A Hierarchical Coordination Language for Interacting Real-Time Tasks

Arkadeb Ghosal, Thomas A. Henzinger, Daniel Iercan,
Christoph M. Kirsch, Alberto Sangiovanni-Vincentelli
presented by: Hannes Payer

University of Salzburg
Compositionality Seminar WS 07/08

December 6, 2007

Real-Time Programming Difficulties

- trial and error - if during a program test some task misses its deadline ⇒ reassigning of task priorities
- prove timing of a program using scheduling theory and/or formal verification
- scheduling analysis becomes difficult when the program structure is irregular
- formal techniques are difficult due to state space explosion
- part of the problem: timing is often defined in an indirect way, through low-level constructs (priorities)

HTL

- HTL . . . Hierarchical Timing Language
- HTL is a programming language for hard real-time systems
- critical timing constraints are specified within the language, and ensured by the compiler
- high-level coordination language for interacting hard real-time tasks

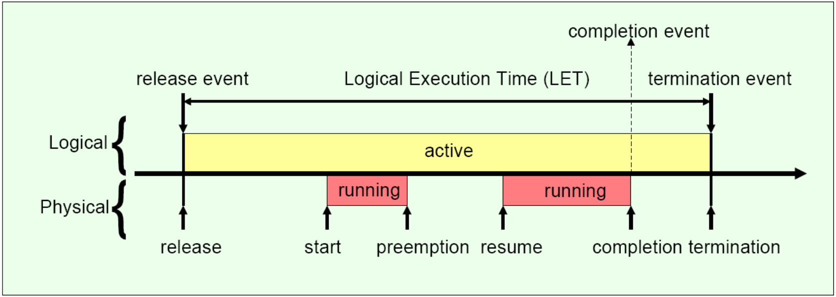
HTL

- HTL programs determine portable and predictable real-time behavior of periodic software tasks running on a possibly distributed system of host computers
- individual tasks can be implemented in “foreign” languages
- more general than Giotto because it offers hierarchical layers of abstraction

Tasks

- the computational unit of HTL are LET tasks
- LET model decouples the times when the task reads input and writes output from the time when the task executes
- release and termination events, which are triggered by clock ticks or sensor interrupts, determine the LET of the task
- a LET task is time-safe if it completes execution before the termination event occurs (on some given hardware)
- time-safe LET tasks are time and value deterministic, portable and composable

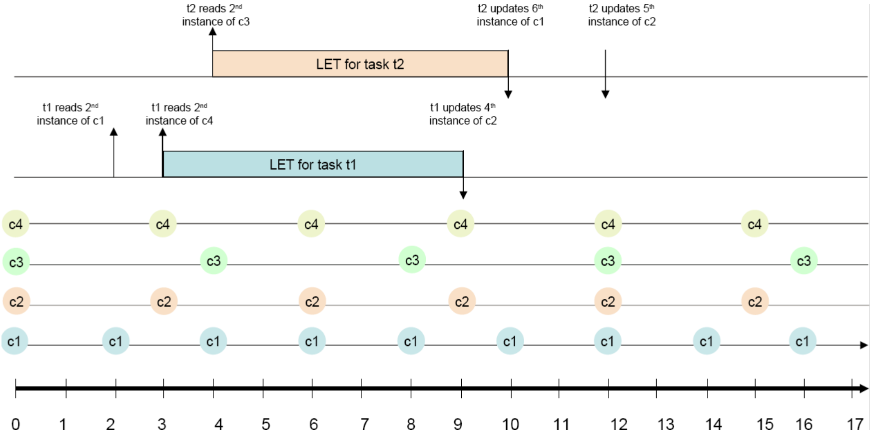
LET


completion event
release event
Logical Execution Time (LET)
termination event
Logical
active
Physical
running
running
release
start
preemption
resume
completion
termination

Communicators

- the communication model for HTL is centered around communicators
- a communicator is a typed variable that can be accessed only at specific time instances
- time instances are periodic and specified through a communicator period
- sensors and actuators are communicators, but communicators can also be used to exchange data between tasks
- the latest read instance determines the release time
- the earliest write instance determines the termination time

Communicators



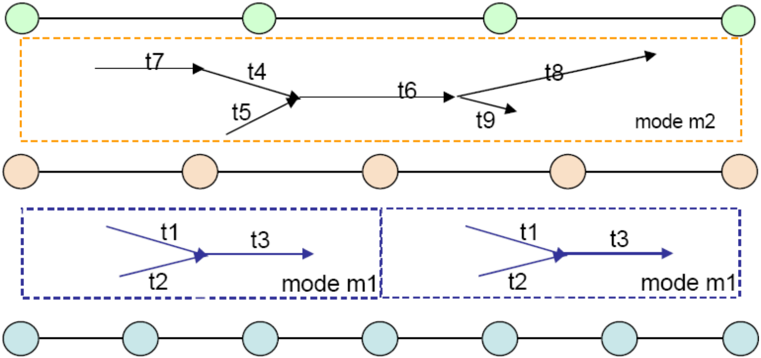
Ports

- direct communication between tasks is allowed for tasks with identical frequencies
- tasks with different frequencies can only communicate via communicators
- direct communication ensures zero latency
- a port is a variable with fixed data type but not bound to time instances

