Distributed, Modular HTL*

Thomas A. Henzinger
EPFL and IST Austria
tah@ist.ac.at

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

Eduardo R. B. Marques
University of Porto
edrdo@dcc.fc.up.pt

Ana Sokolova
University of Salzburg
anas@cs.uni-salzburg.at

***Abstract*—The Hierarchical Timing Language (HTL) is a real-time coordination language for distributed control systems. HTL programs must be checked for well-formedness, race freedom, transmission safety (schedulability of inter-host communication), and time safety (schedulability of host computation). We present a modular abstract syntax and semantics for HTL, modular checks of well-formedness, race freedom, and transmission safety, and modular code distribution. Our contributions here complement previous results on HTL time safety and modular code generation. Modularity in HTL can be utilized in easy program composition as well as fast program analysis and code generation, but also in so-called runtime patching, where program components may be modified at runtime.**

I. Introduction

The Hierarchical Timing Language (HTL) [1] is a real-time coordination language for distributed control systems.

HTL essentially consists of four building blocks called program, module, mode, and task. An HTL program is a hierarchical, tree-like structure whose root node must be a program block. The immediate successors of a program block can be any number of module blocks, which are executed in parallel. The immediate successors of a module block can be any number of mode blocks, with one mode identified as start mode and some mode switching logic. During execution only one mode per module can be active at any time. The immediate successors of a mode block can be a (refinement) program block and any number of task blocks, which are executed periodically in parallel. Non-concurrent, concrete tasks (with implementations) in a refinement program refine a single abstract task (without implementation) which is to be understood as a placeholder for schedulability. Upon invocation, the implementation of a concrete task, which is written in some language other than HTL, such as C, for example, computes new output from its input. Task input may come from output of other tasks running either in the same mode, implicitly creating a task precedence relation, or in other modules, with possibly different periods, but only through so-called communicators, which are periodically updated program-wide variables with their own periods independent of any task periods. The execution of HTL programs on multiple hosts may be distributed at the level of modules with all inter-host communication done through communicators. Duplicates of the same module may execute on multiple hosts for fault tolerance [2].

HTL programs must be checked for well-formedness (of syntax), race freedom (of communicator updates), transmission safety (schedulability of inter-host communication), and time safety (schedulability of host computation). An optional check of reliability of communicator updates on unreliable hardware [2] may also be performed but is not considered here. The key feature of HTL is that the race-free, transmission-safe, and time-safe execution of well-formed programs is time-deterministic, that is, the computed values and update times of communicators are input-determined and therefore predictable [1].

In this paper, we present (1) a modular abstract syntax and semantics for HTL, (2) modular checks of well-formedness, race freedom, and transmission safety, and (3) modular code distribution. The key to (3) is the modular transmission safety check, ensuring that each communicator value can be communicated within a single communicator period. The original definition of HTL does not provide (1)–(3). Only time safety checking of refinement programs [1] and HTL code generation [3] has already been done modularly. The key result of refinement in HTL is that well-formed concrete programs that refine time-safe abstract programs are also time-safe [1]. Our contributions here complete the distributed and modular design of HTL, except for time safety checking of top-level programs (for which there are no abstractions), which remains non-modular.

Modularity in HTL is important for design scalability (program composition, analysis, and code generation) but also enables efficient program modifications at runtime, called runtime patching, while maintaining predictable behavior [4]. Runtime patching is a semantically well-defined method to modify program components at runtime, and may eventually be used as software foundation for addressing uncertainty in control systems.

*Supported by the EU ArtistDesign Network of Excellence on Embedded Systems Design, the EU project COMBEST, the Austrian Science Funds P18913-N15 and V00125, and Fundação para a Ciência e Tecnologia funds SFRH/BD/29461/2006 and PTDC/EIA/71462/2006.

We first introduce HTL informally by example in Section II. We then discuss HTL compilation and runtime patching in Section III. The modular abstract syntax and semantics for HTL are defined in Section IV, followed by a detailed discussion of modularity in HTL in Section V. Some related work is pointed out in Section VI. The conclusion is in Section VII.

II. Overview of HTL

A. An HTL example application

Figure 1 illustrates a three-tank system (3TS) [5] consisting of three tanks, Tank_1, Tank_2, and Tank_3, with respective evacuation taps Tap_1, Tap_2, and Tap_3, and tank inter-connecting taps $\text{Tap}_{1,3}$ and $\text{Tap}_{2,3}$. Two pumps Pump_1 and Pump_2 actuate on the system by controlling the flow of water into Tank_1 and Tank_2. The aim is to maintain the water level in the tanks both in the normal case with no perturbations (no water leaves the tanks through the tanks' taps) and in the case of perturbations (water leaks). To control the pumps, a proportional (P) controller is used in the absence of perturbations, and two proportional-integrative (PI) controllers are used when there are perturbations, one with slow integration speed for an estimated low control error, the other with faster integration speed. All the controllers work with a frequency of 2Hz.

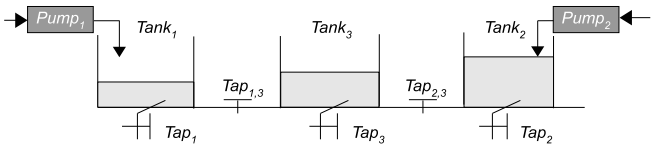

Fig. 1. Three-tank system [5]

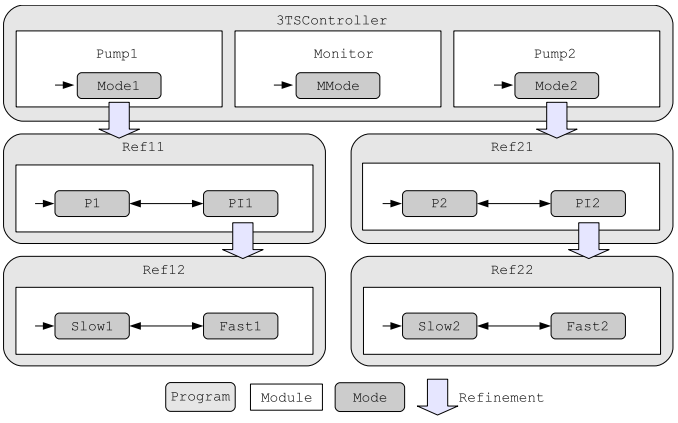

Fig. 2. HTL 3TS controller [3]

To control this system, an HTL program has been implemented [3] running on three hosts with its high-level structure shown in Figure 2. Two hosts have direct access to the two pumps, respectively. The remaining host serves as a monitoring interface to an operator. The HTL top-level program `3TSController` consists of three concurrently running modules `Pump1`, `Monitor`, and `Pump2`, each mapped to one of the mentioned hosts. Each module is organized in modes, which describe switchable configurations of operation in the module, with each mode defining the invocation of a set of real-time task invocations over a time period, some of which can be abstract placeholders for hierarchical refinement. In the example, the pump control modules `Pump1` and `Pump2` are symmetrical in structure. Each has a single mode that is further refined into a program with a single module, switching between modes of P-control and PI-control according to tap perturbations. The PI mode is further refined by a program that defines the "slow"-PI and "fast"-PI control modes.

Figure 3 illustrates a possible execution of `3TSController`. The control starts without water-level perturbations in the system, and modes `P1` and `P2` remain active for some time. Due to a perturbation in the water level of the first tank, there is a mode switch to mode `PI1` within `Ref11` at time 1, and, by refinement, program `Ref12` and mode `Slow1` within `Ref12` become active. Later, at time 2, due to a high control error, there is a mode switch from `Slow1` to `Fast1` within `Ref12`. A similar behavior of the controller of the second tank can be observed, with perturbations making `Ref21`/`PI2` become active at time 1.5, and `Slow2` switching to `Fast2` within `Ref22` at time 3. As perturbations in the first tank decrease, there is a mode switch from `Fast1` to `Slow1` at time 3.5, and at higher level, from mode `PI1` to `P1` at time 4.5.

