Event Driven Real-Time Programming

CHESS Review
University of California, Berkeley, USA

May 10, 2004

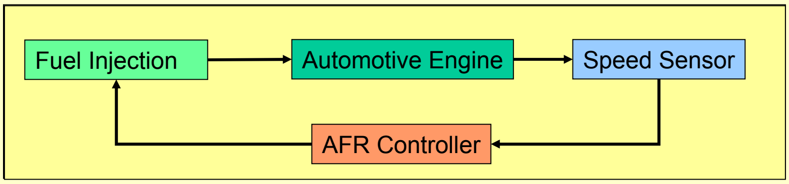
Arkadeb Ghosal
Joint work with Marco A. Sanvido, Christoph M. Kirsch and Thomas A. Henzinger

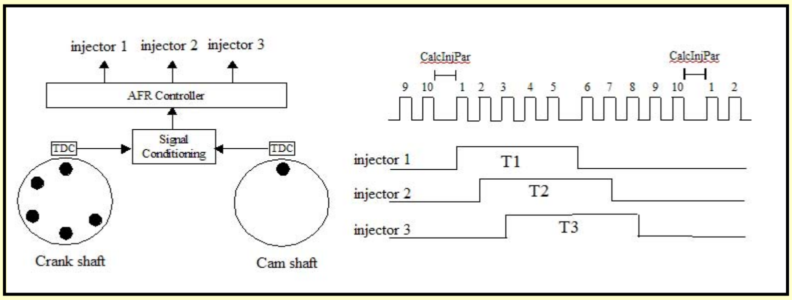
University of California, Berkeley

Overview

- Introduction
- Language Features
  - The LET model
  - Language Constructs
  - Event Scoping
- Analysis
- Implementation
- Ongoing Work

Control System





Implementation Strategies

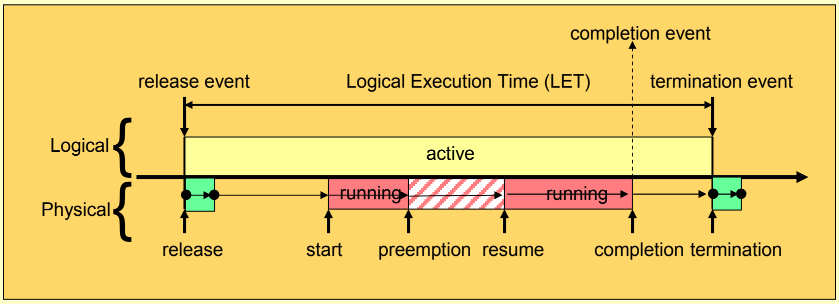
- Traditional
  - Uses priorities to specify the relative deadlines of software tasks
  - Supports efficient code generation based on scheduling theory
  - Run-time behavior is highly non-deterministic
- Synchronous Languages
  - Esterel, Lustre, Signal
  - Based on synchrony assumption: task computation takes negligible execution time
  - Shows deterministic behavior
- Timed Languages
  - Based on Logical Execution Time (LET) for tasks
  - Giotto
    - Time Triggered
  - xGiotto
    - Event Triggered
    - Scoping of events

Logical Execution Time

The logical and physical execution times are depicted below. The events controlling a task behavior are:

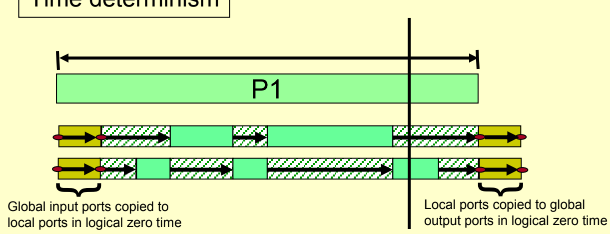
Event generated by the environment:
- release
- termination