Modes

- HTL generalizes the LET model from tasks to task groups
- a set of interacting tasks with the same frequency form a mode with a specified mode period
- tasks within a mode may interact through ports
- tasks in different modes can only communicate through communicators

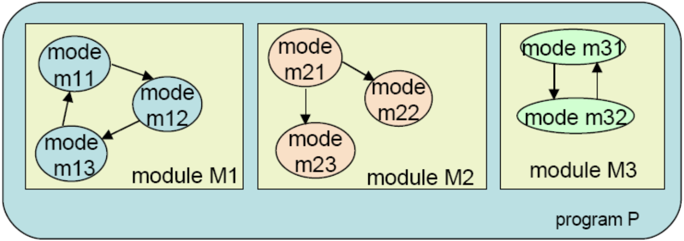
Modes



Modules

- a mode switch can occur at the end of a mode period, which are triggered by conditions on communicator and port values
- a network of modes and mode switches is called a module
- an HTL program is a set of modules and a set of communicators
- modes within a module are composed sequentially
- modes from different modules are composed in parallel and may interact through communicators
- one mode in each module is specified as the start mode

Modules & Modes



Refinement I

- an abstract task is a temporally conservative placeholder for a concrete task with an implementation
- an abstract task has a frequency, specific I/O times, dependencies, and WCET, but no implementation
- refinement is useful for compact representation and simplifying program analysis
- an HTL program is schedulable if the top-level program (without considering any refinement) is schedulable ⇒ avoids a combinatorial explosion
- an HTL program with multiple levels of refinement can be translated into an equivalent flat program without refinement

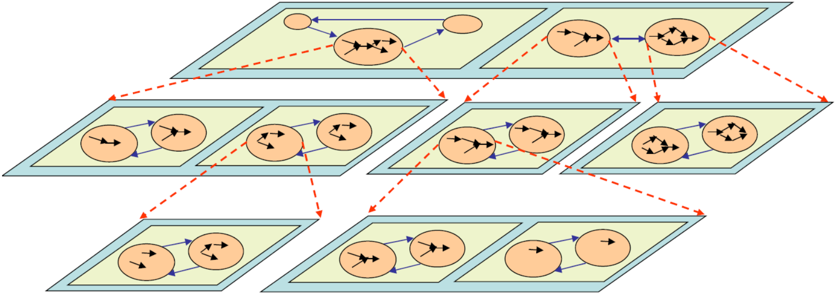
Refinement II

- 3 constraints ensure that if the task in the top-level (abstract) program can be scheduled, also its refinement (concrete) task can be scheduled:
  - the latest communicator read must be equal to or earlier then that of the top-level task
  - every task that precedes the refined task must refine a task that precedes the abstract task
  - the WCET of the refined task must be less than or equal to the WCET of the abstract task

Refinement III

- refinement can represent both choice and change of behavior
  - choice: is expressed when an abstract task t in a mode m is the parent of different tasks in several modes of a program that refines m
  - change: is expressed by having a task that refines t reading from and writing to different communicators than t does

Modules & Modes



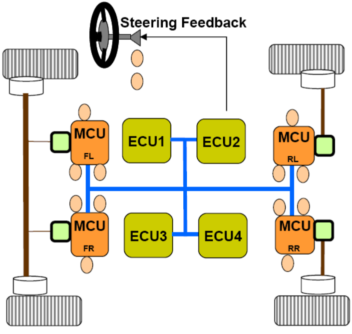
Distribution

- different modules can run on different hosts
- several hosts interact with each other through communication channels
- distribution is specified through a mapping of top-level modules to hosts
- code generation and schedulability analysis must take the distribution into account
- the LET model is extended to include both WCETs as well as worst-case output transmission times WCTTs

Extensibility/Compositionality

- parallel modules can be appended to the implementation without changing the timing behavior of the implementation (horizontally)
- the refinement concept can be used to provide temporal space for future extensions (vertically)

Steer-By-Wire



● Sensors: steer angle/ torque, wheel angle, motor torque/ current, friction, power, pitch, yaw, roll

🟩 Actuators: rack electric motors

SENSORS

Driver Interface

Desired steer
Desired Torque

Vehicle Interface

Wheel Angle
Motor Current
Motor Torque
Speed
Rolling Friction
Power
Pitch/ Yaw/ Roll

FUNCTIONALITIES

- Power Coordinator Unit
- Motor Actuator Controller
- Steer Feedback
- Supervisory Control
- Fault Handling