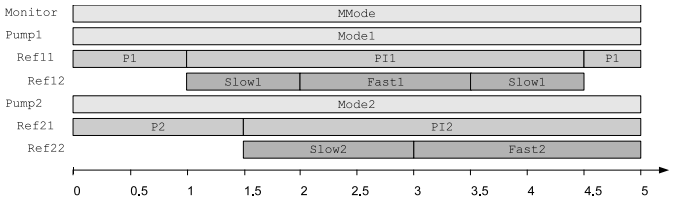

Fig. 3. Sample `3TSController` execution

B. HTL programs, components, and blocks

An HTL program is a tree-like structure of HTL blocks (program, module, mode, task), where the root node is a program block. A component of an HTL program is a subtree of the program. We now describe in more detail the four HTL blocks, and in addition, explain the hierarchical refinement relation, discuss the advantage of hierarchical programs over flat programs, and present the definition of a platform on which an HTL program executes.

Task. A task T in HTL, depicted in Figure 4, is defined by a sequential code procedure `f` without internal synchronization and with isolated memory space, and sets `In`, `Out`, and `Priv` of input, output, and private persistent variables called ports. Each port p is characterized by a set $\texttt{Type}(p)$, defining the domain of p, and an initial value $\texttt{init}(p) \in \texttt{Type}(p)$ that is undefined (equals $\perp$) for non-private ports. The code procedure `f` is implemented in an external language (e.g. C or Java), and its flow of execution comprises an assignment of the task's input ports (`In`) and the actual execution of the procedure code that, upon completion, produces values written to the task's output ports (`Out`) and updates the task's private ports (`Priv`). The private ports enable storing state across task invocations. This task scheme resembles port-based objects [6].

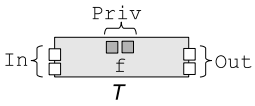

Fig. 4. An HTL task

The HTL task model is a generalization of the Giotto task model [7], based on the notion of logical execution time (LET) of tasks, which is defined by release and termination events. The release time is the latest of logical times for task inputs to become available and the termination time is the earliest of logical times when task outputs may be made available. The LET of a task defines a logical, platform-independent interval for execution of the task's code procedure. All task inputs and outputs can be understood as logically available at the release and termination time, respectively. In an actual execution on a concrete platform, the task procedure may start later than the release event, or complete before the termination event, and be preempted any number of times (if preemption is supported by the platform), as illustrated in Figure 5. The key property of the LET model is that the functional aspects and some key non-functional aspects of programs such as I/O and memory management are preserved across different platforms and workloads as long as there are sufficient computational resources [8], [7], [9].

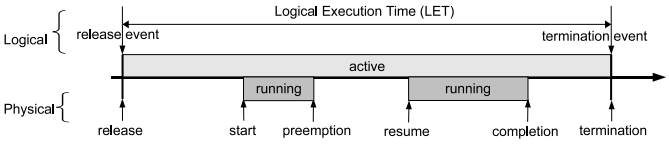

Fig. 5. Logical and physical execution of a task in HTL [1]

A task in HTL is invoked periodically on the timeline. It gets its inputs (resp. produces outputs) from (for) other tasks ports that run in the same period, or from (to) variables called communicators. Figure 6 illustrates the invocation of a task that interfaces with three communicators and other tasks. The inputs and outputs fed from and to other tasks ports define task precedences. A communicator c is specified by a period $\Pi(c)$, a set $\texttt{Type}(c)$ that defines the domain of c, and an initial value $\texttt{init}(c) \in \texttt{Type}(c)$. Communicators can be read or written periodically according to their own periods, and provide means for tasks invoked with different periods to communicate, or an interface with the external runtime environment. A task invocation can read or write a communicator at logical times that fall within the task's invocation period, and match (are divisible by) the communicator's own period. Communicators and ports are not detailed in Figure 2 for the 3TS example, but also there they provide the communication mechanism for task interaction and interaction with the external environment (eg. water-level sensor readings or pump actuation are defined through communicators).

Mode. A mode m in HTL specifies the invocation of a set of tasks, `Tasks`, in a period of time Δ, the mode period. The invocation of each task in a mode is completely defined by acyclic task precedences and read/write access to

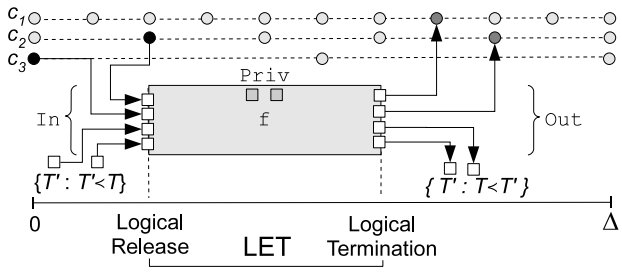

Fig. 6. A task invocation

communicators. The task invocations are established by the mode task invocation relation `Inv` composed of elements of the form (p, p') defining a dependency between an output port p of a task T in `Tasks` and an input port p' of a task T' also in `Tasks`, and of elements (p, c, t) defining a communicator read (resp. write) from (to) communicator c to input (from output) port p at time $t \in [0, \Delta]$ which is a factor of the communicator period. A mode can additionally be refined by a program, as we have already mentioned and will discuss in more detail below.

Figure 7 depicts an example mode with period $\Delta = 10$ and task set $\texttt{Tasks} = \{T_1, T_2, T_3, T_4\}$. We present some details on the task invocation relation for T_1 and T_2 involving the ports p_1 to p_7 and communicators c_1, c_2, and c_3 with periods 1, 2, and 5, respectively. This example also illustrates that the LET of a task is affected both by communicator access and task dependencies: the logical termination time of T_1 is no later than the logical termination of T_2 since there is a dependency from T_1 to T_2, and for the same reason, the logical release time of T_4 is not earlier than the release time of T_3.

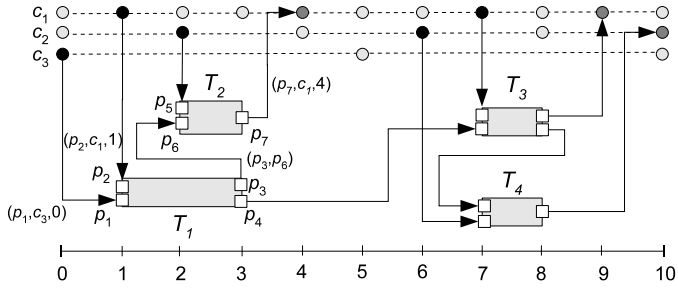

Fig. 7. Mode example

Module. A module M in HTL consists of a set of modes, `Modes`, a starting mode `start` in `Modes`, a mode switching function `switch`, and a set of ports, `Ports`, defining the scope for the ports of tasks in `Modes`. The execution of M corresponds to an execution of its modes in a sequence, starting with `start`, with mode switching taking place at the end of the active mode's period by the evaluation of `switch`. The mode switching function decides the next mode to execute, considering the last active mode and the current values of ports and communicators accessed by tasks in the scope of the module. Modules declared at the top level can have modes with distinct periods that are harmonic with the communicator periods accessed by any mode in the module.

Figure 8 illustrates the structure and a possible execution of an example module M, with 3 modes, m_1, m_2, and m_3 with periods 2, 4, and 3, respectively, with m_1 being the starting mode of M.

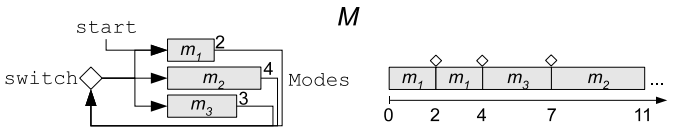

Fig. 8. A module with different mode periods

Program. A program P in HTL consists of a set of modules, `Modules`. If a program P is a top-level program, then it also has a set of communicators, `Comms`. The modules in a program execute concurrently, as illustrated in Figure 3 for the 3TS system, interacting through the program's communicators.

