let task invocation for a task t be inv.

Trigger. A *trigger* $\mathtt{g}$ is a tuple $(\mathtt{gtyp},\mathtt{e},\mathtt{a})$ where $\mathtt{gtyp} \in \{\mathtt{w},\mathtt{s},\mathtt{d},\mathtt{r}\}$ denotes write, switch, read and release trigger 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 $(\mathtt{n},\mathtt{complete})$ where $\mathtt{n} \in \mathbb{N}_{\geq 0}$ represents number of time tick events and $\mathtt{complete} = \{compl(\mathtt{t}_1),...,compl(\mathtt{t}_n)\}$ is a set of completion events for tasks. A trigger is *enabled* when $\mathtt{n} = 0$ and $\mathtt{complete} = \phi$ for the corresponding event instance. A configuration is *waiting* if none of the triggers in any trigger set is enabled; otherwise the configuration is *non-waiting*. The four types of triggers are:

- a *write trigger* is a trigger $(\mathtt{gtyp},\mathtt{e},\mathtt{a})$ where $\mathtt{gtyp} = \mathtt{w}$, $\mathtt{e}$ is an event instance and action $\mathtt{a}$ is a tuple $(\mathtt{c},\mathtt{i},\mathtt{t})$ where $\mathtt{c}$ is a communicator, $\mathtt{i} \in \mathbb{N}_{>0}$ and $\mathtt{t}$ is a task.
- a *switch trigger* is a trigger $(\mathtt{gtyp},\mathtt{e},\mathtt{a})$ where $\mathtt{gtyp} = \mathtt{s}$, $\mathtt{e}$ is an event instance, and action $\mathtt{a}$ is a pair $(\mathtt{sw},\mathtt{m})$ where $\mathtt{sw}$ is a mode switch in mode $\mathtt{m}$.
- a *read trigger* is a trigger $(\mathtt{gtyp},\mathtt{e},\mathtt{a})$ where $\mathtt{gtyp} = \mathtt{d}$, $\mathtt{e}$ is an event instance, and action $\mathtt{a}$ is a tuple $(\mathtt{t},\mathtt{c},\mathtt{i})$ where $\mathtt{t}$ is a task, $\mathtt{c}$ is a communicator and $\mathtt{i} \in \mathbb{N}_{\geq 0}$.
- a *release trigger* is a trigger $(\mathtt{gtyp},\mathtt{e},\mathtt{a})$ with $\mathtt{gtyp} = \mathtt{r}$, $\mathtt{e}$ is an event instance and action $\mathtt{a}$ is a task $\mathtt{t}$.

Successor. A configuration $\mathtt{u}'$ is a *successor* of configuration $\mathtt{u}$ if $\mathtt{u}$ is waiting and a completion event or time event occurs, or $\mathtt{u}$ is non-waiting and a write/ switch/ read/ release trigger is handled. There are five types of successors: event / write/ switch/ read and release successor; next we will define each of the above successors.

Event successor. Configuration $\mathtt{u}'$ is an *event successor* if configuration $\mathtt{u}$ is waiting and an event occurs. Three possible scenarios are:

- *a task completion event occurs*: Trigger set for each host $\mathtt{h}_i$ is updated. Let the completion event for $\mathtt{h}_i$ be $compl(\mathtt{t}_i)$ (ϕ if no completion event occurred for the host). The output ports of $\mathtt{t}_i$ are updated from local output variables of the task i.e. for all ports $\mathtt{p} \in \mathtt{iprts}(\mathtt{inv}_i)$, $\mathtt{val}[\mathtt{p}] = \mathtt{val}[\mathtt{lov}^{\mathtt{t_i}}(\mathtt{p})]$. Triggers whose event depends on $compl(\mathtt{t}_i)$ are updated; rest of the triggers remain the same. For all triggers $\mathtt{g} = (.,\mathtt{e},.) \in \mathtt{trgs}_i(\mathtt{u})$ there exists a trigger $\mathtt{g}' = (.,\mathtt{e}',.) \in \mathtt{trgs}_i(\mathtt{u}')$. If $\mathtt{e} = (.,\mathtt{complete})$ and $compl(\mathtt{t}_i) \in \mathtt{complete}$ then $\mathtt{e}' = (.,\mathtt{complete}')$ with $\mathtt{complete}' = \mathtt{complete}/compl(\mathtt{t}_i)$; otherwise $\mathtt{e}' = \mathtt{e}$. Task $\mathtt{t}_i$ is removed from task set i.e. $\mathtt{tasks}_i(\mathtt{u}) = \mathtt{tasks}_i(\mathtt{u}')/\mathtt{t}_i$.
- *a time tick event occurs*: For each trigger set $\mathtt{trgs}_i$, all triggers are updated. For all triggers $\mathtt{g} = (.,\mathtt{e},.) \in \mathtt{trgs}_i$ there exists a trigger $\mathtt{g}' = (.,\mathtt{e}',.) \in \mathtt{trgs}'$. If $\mathtt{e} = (\mathtt{n},.)$ with $\mathtt{n} > 0$ then $\mathtt{e}' = (\mathtt{n}-1,.)$; otherwise $\mathtt{e}' = \mathtt{e}$. The variable state and task set remains same i.e. $\mathtt{state}_i(\mathtt{u}') = \mathtt{state}_i(\mathtt{u})$ and $\mathtt{tasks}_i(\mathtt{u}') = \mathtt{tasks}_i(\mathtt{u})$.
- *a time tick event and a task completion event occurs simultaneously*: Trigger set for each host $\mathtt{h}_i$ is updated. Let the completion event for $\mathtt{h}_i$ be $compl(\mathtt{t}_i)$ (ϕ if no completion event occurred for the host). The output ports of $\mathtt{t}_i$ are updated from local output variables of the task i.e. for all ports $\mathtt{p} \in \mathtt{iprts}(\mathtt{inv}_i)$, $\mathtt{val}[\mathtt{p}] = \mathtt{val}[\mathtt{lov}^{\mathtt{t_i}}(\mathtt{p})]$. Triggers whose event depends on $compl(\mathtt{t}_i)$ and/or have a non-zero time tick count are updated; rest of the triggers remain the same. For all triggers $\mathtt{g} = (.,\mathtt{e},.) \in \mathtt{trgs}_i(\mathtt{u})$ there exists a trigger $\mathtt{g}' = (.,\mathtt{e}',.) \in \mathtt{trgs}_i(\mathtt{u}')$. If $\mathtt{e} = (\mathtt{n},\mathtt{complete})$ and
  - $compl(\mathtt{t}_i) \in \mathtt{complete}$ and $\mathtt{n} \not> 0$ then $\mathtt{e}' = (\mathtt{n},\mathtt{complete}')$ with $\mathtt{complete}' = \mathtt{complete}/compl(\mathtt{t}_i)$,
  - $compl(\mathtt{t}_i) \notin \mathtt{complete}$ and $\mathtt{n} > 0$ then $\mathtt{e}' = (\mathtt{n}-1,\mathtt{complete})$,
  - $compl(\mathtt{t}_i) \in \mathtt{complete}$ and $\mathtt{n} > 0$ then $\mathtt{e}' = (\mathtt{n}-1,\mathtt{complete}')$ with $\mathtt{complete}' = \mathtt{complete}/compl(\mathtt{t}_i)$;

  otherwise $\mathtt{e}' = \mathtt{e}$. Task $\mathtt{t}_i$ is removed from task set i.e. $\mathtt{tasks}_i(\mathtt{u}) = \mathtt{tasks}_i(\mathtt{u}')/\mathtt{t}_i$.

If a completion event occurs without a simultaneous time tick, then $\mathtt{u}'$ is a *completion event successor*; any other event successor is a *time event successor*.

Write successor. Configuration $\mathtt{u}'$ is a *write successor* of non-waiting configuration $\mathtt{u}$ if an enabled write trigger is handled at $\mathtt{u}$.

Without loss of generality, say the write trigger be $\mathtt{g} = (\mathtt{w},\mathtt{e},\mathtt{a}) \in \mathtt{trgs}_i(\mathtt{u})$ with $\mathtt{e} = (0,\phi)$ and $\mathtt{a} = (\mathtt{c},\mathtt{i},\mathtt{t})$. The trigger is handled as follows. Value of communicator $\mathtt{c}$ is updated. If task $\mathtt{t}$ is executed on host $\mathtt{h}_i$, then $\mathtt{val}[\mathtt{c}] = \mathtt{val}[\mathtt{lov}^{\mathtt{t}}(\mathtt{c},\mathtt{i})]$; otherwise $\mathtt{val}[\mathtt{c}] = \mathtt{val}[\mathtt{lov}^{\mathtt{t}}_{\mathtt{h_i}}(\mathtt{c},\mathtt{i})]$. Values for rest of the communicators and ports remain the same. Trigger $\mathtt{g}$ is removed from trigger set, $\mathtt{trgs}_i(\mathtt{u}') = \mathtt{trgs}_i(\mathtt{u})/\mathtt{g}$ and task set remains same, $\mathtt{tasks}_i(\mathtt{u}') = \mathtt{tasks}_i(\mathtt{u})$. For all other hosts $\mathtt{h}_j \in \mathtt{hset}/\mathtt{h}_i$, the variable state, trigger set and task set remains the same i.e. $\mathtt{state}_j(\mathtt{u}') = \mathtt{state}_j(\mathtt{u})$, $\mathtt{trgs}_j(\mathtt{u}') = \mathtt{trgs}_j(\mathtt{u})$ and $\mathtt{tasks}_j(\mathtt{u}') = \mathtt{tasks}_j(\mathtt{u})$.

Switch successor. Configuration $\mathtt{u}'$ is a *switch successor* of non-waiting configuration $\mathtt{u}$ if an enabled switch trigger is handled. An enabled switch trigger can be handled if no write triggers are enabled and no switch triggers of the ancestors are enabled.

Without loss of generality, the enabled switch trigger be $\mathtt{g} \in \mathtt{trgs}_i(\mathtt{u})$. No write trigger is enabled i.e. there exist no trigger $(\mathtt{w},\mathtt{e},.) \in \mathtt{trgs}_k(\mathtt{u})$ with $\mathtt{e} = (0,\phi)$ for all hosts $\mathtt{h}_k \in \mathtt{hset}$. Let trigger $\mathtt{g} = (\mathtt{s},\mathtt{e},\mathtt{a}) \in \mathtt{trgs}_i(\mathtt{u})$ where $\mathtt{e} = (0,\phi)$, $\mathtt{a} = (\mathtt{sw},\mathtt{m})$ and $\mathtt{sw} = (\mathtt{cnd},\mathtt{m}')$ and there is no trigger $\mathtt{g}'' = (\mathtt{s},\mathtt{e},\mathtt{a}) \in \mathtt{trgs}_i(\mathtt{u})$ such that $\mathtt{e} = (0,\phi)$ and $\mathtt{a} = (.,\mathtt{m}'')$ where $\mathtt{m}'' \in \mathtt{ancestors}(\mathtt{m})$. There are three possible scenarios:

