System and Language Support for Timing Constraints

Insup Lee
Department of Computer and Information Science
University of Pennsylvania

Originally prepared by Sebastian Fischmeister
Modified by Insup Lee for CIS 541, Spring 2010

Goals

- Understand different concepts about temporal constraints.
- Understand how temporal constraints can be incorporated into a programming language.
- Discuss how you would design your language.
Overview of Temporal Constraints

Why Temporal Constraints?

- A number of control applications puts temporal constraints on the control software.
  - Engine simulation: 1kHz recording frequency over a distributed system
  - Clock synchronization: down to 1 nanosecond
  - Industrial process control
  - Drive-by-wire
  - Anti-lock brakes
  - Pacemakers
  - Helicopter control
    - 200 Hz pilot stick, 400 Hz sensors, 200 Hz flight control, 1kHz actuator electronics
  - Heating control: 10 seconds
Temporal Constraints

- Real-time is about producing the correct result at the right time.

<table>
<thead>
<tr>
<th>Value</th>
<th>Timing</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wrong</td>
<td>Too late</td>
<td>Failure</td>
</tr>
<tr>
<td>Wrong</td>
<td>On time</td>
<td>Failure</td>
</tr>
<tr>
<td>Correct</td>
<td>Too late</td>
<td>Failure</td>
</tr>
<tr>
<td>Correct</td>
<td>On time</td>
<td>Ok</td>
</tr>
</tbody>
</table>

- Temporal constraints are a way to specify when the value is on time.

Types of Temporal Constraints

- Absolute temporal constraints
  - Measured with respect to a global clock
  - Xmas tree should light up between 5pm and 7am from November 27th 2006 until December 27th 2006

- Relative temporal constraints
  - Measured with respect to a local clock
  - The ventilation task should restart in five seconds

- Timing violation
  - Occurs when a temporal constraint is violated
Types of Temporal Constraints

- Hard temporal constraints
- Soft temporal constraints
- Firm temporal constraints
- Deterministic temporal constraints

Soft Temporal Constraints

- A soft real-time system is one where the response time is normally specified as an average value. This time is normally dictated by the business or market.

- A single computation arriving late is not significant to the operation of the system, though many late arrivals might be.

- Ex: Airline reservation system - If a single computation is late, the system’s response time may lag. However, the only consequence would be a frustrated potential passenger.
Hard Temporal Constraints

- A **hard real-time system** is one where the response time is specified as an absolute value. This time is normally dictated by the environment.

- A system is called a hard real-time if tasks always must finish execution before their deadlines or if messages always are delivered within a specified time interval.

- Hard real-time is often associated with safety critical applications. A failure (e.g., missing a deadline) in a safety-critical application can lead to loss of human life or severe economical damage.

Firm Temporal Constraints

- In a **firm real-time system** timing requirements are a combination of both hard and soft ones. Typically the computation will have a shorter soft requirement and a longer hard requirement.

- Ex: Ventilator – The system must ventilate a patient so many times within a given time period. But a few second delay in the initiation of the patient’s breath is allowed, but not more.
Deterministic Temporal Constraints

- In a **temporal deterministic real-time system** timing requirements are deterministic. An external observer can tell the temporal state at any time.

- A system with deterministic temporal constraints finishes execution exactly at the deadline (not before [hard] and not about [soft]).

- Ex. Similar to hard real-time systems, however, temporal determinism simplifies guaranteeing compositionality.

---

Real-Time Spectrum

```
No RT       Soft RT     Hard RT

Computer simulation  User interface  Internet video, audio  Cruise control  Tele communication  Flight control  Electronic engine
```
Terminology of Temporal Constraints

Tasks, Job

- A **task** is a piece of code that can be executed many times with different input data. (thread or process)

- A **job** is an instance of a task.
Parameters

- **Release or Arrival Time** ($r_i$)
  - is the time at which the task becomes ready for execution.

- **Computation time** ($C_i$)
  - is the time necessary to the processor for executing the task without interruption.