Hierarchical refinement. Hierarchical refinement is enabled if a subset of the tasks of a mode are declared as abstract. Abstract tasks in a mode have no associated code procedure, but are instantiated by other (concrete or abstract) refinement tasks in the mode's refinement program. A mode m with abstract task set `ATasks` must have a defined refinement program `RefP` such that the implementation of tasks in `ATasks` is partitioned across the modules of `RefP`. The other basic syntactic constraints in task refinement require that the LET of a refinement task R must be larger than that of the abstract task A it refines, as illustrated by Figure 9. This means that the refinement task is less constrained in its execution, and in that sense the abstract task is a convenient conservative abstraction for the refinement task.

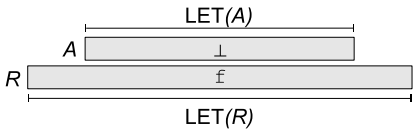

Fig. 9. LET refinement constraint

Hierarchical vs. flat programs. Hierarchical refinement does not add expressiveness to HTL, as it is possible to "flatten" any hierarchical program into a "flat" program without refinement. Hierarchical refinement does however offer the flexibility of hierarchical encapsulation, which in turn can leverage the effort in compiling a program. In general, even beyond HTL, the flat version of a hierarchical program is typically larger and takes more effort to analyze, eventually leading to state-space explosion during compilation. Also, in general, flattening a program may be an unfeasible option if components are "black boxes" with only known "public interface" (see e.g. [10]), or the internal behavior of components is too heterogeneous, requiring an agreed interface to compose them together (see e.g. [11]).

In Figure 10 we depict the structure of a flat HTL program equivalent to the hierarchical 3TS controller of Figure 2, illustrating some issues concerning flat vs. hierarchical programs. The flat program contains "flat modes", accounting

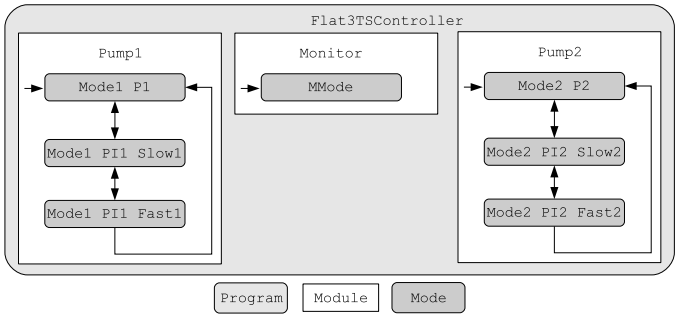

Fig. 10. Flat 3TS controller

for all possible combinations of modes in the refinement level. Furthermore, "flatness" may turn into "fatness" for each flattened component: an abstract mode like `Mode1` in Figure 2 contains abstract tasks but also concrete tasks, so in that case its concrete functionality needs to be repeated three times in the flat program. Finally, and crucially, the program becomes harder to check at the top level, as the number of possible combinations of concurrent top-level mode invocations grows exponentially. We have only one possible top-level combination in the hierarchical 3TS controller (`Mode1`/`Mode2`/`Monitor`), in contrast to nine combinations in the flat program.

C. HTL platform characterization

An HTL program, by definition, does not include a description of the platform on which it executes. The platform definition is encoded separately, or conveyed through programmer annotations.

A platform for the execution of an HTL program P is defined by: (1) A set of hosts, `Hosts`, each with a uniprocessor maintaining a mirror image of the values of all communicators, employing a preemptive EDF scheduling policy, and communicating over a reliable, time-synchronized broadcast network. (2) A distribution $\texttt{dmap} : \texttt{Modules}(P) \rightarrow \texttt{Hosts}$ mapping modules of P to hosts in the platform. Each concrete task T, declared in some module $M \in \texttt{Modules}(P)$, runs on host $\texttt{dmap}(M)$. (3) A WCET mapping $\texttt{wcet} : \texttt{Tasks}(P) \rightarrow \mathbb{Q}_{>0}$ indicating the WCET estimate for all abstract or concrete tasks declared in P. The WCET is a bound on the execution of a task procedure in the absence of any concurrency. If T is an abstract task, then $\texttt{wcet}(T)$ should be understood as an abstract annotation for the upper bound on the WCET for any refinements of T. (4) A WCTT mapping $\texttt{wctt} : \texttt{Comms}(P) \rightarrow \mathbb{Q}_{>0}$. For each communicator c, $\texttt{wctt}(c)$ is a bound on the worst-case time the network needs to transmit a single communicator datum, in the absence of concurrency.

This characterization of platform differs in the formulation of the WCTTs from the original characterization [1]. The difference is that WCTTs are characterized here as raw time needed to transmit a datum, in the absence of concurrency, rather than in the presence of worst-case concurrency. Instead of abstracting the complete network with a worst-case characterization, we consider it here in a form that allows modular transmission safety checking and modular code distribution.

III. Compilation and Runtime Patching

The desired key property of HTL programs is time-determinism, meaning that their functional and temporal behavior is repeatable, i.e., for every timed sequence of inputs there is a unique timed sequence of outputs. HTL compilation is the process of checking whether an HTL program is time-deterministic on a given, possibly distributed target platform, and of generating code that runs on that platform. HTL runtime patching is the process of modifying components of an HTL program at runtime using HTL compilation to re-check correctness and re-generate code.

A. HTL compilation

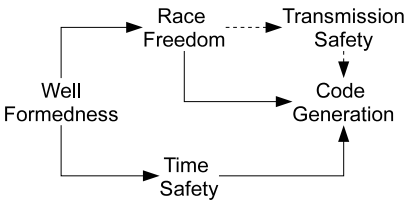

Fig. 11. HTL compilation stages

HTL compilation can be seen as proceeding in stages to check if a program is time-deterministic, as illustrated in Figure 11, before generating code. First, any program specification needs to be checked for well-formedness with respect to the syntactic constraints of the language. Then, race freedom must be established, meaning that no communicator is assigned different values at the same time. The existence of a race clearly leads to non-determinism. Along with race freedom, time safety must also be checked, in the traditional sense of real-time systems, meaning that the computation of a program on each host of the platform is schedulable, taking into account the mapping of modules to hosts and the WCET of tasks. For distributed platforms, a transmission safety check is also necessary, verifying that there exists a schedule for the networked communication between different hosts, based on the WCTT of communicators. In particular, the check verifies that race-free communicator updates can be broadcast during the respective communicators' periods before the update. Time and transmission safety are checked independently, in contrast to [1], by requiring tasks to complete so that their outputs are available at least one communicator period early.

Well-formed, race-free, time-safe, and transmission-safe programs are time-deterministic [1]. Code generation starts after checking these properties. There is a non-modular, flattening HTL compiler generating code that may be exponentially larger than the input program [1], and a modular, hierarchy-preserving HTL compiler [3] generating code that is linear in the size of the input program. The modular compiler can also compile HTL programs on the level of individual components. Below top level, the modular compiler can even check time-determinism and generate code incrementally by only considering a limited context of the component or even no context at all. Section V introduces a framework to quantify the degree of compositionality in modular compilation and applies the framework to HTL.

B. HTL runtime patching

Modular and incremental compilation in particular is not just a prerequisite for scalable software development but may also lead to interesting applications at runtime. Uncertainty in control systems, for example, makes it difficult if not impossible to design control software at compile time that is able to address infinitely many environment states at runtime. Run-time code modifications provide a way to accomplish this. However, semantics and efficiency of code modifications at runtime requires a robust foundation. Runtime patching is the process of modifying code incrementally (for efficiency) at runtime such that the resulting system behavior could be reproduced (for semantics) by a program containing complete copies of the unpatched and the patched program, switching from the unpatched to the patched copy at the time of the patch [4]. Therefore, the semantics of the language in which the patched programs are written also defines the semantics of code modifications through runtime patching.