Events generated by the platform:
- start
- preempt
- resume
- completion



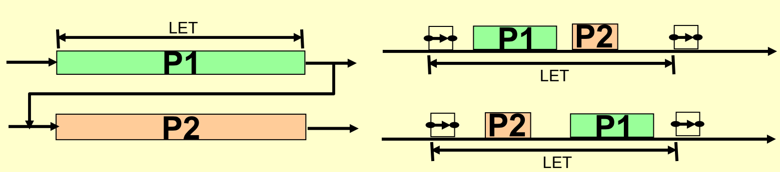
Logical Execution Time

Time determinism



Value of output port remains invariant at any instant independent of execution pattern

Value determinism



Reactions and Triggers

- A trigger maps an event to a reaction
  - When the event occurs the reaction is invoked
- A reaction defines
  - New triggers
  - A termination event
- Events of a reaction block are the events of its triggers and the termination event
  - A reaction block defines a scope for its events
- When a trigger is invoked
  - The events of the new reaction block are enabled (active)
  - The events of the callee reaction block become passive (inactive)

Reaction Block

Reaction Name

```
react R1 {
    when [e2] react R2;
    when [e3] react R3;
} until [e7];
```

Triggers

Until Event

Events of R1: e2, e3, e7

Scope of e2, e3, e7: R1

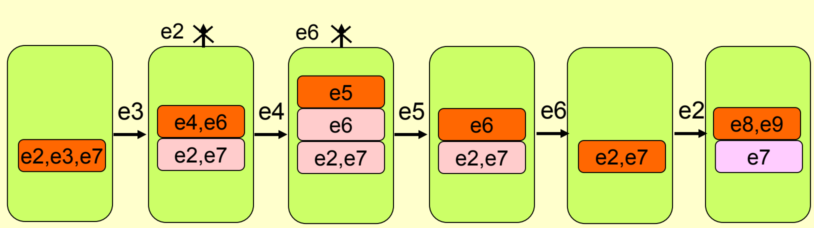
Reactions and Triggers

```
react R1 {
    when [e2] react R2;
    when [e3] react R3;
} until [e7];
```

```
react R3 {
    when [e4] react R4;
} until [e6];
```

```
react R4 {
} until [e5];
```

```
react R2 {
    when [e8] react R5;
} until [e9];
```



Tasks

- Tasks instances are defined by release statements
- Tasks instances
  - released with the invocation of the reaction block
  - terminated with the termination of the reaction block
- LET of the task is given by the life-span of the reaction block

Task Name

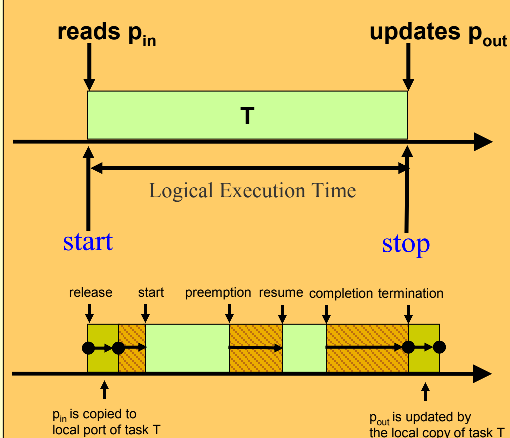
Input Port

Release Statement

```
react R1 {
  release t1 (i1) (o1);
  when [e2] react R2;
  when [e3] react R3;
} until [e7];
```

Output Port

Releasing Tasks




```
port
   pin;
   pout;
event
   start;
   stop;
task T (i) output (o)
   {/* compute */}
react R {
   release T(pin)(pout);
} until [stop]
{ when [start] react R;}
```