- condition $\mathtt{cnd}$ evaluates to false and there exists enabled switch triggers from $\mathtt{m} \longrightarrow$ trigger $\mathtt{g}$ is removed. Formally, if there exists $\mathtt{g}' = (\mathtt{s},\mathtt{e}',\mathtt{a}') \in \mathtt{trgs}_i(\mathtt{u})$ with $\mathtt{e}' = (0,\phi)$ and $\mathtt{a}' = (.,\mathtt{m})$, then $\mathtt{trgs}_i(\mathtt{u}') = \mathtt{trgs}_i(\mathtt{u})/\mathtt{g}$.
- condition $\mathtt{cnd}$ evaluates to false and there exists no other enabled switch trigger from $\mathtt{m} \longrightarrow$ trigger $\mathtt{g}$ is removed and mode $\mathtt{m}$ is invoked. Formally, if there does not exist a trigger $\mathtt{g}' = (\mathtt{s},\mathtt{e}',\mathtt{a}') \in \mathtt{trgs}_i(\mathtt{u})$ with $\mathtt{e}' = (0,\phi)$ and $\mathtt{a}' = (.,\mathtt{m})$, then i.e. $\mathtt{trgs}_i(\mathtt{u}') = \mathtt{trgs}_i(\mathtt{u})/\mathtt{g}$; invoking of mode $\mathtt{m}$ (see below) may add more triggers to trigger set $\mathtt{trgs}_i(\mathtt{u}')$.
- condition $\mathtt{cnd}$ evaluates to true $\longrightarrow$ all enabled switch triggers corresponding to $\mathtt{m}$ and descendants of $\mathtt{m}$ are removed from the trigger set and all modes in $\mathtt{startSet}(\mathtt{m}')$ are invoked. Let all such triggers be $\mathtt{g}_{grp}$ i.e. $\mathtt{g}' \in \mathtt{g}_{grp}$ if $\mathtt{g}' = (\mathtt{s},\mathtt{e}',\mathtt{a}') \in \mathtt{trgs}_i(\mathtt{u})$ with $\mathtt{e}' = (0,\phi)$ and $\mathtt{a}' = (.,\mathtt{m}''')$ where either $\mathtt{m}''' = \mathtt{m}$ or $\mathtt{m} \in \mathtt{ancestors}(\mathtt{m}''')$. The group of triggers are removed i.e. $\mathtt{trgs}_i(\mathtt{u}') = \mathtt{trgs}'(\mathtt{u}')/\mathtt{g}_{grp}$; invoking of modes may add more triggers to $\mathtt{trgs}_i(\mathtt{u}')$.

The variable state and task set remains same i.e. $\mathtt{state}_i(\mathtt{u}') = \mathtt{state}_i(\mathtt{u})$ and $\mathtt{tasks}_i(\mathtt{u}') = \mathtt{tasks}_i(\mathtt{u})$. For all other hosts $\mathtt{h}_j \in \mathtt{hset}/\mathtt{h}_i$, the variable state, trigger set and task set remains the same i.e. $\mathtt{state}_j(\mathtt{u}') = \mathtt{state}_j(\mathtt{u})$, $\mathtt{trgs}_j(\mathtt{u}') = \mathtt{trgs}_j(\mathtt{u})$ and $\mathtt{tasks}_j(\mathtt{u}') = \mathtt{tasks}_j(\mathtt{u})$.

Invocation of mode a $\mathtt{m}$ adds a read trigger for every communicator instance read by a task invocation in $\mathtt{m}$, a write trigger for every communicator instance written by a task invocation in $\mathtt{m}$, a release trigger for every task invoked in $\mathtt{m}$ and a switch trigger for every mode switch in $\mathtt{m}$. Formally, it involves the following steps:

- for all task invocation $\mathtt{inv} = (\mathtt{t}, \mathtt{ailist}, \mathtt{aolist}, .) \in \mathtt{invocs}(\mathtt{m})$ where $\mathtt{t}$ has a concrete declaration[4]:
  - if $\exists k$ such that $\mathtt{ailist}[k] = (\mathtt{c}, \mathtt{i})$ and $(\mathtt{c}, ., ., \pi_{\mathtt{c}})$ is the communicator declaration, then trigger $\mathtt{g} = (\mathtt{d}, \mathtt{e}, \mathtt{a})$ with $\mathtt{e} = \mathtt{i} \cdot \pi_{\mathtt{c}}$ and $\mathtt{a} = (\mathtt{t}, \mathtt{c}, \mathtt{i})$ is added to $\mathtt{trgs}_i(\mathtt{u}')$.
  - if $\exists j$ such that $\mathtt{aolist}[j] = (\mathtt{c}, \mathtt{i})$ and $(\mathtt{c}, ., ., \pi_{\mathtt{c}})$ is the communicator declaration, then trigger $\mathtt{g} = (\mathtt{w}, \mathtt{e}, \mathtt{a})$ with $\mathtt{e} = \mathtt{i} \cdot \pi_{\mathtt{c}}$ and $\mathtt{a} = (\mathtt{c}, \mathtt{i}, \mathtt{t})$ is added to $\mathtt{trgs}_i(\mathtt{u}')$.
  - trigger $\mathtt{g} = (\mathtt{r}, \mathtt{e}, \mathtt{a})$ with $\mathtt{e} = (\mathtt{n}, \mathtt{complete})$ (defined below) and $\mathtt{a} = \mathtt{t}$ is added to $\mathtt{trgs}_i(\mathtt{u}')$. The time tick count is set to transitive read time of $\mathtt{inv}$ i.e. $\mathtt{n} = \mathtt{r}^*(\mathtt{inv})$ and $\mathtt{complete}$ is the set of completion events of preceding task invocations with concrete declaration i.e. $compl(\mathtt{t}') \in \mathtt{complete}$ if $(\mathtt{t}', ., ., .) \in \mathtt{prec}(\mathtt{inv})$ and $\mathtt{t}'$ has a concrete declaration.
- for each mode switch $\mathtt{sw} = (\mathtt{cnd}, \mathtt{m}') \in \mathtt{switches}[\mathtt{m}]$, trigger $(\mathtt{s}, \mathtt{e}, \mathtt{a})$ with $\mathtt{e} = (\pi[\mathtt{m}], \phi)$ and $\mathtt{a} = (\mathtt{sw}, \mathtt{m})$ is added to $\mathtt{trgs}_i(\mathtt{u}')$.

Read successor. Configuration $\mathtt{u}'$ is a *read successor* of non-waiting configuration $\mathtt{u}$ if an enabled read trigger is handled; an enabled read trigger can be handled if no write/ switch trigger is enabled for any host.

Without loss of generality let the enabled read trigger be $\mathtt{g} = (\mathtt{d}, \mathtt{e}, \mathtt{a}) \in \mathtt{trgs}_i(\mathtt{u})$ with $\mathtt{e} = (0, \phi)$ and $\mathtt{a} = (\mathtt{t}, \mathtt{c}, \mathtt{i})$. No write or switch trigger are enabled i.e. for all triggers $(\mathtt{gtyp}, \mathtt{e}, .) \in \mathtt{trgs}_k(\mathtt{u}')$ with $\mathtt{gtyp} = \mathtt{w}|\mathtt{s}$, $\mathtt{e} \neq (0, \phi)$ for all hosts $\mathtt{h}_k \in \mathtt{hset}$. Local input of $\mathtt{t}$ is loaded with the value of communicator $\mathtt{c}$ i.e. $\mathtt{val}[\mathtt{liv}^{\mathtt{t}}(\mathtt{c}, \mathtt{i})] = \mathtt{val}[\mathtt{c}]$. Trigger $\mathtt{g}$ is removed from trigger set, $\mathtt{trgs}_i(\mathtt{u}') = \mathtt{trgs}_i(\mathtt{u})/\mathtt{g}$; variable state and task set remains same i.e. $\mathtt{state}_i(\mathtt{u}') = \mathtt{state}_i(\mathtt{u})$ and $\mathtt{tasks}_i(\mathtt{u}') = \mathtt{tasks}_i(\mathtt{u})$. For all other hosts $\mathtt{h}_j \in \mathtt{hset}/\mathtt{h}_i$, the variable state, trigger set and task set remains the same i.e. $\mathtt{state}_j(\mathtt{u}') = \mathtt{state}_j(\mathtt{u})$, $\mathtt{trgs}_j(\mathtt{u}') = \mathtt{trgs}_j(\mathtt{u})$ and $\mathtt{tasks}_j(\mathtt{u}') = \mathtt{tasks}_j(\mathtt{u})$.

Release successor. Configuration $\mathtt{u}'$ is a *release successor* of non-waiting configuration $\mathtt{u}$ if an enabled release trigger is handled; an enabled release trigger can be handled if no write/ switch/ read trigger is enabled for any host.

Without loss of generality let the enabled release trigger be $\mathtt{g} = (\mathtt{r}, \mathtt{e}, \mathtt{a}) \in \mathtt{trgs}_i(\mathtt{u})$ with $\mathtt{e} = (0, \phi)$ and $\mathtt{a} = \mathtt{t}$. No write/ switch/ read trigger are enabled i.e. for all triggers $(\mathtt{gtyp}, \mathtt{e}, .) \in \mathtt{trgs}_k(\mathtt{u}')$ with $\mathtt{gtyp} \neq \mathtt{r}$, $\mathtt{e} \neq (0, \phi)$ for all hosts $\mathtt{h}_k \in \mathtt{hset}$. Trigger $\mathtt{g}$ is removed from trigger set i.e. $\mathtt{trgs}_i(\mathtt{u}') = \mathtt{trgs}_i(\mathtt{u})/\mathtt{g}$. Input ports of the task invocation are copied to local input variables i.e. for all ports $\mathtt{p} \in \mathtt{iprts}(\mathtt{inv})$, $\mathtt{val}[\mathtt{liv}^{\mathtt{t}}(\mathtt{p})] = \mathtt{val}[\mathtt{p}]$. Task $\mathtt{t}$ is added to task set i.e. $\mathtt{tasks}_i(\mathtt{u}') = \mathtt{tasks}_i(\mathtt{u}) \cup \{\mathtt{t}\}$. Variable state and task set remains the same i.e. $\mathtt{state}_i(\mathtt{u}') = \mathtt{state}_i(\mathtt{u})$. For all other hosts $\mathtt{h}_j \in \mathtt{hset}/\mathtt{h}_i$, the variable state, trigger set and task set remains the same i.e. $\mathtt{state}_j(\mathtt{u}') = \mathtt{state}_j(\mathtt{u})$, $\mathtt{trgs}_j(\mathtt{u}') = \mathtt{trgs}_j(\mathtt{u})$ and $\mathtt{tasks}_j(\mathtt{u}') = \mathtt{tasks}_j(\mathtt{u})$.

Trace. The *initial* configuration of a program is as follows; the variable states consists of the initial values of ports and communicators, trigger sets consists of triggers by invoking start set for each of the start modes for all top-level modules and the task sets being empty. The *trace* of a program is a sequence of configurations $\mathtt{u}_0, \mathtt{u}_1, \ldots$ where $\mathtt{u}_0$ is the initial configuration and for any two consecutive configurations $\mathtt{u}_{i-1}, \mathtt{u}_i$ (where $i \in \mathbb{N}_{>0}$), configuration $\mathtt{u}_i$ is a valid time event/ completion event/ write/ switch/ read/ release successor of $\mathtt{u}_{i-1}$; the configuration pair $(\mathtt{u}_{i-1}, \mathtt{u}_i)$ is referred as a time event/ completion event/ write/ switch/ read/ release *transition* respectively.