ACTUATORS

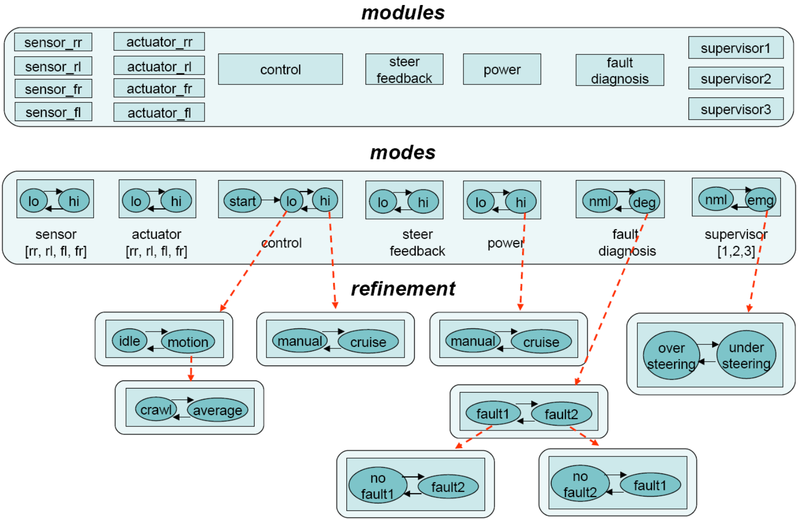
Vehicle Actuation

Wheel Motor
Actuation

Driver Feedback

Steer Feedback
Warning

Steer-By-Wire



modules

sensor_rr
sensor_rl
sensor_fr
sensor_fl
actuator_rr
actuator_rl
actuator_fr
actuator_fl
control
steer feedback
power
fault diagnosis
supervisor1
supervisor2
supervisor3

modes

lo hi
sensor [rr, rl, fl, fr]

lo hi
actuator [rr, rl, fl, fr]

start lo hi
control

lo hi
steer feedback

lo hi
power

nml deg
fault diagnosis

nml emg
supervisor [1,2,3]

refinement

idle motion

crawl average

manual cruise

manual cruise

over steering under steering

fault1 fault2

no fault1 fault2

no fault2 fault1

Implementation

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
- generates E code for the flattened program targeting the E machine

Well-Formedness, Well-Timedness, Schedulability

The compiler . . .

- verifies that any concrete task refines its parent task
- performs an EDF-scheduling test on the abstract, top-level portion of the input program
- adds the WCTT for broadcasting the output port values of each task to the WCET of the task (distributed HTL programs)

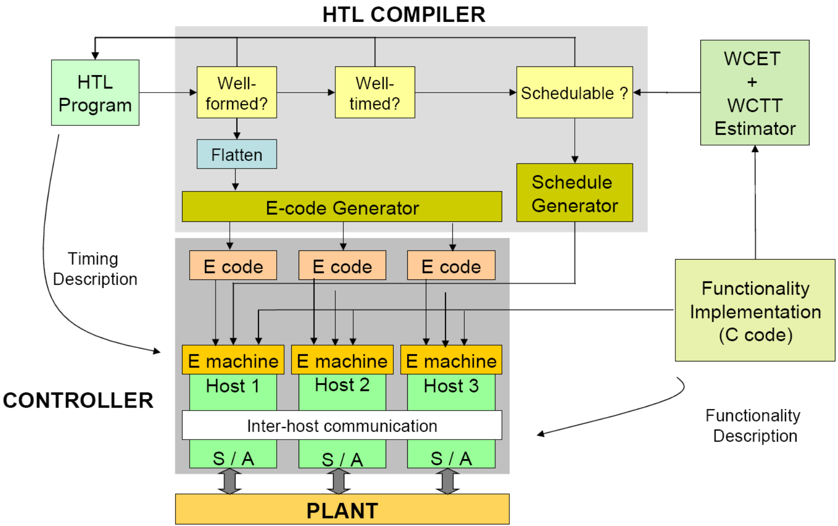
Flattening

- flattening works by essentially computing the product of all modes in the refinement of each top-level module of the original program
- mode switches in more abstract modules need to be checked before mode switches in more concrete modules
- flattening an HTL program may in theory result in generated code that is exponentially larger than the size of the input program
- APGES 2007: *Separate Compilation of Hierarchical Real-Time Programs into Linear-Bounded Embedded Machine Code*

Schedulability

- the schedulability problem is solved only for the top-level
- scheduling task execution during time slots in which the parent task is executed
- HTL guarantees that top-level schedulability is a sufficient condition for schedulability
- EDF scheduling algorithm is used for top-level schedulability on a single host

Implementation



Conclusion

- HTL 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
- lower levels are schedulable if higher levels of abstraction are schedulable
- in general, checking refinement in HTL is exponentially faster then checking time-safety (schedulability)
- abstract HTL programs are temporally conservative approximations of concrete HTL programs