- **Deadline** ($d_i$)
  - is the time before which a task should be complete to avoid damage to the system.
  - Relative Deadline ($D_i$): $D_i = d_i - r_i$

- **Start time** ($s_i$)
  - is the time at which the task starts its execution.

Parameters

- **Finishing time** ($f_i$)
  - is the time at which the task finishes its execution.

- **Laxity** (Slack time) ($X_i$)
  - $X_i = d_i - r_i - C_i$ is the maximum time a task can be delayed on its activation to complete within its deadline.
Temporal Constraint Specifications

Task Types

- A periodic task has invocations within regular time intervals.
  - E.g., reading a heat sensor.
- A sporadic task has unknown arrival times, but have bounds such as maximum frequency.
  - E.g., routinely memory status check.
- An aperiodic task has an unknown arrival time.
  - E.g., an emergency shutoff.
**Frequency, Period**

- **Period, frequency:**
  - $T_1$: Period=10ms, Frequency=2

- **Period:**
  - $T_2$: Period=10ms

- **Frequency**
  - $T_3$: Frequency=400Hz

---

**Additional Terms**

- **Execution time:** total time of execution of a specific task
- **Elapse time:** the task’s execution time + all delays

- **Maximum time constraint:** no more than $t$ time units will elapse
- **Minimum time constraint:** no less than $t$ time units will elapse
Hyper-Period

- Hyper-Period is the time span after which the system repeats its behavior.
  - T₁: Period=10ms, Frequency=2
  - T₂: Period=10ms
  - T₃: Frequency=400Hz

- Hyper period = 10ms

Basic Model
Example

Independent-Digit Clock

- Consider a clock with each digit as an independent task.

\[
\begin{align*}
T_1: P &= 1\text{s} \\
T_2: P &= 10\text{s} \\
T_3: P &= 60\text{s} \\
T_4: P &= 600\text{s}
\end{align*}
\]
Properties

- Timliness is key
  - Invalid time value displayed

- Jitter accumulates and causes incorrect display.

- Value outputs need to be synchronized.

- Nearly no computation required.

Implicit Temporal Control
Foreground/Background System

- Using **super-loops** as the main routine with two levels: the task level and the interrupt level.
  - Task level (aka background): executes modules
  - Interrupt level (aka foreground): handles asynchronous events via ISRs.

- Foreground can preempt the background, thus:
  - Critical tasks must be in the foreground part.
  - Task level response = an ISR prepares data for the super-loop.

- Used for small devices (e.g., microcontrollers in microwaves, washers, dryers, radio)
Foreground/Background Model

Code for the Example

```c
void main(void) {
    unsigned short val;  unsigned int i;

    while ( 1 ) {
        val = get_curr_sec();
        val++;
        update_display_seconds(val);

        if (val%60 == 0 ) {
            // update tens
        }
        ...
        // may have nested loops, if too short
        i=WHILE_INSTRUCTIONS_PER_SECOND
        while( --i );
    }
}
```
Foreground/Background Properties

- Simple system/low overhead
  - No maintenance, basically no "system" at all

- Difficult to specify temporal behavior
  - F/B systems require hand tuning to meet a timing criteria; if the system is not responsive enough, then the developer will optimize the super-loop.

- Sensitive to changes
  - Changing a module constantly changes the timing of the super-loop.
  - Changing code in an ISR changes may change the overall timing behavior.

Programming-Language Timing Control
Type of Specification

- **Program-based temporal constraints**
  - Programmed in the target language.
  - Often mix program logic and temporal behavior

- **Specification-based temporal constraints**
  - Temporal constraints are specified in a separate language (=> Coordination language)
  - Can be high-level, e.g., *task A freq 0.2*

Temporal Scopes
Temporal Scopes

- **Source:** [Lee 1985], the Distributed Programming System (DPS).
- Temporal scopes and DPS describes a system to specify generic temporal constraints at the statement level.
- The main goals for temporal scopes are:
  - Provide language constructs for specifying timing constraints,
  - Apt for distributed systems,
  - Extend an existing language, and
  - Run-time monitoring and exception handling.
- Its properties are:
  - The program is configured offline.
  - All processes are created before start-up.
    - **No dynamic create of RT processes.**
  - The system has two modes: initialization and operation.
- Timing support is specification-based.