Configuration graph. In an HTL program there are finitely many communicators, ports, task invocations, modes, modules and programs. If the communicators and ports have finitely many values then the HTL program is *propositional*. A propositional HTL program has finitely many configurations. The relation between the configurations and

[4] In the case of abstract program both abstract and concrete declarations are considered.

the transitions of a propositional HTL program can be represented by a labeled transition graph. A *labeled transition graph* $\mathcal{G} = (V, V^0, \Sigma, \sigma)$ consists of a finite set of vertices V, a set $V^0 \subseteq V$ of initial vertices, a set Σ of labels and a relation $\sigma \subseteq V \times V$ of edges, such that all the relations σ are labeled by a label $a (a \in \Sigma)$. The labeled transition graph for configurations is referred as *configuration graph.* A configuration graph, $\mathcal{G}_{\mathtt{P}}$, of an HTL program $\mathtt{P_h}$ is a labeled transition graph, $(V_{\mathtt{P}}, V^0_{\mathtt{P}}, \Sigma_{\mathtt{P}}, \sigma_{\mathtt{P}})$, where 1, $V_{\mathtt{P}}$ is the set of all possible configurations for P, 2, $V^0_{\mathtt{P}}$ is the starting configuration, 3, $\Sigma_{\mathtt{P}}$ is the set of all possible completion event transitions, time tick transitions, write transitions, read transitions, switch transitions and release transitions for P and 4, $\sigma_{\mathtt{P}} \subseteq V_{\mathtt{P}} \times V_{\mathtt{P}}$ such that $(v_{\mathtt{P}}, v'_{\mathtt{P}}) \in \sigma_{\mathtt{P}}$ iff c' is a valid completion event successor, time event successor, write successor, read successor, switch successor or release successor of u, where $v_{\mathtt{P}}, v'_{\mathtt{P}}$ denotes the configurations c, c' respectively.

Properties of well-formed program. The following are some of the interesting properties of the execution trace for well-formed HTL programs. The effect of well-timedness will be discussed in Section 7.

Mode switches for a mode (and the respective ancestors and descendant modes) are enabled simultaneously; the triggers are handled in order from top-level modes. Period of a mode m and its ancestors are identical which implies that the respective switch triggers will be enabled simultaneously. Constraint on trigger handling ensures that switch triggers of m are handled only if no switch triggers of ancestors is enabled. If switch of an ancestor of m evaluates to true, then all switch triggers related to m are removed from the trigger set, thus prioritizing the switches of parents over refinement modes. Mode switching for a well-formed program is explained through an example in Figure 15.

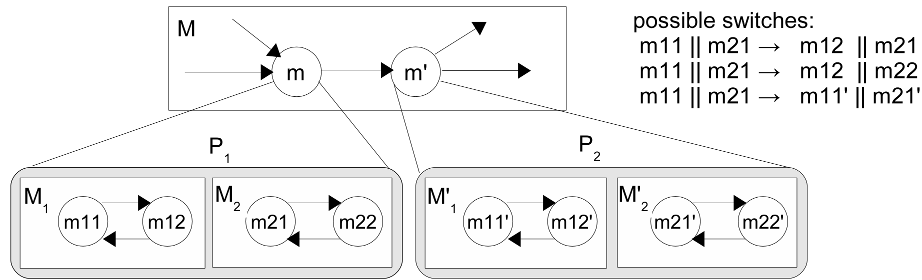


Figure 15: Mode switching through hierarchy

Modes m and m′ are refined by programs P and P′ respectively. Consider a scenario when m, $\mathtt{m}_{11}$ and $\mathtt{m}_{21}$ are executing; from well-formedness constraints periods of all three are identical. At the end of the period mode switches of all the three modes would be checked; there are four possible scenarios:

- switch condition for m is true (the switch condition of refined modes are not checked). The scenario switches to parallel execution of m′, $\mathtt{m}'_{11}$ and $\mathtt{m}'_{21}$ (assuming $\mathtt{m}'_{11}$ and $\mathtt{m}'_{21}$ are start modes of respective modules in P′).
- switch condition for m and $\mathtt{m}_{21}$ are false and switch condition for $\mathtt{m}_{11}$ is true; the scenario switches to parallel execution of m, $\mathtt{m}_{12}$ and $\mathtt{m}_{21}$.
- switch condition for m and $\mathtt{m}_{11}$ are false and switch condition for $\mathtt{m}_{21}$ is true; the scenario switches to parallel execution of m, $\mathtt{m}_{11}$ and $\mathtt{m}_{22}$.
- switch condition for m is false; and switch condition for $\mathtt{m}_{11}$ and $\mathtt{m}_{12}$ is true; the scenario switches to parallel execution of m, $\mathtt{m}_{12}$ and $\mathtt{m}_{22}$.

The only source of non-determinism is the sensor communicators. Except the sensor communicators, which are updated by the environment (through device drivers), the communicators and ports are updated by tasks. For well-formed

HTL program, tasks of only one module can write to a communicator and only one task in a mode (of the module) can write to a communicator instance and to a port. In other words, there is no race on communicator instances and ports. The semantics ensures that communicator updates are done before mode switch checks/ communicator reads and task releases. The switch/ read/ release triggers can update trigger sets and task sets but cannot modify the variable state; this ensures that values of ports and communicators are consistent after all write trigger have been handled until a new event arrives.

Any execution trace from a non-waiting configuration $\mathtt{u}$ *with no enabled write triggers will converge at an unique waiting configuration* $\mathtt{u}'$. Once the write triggers have been handled, communicators and ports cannot be modified before the next event transition. Mode switches being deterministic (at most one switch can be enabled at a given instance) mode invocations are deterministic. In other words, irrespective of the order of handling of switch triggers, the path would lead to an unique configuration $\mathtt{u}_1$ without any enabled switch triggers. Handling of read triggers do not add new triggers, modify existing triggers (other than removing the trigger being handled) or update the variable states. This ensures that irrespective of the order of handling read triggers from $\mathtt{u}_1$, there exists an unique configuration $\mathtt{u}_2$ without any enabled read triggers. Similarly handling of release triggers do not add new triggers, modify existing triggers (other than removing the trigger being handled) or update the variable states. This ensures that irrespective of the order of handling release triggers from $\mathtt{u}_2$, there exists an unique configuration $\mathtt{u}_3$ without any enabled release triggers. Configuration $\mathtt{u}_3$ being unique must be same as $\mathtt{u}'$.

6 HTL-E code Compiler

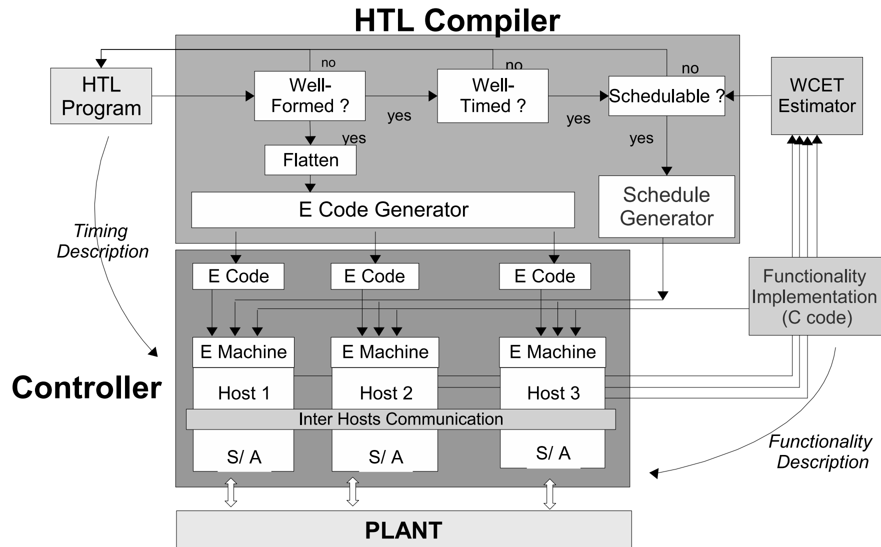


Figure 16: Structure of compiler and target

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

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 the E machine is written in C as well. Released tasks are dispatched for execution by an EDF scheduler that is external to the E machine and also implemented in C. Figure 16 depicts the structure of the compiler and the target architecture. The compiler generates E code for each host separately, i.e., each host runs its own E machine. In our experiments, the hosts communicate via sockets and standard Ethernet. In the following, we explain each phase of the HTL compiler.

Checking well-formedness and well-timedness as well as schedulability. Before flattening the input program, the compiler first checks the well-formedness and well-timedness of the program. In particular, the compiler verifies that any concrete task indeed refines its parent task in order 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 it follows from our result in Section 7 that the whole input program is schedulable. This result also applies to distributed HTL programs as long as the worst-case latency for broadcasting all output ports of each task has been added to the worst-case execution time of the task, and the worst-case latency includes the time it takes to resolve any collisions even when all hosts try to broadcast at the same time. In our current implementation, all output ports of each task are always broadcast to all other hosts as soon as the task completes execution. Communication and scheduling techniques that may utilize the network more efficiently, e.g., [5], can also be used but are not implemented.

Flattening. The HTL compiler flattens a given well-formed and well-timed HTL program into a semantically equivalent flat HTL program that only contains modules on the top level in a straightforward way. 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. 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. Refer Appendix C for details on algorithms used for flattening.

Flattening an HTL program may in theory result in generated code that is exponentially larger with respect to the size of the input program (i.e., number of refinement levels). 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 [6]. 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. We assume that completion events only occur strictly between time ticks, i.e., not at time ticks. An extension of the original trigger implementation to handle completion events in addition to time ticks

has been done. The original E machine implementation itself was not affected. 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 the address a if the predicate cnd evaluates to true. We call cnd a *condition*, which observes port or communicator states such as sensor readings and task outputs. A *jump*(a) instruction is an absolute jump to the E code address a and a *return* instruction completes the execution of an E code sequence.

E code generation. For a given HTL program and a mapping of its top-level modules to hosts, the HTL compiler generates E code for each host separately. The idea is to compile repeatedly the whole program for each host and generate E code that implements the whole program except that the tasks of the modules not mapped to a host are not released on that host. In other words, the generated E code is identical on all hosts except for the instructions that release the tasks. This also means that each host has copies of all communicators and ports. In order to make sure that the distributed system maintains a consistent state of all communicators and ports, each task broadcasts the values of its output ports as soon as the task completes an invocation, i.e., communication is done by the tasks, not by the E code. Therefore, for a schedulability analysis, the communication latency needs to be part of the WCET of each task as we have mentioned earlier. More efficient approaches are possible but have not been implemented such as broadcasting only those output ports of a task that are actually written to communicators that are read by tasks running on other hosts.

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*$[\texttt{m}, i]$. 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.

The compiler generates E code for the flattened input program by invoking Algorithm 1 for each host, which in turn invokes Algorithm 2 to generate E code for each module of the program, which finally invokes Algorithm 3 to generate E code for each mode of each module. The following is a set of auxiliary operators used for code generation:

- $\texttt{init}(x)$ is the driver that initializes the communicator or port x
- $\texttt{readDrivers}(\texttt{m}, \texttt{u})$ is the set of drivers that load the tasks in mode m with values of the communicators that are read by these tasks at unit u
- $\texttt{writeDrivers}(\texttt{m}, \texttt{u})$ is the set of drivers that load the communicators with the output of the tasks in mode m that write to these communicators at unit u
- $\texttt{portDrivers}(\texttt{t})$ is the set of drivers that load task t with the values of the ports of the tasks on which t depends
- $\texttt{complete}(\texttt{t})$ is the set of events that signal the completion of the tasks on which task t depends and that signal the time instants at which communicators are read by t
- $\texttt{releasedTasks}(\texttt{m}, \texttt{u})$ is the set of tasks in mode m with no precedences that are released at unit u
- $\texttt{precedenceTasks}(\texttt{m})$ is the set of tasks in mode m with precedences

Algorithm 1 generates instructions to initialize all communicators and modules. Here, we use instructions of the form $\mathit{future}(0, \mathit{module_address}[\mathtt{M}])$, which effectively execute the E code at the address $\mathit{module_address}[\mathtt{M}]$ similar to a jump-to-subroutine instruction. However, the actual mechanism is more complicated: for a $\mathit{future}(0,a)$ instruction the E machine appends the already enabled trigger-address pair $(\mathtt{true},a)$ 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 $(\mathtt{true},a)$ 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 $(\mathtt{true},0)$ in the queue.