Tasks

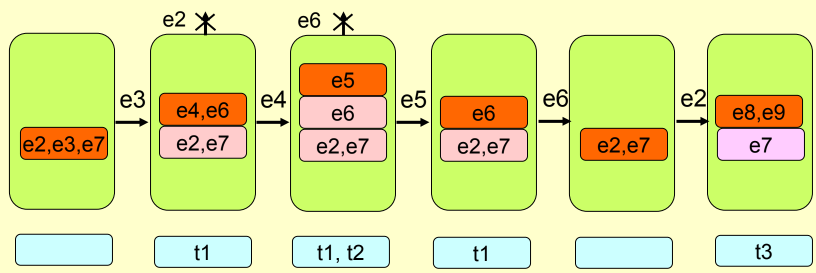
- Tasks are released with the invocation of the reaction block
- Tasks are terminated with the termination of the reaction block

```
react R1 {
    when [e2] react R2;
    when [e3] react R3;
} until [e7];
```

```
react R3 {
    release t1;
    when [e4] react R4;
} until [e6];
```

```
react R4 {
    release t2;
} until [e5];
```

```
react R2 {
    release t3;
    when [e8] react R5;
} until [e9];
```



Handling Events

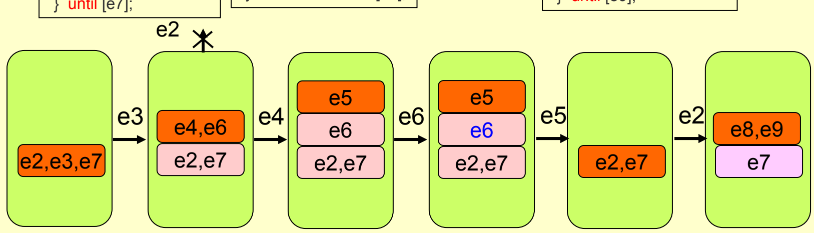
- A reaction block defines a scope: this implicitly denotes the scope of an event
- When an active trigger is invoked, the called reaction becomes the active scope and the caller reaction, the passive scope
- The event of a passive scope can be
  - Ignored (**forget**)
  - Postponed until its scope becomes active again (**remember**)

```
react R1 {
    when [e2] react R2;
    when [e3] react R3;
} until [e7];
```

```
react R3 {
    when [e4] react R4;
} until remember [e6];
```

```
react R4 {
} until [e5];
```

```
react R2 {
    when [e8] react R5;
} until [e9];
```



Parallelism

- A trigger may invoke multiple reaction blocks in parallel.
- When the trigger is invoked all the reactions become active simultaneously.
- The parent block is active only when all the parallel reaction blocks have terminated.

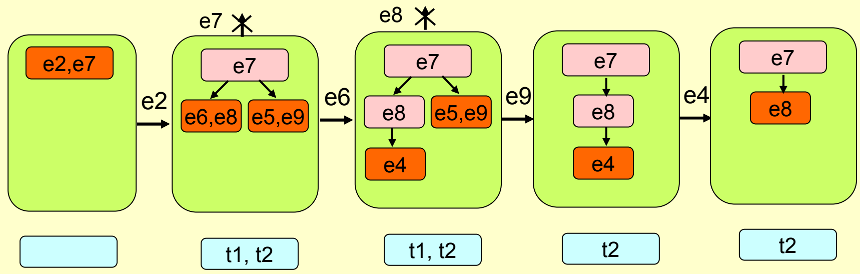
```
react R1 {
   when [e2] react R2 || R3;
} until [e7];
```

```
react R3 {
   release t1;
   when [e5] react R5;
} until [e9];
```

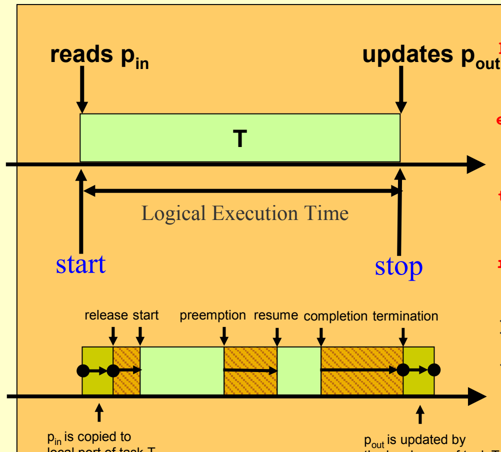
```
react R2 {
   release t2;
   when [e6] react R6;
} until [e8];
```

```
react R6 {

} until [e4];
```