Runtime patching can be applied to HTL programs using mode switching as a means to express the switch from unpatched to patched versions. HTL components can be patched at runtime as long as there exists a time-deterministic HTL program that can mimick the patch through mode switching. Sufficient conditions for runtime patching cold (non-executing) and even hot (executing) HTL components have been discussed elsewhere [4]. An implementation of HTL runtime patching is future work. The modularity framework and analysis presented in this paper provide a foundation for it.

IV. Modular syntax and semantics

$$
\begin{aligned}
\langle \text{Top Program} \rangle &\to \texttt{Comms} : \langle \text{Communicator} \rangle^{+} , \\
&\quad \langle \text{Program} \rangle_{\texttt{Comms},\emptyset} . \\
\langle \text{Program} \rangle_{\texttt{Comms},\texttt{Ports}} &\to P : \text{Name} , \\
&\quad \texttt{Modules} : \langle \text{Module} \rangle^{+}_{\texttt{Comms},\texttt{Ports}} . \\
\langle \text{Communicator} \rangle &\to c : \text{Name} , \\
&\quad \texttt{Type} : \text{Type} , \\
&\quad \texttt{init} : \texttt{Type}, \\
&\quad \Pi : \mathbb{Q}_{>0} . \\
\langle \text{Module} \rangle_{\texttt{Comms},\texttt{Ports}_0} &\to M : \text{Name} , \\
&\quad \texttt{Ports} : \langle \text{Ports} \rangle^{+} , \\
&\quad \texttt{Modes} : \langle \text{Mode} \rangle^{+}_{\texttt{Comms},\texttt{Ports}_0 \cup \texttt{Ports}} , \\
&\quad \texttt{start} : \texttt{Modes}, \\
&\quad \texttt{switch} : \texttt{Modes}^{\texttt{Modes} \times \mathcal{V}(\texttt{Comms} \cup \texttt{Ports})} . \\
\langle \text{Port} \rangle &\to p : \text{Name} , \\
&\quad \texttt{Type} : \text{Type} , \\
&\quad \texttt{init} : \texttt{Type} . \\
\langle \text{Mode} \rangle_{\texttt{Comms},\texttt{Ports}} &\to m : \text{Name} , \\
&\quad \Delta : \mathbb{Q}_{>0} , \\
&\quad \texttt{Tasks} : \langle \text{Task} \rangle^{+}_{\texttt{Ports}} , \\
&\quad \texttt{Inv} \subseteq (\texttt{Ports} \times \texttt{Ports}) \cup (\texttt{Ports} \times \texttt{Comms} \times \mathbb{Q}_{\geq 0}) , \\
&\quad \texttt{RefP} : \langle \text{Program} \rangle_{\texttt{Comms},\texttt{Ports}} \mid \bot . \\
\langle \text{Task} \rangle_{\texttt{Ports}} &\to T : \text{Name} , \\
&\quad \texttt{In} \subseteq \texttt{Ports}, \\
&\quad \texttt{Out} \subseteq \texttt{Ports}, \\
&\quad \texttt{Priv} \subseteq \texttt{Ports}, \\
&\quad \texttt{f} : \mathcal{V}(\texttt{Out} \cup \texttt{Priv})^{\mathcal{V}(\texttt{In} \cup \texttt{Priv})} \mid \bot .
\end{aligned}
$$

Fig. 12. HTL abstract syntax grammar

A. Syntax

HTL programs are formally defined via an abstract syntax [1], which is implemented by compilers in a textual or

Top level semantics

$$\frac{}{s_0 = (0, v_0(P), \texttt{invoke}(P, 0))}\ (\texttt{INIT})$$

$$\texttt{E} = \{\texttt{complete}, \texttt{write}, \texttt{switch}, \texttt{read}, \texttt{time}\}$$

$$\frac{s \overset{\texttt{complete}}{\Longrightarrow^{\bullet}} s_1 \overset{\texttt{write}}{\Longrightarrow^{\bullet}} s_2 \overset{\texttt{switch}}{\Longrightarrow^{\bullet}} s_3 \overset{\texttt{read}}{\Longrightarrow^{\bullet}} s_4 \overset{\texttt{time}}{\Longrightarrow} s'}{s \longrightarrow s'}\ (\texttt{TOP})$$

$$\frac{e \in \texttt{E} \setminus \{\texttt{time}\} \quad s \overset{e}{\underset{l_1}{\Longrightarrow}} s_1 \overset{e}{\underset{l_2}{\Longrightarrow}} \ldots \overset{e}{\underset{l_n}{\Longrightarrow}} s_n \overset{e}{\not\Longrightarrow} \cdot \quad n \geq 0 \quad \forall i,\ l_i = \varepsilon \vee \forall i \neq j,\ l_i \neq l_j}{s \overset{e}{\Longrightarrow^{\bullet}} s_n}\ (\texttt{MICRO})$$

Invocation and transitions for program $(P, \texttt{Modules})$

$$\texttt{invoke}(P, t) = \{\langle M, \texttt{invoke}(M, t)\rangle \mid M \in \texttt{Modules}\}$$

$$\frac{e \in \texttt{E} \setminus \{\texttt{time}\} \quad \langle M, A_M\rangle \in A \quad (t, v, A_M) \overset{e}{\underset{l}{\Longrightarrow}} (t, v', A'_M)}{(t, v, A) \overset{e}{\underset{l}{\Longrightarrow}} (t, v', (A \setminus \{\langle M, A_M\rangle\}) \cup \{\langle M, A'_M\rangle\})}\ (\texttt{COMP1})$$

$$\frac{\forall \langle M, A_M\rangle \in A, (t, v, A_M) \overset{\texttt{time}}{\Longrightarrow} (t', v, A_M)}{(t, v, A) \overset{\texttt{time}}{\Longrightarrow} (t', v, A)}\ (\texttt{TIME1})$$

Invocation and transitions for a module $(M, \texttt{Ports}, \texttt{Modes}, \texttt{start}, \texttt{switch})$
$(\texttt{PrivPorts} = \{p \in \texttt{Priv}(T) \mid T \in \texttt{Tasks}(m) \wedge m \in \texttt{Modes}\})$

$$\texttt{invoke}(M, t) = \langle \texttt{start}, \texttt{invoke}(\texttt{start}, t)\rangle$$

$$\frac{e \in \texttt{E} \setminus \{\texttt{switch}\} \quad A = \langle m, A_m\rangle \quad (t, v, A_m) \overset{e}{\underset{l}{\Longrightarrow}} (t', v', A_m)}{(t, v, A) \overset{e}{\underset{l}{\Longrightarrow}} (t', v', A)}\ (\texttt{COMP2})$$

$$\frac{\begin{array}{c}(t, v, A_m) \overset{\texttt{switch}}{\Longrightarrow} (t, v', A'_m) \\ m_{new} = \texttt{switch}(m, v(\texttt{Comms} \cup \texttt{Ports})) \\ (m_{new} = m \wedge A_{new} = A'_m) \vee A_{new} = \texttt{invoke}(m_{new}, t)\end{array}}{(t, v, \langle m, A_m\rangle) \overset{\texttt{switch}}{\Longrightarrow} (t, [\texttt{Ports} \setminus \texttt{PrivPorts} := \bot]\ v', \langle m_{new}, A_{new}\rangle)}\ (\texttt{SWITCH1})$$