Algorithm 1 GenerateECodeForProgramOnHost($\mathtt{P},\mathtt{h}$)

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

Algorithm 2 generates instructions to initialize all ports of 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. We denote the compiler's program counter by *PC*.

Algorithm 2 GenerateECodeForModuleOnHost($\mathtt{M},\mathtt{h}$)

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

Algorithm 3 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 $\mathit{mode_address}[\mathtt{m}']$ of the target mode $\mathtt{m}'$, not the $\mathit{unit_address}[\mathtt{m}',0]$, since only at most one mode switch per time instant may occur. At each time instant, the generated E code uses $\mathit{future}(0,a)$ instructions to write communicators always before any communicators are read making sure that the latest communicator values are used across all modules. Communicator and port values do not need to 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 port on each host. Additional memory is not required.

Algorithm 3 GenerateECodeForModeOnHost(m, h)

```
u := 0
while u < π[m]/γ[m] do
 set unit_address[m,u] to PC and fix up
 // update communicators with task output
 ∀d ∈ writeDrivers(m,u):emit(call(d))
 if (u = 0)
  // begin mode after other modules updated communicators
  emit(future(0,PC+2))
  emit(return)
  // check mode switches
  ∀(cnd,m′) ∈ switches(m):emit(if(cnd,mode_address[m′]))
  set mode_address[m] to PC and fix up
 else
  // continue mode after other modules updated communicators
  emit(future(0,PC+2))
  emit(return)
 end if
 if (mode m is contained in a module on host h)
  // read communicators into tasks
  ∀d ∈ readDrivers(m,u):emit(call(d))
  // release tasks with no precedences
  ∀t ∈ releasedTasks(m,u):emit(release(t))
  if (u = 0)
   // release tasks with precedences
   ∀t ∈ precedenceTasks(m):
    // wait for tasks on which t depends to complete
    emit(future(complete(t),PC+2))
    emit(jump(PC+6))
    // 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
    ∀d ∈ portDrivers(t):emit(call(d))
    emit(release(t))
    emit(return)
  end if
 end if
 // continue mode after γ[m] time
 emit(future(γ[m],unit_address[m,u+1 mod π[m]/γ[m]]))
 emit(return)
 u := u+1
end while
```

7 Schedulability

An execution trace of an HTL program is a (possibly infinite) sequence of configurations. The schedulability analysis for HTL checks that traces generated from an HTL program has three properties: 1, a task writing to a communicator must have terminated before the communicator update, 2, two instances of the same task do not overlap and 3, if an host is transmitting all other hosts are listening (i.e. neither executing nor transmitting). A *scheduler* decides which task to be executed on each host. The scheduler 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 the task. The scheduler is a discrete-time one i.e. it takes decisions at a periodic event. The periodic event is assumed to be a global clock tick (i.e. clocks of all the hosts are synchronized). The clock is harmonic to the program clock (which is the minimum interval at which any communicator is accessed i.e. the highest common factor for all communicators and mode periods). The worst case execution and transmission times for tasks are specified as multiple of clock ticks. Under this assumption, a task completion event occurs simultaneously at a host clock tick; in other words, without loss of generality all event successors in the execution trace are time event successors. In the analysis below, a time tick will refer to clock advancement of the global clock[5]. If the trace generated by a scheduler maintains the first two properties mentioned above then it is time safe and if the third property is maintained then it is transmission safe. A scheduler is safe if it is both time and transmission safe.

Ready set. Given a configuration $\mathtt{u}$ and an host $\mathtt{h}_i$, *ready set* $\mathtt{ready}(\mathtt{u},\mathtt{h}_i)$ (or $\mathtt{ready}_i(\mathtt{u})$ in short) is a set of tasks for which the corresponding release triggers has been enabled i.e. the tasks would be added to task set before the next waiting configuration; formally, $\mathtt{t}_i \in \mathtt{ready}(\mathtt{u},\mathtt{h}_i)$ if there exists trigger $(\mathtt{s},(0,\phi),\mathtt{t}_i) \in \mathtt{trgs}_i(\mathtt{u})$.

Time-on-host set. Given a configuration $\mathtt{u}$ and an host $\mathtt{h}_i$, *time-on-host set* $\mathtt{toh}(\mathtt{u},\mathtt{h}_i)$ (or $\mathtt{toh}_i(\mathtt{u})$ in short) maps each released task to the amount of cpu time allocated for the execution and transmission of output (of the task); formally, set $\mathtt{toh}(\mathtt{u},\mathtt{h}_i)$ consists of triples $(\mathtt{t}_i,\mathtt{n_e},\mathtt{n_r})$ where $\mathtt{t}_i \in \mathtt{tasks}_i(\mathtt{u})$ and $\mathtt{n_e} \in \mathbb{N}_{\geq 0}$ and $\mathtt{n_r} \in \mathbb{N}_{\geq 0}$ denote the remaining execution and transmission time (for $\mathtt{t}_i$) respectively.

The time-on-host set is updated as follows. Let $\mathtt{u}$ and $\mathtt{u}'$ be consecutive configurations. If $\mathtt{u}'$ is a write/read/switch successor of $\mathtt{u}$, then $\mathtt{toh}_i(\mathtt{u}') = \mathtt{toh}_i(\mathtt{u})$. If $\mathtt{u}'$ is a release successor and the release trigger being handled is $(\mathtt{s},(0,\phi),\mathtt{t}) \in \mathtt{trgs}_i)$ then $\mathtt{toh}_i(\mathtt{u}') = \mathtt{toh}_i(\mathtt{u}) \cup (\mathtt{t},\mathtt{wcet}(\mathtt{t}),\mathtt{wctt}(\mathtt{t}))$. If $\mathtt{u}'$ is a time event successor and the scheduler decides to schedule task $\mathtt{t}_i$ on host $\mathtt{h}_i$ then:

- for all tuples $(\mathtt{t}',\mathtt{n_e},\mathtt{n_r}) \in \mathtt{toh}_i(\mathtt{u})$ where $\mathtt{t}' \neq \mathtt{t}$ there exists tuple $(\mathtt{t}',\mathtt{n_e},\mathtt{n_r}) \in \mathtt{toh}_i(\mathtt{u}')$.
- for tuple $(\mathtt{t},\mathtt{n_e},\mathtt{n_r}) \in \mathtt{toh}_i(\mathtt{u})$ if
– if $\mathtt{n_e} > 0$ then $(\mathtt{t},\mathtt{n_e}-1,\mathtt{n_r}) \in \mathtt{toh}_i(\mathtt{u}')$.
– if $\mathtt{n_e} = 0$ and $\mathtt{n_r} > 1$ then $(\mathtt{t},\mathtt{n_e},\mathtt{n_r}-1) \in \mathtt{toh}_i(\mathtt{u}')$.
– if $\mathtt{n_e} = 0$ and $\mathtt{n_r} = 1$, then tuple $(\mathtt{t},.,.) \notin \mathtt{toh}_i(\mathtt{u}')$.

Scheduler. Let γ be a non-empty finite trace for HTL program $\mathtt{P}$ (on a set of hosts $\mathtt{hset}$) and $\mathtt{u}$ be the last configuration of γ. Given a non-empty finite trace γ such that $\mathtt{u}$ is waiting[6], *scheduler* is a function that returns a mapping $\mathtt{tmap}$ from host $\mathtt{h}_i$ to ϕ or a task $\mathtt{t}_i \in \mathtt{tasks}_i(\mathtt{u})$. An infinite trace $\gamma = \mathtt{u}_0,\mathtt{u}_1,...$ (where $\mathtt{u}_0$ is the initial configuration) is said to be generated by scheduler $\mathtt{sch}$ if for every non-empty finite prefixes $\gamma' = \mathtt{u}_0,\mathtt{u}_1,...,\mathtt{u}_j$ of γ where $\mathtt{u}_j$ is waiting, $\mathtt{sch}(\gamma') = \mathtt{tmap}$ such that $\mathtt{tmap}(\mathtt{h}_i) = \mathtt{t}_i$ (or ϕ), where $\mathtt{t}_i \in \mathtt{tasks}_i(\mathtt{u})$ and $\mathtt{t}_i$ has not

[5] We will also assume each task invocation has an unique task name i.e. a task name uniquely identifies the task invocation.

[6] The successor of $\mathtt{u}$ is a always a time event successor under the initial assumption that execution and transmission times are provided in multiple of clock ticks.

completed execution (i.e. $(\mathtt{t}_i, \mathtt{n_e}, \mathtt{n_r}) \in \mathtt{toh}_i(\mathtt{u})$ with $\mathtt{n_e} > 0$) or $\mathtt{t}_i$ has completed execution but not transmission (i.e. $(\mathtt{t}_i, \mathtt{n_e}, \mathtt{n_r}) \in \mathtt{toh}_i(\mathtt{u})$ with $\mathtt{n_e} = 0$ and $\mathtt{n_r} > 0$). If $(\mathtt{t}_i, 0, 1) \in \mathtt{toh}_i(\mathtt{u})$ then a completion event $compl(\mathtt{t}_i)$ is raised at the next time tick event. Intuitively, at each time step the scheduler decides which task to be executed/ transmitted on each host (it may also decide to keep the host idle).

Time safety. An HTL program executes as intended if task $\mathtt{t}_i$ (on host $\mathtt{h}_i$) completes execution before a communicator is updated by the output (of $\mathtt{t}_i$) or another instance of $\mathtt{t}_i$ is scheduled; an execution trace satisfying the above behavior is a *time safe* trace. A time safe trace is generated by time safe configurations. A configuration is time safe (1) if a communicator is being updated by the evaluation of a task then the task and all the predecessor tasks must have completed execution and, (2) if a task is being released then any other instance of the task must have terminated. A configuration $\mathtt{u}$ is *time safe* if there exists a configuration $\mathtt{u}'$ such that

- if $\mathtt{u}'$ is a write successor of $\mathtt{u}$ and the write trigger being handled is $(\mathtt{w}, (0, \phi), (\mathtt{c}, \mathtt{t})) \in \mathtt{trgs}_i(\mathtt{u})$, then $\mathtt{t}' \notin \mathtt{tasks}_i(\mathtt{u})$ and $\mathtt{t}' \notin \mathtt{ready}_i(\mathtt{u})$ where $\mathtt{t}' = \mathtt{t}$ or $\mathtt{t}'$ is a predecessor of $\mathtt{t}$, and
- if $\mathtt{u}'$ is a release successor of $\mathtt{u}$ and the release trigger being handled is $(\mathtt{r}, (0, \phi), \mathtt{t}) \in \mathtt{trgs}_i(\mathtt{u})$, then $\mathtt{t} \notin \mathtt{tasks}_i(\mathtt{u})$.

A scheduler $\mathtt{sch}$ is *time safe* if all traces generated by the scheduler is time-safe.

Transmission safety. An HTL program transmits as intended if when an host is transmitting the output of a task, no other host is executing or transmitting; an execution trace satisfying the above behavior is a *transmission safe* trace. Consider a finite prefix $\gamma^* = \mathtt{u}_0, \mathtt{u}_1, ..., \mathtt{u}_j$ (for infinite trace γ) where $\mathtt{u}_j$ is waiting and $\mathtt{sch}(\gamma^*) = \mathtt{tmap}$. The scheduler is *transmission safe* if there exists host $\mathtt{h}_i$ such that $\mathtt{tmap}(\mathtt{h}_i) = \mathtt{t}_i$ and $(\mathtt{t}_i, 0, \mathtt{n_r}) \in \mathtt{toh}_i(\mathtt{u}_j)$ then $\mathtt{tmap}(\mathtt{h}_j) = \phi$ for all hosts $\mathtt{h}_j \in \mathtt{hset}/\mathtt{h}_i$.