Timing Specification

- **Deadline.** The latest time in which the execution of a temporal scope can be completed.
- **Minimum delay.** The minimum amount of time that should pass before starting the execution of a temporal scope.
- **Maximum delay.** the maximum amount of time that should pass before starting the execution of a temporal scope.
- **Maximum execution time.** The maximum computation time necessary for the execution of a temporal scope.
- **Maximum elapse time.** The maximum execution time plus all user-defined delay during the execution of a temporal scope.
Timing Specification

Max. elapse time
Max. delay
Min. delay
Gap
Release
exe_1
deadline
t
exe_2
Deadline

Max. execution time = WCET

The Temporal Scope

- \text{start} \text{<delay-part>} [ \text{<exe-part>} ] [ \text{<dl-part>} ]
do
  \text{<start-body>}
  [\text{<exceptions>}] 
end

- \text{<delay-part>}:==\text{now}|\text{at} \text{<abs-time>}|\text{after} \text{<rel-time>}
- \text{<exe-part>}:==\text{execute} \text{<rel-time>}|\text{elapse} \text{<rel-time>}
- \text{<dl-part>}:==\text{by} \text{<abs-time>}|\text{within} \text{<rel-time>}

\textbf{Examples:}
- \text{Start after 10 sec do ... end}
- \text{Start at (9h:00m) within 10 sec do ... end}
Repetitive Temporal Scope

- from <start_time> to <end time> every <period>
  execute <exec_time> within <deadline> do
  <stmts>
  [<exceptions>]
  end

- Example:
  - from (8h:00m) to (18h:00m) every (0h:30m)
    within 10 sec do
    relax_eyes()
  end

Consecutive Temporal Scope

- cstart <delay_1> [<execute_1>] [<deadline_1>] do
  <stmts_1>
  [<exceptions_1>]
- cstart <delay_2> [<execute_2>] [<deadline_2>] do
  <stmts_2>
  [<exceptions_2>]
- cstart <delay_n> [<execute_n>] [<deadline_n>] do
  <stmts_n>
  [<exceptions_n>]
- end

- Example:
  - cstart within 2 sec do fill_glass_with_water()
  - cstart after 2 sec do empty_glass() end
Temporal Scopes Task Life Cycle

Temporal Scopes Example

from 00:00 to 59:59 every 10s execute 20ms within 1s
do
    var ctr;
    ctr=get_cur_tsecs();
    ctr=(ctr+1)%6;
    set_cur_tsecs(ctr);
    exception
        display_warning_light();
end
PEARL Overview

- Acronym for Process Automation Real-time Language
- Aimed to be a high-level programming language with elaborate constructs for programming temporal constraints.

- Developed at the same time as PASCAL, so both share similar syntax.
- PEARL forbids recursive procedures to eliminate out-of-memory errors.
- Strong emphasis on the I/O part, because of its target domain.

- Standardized as DIN 66253
- PEARL-90 is the revised version
PEARL Task Life Cycle

Timing Specification

StartCondition ::= 
  AT Expression§Time [ Frequency ] 
  | AFTER Expression§Duration [ Frequency ] 
  | WHEN Name§Interruption [ AFTER Expression§Duration ] [ Frequency ] 
  | Frequency

Frequency ::= 
  ALL Expression§Duration [ { UNTIL Expression§Time } 
  | { DURING Expression§Duration } ]

Examples:
- ALL 0.00005 SEC ACTIVATE Highspeedcontroller;
- AT 12:00 ALL 4 SEC UNTIL 12:30 ACTIVATE lunchhour;
- WHEN fire ACTIVATE extinguish;
PEARL Example