Invocation and transitions for a mode $(m, \Delta, \texttt{Tasks}, \texttt{Inv}, \texttt{RefP})$
$(\texttt{CTasks} = \{T \in \texttt{Tasks} \mid \texttt{f}(T) \neq \bot\})$

$$\texttt{invoke}(m, t) = \begin{cases} \langle t, \bot\rangle & , \texttt{RefP} = \bot \\ \langle t, \texttt{invoke}(\texttt{RefP}, t)\rangle & , \texttt{RefP} \neq \bot \end{cases}$$

$$\frac{e \in \texttt{E} \setminus \{\texttt{switch}, \texttt{time}\} \quad (t, v, A_r) \overset{e}{\underset{l}{\Longrightarrow}} (t, v', A_r)}{(t, v, \langle a, A_r\rangle) \overset{e}{\underset{l}{\Longrightarrow}} (t, v', \langle a, A_r\rangle)}\ (\texttt{REFINE})$$

$$\frac{(\texttt{RefP} = A'_r = \bot) \vee (t, v, A_r) \overset{\texttt{switch}}{\Longrightarrow} (t, v, A'_r)}{(t, v, \langle t - \Delta, A_r\rangle) \overset{\texttt{switch}}{\Longrightarrow} (t, v, \langle t, A'_r\rangle)}\ (\texttt{SWITCH2})$$

$$\frac{\begin{array}{c}t < t' \leq a + \min_{>t-a}\{\delta \mid (p, \cdot, \delta) \in \texttt{Inv} \vee \delta = \Delta\} \\ A = \langle a, A_r\rangle \quad \texttt{RefP} = \bot \vee (t, v, A_r) \overset{\texttt{time}}{\Longrightarrow} (t', v, A_r)\end{array}}{(t, v, A) \overset{\texttt{time}}{\Longrightarrow} (t', v, A)}\ (\texttt{TIME2})$$

$$\frac{\begin{array}{c}A = (a, \cdot) \quad \forall p \in \texttt{In}(T), v(p) \neq \bot \\ T \in \texttt{CTasks} \quad t - a \leq \texttt{Termination}(T) \\ v' = [\texttt{Out}(T) \cup \texttt{Priv}(T) := \texttt{f}(T)(v(\texttt{Priv}(T) \cup \texttt{In}(T))]\ v\end{array}}{(t, v, A) \overset{\texttt{complete}}{\underset{T}{\Longrightarrow}} (t, v', A)}\ (\texttt{COMPLETE1})$$

$$\frac{\begin{array}{c}A = (a, \cdot) \quad \forall p \in \texttt{In}(T), v(p) \neq \bot \\ T \in \texttt{CTasks} \quad t - a < \texttt{Termination}(T)\end{array}}{(t, v, A) \overset{\texttt{complete}}{\underset{T}{\Longrightarrow}} (t, v, A)}\ (\texttt{COMPLETE2})$$

$$\frac{(p, p') \in \texttt{Inv} \quad v(p) \neq \bot \quad v(p') = \bot}{(t, v, A) \overset{\texttt{read}}{\Longrightarrow} (t, [p' := v(p)]\ v, A)}\ (\texttt{DEP})$$

$$\frac{\begin{array}{c}A = (a, \cdot) \quad (c, p, t - a) \in \texttt{Inv} \\ p \in \texttt{In}(T) \quad v(p) = \bot \quad T \in \texttt{CTasks}\end{array}}{(t, v, A) \overset{\texttt{read}}{\Longrightarrow} (t, [p := v(c)]\ v, A)}\ (\texttt{READ})$$

$$\frac{\begin{array}{c}A = (a, \cdot) \quad (c, p, t - a) \in \texttt{Inv} \\ p \in \texttt{Out}(T) \quad v(p) \neq \bot \quad T \in \texttt{CTasks}\end{array}}{(t, v, A) \overset{\texttt{write}}{\underset{p,c}{\Longrightarrow}} (t, [c := v(p)]\ v, A)}\ (\texttt{WRITE})$$

Fig. 13. HTL semantics

visual representation [8], [12]. Figure 12 depicts the abstract syntax in grammar-like notation, defining the various blocks and the scope of ports and communicators within a program. In the syntax presentation we use the symbol $+$ as in regular expressions, for the set of non-empty sequences of elements in a given set. In addition to the syntax rules (and simple scope and naming constraints detailed in [1]), a well-formed HTL program must conform to a set of constraints regarding the LET of tasks and hierarchical refinement, discussed below.

LET constraints. In a mode m, with period Δ, a task invocation may access (read/write) communicators only at logical times in $[0, \Delta]$ that are factors of the communicators' periods. No read may happen at time Δ, and no write at time 0. Also, Δ must be a factor of every communicator period referenced by other modes in the same module, in order that mode switching to any mode induces an aligned time window for all communicators. In addition, the task precedence relation $\prec_m$ for the tasks in m must be acyclic.

Each task $T \in \texttt{Tasks}(m)$ must have a well-defined LET interval $\texttt{LET}(T) = [\texttt{Release}(T), \texttt{Termination}(T)]$ such that $\texttt{Release}(T)$ is the maximum of times t such that $(p, c, t) \in \texttt{Inv}(m)$ for $p \in \texttt{In}(T)$ or such that $t = \texttt{Release}(T_0)$ for $T_0 \prec_m T$, and $\texttt{Termination}(T)$ is the minimum of times t such that $(p, c, t + \Pi(c)) \in \texttt{Inv}(m)$ for $p \in \texttt{Out}(T)$ or such that $t = \texttt{Termination}(T_0)$ for $T \prec_m T_0$. Note that for communicator writes we take into account the transmission time, which takes at most one communicator period $\Pi(c)$. Hence, if $(p, c, t) \in \texttt{Inv}(m)$ for $p \in \texttt{Out}(T)$, then at time t the corresponding p-output of T is both written to the communicator c and has already been transmitted via the network.

Refinement constraints. A mode m is refined if and only if it contains at least one abstract task, i.e., $A \in \texttt{Tasks}(m)$ with $\texttt{f}(A) = \bot$. In that case, each refinement mode m' within $\texttt{RefP}(m)$ must have the same period as m, and each of its tasks must be a proper refinement of a unique abstract task A.

A task T is a proper refinement of A if and only if the following four conditions are met: (1) The LET of T contains that of A, $\texttt{LET}(T) \supseteq \texttt{LET}(A)$; (2) Any precedence of T is a precedence of A, directly or by refinement: if $T_0 \prec_m T$ then T_0 is either a refinement of a precedence of A, or a concrete task in the parent mode that is a precedence of A; (3) T writes to all of the output ports of A, $\texttt{Out}(T) \supseteq \texttt{Out}(A)$; and (4) T only writes to communicators to which A also writes. We note that condition (4) is not present in [1], but it certainly characterizes a well-defined refinement and is necessary for establishing race freedom in the simple way hinted in [1]. Whenever an execution platform is specified for the program,

the following condition must also be satisfied: (5) The WCET of T is less than or equal to the WCET of A. Note that an abstract task is not an interface for communication of refinement tasks, but merely a placeholder for schedulability ensuring race freedom.

B. Semantics

An operational semantics was given for HTL in [1], in close relation to the time-triggered execution of the E machine [9]. Here we briefly present a more abstract modular semantics defined compositionally in terms of the HTL blocks.

A state of a top program P is a triple $s = (t, v, A)$ where $t \in \mathbb{Q}_{\geq 0}$ is the current time, v is a valuation of the variables of P, and A is an activation of P. Activations are defined compositionally: for a program P, an activation $A \in \texttt{Act}(P)$ is a set

$$A = \{\langle M, A_M\rangle \mid M \in \texttt{Modules}(M), A_M \in \texttt{Act}(M)\}$$

where $A_M = \langle m, A_m\rangle \in \texttt{Act}(M)$ is an activation for the module M, with $m \in \texttt{Modes}(M)$ being the active mode, and $A_m \in \texttt{Act}(m)$ its activation. A mode activation for m is another pair $A_m = \langle a, A_r\rangle$, with activation time $a \in \mathbb{Q}_0^+$, and a refinement activation $A_r = \bot$ if $\texttt{RefP}(m) = \bot$, or $A_r \in \texttt{Act}(\texttt{RefP}(m))$ otherwise.