Definition 1 *A scheduler* $\mathtt{sch}$ *is safe if the scheduler is both time safe and transmission safe.*

Definition 2 *Given an HTL program* $\mathtt{P}$*, a set of hosts* $\mathtt{hset}$ *and a host map* $\mathtt{hmap}$ *(from top-level modules to hosts), the schedulability problem for* $\mathtt{P}$ *returns* $\mathtt{true}$ *if there exists a safe scheduler for* $\mathtt{P}$*,* $\mathtt{false}$ *otherwise.* If the schedulability problem returns true, then the program $\mathtt{P}$ is *schedulable*.

Theorem 1 *Given a well-formed program* $\mathtt{P}$*, if abstract program* $\mathtt{abs(P)}$ *is schedulable then* $\mathtt{P}$ *is schedulable.*

Proof. Assume there exists no safe scheduler for $\mathtt{P}$. Let the safe scheduler for $\mathtt{abs(P)}$ be $\mathtt{sch}'$. We will construct a scheduler $\mathtt{sch}$ for $\mathtt{P}$ as follows. Consider non-empty finite prefix γ'^* and γ^* for infinite traces γ' (for $\mathtt{abs(P)}$) and γ (for $\mathtt{P}$) respectively. Let $\gamma'^* = \mathtt{u}'_0, \mathtt{u}'_1, ..., \mathtt{u}'_j$ where $\mathtt{u}'_j$ is waiting, $n \in \mathbb{N}_{>0}$ time transitions precede $\mathtt{u}'_j$ and $\mathtt{sch}'(\gamma'^*) = \mathtt{tmap}'$. Let $\gamma^* = \mathtt{u}_0, \mathtt{u}_1, ..., \mathtt{u}_k$ where $\mathtt{u}_k$ is waiting, n time transitions precede $\mathtt{u}_k$ and $\mathtt{sch}(\gamma'^*) = \mathtt{tmap}$; task map $\mathtt{tmap}$ is defined as follows:

(a) $\mathtt{tmap}(\mathtt{h}_i) = \mathtt{t}$ if $\mathtt{tmap}'(\mathtt{h}_i) = \mathtt{t}$ where $\mathtt{t}$ is a concrete task of top-level modules of $\mathtt{P}_{\mathtt{h}_i}$,
(b) $\mathtt{tmap}(\mathtt{h}_i) = \phi$ if $\mathtt{tmap}'(\mathtt{h}_i) = \phi$,

(c) $\mathtt{tmap}(\mathtt{h}_i) = \phi$ if $\mathtt{tmap}'(\mathtt{h}_i) = \mathtt{t}'$ and there exists no task $\mathtt{t} \in \mathtt{tasks}_i(\mathtt{u}_k)$ such that $\mathtt{t}'$ is top-level parent of $\mathtt{t}$.
(d) $\mathtt{tmap}(\mathtt{h}_i) = \mathtt{t}$ if $\mathtt{tmap}'(\mathtt{h}_i) = \mathtt{t}'$ and there exists task $\mathtt{t} \in \mathtt{tasks}_i(\mathtt{u}_k)$ such that $\mathtt{t}'$ is top-level parent of $\mathtt{t}$.

Let task $(\mathtt{t}, ., ., .) \in \mathtt{invocs}(\mathtt{m})$ and $\mathtt{m}$ is defined in a program other than the top-level program; under the assumption that every task invocation has unique task name and program structure of well-formed programs, $\mathtt{m}$ is unique for $\mathtt{t}$. Let the last activation of $\mathtt{m}$ is at a configuration $\mathtt{u}_m$ and there are n_o (where $n_o \in \mathbb{N}$ and $0 \leq n_o \leq n$) time transitions between $\mathtt{u}_m$ and $\mathtt{u}_k$ on trace γ^* and switch triggers for current invocation of $\mathtt{m}$ are enabled after n_s time transitions. Consider $\mathtt{t}'$, the top-level parent of $\mathtt{t}$ is invoked in mode $\mathtt{m}'$. Let the last activation of $\mathtt{m}'$ is at a configuration $\mathtt{u}'_{m'}$ and there are n'_o (where $n'_o \in \mathbb{N}$ and $0 \leq n'_o \leq n$) time transitions between $\mathtt{u}'_{m'}$ and $\mathtt{u}_j$ on trace γ'^* and switch triggers for current invocation of $\mathtt{m}'$ are enabled after n'_s time transitions.

Observation 1 *Invocation of* $\mathtt{m}$ *coincides with the invocation of mode* $\mathtt{m}'$. The environment behavior is identical for traces γ and γ' and period of a mode is same as that of its ancestors. This implies that $n_o = n'_o$ and $n'_s = n_s$. In other words period of $\mathtt{t}$ in γ and period of $\mathtt{t}'$ in γ' overlaps.

Observation 2 *For all modes active at* $\mathtt{u}_k$*, the corresponding top-level parent mode is active at configuration* $\mathtt{u}_j$. The proof is by contradiction. Let mode $\mathtt{m}_1$ be enabled at $\mathtt{u}_k$ while the corresponding top-level mode $\mathtt{m}_n$ has not been enabled at $\mathtt{u}_j$. There are two possibilities. First, top-level parent $\mathtt{m}_n$ is not in $\mathtt{P}$. This is not possible as for well-formed programs there is only one top-level program (constraint C1.1). Second, $\mathtt{m}_n$ has terminated while mode $\mathtt{m}$ is active. This is also not feasible: (1) mode $\mathtt{m}_1$ has an unique top-level parent, (2) when mode $\mathtt{m}_n$ terminates all modes in subsequent refinements terminate and (3) when $\mathtt{m}_n$ switches, switch triggers for all modes in refinements are removed and thus eliminating the possibility of modes in refinement programs switching between themselves when the top-level parent in not active.

From the above observations, period for each task in $\mathtt{tasks}(\mathtt{u}_k)$ on trace γ coincides with the period of the corresponding top-level parent task in γ'.

Observation 3 *For each task* $\mathtt{t}' \in \mathtt{tasks}'_i(\mathtt{u}_j)$ *there exists at most one task* $\mathtt{t} \in \mathtt{tasks}_i(\mathtt{u}_k)$ *such that* $\mathtt{t}'$ *is a top-level parent of* $\mathtt{t}$. In other words, a task has an unique top-level parent task relative to all tasks that can execute in parallel. Program $\mathtt{P}$ being well-formed task $\mathtt{t}$ has an unique parent relative to all other tasks in $\mathtt{m}$ and tasks in all modes of sibling modules of $\mathtt{M}$ (where $\mathtt{m} \in \mathtt{modes}(\mathtt{M})$). Using the above constraint and program structure of well-formed programs, it can be proved that for any two tasks either they have different top-level parents or they cannot execute in parallel; refer Section 4.7 for the complete proof.

Observation 4 *If* $(\mathtt{t}, \mathtt{n_e}, \mathtt{n_r}) \in \mathtt{ready}_i(\mathtt{u}_k)$*, and* $(\mathtt{t}', \mathtt{n_e}', \mathtt{n_r}') \in \mathtt{ready}_i(\mathtt{u}_j)$ *then* $\mathtt{n_e} \leq \mathtt{n_e}'$ *and* $\mathtt{n_r} \leq \mathtt{n_r}'$. Let $\mathtt{t}$ does not have preceding tasks i.e. the release depends only on the transitive read time of $\mathtt{t}$. Say $\mathtt{t}$ has been released n_l time transitions earlier (where $n_l \in \mathbb{N}$ and $n_l \leq n_o$) in γ^*, and $\mathtt{t}'$ must have been released $n'_{l'}$ time transitions earlier (where $n'_{l'} \in \mathbb{N}$ and $n'_{l'} \leq n_o$) in γ'^*. From from well-formedness, $\mathtt{r}^*(\mathtt{t}) \leq \mathtt{r}^*(\mathtt{t}')$ which implies $(n_o - n_l) \leq (n_o - n'_{l'})$ or $n'_{l'} \leq n_l$. Given the definition of scheduler $\mathtt{sch}$ and well-timedness constraints ($\mathtt{wcet}(\mathtt{t}) \leq \mathtt{wcet}(\mathtt{t}')$ and $\mathtt{wctt}(\mathtt{t}) \leq \mathtt{wctt}(\mathtt{t}')$), observation 2 holds for $\mathtt{t}$ and $\mathtt{t}'$. This also implies that $\mathtt{t}$ completes execution before $\mathtt{t}'$.

If $\mathtt{t}$ has precedences, then the claim can be proved by induction. The release event for $\mathtt{t}$ and $\mathtt{t}'$ be $(\mathtt{n}, \mathtt{complete})$ and $(\mathtt{n}', \mathtt{complete}')$. The time ticks $\mathtt{n} \leq \mathtt{n}'$ (from constraint of transitive release time). For each $\mathit{compl}(\mathtt{t}_i) \in \mathtt{complete}$,

there is a $compl(\mathtt{t}_i')\in\mathtt{complete}'$ in the release trigger for $\mathtt{t}'$ where $\mathtt{t}_i'$ is parent of $\mathtt{t}_i$. By inductive hypothesis, completion event for each $\mathtt{t}_i$ cannot be later than that of $\mathtt{t}_i'$ (the base case of the argument has been discussed above). In other words all of the completion events in $\mathtt{complete}'$ should have occurred before the completion events in $\mathtt{complete}$ which implies that $\mathtt{t}'$ must have been released later than $\mathtt{t}$.

The observation also argues the necessity of four cases in definition of $\mathtt{sch}$: (a) $\mathtt{t}$ is a concrete task of the top-level mode and hence is executing in both abstract and concrete program, (b) $\mathtt{t}$ has not been released, $\mathtt{t}'$ may (or may not) have been released, (c) $\mathtt{t}$ has been released but not terminated, $\mathtt{t}'$ has been released and terminated and (d) both $\mathtt{t}$ and $\mathtt{t}'$ have been released but neither have terminated.

Claim 7 *Scheduler* $\mathtt{sch}$ *is time safe.*

Let $\mathtt{t}'$ updates a communicator $\mathtt{c}'$ at configuration $\mathtt{u}_p'$ and task $\mathtt{t}$ updates a communicator $\mathtt{c}$ at $\mathtt{u}_q$.

Observation 5 *Task* $\mathtt{t}$ *or none of its preceding tasks are in* $\mathtt{tasks}_i(\mathtt{u}_q)$ *or* $\mathtt{ready}_i(\mathtt{u}_q)$. Let no other communicator is updated by $\mathtt{t}'$ or no switch trigger for mode $\mathtt{m}'$ is handled between $\mathtt{u}_j'$ and $\mathtt{u}_p'$. Similarly, let no other communicator is updated by $\mathtt{t}$ and no switch trigger for mode $\mathtt{m}$ is handled in between $\mathtt{u}_k$ and $\mathtt{u}_q$. Let there be $n_{c'}'\in\mathbb{N}_{>0}$ time transitions between $\mathtt{u}_j'$ and $\mathtt{u}_p'$ and $n_c\in\mathbb{N}_{>0}$ time transitions between $\mathtt{u}_k$ and $\mathtt{u}_q$. Trace γ' is time safe; so $\mathtt{t}'$ or no task preceding $\mathtt{t}'$ in $\mathtt{tasks}_i(\mathtt{u}_p')$ or $\mathtt{ready}_i(\mathtt{u}_p')$. Without loss of generality, let $\mathtt{t}'$ terminates after $n_{t'}'$ time transitions ($n_{t'}'\in\mathbb{N}$ and $n_{t'}'\le n_{c'}'$). If $\mathtt{t}$ terminates after n_t time transitions ($n_t\in\mathbb{N}$) then $n_t\le n_{t'}'$ (from discussion of observation 1). Also $n_{c'}'\le n_c$ as $\tau^*(\mathtt{t}')\le\tau^*(\mathtt{t})$. From above we have, $n_t\le n_{t'}'\le n_{c'}'\le n_c$ which implies that $\mathtt{t}\notin\mathtt{tasks}_i(\mathtt{u}_q)$ and $\mathtt{t}\notin\mathtt{tasks}_i(\mathtt{u}_q)$. If task $\mathtt{t}$ has been terminated all the preceding tasks must have terminated. There is a special case when $n_t=n_{t'}'=n_{c'}'=n_c=n_s$ (i.e. the communicator update and mode switch check would be enabled simultaneously). However from operational semantics, the communicator update would be handled before switch check; thus excluding the possibility of adding $\mathtt{t}$ in task set by new mode invocations. Thus $\mathtt{u}_q$ is time safe.