Environment Assumption




```
port
   pin;
   pout;
event
   start;
   stop;
task T (i) output (o)
   {/* compute */}

react R {
   release T(pin)(pout);
   when [now] react {} until [3ms];
} until [3ms]stop]

{ when [start] react R; }
```

xGiotto: Basic Constructs

- Reaction Blocks
  - Basic programming blocks in xGiotto
  - Consists of release statements and trigger statements along with an termination information
  - Releases tasks and invokes triggers
    - react {reaction block} until [event];

```
reaction() {
    release task1 (i1) (o1);
    release task2 (i2) (o2);
    when event1 react block1;
    whenever event2 react block2;
} until event;
```

- Release Instruction
  - Tasks are released with the invocation of the reaction block
  - Tasks are terminated with the termination of the reaction block
    - release task (input ports) (output ports);
- Trigger Statements
  - Defines the invoking action associated with an event
    - when [event] reaction block;
  - Repetition construct using whenever

Structuring Events

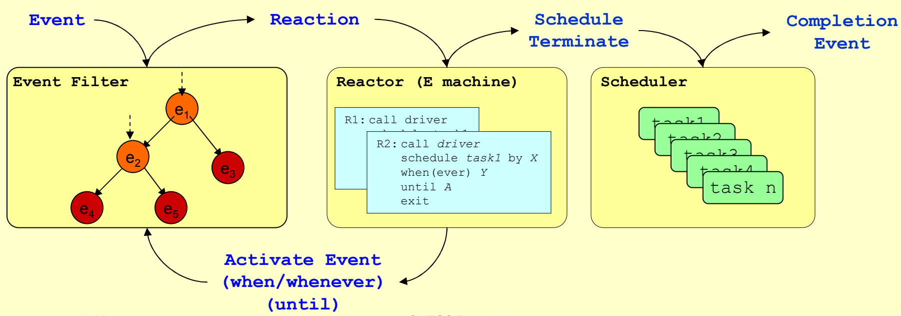
- Scoping of events
  - A reaction block defines a scope: this implicitly denotes the scope of an event.
  - When an active trigger is invoked, the called reaction becomes the active scope and the caller reaction, the passive scope.
  - The tree of scopes and the state of program variables denotes the state of the program.
- Handling of events (of a passive scope)
  - It may be ignored (**forget**)
  - It may be postponed until its scope becomes active again (**remember**)
  - It may disable trigger statements of all descendent blocks and thus speeding up their termination (**asap**)
- Invoking reactions in parallel
  - Wait-parallelism
  - Asap-parallelism
- Embedding Environment Assumption
  - Event calculus

The Program Flow

Event Filter:
The Event Filter implements the event scoping mechanism and filter the incoming event. It determines which event needs to be reacted upon depending upon the event qualifiers – forget, remember or asap .

Reactor:
The Reactor executes the specified reaction and activates new events (when/whenever/until) and activates and terminates tasks (release).

Scheduler:
The Scheduler chooses from the active tasks, a task to be executed on the given platform (CPU). The scheduler generates an event at task completion.



AFR Controller

port
/* fuel ports */
/* pulse ports */

event
teeth; synch; stop;

task set { /* opens the valve */ }
task reset { /* closes the valve */ }
task dec {/* pulse generation */}

task CalcFuelInj
{/* fuel parameter computation */}

```
react channel2 {
  react { } until [5ms : teeth];
  when remember [5ms : teeth] react {release set} until [ms];
  react
    loop react {release reset; dec;  } until [ms];
} until asap [50ms : 9teeth]
```

```
react calcFuel { release CalcFuelInj; } until [10ms : teeth];
```