WHEN start ALL 1 sec UNTIL stop ACTIVATE clock_sec;
WHEN start ALL 10 sec UNTIL stop ACTIVATE clock_tsec;
WHEN start ALL 60 sec UNTIL stop ACTIVATE clock_min;
WHEN start ALL 600 sec UNTIL stop ACTIVATE clock_tmin;

clock_tsec: TASK PRIO 2;
   DCL ctr INTEGER;
BEGIN
   GET ctr FROM DISPLAY_T_ONES;
   ctr := (ctr+1)%6;
   PUT ctr TO DISPLAY_T_ONES;
END

The ARTS Kernel &
The Time Fence Protocol
Time Fence in the ARTS Kernel

- Source: [Tokuda, Mercer, 1998].
- The time-fence protocol allows for temporal constraints in a distributed real-time system. The time-fence protocol is built into the ARTS kernel.
- The ARTS kernel aims at distributed real-time systems.
- The \textit{artsobject} is the abstraction for computation:
  - The \textit{artsobject} has a WCET.
  - The \textit{artsobject} minimizes inter-module dependence.
  - It provides time-encapsulation (however, the designer must guarantee this).
- Timing support is specification-based.

Thread Life Cycle

[Diagram showing the thread life cycle with states such as Init, Ready, Running, Suspended, and transitions like Release, Start, and Completion.]
Function Life Cycle

 Specification

// An example of a real-time thread
Thread Sample._Artobject::RT_Thread( )
//# priority, stack_size, wcet, period, phase, delay
{ //thread body …
    ThreadExit( );
}

The implementation also allows for object methods:

 type opt1 (type arg .... ); //# within time except opr()
Stopwatch Example

Thread Minutes::RT_Thread() // # 2, __, 10ms, 10s, 0, 0s
{
    // thread body
    int tens_seconds = get_cur_tens_seconds();
    tens_seconds = (tens_seconds + 1) % 6;
    set_cur_seconds(tens_seconds);

    ThreadExit( ); // reincarnate this thread
}

The Time Fence Protocol

- The system scheduler checks for transient overloads (not enough CPU cycles) and rejects tasks in case of such an overload.

- Each RT computation has a WCET.
- The time fence uses the deadline to set a timer.
- The scheduler checks schedulability using the time fence and the WCET.

\[ \text{Callee}_{wrtv} < \text{Caller}_{ctv} - 2*\text{comm} + \text{clockdrift} \]

- Comm can include communication overhead for the distributed system.
Esterel

Synchronous Model

Scheduled Model

- Event
- Deadline
- Scheduled Computation
- Response Time

Synchronous Model

- Event
- Event
- Synchronous Computation
### Synchronous Model

- **Scheduled Implementation**
  - Event → Deadline
  - Response Time

- **Synchronous Implementation**
  - Event → Event
  - Response Time

### Basic Concepts

- **Specification language** has been specialized for reactive systems.
- **Reactive system:**
  - In continuous interaction with its environment.
  - A reaction begins when the system receives an input event and ends when it generates the corresponding output event.
- **Black-box approach**
  - Inputs produce outputs, continuously.
  - Only define relationships between input and output events.
  - A task may be complex; but, you don’t care.
Basic Concepts

- **Based on synchronous model of time** (synchrony hypothesis)
  - The underlying machine is infinitely fast, and hence, the reaction of the system to an input event is instantaneous; in between reactions, the system is idle.
  - No reaction intervals → only reaction instants → reactions do not overlap.
  - The synchrony hypothesis simplifies the behavioral specification of reactive systems (see the example later on).
  - Looks flawed (or sounds unreal), but the machine must react to an input event before the next input event arrives.

<table>
<thead>
<tr>
<th>Event</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Basic Concepts

- **Determinism**
  - A non-deterministic system does not have a unique response to a given input event → the external observer cannot predict the response.
  - Example:
    - Waiting for 60 seconds and then(??) signal "minute".
    - Broadcasting the signal, timing delays.