Observation 6 *Two invocations of* $\mathtt{t}$ *cannot overlap.* There are two possibilities - 1, $\mathtt{t}$ writes to a communicator at the end of mode period and 2, $\mathtt{t}$ writes to a port (but not any communicator). The first case has been discussed above. Time-safety of γ ensures that execution of $\mathtt{t}'$ (irrespective of whether it writes to a communicator or not) is complete after $n_{t'}'\le n_s$ time event transitions. We know $n_t\le n_{t'}'$. If $n_t=n_{t'}'=n_s$ the operational semantics ensure that task is removed from task set before mode $\mathtt{m}$ is invoked i.e. another instance of $\mathtt{t}$ is invoked.

Claim 8 *Scheduler* $\mathtt{sch}$ *is transmission safe.*

Let host $\mathtt{h}_i$ be transmitting the evaluation. So $\mathtt{tmap}'(\mathtt{h}_i)=\mathtt{t}'$ with $(\mathtt{t}',0,\mathtt{n_r}')\in\mathtt{toh}_i(\mathtt{u}_j')$. Scheduler $\mathtt{sch}'$ being transmission safe, for all hosts $\mathtt{h}_j\in\mathtt{hset}/\mathtt{h}_i$, $\mathtt{tmap}'(\mathtt{h}_j)=\phi$. From definition of $\mathtt{sch}$, $\mathtt{tmap}(\mathtt{h}_j)=\phi$. The last proof shows that $\mathtt{t}$ cannot complete execution later than $\mathtt{t}'$. This along with the definition of $\mathtt{sch}$ implies that the transmission for the two tasks start at the same instance from the start of program execution. From well-formedness constraints $\mathtt{wctt}(\mathtt{t})\le\mathtt{wctt}(\mathtt{t}')$. Thus either $\mathtt{t}\notin\mathtt{tasks}_i(\mathtt{u}_k)$ or $(\mathtt{t},0,\mathtt{n_r})\in\mathtt{toh}_i(\mathtt{u}_k)$ with $\mathtt{n_r}\le\mathtt{n_r}'$. In the first case, $\mathtt{tmap}(\mathtt{h}_i)=\phi$, in the second case $\mathtt{tmap}(\mathtt{h}_i)=\mathtt{t}$. The observation implies that when $\mathtt{t}$ is being transmitted, all other hosts are idle. In other words $\mathtt{sch}$ is transmission safe.

Observation 3 and 4 proves Claim 1 which along with Claim 2 shows that scheduler $\mathtt{sch}$ is safe. This contradicts the initial assumption and concludes the proof.

8 Related Works

Timed languages. HTL builds on the LET concept pioneered by the Giotto language [1]. LET-based languages include TDL [7], which like Giotto is restricted to periodic tasks; Timed Multitasking (TM) [8], which defines LET properties through deadlines; and xGiotto [9], an event-triggered LET language. HTL differs in that logical execution times are defined through the reading and writing of communicator instances. This adds considerable flexibility, and naturally supports aperiodic tasks and hierarchical refinement. In periodic languages such as Giotto, only very restricted forms of refinement are possible, and in event-triggered languages, such as xGiotto, scheduling quickly becomes intractable. Refer Appendix B for a detailed comparison between Giotto and HTL.

Synchronous languages. Esterel [10], Lustre [11], and Signal [12] are based on the synchrony assumption that the execution platform is sufficiently fast as to complete the execution before the arrival of the next environment event occurs. Similar to timed languages, the resulting behavior of synchronous languages is highly deterministic, and hence amenable to efficient formal verification. HTL differs from synchronous languages in the program structure, which supports the refinement of tasks into task groups with precedences.

Real-time languages for specialized domains. nesC [13] is targeted towards network-based applications for small, distributed sensor devices. Erlang [14] is a concurrent functional programming language for real-time systems, specifically for telephony and telecommunication. Flex [15] extends C++ by introducing explicit real-time constraints and offers flexible trade-offs between time, resources and precision. Timber [16] is a programming language for implementing event-driven real-time systems. nesC and Erlang are targeted toward soft real-time requirements, while HTL is targeted toward hard real-time. Unlike in HTL, program execution in nesC and Timber is not schedule independent. Flex allows tasks to miss a deadline (in which case it provides an imprecise computation); in HTL this is a run-time error. None of the above languages supports a compositional communicator model and hierarchical task refinement.

9 Conclusion

In this report we present HTL, a hierarchical coordination language for safety critical hard real-time applications. HTL is built upon the LET 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 present the restrictions on a general HTL program to guarantee schedulability of lower levels if higher levels of abstraction are schedulable. The report presents the operational semantics for HTL and discusses the implementation of an HTL compiler to generate code for the Embedded Machine.

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] FlexRay: Road Vehicles - Interchange of Digital Information - Controller Area Network (CAN) for High-Speed Communication. (International Standards Organisation (ISO). ISO Standard -11898, Nov 1993)

[4] CAN: (Flexray communications system specifications 2.1, 2005)

[5] Tindel, K., Clark, J.: Holistic schedulability for distributed hard real-time systems. Microprocessing and Microprogramming - Euromicro Journal (Special Issue on Parallel Embedded Real-Time Systems) (1994)

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

[7] Templ, J.: Tdl specification and report. Technical Report T004, Department of Computer Science, University of Salzburg (2004)

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

[9] 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)

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

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

[12] Guernic, P.L., Borgne, M.L., Gauthier, T., Maire, C.L.: Programming real time applications with signal. Proceedings of the IEEE (1991)

[13] 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

[14] Armstrong, J., Virding, R., Wikström, C., Williams, M.: Concurrent Programming in Erlang. Prentice-Hall (1992)

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

[16] Carlsson, M., Nordlander, J., Jones, M.: The semantic layers of timber. In: The First Asian Symposium on Programming Langauges and Systems (APLAS), Springer Verlag (2003)

A Concrete Syntax

```
Package htlc;


Helpers

all       = [0 .. 0xFFFF];
lowercase = ['a' .. 'z'];
uppercase = ['A' .. 'Z'];
digit     = ['0' .. '9'];
hex_digit = [digit + [['a' .. 'f'] + ['A' .. 'F']]];

tab = 9;
cr  = 13;
lf  = 10;
eol = cr lf | cr | lf; // This takes care of different platforms

not_cr_lf      = [all -[cr + lf]];
not_star       = [all -'*'];
not_star_slash = [not_star -'/'];

blank         = (' ' | tab | eol)+;
short_comment = '//' not_cr_lf* eol;
long_comment  = '/*' not_star* '*'+ (not_star_slash not_star* '*'+)* '/';
comment       = short_comment | long_comment;

letter = lowercase | uppercase | '_';
name   = letter (letter | digit)*;
ident  = name ('.' name)*;


Tokens

program      = 'program';
communicator = 'communicator';
sensor       = 'sensor';
actuator     = 'actuator';
general      = 'general';
period       = 'period';
uses         = 'uses';
module       = 'module';
start        = 'start';
import       = 'import';
export       = 'export';
task         = 'task';
output       = 'output';
input        = 'input';
```

```
state        = ’state’;
parent       = ’parent’;
function     = ’function’;
update       = ’update’;
port         = ’port’;
mode         = ’mode’;
invoke       = ’invoke’;
switch       = ’switch’;
wcet         = ’wcet’;
init         = ’init’;
host         = ’host’;

ident  = ident;
number = digit+;

semicolon = ’;’;
comma     = ’,’;
dot       = ’.’;
colon     = ’:’;

greater_than  = ’>’;
less_or_equal = ’<=’;
assign        = ’:=’;

l_par     = ’(’;
r_par     = ’)’;
l_brace   = ’’;
r_brace   = ’’;
l_bracket = ’[’;
r_bracket = ’]’;

blank   = blank;
comment = comment;


Ignored Tokens

blank, comment;


Productions

program_declaration_list = program_declaration*;
program_declaration      = program [program_name]:ident
                            l_brace
                              communicator_declaration_list?
                              module_declaration_list
                            r_brace;
```

```
communicator_declaration_list = communicator communicator_declaration* ;
communicator_declaration      = [type_name]:ident
                                [communicator_name]:ident
                                period [communicator_period]:number
                                init [init_driver]:ident
                                semicolon;

module_declaration_list = module_declaration*;
module_declaration      = module [module_name]:ident
                          host_declaration?
                          start [start_mode]:ident
                          l_brace
                            port_declaration_list?
                            task_declaration_list
                            mode_declaration_list
                          r_brace;

host_declaration = l_bracket
                     [host_name]:ident
                     [host_ip]:ip_declaration colon
                     [host_port]:number
                   r_bracket;
ip_declaration   = [a]:number [d1]:dot
                   [b]:number [d2]:dot
                   [c]:number [d3]:dot
                   [d]:number;

port_declaration_list = port port_declaration*;
port_declaration      = [port_type]:ident
                        [port_name]:ident assign [init_driver]:ident semicolon;

task_declaration_list = task_declaration*;
task_declaration      = task [task_name]:ident
                        input [input_formal_ports]:formal_ports
                        state [state_formal_ports]:state_ports
                        output [output_formal_ports]:formal_ports
                        task_function?
                        task_wcet?
                        semicolon;

task_function = function [function_name]:ident;
task_wcet     = wcet [wcet_map]:number;

formal_ports     = l_par formal_port_list? r_par ;
formal_port_list = concrete formal_port formal_port_tail* | (formal_port+) ;
formal_port_tail = comma formal_port ;
formal_port      = [type_name]:ident [port_name]:ident ;

state_ports     = l_par state_port_list? r_par ;
```

```
state_port_list = concrete state_port state_port_tail* | (state_port+) ;
state_port_tail = comma state_port ;
state_port      = [type_name]:ident [state_name]:ident
                  assign [init_driver]:ident ;

mode_declaration_list = mode_declaration*;
mode_declaration      = mode [mode_name]:ident
                        period [mode_period]:number
                        refine_program?
                        l_brace
                          task_invocation_list
                          mode_switch_list
                        r_brace;

refine_program = program [program_name]:ident;

task_invocation_list = task_invocation*;
task_invocation      = invoke [task_name]:ident
                       input [input_actual_ports]:actual_ports
                       output [output_actual_ports]:actual_ports
                       parent_task?
                       semicolon;
parent_task          = parent [task_name]:ident;

actual_ports     = l_par actual_port_list? r_par ;
actual_port_list = concrete actual_port actual_port_tail* | (actual_port+) ;
actual_port_tail = comma actual_port ;
actual_port      = concrete [port_name]:ident | communicator_instance ;

communicator_instance = l_par
                          [communicator_port_name]:ident comma
                          [communicator_instance_number]:number
                        r_par ;

mode_switch_list = mode_switch*;
mode_switch      = switch
                   l_par
                     [condition_function]:ident
                     switch_ports
                   r_par
                   [destination_mode]:ident
                   semicolon;

switch_ports     = l_par switch_port_list? r_par ;
switch_port_list = concrete switch_port switch_port_tail* | (switch_port+) ;
switch_port_tail = comma switch_port ;
switch_port      = [port_name]:ident;
```