A program trace $s_0 \longrightarrow s_1 \longrightarrow s_2 \longrightarrow \cdots$ is defined by a two-layer operational semantics shown in Figure 13. The initial program state of P (INIT rule) is $s_0 = (0, v_0(P), \texttt{invoke}(P, 0))$, where $v_0(P)$ assigns the initial value $\texttt{init}(\texttt{x})$ to each communicator or port x in P. In the initial activation $\texttt{invoke}(P, 0)$, the active mode of each module is set to be the specified start mode. A successor state s' of s is reached after a sequence of task completions, writes, reads, and mode switches, defining "micro"-successors, before an elapse of time (TOP rule). We write e for any of these events (comprising the set of events E). Each e-"micro"-successor, except for the time elapse, comprises a closed sequence of e-events with distinct labels or no labels at all (MICRO rule). The labels are a convenience artifact that allows us to model that any (complete or write) event transition is enabled only once at a time instant.

The handling of program events is modeled by the second layer of the semantics, modularly. A program event is either an independent event in one of its modules (COMP1) or a synchronized elapse of time in all of its modules (TIME1). The behavior of a module reduces to the behavior of the current active mode (COMP2), until mode switching is due and the active mode may change (SWITCH1). At the level of modes, the behavior is defined by the concrete tasks of the mode, and compositionally by the refinement program (REFINE). Switching is enabled at the end of the mode's period (SWITCH1), and time elapses with an offset up to the next logical event in the mode (TIME2). Both switching and time elapsing are synchronized events with the mode's refinement program. In the lowest level of concrete task invocations in a mode, each task completion is modeled as completing non-deterministically at an instant that is within the LET of the task (COMPLETE1 and COMPLETE2). Other than that, the state may change by appropriate updates of ports and communicators (DEP, READ, and WRITE) due to task precedences and communicator reads/writes. Note that the semantics defines only the time-deterministic behavior of the program and not an actual execution on a platform. In this respect it qualifies as "abstract".

V. Modularity

A. Modularity framework

We consider the following framework, illustrated in Figure 14. Given a top-level program P_o for which property or aspect φ holds, and a component C_o within P_o that is updated to a component (patch) C yielding top-level program P, we wish to establish φ for P. Let $\mathcal{A}$ be the algorithm used for establishing φ. The modularity of φ given $\mathcal{A}$, in regard to P and C, is a pair

$$\left(\mathcal{D}_\varphi^{\mathcal{A}}(C, P), \mathcal{C}_\varphi^{\mathcal{A}}(C, P)\right)$$

where $\mathcal{D}_\varphi^{\mathcal{A}}(C, P)$ stands for the part of P that needs to be re-analyzed in order to establish φ, called the modular dependency context and $\mathcal{C}_\varphi^{\mathcal{A}}(C, P)$ is the complexity function measuring the effort in doing so in terms of the size of the dependency context, called the modular complexity.

For comparison, we also denote by $\overline{\mathcal{C}}_\varphi^{\mathcal{A}}(P)$ the complexity of the total effort of checking φ for P using $\mathcal{A}$. Note that, in the modularity framework, we exclude the case $C_o = P_o$, i.e., $C = P$, since then the modular complexity equals $\overline{\mathcal{C}}_\varphi^{\mathcal{A}}(P)$.

Hence, our framework is one for incremental compilation, applicable when a certain component of a program is updated, and thus the program requires re-compilation. The framework also subsumes plain incremental compilation, in case when the old component C_o is the "empty component".

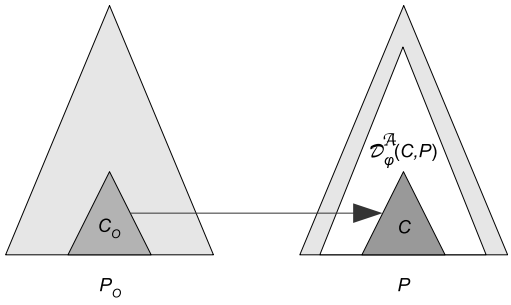

Fig. 14. Modularity

The modular dependency context $\mathcal{D}_\varphi^{\mathcal{A}}(C, P)$ alone is already a good measure of modularity, and is in close relation to the modular complexity. In the best-case scenario, we will have $\mathcal{D}_\varphi^{\mathcal{A}}(C, P) = C$, meaning that only C needs to be re-checked. This is the case when φ (and $\mathcal{A}$) is "fully modular" for C with regard to P, or C can be checked to be a refinement of C_o in regard to φ. In the worst-case scenario, we will have $\mathcal{D}_\varphi^{\mathcal{A}}(C, P) = P$ and $\mathcal{C}_\varphi^{\mathcal{A}}(C, P) = \overline{\mathcal{C}}_\varphi^{\mathcal{A}}(P)$, meaning that P needs to be fully re-checked, i.e., φ (or $\mathcal{A}$) is "non-modular" for C in the context of P.

From now on, if φ, $\mathcal{A}$, C, and P are fixed and clear, we write $\mathcal{D}$ and $\mathcal{C}$, for the dependency context and the complexity, respectively. Similarly, we write $\overline{\mathcal{C}}$ for the total complexity.

B. HTL modularity aspects

We characterize HTL modularity for all cases of interest. For each property φ, from the compilation stages of Figure 11, we discuss an algorithm $\mathcal{A}$ used to validate φ on P. Before presenting the modularity of the property, i.e., $\mathcal{D}$ and $\mathcal{C}$ for all possible components, we first discuss $\overline{\mathcal{C}}$, the total complexity of checking φ by $\mathcal{A}$. Note that in case of a dependency between properties, i.e., an arrow in Figure 11, we only consider the effort of checking the property of interest and assume that the needed predecessor property has already been checked. For example, in order to establish transmission safety a program needs to be race free (and in order to be race free, it must be well-formed). When discussing the complexity of the transmission-safety check, we assume the program is already proven race free (and hence also well-formed), and consider only the complexity that is essential to the transmission safety check.

Our results are summarized in Table I, showing that top-level time safety is non-modular, whereas all other properties are modular. Moreover, refinement-level race freedom, transmission safety, refinement-level time safety, and code generation are fully modular properties. In the rest of the paper we discuss the results presented in Table I. Let us start by introducing the needed notation.

A program P has a refinement height r_P, and the following upper bounds on its size: n_c communicators, n_M top modules, n_m modes per top module, n_T tasks per mode, n_w communicator writes per task, n_a communicator accesses per task, n_p ports per task, and periods of modes with maximum value $\Delta_{\max}$. We also use the notation $n^P_{m\downarrow}$ for the bound on the total number of modes in P calculated as $n^P_{m\downarrow}(r_P) = (n_M \cdot n_m)^{r_P+1}$, and $n^P_{T\uparrow}$ for the bound of the total number of top-level tasks in P calculated as $n^P_{T\uparrow} = n_M n_m n_T$.

We distinguish between top-level and refinement-level components. A top-level component can have a root node module, mode, or task. A refinement-level component can have a root node program, module, mode, or task. Similar as for programs, r_C denotes the refinement height of a component C, $n^C_{m\downarrow}$ an upper bound on the total number of modes in it, and $n^C_{T\uparrow}$ an upper bound on the number of top-level tasks. These two notions can be calculated for each type of a component. For example, if C is a refinement program, then $n^C_{m\downarrow} = (n_M \cdot n_m)^{r_C+1}$ and $n^C_{T\uparrow} = 0$; whereas if C is a top-level component with root node a module, then $n^C_{m\downarrow} = (n_M \cdot n_m)^{r_C} \cdot n_m$ and $n^C_{T\uparrow} = n_m\, n_T$.

For simplicity, we assume that the mode switching function of each module induces a fully connected mode-switching graph. This is not a restriction but actually creates a worst-case scenario since it subsumes all possible mode-switching behaviors.