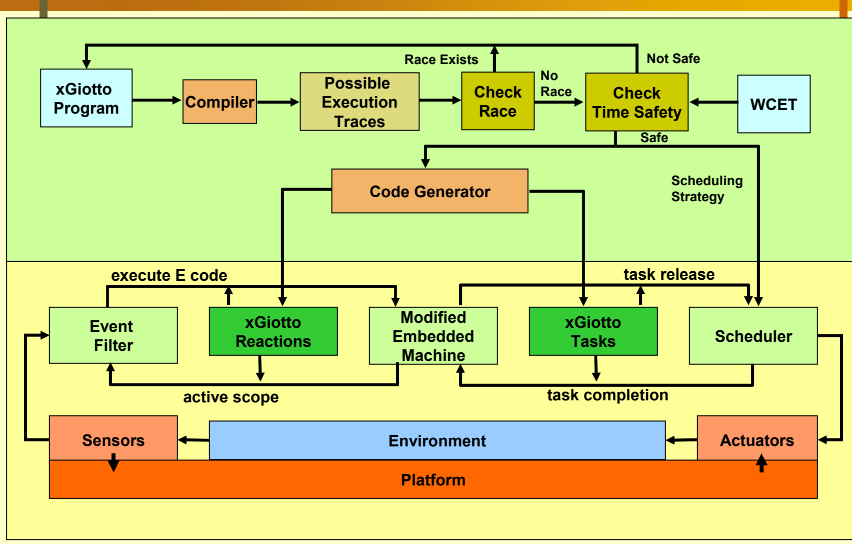
```
react controller {
   react calcFuel;
   when remember [teeth]
        react channel1 ||  react channel2  || … ;
} until remember [10teeth]
```

```
react start {
   whenever remember [10teeth] react controller;
} until [stop];
```

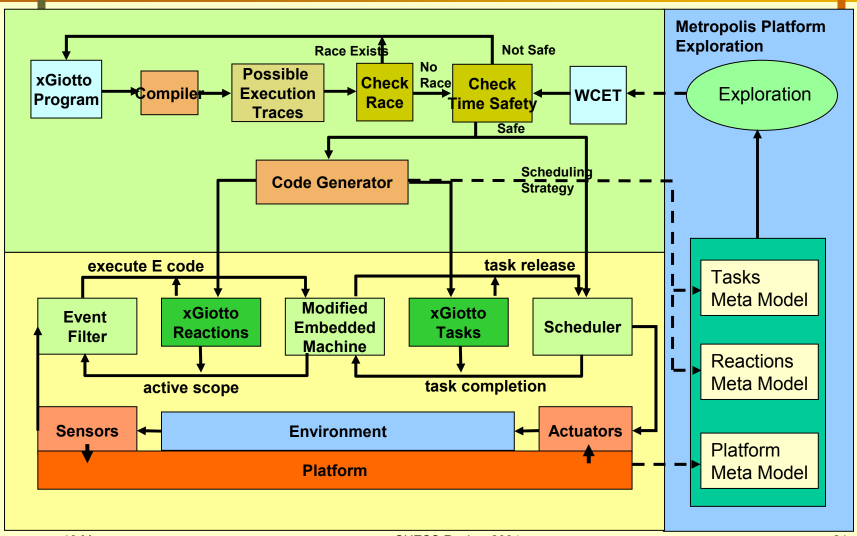
Analysis

- Platform independent
  - Race Condition Detection
    - Verifying whether a port may be updated by multiple task invocations and thus leading to non-determinism
  - Resource Size Analysis
    - Predicting the run-time memory requirements for executing an **xGiotto** program: the bound on the size of the event filter and scopes (trigger queue size and active task set size).
- Platform dependant
  - Schedulability Analysis
    - Ensuring that all the task invocations get access to the executing platform at least equal to their worst-case-execution times before their termination

Implementation



Implementation



Ongoing Works

- Implementation
  - Generate code
    - Embedded Virtual Machine code
    - Metropolis Meta-model
  - Porting to RTOS
    - EVM, JVM, OSEK
  - Case studies
    - Porting AFR controller on OSEK
- Analysis
  - Defining the run-time system for xGiotto
  - Schedulability check in time polynomial to the size of the program
- Future Direction
  - Sub-classes of xGiotto
    - Definition, inter relation and effectiveness towards event-driven programming
  - Type Checking

Thank You !