B Giotto vs. HTL

B.1 Comparison

The difference between Giotto and HTL is discussed based on the following example. There are three sensors (s_1, s_2, s_3), two actuators (a_1, a_2) and four tasks (t_1, t_2, t_4, t_5). The intended execution starts with tasks t_1, t_2, t_4; upon some predefined condition task t_4 is replaced by t_5 (the reverse switch is also possible). The data flow between tasks, sensors, and actuators is as follows: t_1 reads from s_1, t_2 reads the output of t_1 and sensor s_2, and updates actuator a_1, t_4 (t_5) reads sensor s_3 and updates actuator a_2. Tasks t_1 and t_2 should be executed every 10 ms, while tasks t_4 and t_5 should be executed every 5 ms. Figure 17 and Figure 18 shows (simplified) Giotto and HTL code.

```
mode m1() period 10 {
  actfreq 1 do a1(a1_drv);
  actfreq 2 do a2(a2_drv);

  exitfreq 2 do m2(sw_drv);

  taskfreq 1 do t1(drv1);
  taskfreq 1 do t2(drv2);
  taskfreq 2 do t4(drv4);
}
```

```
mode m2() period 10 {
  actfreq 1 do a1(a1_drv);
  actfreq 2 do a2(a2_drv);

  exitfreq 2 do m1(sw_drv);

  taskfreq 1 do t1(drv1);
  taskfreq 1 do t2(drv2);
  taskfreq 2 do t5(drv5);
}
```

Figure 17: Giotto modes

HTL implementation reduces latency than an equivalent Giotto implementation. In Giotto code, t_2 can read the output of t_1 only at period boundaries (even if t_1 terminates earlier than the period). In other words, there is a delay of one period. On the other hand, t_2 reads the output of t_1 in HTL implementation as soon as t_1 completes.

```
program P {
 communicator
  s1, s2, s3, a1, a2
 module M10 start m10 {
  port p1
  task t1 // concrete decl.
  task t2 // concrete decl.
  mode m10 period 10 {
   invoke t1 input (s1,0)
             output (p1)
   invoke t2 input (s2,0)
             output (a1,10)
}

 module M5 start m5 {
  task t3 // concrete decl.
  mode m5 period 5 program refP {
   invoke t3 input (s3,0)
             output (a2,5)
}
```

```
program refP {
 module refM start refm1 {
 task t4 // concrete decl.
 task t5 // concrete decl.
 mode refm1 period 5 {
  invoke t4 input (s3,0)
    output (a2,5) parent t3;
  switch (cond, refm2);
 }
 mode refm2 period 5 {
  invoke t5 input (s3,0)
    output (a2,5) parent t3;
  switch (cond, refm1);
 }

}
```

Figure 18: HTL code fragements

HTL implementation reduces latency for sensor readings. In Giotto, the sensors are read at the start of task periods; thus s_1 is read once every 10 ms. In HTL, the sensors can be read in the middle of task periods. For example, the communicator instance of communicator s_2, can be set to a number between 0 and 9 to indicate which sensor instance should be read within the period. If need be, the task can read multiple sensor instances within the period.

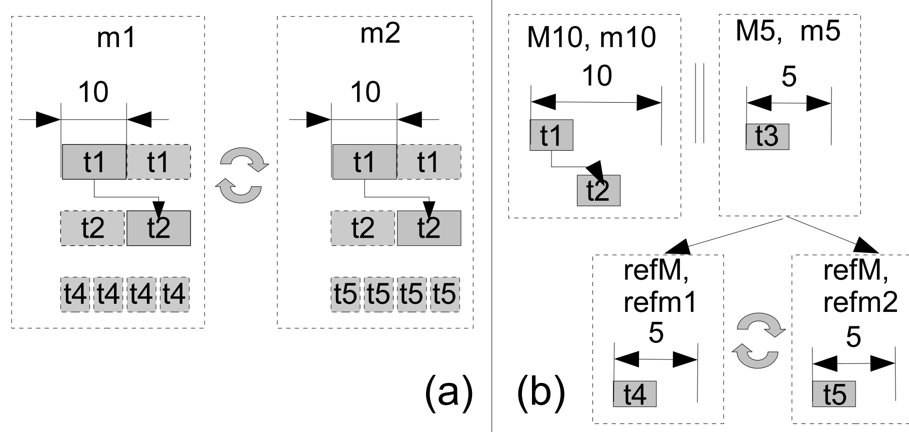


Figure 19: Giotto (a) and HTL (b) implementation diagrams;

HTL allows more structure than Giotto specification. The Giotto modes are different by only one task; mode m_2 invokes t_5 in place of t_4 (both of period 5). In HTL, the tasks are partitioned for efficient handling; mode m10 invoked tasks t_1 and t_2 and mode m5 invokes an abstract task t_3 (to be used a placeholder for both t_4 and t_5). Mode m5 is then refined by program refP which consists of two modes switching between themselves; mode refm1 invokes t_4 and mode refm2 invokes t_5. This helps in code reduction and better structure with increase in choices.

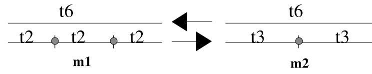
B.2 Giotto-to-HTL

As we showed previously HTL is more powerful then Giotto, this means that there should be an algorithm which converts a Giotto program to an HTL program. In this subsection we present the conversion of a Giotto program to an HTL program based on three examples.

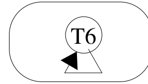
In the first example (Figure 20) we consider a Giotto program which consists of two modes (m_1 and m_2). Mode m_1 has a period of 6 and invokes two tasks: t_2 with a frequency of 3 (a period of 2), and t_6 with a frequency of 1 (a period of 6). Mode m_2 is similar to mode m_1, the only difference being the fact that in mode m_2 task $task_2$ is replaced by t_3, which has a frequency of 2 (a period of 3). Mode m_1 can switch to mode m_2 with a frequency of 3, the revers switch is also possible but with a frequency of 2. The corresponding HTL program will have three modules one for each frequency group that exists in Giotto program (M_2, M_3, and M_6). As it can be seen in Figure 20, module M_6 is straight forward, since it corresponds to the group of frequency 1, which is represented by only one task (t_6), which is invoked in both Giotto modes (it is not influenced by the mode switch); module M_6 contains only one mode which invokes only one task (t_6), and there is no switch. On the other hand modules M_2 and M_3 are much more complex, this is because the frequency groups represented by two modules are influenced by the mode switch. The two HTL modules have to express all possible states in which Giotto program can get due to the non-trivial mode switch. In order to accomplish this goal there has to be introduced empty modes (modes that do not invoke any task).

In the second example (Figure 21) the only difference is that a third mode (m_3) was added to the Giotto program, the following switches were added compare to the first case: mode m_2 can switch to mode m_3, and mode m_3 can switch to mode m_1. Regarding the corresponding HTL program, it can be seen that a new module (M_4) was added, since there is

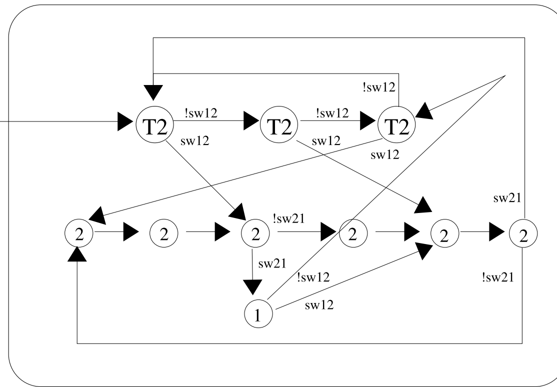
Giotto
Program



Module M6
T6: mode with period 6
and release task t6



Module M2
T2: modes with period 2
and release task t2
2: empty mode with period 1
denotes current
Giotto mode is m2
1: empty mode with period 1
denotes current
Giotto mode is m1



Module M3
T3: modes with period 3
and release task t3
2: empty mode with period 1
denotes current
Giotto mode is m2
1: empty mode with period 1
denotes current
Giotto mode is m1

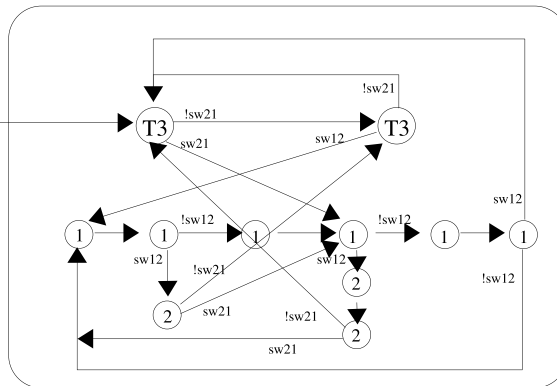
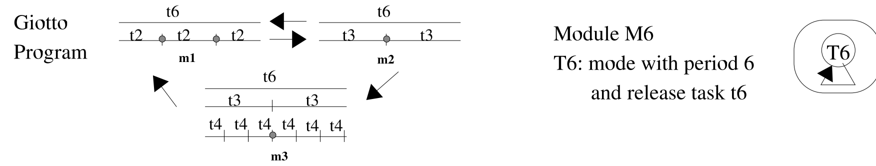


Figure 20: Example 1

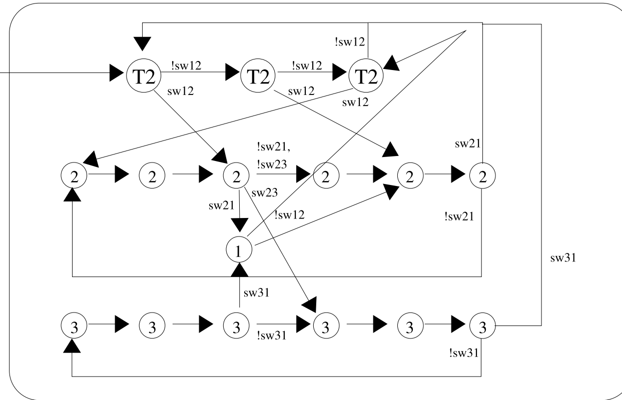
a new frequency group, module M_6 is unchanged since it is not affected by the new added mode, while modules M_2 and M_3 have become more complex, since they have to express more possible states.

In the third example (Figure 22) we can see that there are two groups of frequency (represented by tasks t_6, and t_2) which are not influenced by the mode switch, they will be express in HTL by simple modules (M_6 and M_2, respectively), while the other two groups of frequency (represented by tasks t_3, and t_4) being influenced by the mode switch will lead to complex modules (M_3 and M_4 respectively) in the HTL program.

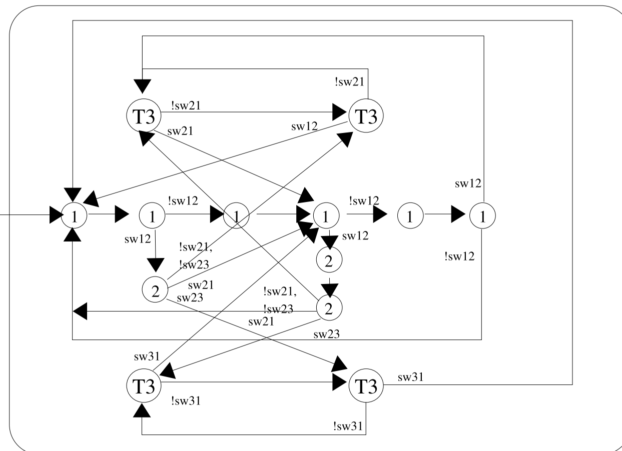
In conclusion when converting a Giotto program to an HTL program, for each group of frequency in Giotto program there has to be a module in the top-level program of the HTL program, the complexity of a module depends on how the mode switches from Giotto program affect the corresponding group of frequency.