Well-formedness. The verification of well-formedness of a program P with respect to the LET and refinement constraints discussed in Section IV is performed inductively on the structure of the program and it can be achieved in linear time (one pass) depending on the product of the total number of modes, number of tasks per mode, and number of ports per task, i.e. $\overline{\mathcal{C}} = \mathrm{O}(n^P_{m\downarrow}(r)\ n_T\ n_p)$. Checking other syntactic constraints is performed in less time, and therefore subsumed by this complexity bound.

For modular compilation, the well-formedness check is also linear, now in the size of the component. The dependency context is C itself (although each refinement task needs to be compared also to its parent task) and the modular complexity is $\mathcal{C} = \mathrm{O}(n^C_{m\downarrow}(r_C)\ n_T\ n_p)$.

Race freedom. A race happens in the execution of a program when two concurrent task invocations write the same communicator at the same time, compromising time-determinism.

We consider a simple race-free sufficiency algorithm. It uses the fact that if no two concurrent tasks write to the same communicators, regardless of the time when the communicators are written, then race freedom is ensured. Moreover, using condition (4) of the refinement constraints in Section IV-A, we observe that two concurrent tasks in an HTL program write to the same communicators if and only if there exist two top-level concurrent tasks in the program that write the same communicators. Hence, for race freedom it is sufficient to check that no two top-level concurrent tasks write to the same communicators. At top level, concurrent tasks are any two tasks within the same mode, or any two tasks in different top-level modules.

More precisely, let $\texttt{write}(T)$ be the set of communicators to which task T writes, and let $\texttt{write}(M)$ be the set of communicators written by any task in any mode of M. We need to ensure that

1. for all top-level modules M_1 and M_2 we have $\texttt{write}(M_1) \cap \texttt{write}(M_2) = \emptyset$, and
2. for every top level mode m, and each two tasks T_1 and T_2 of m, it holds that $\texttt{write}(T_1) \cap \texttt{write}(T_2) = \emptyset$.

The check of 1. can be done in linear time in the number of modules and number of communicators, whereas the check of 2. is linear in the total number of writes of all top-level task. Hence, $\overline{\mathcal{C}} = \mathrm{O}(n^P_{T\uparrow}\ n_w + n_M\ n_c)$.

As for the modularity cases of interest, if C is a well-formed refinement-level component, then there is nothing to check: the dependency context is C, and the complexity is constant time, actually zero time. If C is a top-level component, then re-checking race freedom requires checking 2. for the tasks in C and re-checking 1. Hence, the dependency context is the whole program P, but not all of P needs to be re-checked. There is still some modularity involved, the top-level tasks outside C need not be re-checked. As a result, we get that $\mathcal{C} = \mathrm{O}(n^C_{T\uparrow}\ n_w + n_M\ n_c)$.

Transmission safety. Transmission safety requires that the networked communication of a program, i.e., the broadcast of communicator values, is schedulable, meaning that any transmission of a value of any communicator fits into the communicator's period instance.

In the worst-case transmission scenario of a race-free program, every communicator is updated at every period. Since transmitting an update takes time at most the WCTT of the communicator, scheduling the network communication amounts to scheduling a set of periodic tasks (one per each

φ	C	$\mathcal{D}^{\mathcal{A}}_{\varphi}(C,P)$	$\mathcal{C}^{\mathcal{A}}_{\varphi}(C,P)$	$\overline{\mathcal{C}}^{\mathcal{A}}_{\varphi}(P)$
Well-formedness	any	C	$n^C_{m\downarrow}\ n_T\ n_p$	$n^P_{m\downarrow}\ n_T\ n_p$
Race freedom	top	P	$n^C_{T\uparrow}\ n_w + n_M\ n_c$	$n^P_{T\uparrow}\ n_w + n_M\ n_c$
	ref.	C	1	
Transmission safety	any	C	1	n_c
Time safety	top	P	$(n_m\ \Delta_{max})^{n_M}$	$(n_m\ \Delta_{max})^{n_M}$
	ref.	C	1	
Code generation	any	C	$n^C_{m\downarrow}\ (n_T\ n_a + n_m)$	$n^P_{m\downarrow}\ (n_T\ n_a + n_m)$

n_a	number of communicator accesses per task	n_c	number of communicators	n_M	number of modules per program
n_m	number of modes per module	n_p	number of ports per task	n_T	number of tasks per mode
n_w	number of communicator writes per task	$n^C_{m\downarrow}$	total number of modes in C	$n^P_{m\downarrow}$	total number of modes in P
$n^C_{T\uparrow}$	number of top-level tasks in C	$n^P_{T\uparrow}$	number of top-level tasks in P	$\Delta_{\max}$	maximal value of mode periods

TABLE I
MODULARITY ASPECTS OF HTL

communicator) with execution times equal to the corresponding communicator WCTTs and periods equal to the corresponding communicator periods.

Hence, for network architectures that are flexible with respect to the choice of a scheduling algorithm, a simple utilization based scheduling test and algorithm (e.g. EDF or rate-monotonic scheduling), linear in the number of communicators in the system, suffices to establish transmission safety, $\overline{\mathcal{C}} = \mathrm{O}(n_c)$. An example of such a flexible architecture is an FFT-CAN bus [13]. Also statically scheduled networks like the simplest forms of TDMA buses may allow for a simple utilization-based schedulability check.

Transmission safety is fully modular although the communicators in a program are global: if the original program P_o is transmission safe, then all communicators have already been taken into account in P_o and a change of a component does not require any additional check. So, the dependency context is always C and $\mathcal{C} = \mathrm{O}(1)$.

Time safety. For time safety, a program needs to be checked for schedulability of computation, taking into account the target platform (WCETs) and the program specification (LETs). Time safety can be checked independently from the network communication (the transmission-safety check). The main HTL schedulability result, enabled by the refinement constraints, is that time safety of any program P is ensured by schedulability of all (abstract and concrete) top-level tasks. In such a case, any schedule for the top-level tasks is robust and sustainable (in the sense of e.g. [14], [15]) to a replacement of an abstract task by any of its refinements.

For schedulability at the top level, standard periodic-task scheduling techniques can be used (c.f. [16]). In the absence of mode switching, with a single mode per module, the top level of P is a periodic-task system with task precedences and the complexity of checking top-level schedulability is exponential in the mode periods: $\mathrm{O}(\Delta^{n_M}_{max})$ [17]. This can be improved for special cases, for which there are tractable schedulability tests and algorithms, e.g. synchronous periodic-task systems with deadlines equal to or less than periods. In the general case with mode switching, the top-level schedulability check must consider in the worst case $\mathrm{O}({n_m}^{n_M})$ combinations of concurrent modes at the top level. There is no need to check anything but well-formedness for refinement programs. Hence, the total complexity for checking time safety of a program P is $\overline{\mathcal{C}} = \mathrm{O}((n_m\ \Delta_{max})^{n_M})$.

Let us consider the two cases of interest for modularity analysis. If C is a well-formed refinement-level component, then $\mathcal{D} = C$ and the modular complexity is $\mathcal{C} = \mathrm{O}(1)$ since there is nothing to check. If C is a top-level component, the top-level time-safety analysis of P must be fully re-checked, i.e., time safety is non-modular in this case with dependency context $\mathcal{D} = P$ and modular complexity $\mathcal{C} = \mathrm{O}((n_m\ \Delta_{max})^{n_M})$.

Code generation. There is a flattening HTL compiler targeting the E machine [1] and a modular, hierarchy-preserving HTL compiler targeting the HE machine [3]. A detailed comparison of both compilers can be found in [5]. We highlight the differences, and characterize the complexity and modularity for the HE-compiler.