Module M2
T2: modes with period 2
and release task t2
2: empty mode with period 1
denotes current
Giotto mode is m2
1: empty mode with period 1
denotes current
Giotto mode is m1
3: empty mode with period 1
denotes current
Giotto mode is m3



Module M3
T3: modes with period 3
and release task t3
2: empty mode with period 1
denotes current
Giotto mode is m2
1: empty mode with period 1
denotes current
Giotto mode is m1
3: empty mode with period 1
denotes current
Giotto mode is m3



Module M4
T4: modes with period 1
and release task t4
2: empty mode with period 1
denotes current
Giotto mode is m2
1: empty mode with period 1
denotes current
Giotto mode is m1
3: empty mode with period 1
denotes current
Giotto mode is m3
(incomplete but can be worked
out from the above two
diagrams)

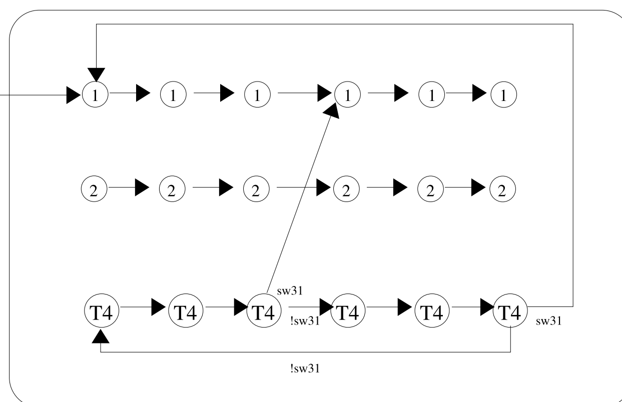
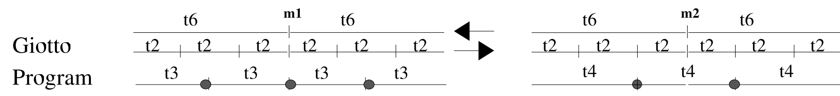
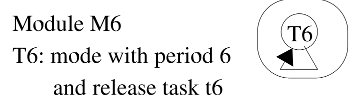
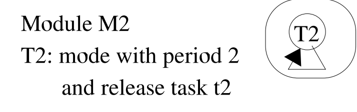
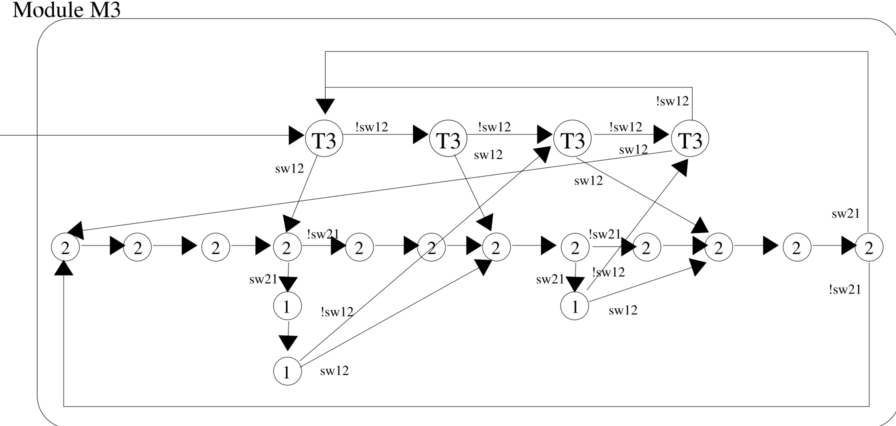


Figure 21: Example 2









Module M4

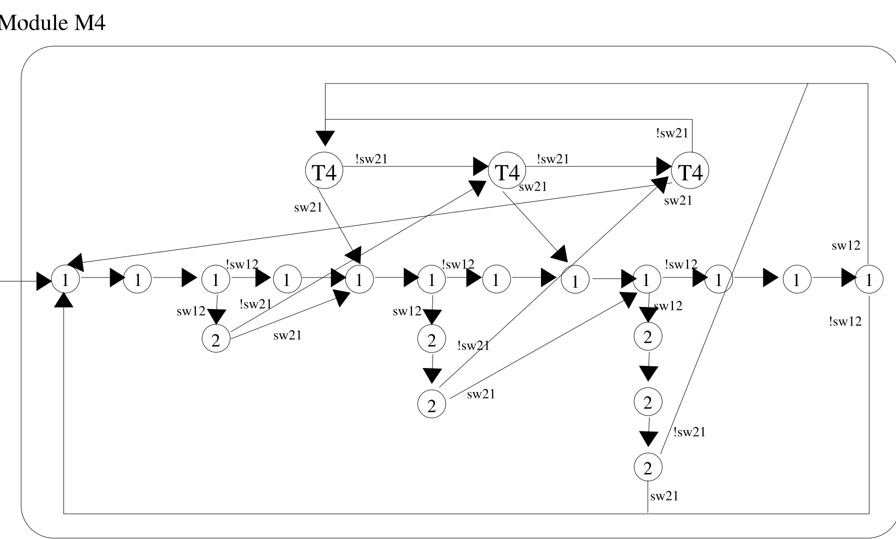
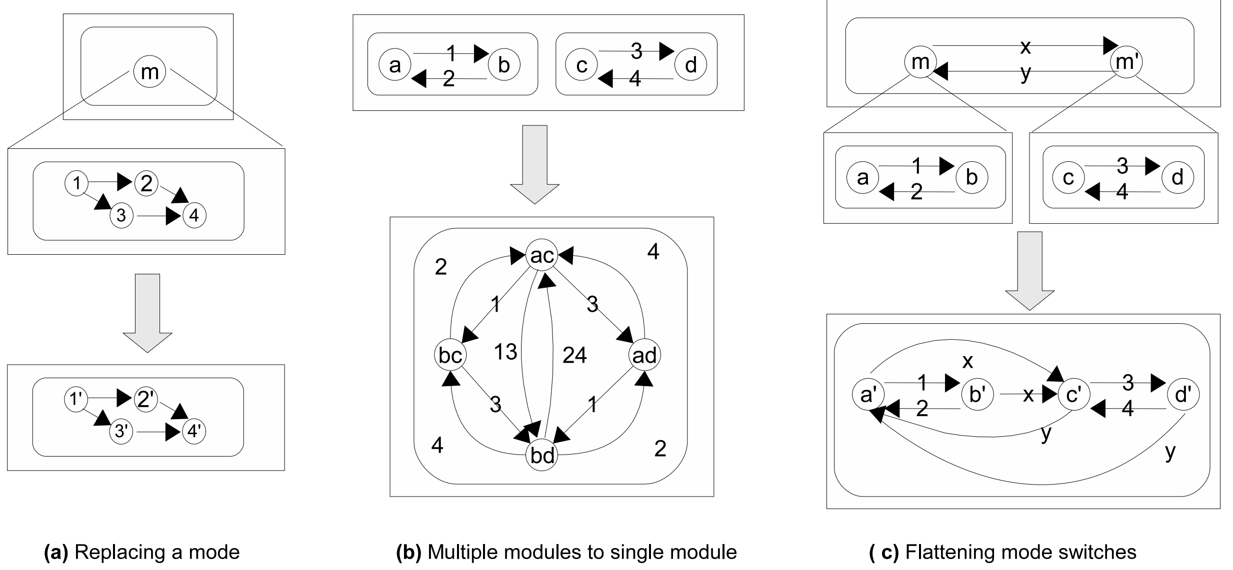


Figure 22: Example 3

C Flattening

Given a well-timed HTL program, $\mathtt{P}$, the program can be converted into a program $\mathtt{P}'$ with same number of modules as in $\mathtt{P}$ but with all mode declaration of the form $(\mathtt{m},.,.)$; in other words all modes in the top-level program with refinement are replaced by modes with no refinement. Instead of presenting a formal translation algorithm we will motivate the case through examples.



(a) Replacing a mode **(b)** Multiple modules to single module **(c)** Flattening mode switches

Figure 23: Flattening HTL program

If the top level program is a leaf program, no further flattening is required; otherwise each mode is replaced by (recursively flattened) refinement program as in Figure 23. Renaming of all communicators, modules, modes, ports, task declarations, task invocations are to be performed first. Next the procedure *FlattenTopLevelProgram* is invoked on the top-level program. If the program is a leaf program then no further action is necessary. Otherwise each mode of the modules which has a refinement has to be flattened. This is carried in two steps. For a mode declaration $(\mathtt{m},.,.,\mathtt{P}')$, first the program is flattened (recursive invocation), second, converting the set of modules to one module and third, merging the single module program with mode $\mathtt{m}$.

Algorithm 4 FlattenTopLevelProgram ($\mathtt{P}$)

// $\mathtt{P}$ is a non-leaf program
for each module $\mathtt{M} \in \mathtt{modules}(\mathtt{P})$
for each $(\mathtt{m},.,.,\mathtt{P}') \in \mathtt{modedecl}(\mathtt{M})$
invoke *FlattenAndConvertToSingleModule* on $\mathtt{P}'$
invoke *MergeModeWithProgram* on $\mathtt{m},\mathtt{P}'$

If a program $\mathtt{P}'$ is not top-level and it is a leaf program the modules can be converted into one single module (Figure 23). This is possible because all modes have identical period; hence composition of the modes can be performed without any need to modify the semantics of mode switches. The conversion is not possible for top-level program as mode periods are different.

Algorithm 5 FlattenAndConvertToSingleModule

```
if P is a leaf program and |modules(P)| = 1
  return;
if P is a leaf program
  define a new module declaration
  (M, portdecl, taskdecl, modedecl, smode)
   M = composition of module names in modules(P)
   portdecl = union of portdecl of all modules in P
   taskdecl = union of concrete task declarations of all modules in P
   empty modedecl and empty smode
  k := |modules(P)|
  for each unique combination m_1,.,m_k of one mode from each module
   create (m, invocs, switches)
   m = m_1 · m_2 · ..m_k
   invocs = invocs_1 ∪ invocs_2 ∪ ..invocs_k
   switches = Power set of mode switches switches,.,switches_k
   add m in moduledecl
  smode is the combination with all start modes
  return
else
  for each module M ∈ modules(P)
   for each (m,.,.,P′) ∈ modedecl(M)
    invoke FlattenAndConvertToSingleModule on P′
    invoke MergeModeWithProgram on m,P′
  invoke FlattenAndConvertToSingleModule on P
  return
```

Algorithm 6 MergeModeWithProgram $(\mathtt{m}, \mathtt{P}')$

```
single module M′ in P′
and let (m, invocs, switches, P′) in modedecl(M)

// modify all mode declarations in modedecl(M′)
  // modify task invocations
   copy all concrete task invocations of m to all modes of M′
  // modify mode switches
   the mode switch conditions are added with
   not-switch-condition for switches in m
   copy all mode switches of m to each mode in M′

// modify M
 copy all task declarations of M′ to M
 copy all port declarations of M′ to M

// modify switches from modes in M′ other than m
for all switches with destination mode m
  replace destination mode by smode of M′

remove (m, invocs, switches, P′) from modedecl(M)
```