The more recent HE-compiler allows modular code generation. The worst-case code size per mode generated by the HE-compiler depends linearly on the number of communicator accesses (n_a) for each task and the number of modes in the same module (for the evaluation of mode switching), c.f. [3] for more details. Hence, the size of the generated code per mode is $\mathrm{O}(n_T\ n_a + n_m)$ and the overall complexity of generating code for P is $\overline{\mathcal{C}} = \mathrm{O}(n^P_{m\downarrow}(r_P)\,(n_T\,n_a + n_m))$. The code for each component C can be generated independently from the code for the rest of the program. So, we have that for any component C, $\mathcal{D} = C$ and $\mathcal{C} = \mathrm{O}(n^C_{m\downarrow}(r_C)(n_T n_a + n_m))$. To be precise, this holds for any component that contains at least one mode. For the corner case of a component with root node task, the modular complexity simplifies to $\mathcal{C} = \mathrm{O}(n_a)$.

Beyond code generation, there is the aspect of code distribution across multiple hosts. In the HTL characterization in [1] different modules exchanged values of ports, rather than values of communicators. As a consequence of exchanging values of ports, the code for (almost) the entire program had to be

executed on all hosts. With the current approach, originating in the decoupling of task executions from network transmissions, only the code of modules mapped to a host needs to execute on that host leading to fully modular code distribution.

VI. Related work

The foundation for HTL is the LET model, introduced in previous work on Giotto [7] and the E machine [9]. Modular refinement-level time safety checking was shown in the original work on HTL [1]. Modular code generation was described for HTL in [3], [5], and for Giotto in [18]. Modular code generation is a relevant issue also in synchronous reactive programming languages, for it allows reusability and integration of components, but calls for different solutions due to the zero-time rather than LET semantics [10]. The quest for verifiable, predictable, and compositional time-determinism of distributed programs, motivated the work on "network code" programs [19], distributed synchronous reactive programs on so-called "loosely"-time-triggered architectures (LTTA) [20], and further back, work on TTA [21], [22].

Exact schedulability analysis of real-time programs is a complex problem, which motivated the development of sustainable [23], compositional [23], [24], and "interface"-based [25], [26] techniques.

There is significant research interest in component-based real-time software in general, and its modular compilation. We highlight some examples of recent work in this area. BIP components [27] provide a framework where correctness of components is inferred compositionally by properties of sub-components. BIP also works as a "verification back-end" for other component-based systems such as AADL [28] and synchronous languages such as Lustre [29]. Ptolemy actors [11] allow for the composition of heterogeneous models of computation and have been considered in verification [30]. Relational interfaces [31] endow (real-time) interfaces [32] with synchronous input-output relations and are compositional with respect to refinement.

VII. Conclusion

We have presented a modular abstract syntax and semantics for HTL, modular checks of well-formedness, race freedom, and transmission safety, and modular code distribution. Our contributions here complement previous results on HTL time safety and modular code generation, and complete the study of modularity in HTL. Modularity in HTL can be utilized in easy program composition as well as fast program analysis and code generation, but also in runtime patching, which may eventually be used as software foundation for addressing uncertainty in control systems. While there is already an implementation of a modular HTL compiler, an implementation of runtime patching for HTL is still future work.

References

[1] A. Ghosal, T. Henzinger, D. Iercan, C. Kirsch, and A. Sangiovanni-Vincentelli, "A hierarchical coordination language for interacting real-time tasks," in Proc. EMSOFT, 2006.

[2] K. Chatterjee, A. Ghosal, D. Iercan, C. Kirsch, T. Henzinger, C. Pinello, and A. Sangiovanni-Vincentelli, "Logical reliability of interacting real-time tasks," in Proc. DATE, 2008.

[3] A. Ghosal, D. Iercan, C. Kirsch, T. Henzinger, and A. Sangiovanni-Vincentelli, "Separate compilation of hierarchical real-time programs into linear-bounded embedded machine code," in Proc. APGES, 2007.

[4] C. Kirsch, L. Lopes, and E. Marques, "Semantics-preserving and incremental runtime patching of real-time programs," in Proc. APRES, 2008.

[5] D. Iercan, "Contributions to the development of real-time programming techniques and technologies," Ph.D. dissertation, Politehnica University of Timisoara, 2008.

[6] D. Stewart, R. Volpe, and P. Khosla, "Design of dynamically reconfigurable real-time software using port-based objects," IEEE Transactions on Software Engineering, 1997.

[7] T. Henzinger, B. Horowitz, and C. Kirsch, "Giotto: A time-triggered language for embedded programming," Proc. of the IEEE, 2003.

[8] J. Auerbach, D. Bacon, D. Iercan, C. Kirsch, V. Rajan, H. Röck, and R. Trummer, "Low-latency time-portable real-time programming with Exotasks," ACM TECS, 2009.

[9] T. Henzinger and C. Kirsch, "The Embedded Machine: predictable, portable real-time code," in Proc. PLDI, 2002.

[10] R. Lublinerman, C. Szegedy, and S. Tripakis, "Modular code generation from synchronous block diagrams — modularity vs. code size," in Proc. POPL, 2009.

[11] J. Eker, J. Janneck, E. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong, "Taming heterogeneity—the Ptolemy approach," Proc. of the IEEE, 2003.

[12] HTL site, http://htl.cs.uni-salzburg.at.

[13] L. Almeida, P. Pedreiras, and J. Fonseca, "The FTT-CAN protocol: Why and how," IEEE Trans. on Industrial Electronics, 2002.

[14] M. Anand and I. Lee, "Robust and sustainable schedulability analysis of embedded software," in Proc. LCTES, 2008.

[15] S. Baruah and A. Burns, "Sustainable scheduling analysis," in Proc. RTSS, 2006.

[16] G. Buttazzo, Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications. Norwell, MA, USA: Kluwer Academic Publishers, 1997.

[17] J. Leung and M. Merrill, "A note on preemptive scheduling of periodic, real-time tasks," Information Processing Letters, 1980.

[18] T. Henzinger, C. Kirsch, and S. Matic, "Composable code generation for distributed Giotto," in Proc. LCTES, 2005.

[19] S. Fischmeister, O. Sokolsky, and I. Lee, "A verifiable language for programming real-time communication schedules," IEEE Trans. on Computers, 2007.

[20] S. Tripakis, C. Pinello, A. Benveniste, A. Sangiovanni-Vincent, P. Caspi, and M. Di Natale, "Implementing synchronous models on loosely time triggered architectures," IEEE Trans. on Computers, 2008.

[21] H. Kopetz and G. Bauer, "The Time-Triggered Architecture," Proc. of the IEEE, 2003.

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

[23] A. Easwaran, M. Anand, and I. Lee, "Compositional analysis framework using EDP resource models," in Proc. RTSS, 2007.

[24] L. Thiele, E. Wandeler, and N. Stoimenov, "Real-time interfaces for composing real-time systems," in Proc. EMSOFT, 2006.

[25] R. Alur and G. Weiss, "RTComposer: a framework for real-time components with scheduling interfaces," in Proc. EMSOFT, 2008.

[26] T. Henzinger and S. Matic, "An interface algebra for real-time components," in Proc. RTAS, 2006.

[27] S. Bensalem, M. Bozga, J. Sifakis, and T. Nguyen, "Compositional verification for component-based systems and application," in Proc. ATVA, 2008.

[28] M. Chkouri, A. Robert, M. Bozga, and J. Sifakis, "Translating AADL into BIP-application to the verification of real-time systems," in Proc. MoDELS Workshops, 2008.

[29] M. Bozga, V. Sfyrla, and J. Sifakis, "Modelling synchronous systems in BIP," in Proc. EMSOFT, 2009.

[30] Y. Zhou and E. A. Lee, "Causality interfaces for actor networks," ACM TECS, 2008.

[31] S. Tripakis, B. Lickly, T. Henzinger, and E. Lee, "On relational interfaces," in Proc. EMSOFT, 2009.

[32] L. de Alfaro and T. A. Henzinger, "Interface theories for component-based design," in Proc. EMSOFT, 2001.