    ```
    loop
        delay 60; B.MINUTE; (C.MINUTE)
    end
    ```

  - Esterel guarantees determinism
    - All statements and constructs are well defined (syntax and semantics).
    - A compiler checks the program and ensures determinism.
Signal Handling: Example

- Example program:

```plaintext
pause;
emit A;
emit B;
present B then emit C; end
pause;
emit C;
```

Example StopWatch

```plaintext
module SW1:
input START, STOP, MS;
output TIME(integer);
relation START $ STOP;
var count := 0 : integer
in
  await immediate START;
  abort every immediate MS do
    count := count + 1;
    emit TIME(count);
  end
  when STOP
    % pause;
    sustain TIME(count);
end var
end module
```
Real-Time Specification for Java (RTSJ)

Introduction

- The correct name is: Real-Time Specification for Java (RTSJ).
- Guiding Principles:
  - **Applicability to Java Environments**: The RTSJ shall not include specifications that restrict its use to particular Java environments.
  - **Backward Compatibility**: The RTSJ shall not prevent existing, properly written, non-real-time Java programs from executing on implementations of the RTSJ.
  - **Write Once, Run Anywhere**.
  - **Current Practice vs. Advanced Features**: The RTSJ should address current real-time system practice as well as allow future implementations to include advanced features.
  - **Predictable Execution**: The RTSJ shall hold predictable execution as first priority in all trade-offs.
  - **No Syntactic Extension**.
  - **Allow Variation in Implementation Decisions**.
Overview

- RT Java consists of an RTJVM and the RTSJ class library.
- RTSJ-compliant JVMs can be considered Real-Time Java Virtual Machines (RTJVMs).
- Resides in the package javax.realtime with modifications to the non RT Java such as
  - A RT Thread class extending java.lang.Thread
  - Sophisticated scheduling support
  - No mandatory RT garbage collection, instead memory partitioning
  - Raw memory access for device drivers

Handling of Time

- Clock:
  - A clock marks the passing of time.
  - System.getRealtimeClock() for singletons.
  - Can have an arbitrary resolution (see RelativeTime).
- Based on the clock, a number of classes dealing with time exist:
  - **HighResolutionTime**: is an abstract class and the base class for all time-related classes. Used to express time with nanosecond accuracy.
  - **AbsoluteTime**: represents a specific point in time given by milliseconds plus nanoseconds past some point in time fixed by the clock.
  - **RationalTime**: represents a time interval that is divided into subintervals by some frequency. Used to periodic events, threads, and feasibility analysis.
  - **RelativeTime**: is generally used to represent a time relative to now
- All time objects must maintain nanosecond precision.
Real-Time Threads

- Two types of threads:
  - NoHeapRealtimeThread
  - RealtimeThread
- Release parameters specify the thread’s behavior in the time domain:
  - PeriodicParameters: indicates that the schedulable object is released on a regular basis.
  - SporadicParameters: notes that the associated schedulable object’s run method will be released aperiodically but with a minimum time between releases.
  - AperiodicParameters: characterizes a schedulable object that may be released at any time.

Task Life Cycle
Stopwatch Example

```java
public class TSec extends RealTimeThread {
    public void run() {
        while (true) {
            int val = getCurrentTSecValue();
            val = (val + 1) % 6;
            setCurrentTSecValue(val);
            waitForNextPeriod();
        }
    }

    TMin createInstance() {
        PeriodicParameters pp = new PeriodicParameters(
            offset, // the period
            new RelativeTime(10.0*SECONDS),
            new RelativeTime(5.0), // the cost
            new RelativeTime(10.0*SECONDS), // the deadline
            null, null);

        return new TSec(priority, pp);
    }
}
```

Giotto
Overview

- Source: [T. Henzinger et al, 2002]

- One of the main issues was to create verifiable RT programs.

- Rigid control of the system’s behavior.
  - Input/output values are buffered in ports (similar to the process image with PLCs)
  - Value determinism
  - Time determinism

- An embedded machine controls the task’s execution.

Logical Execution Time

Logical execution time = Logical computation time

Reading input ports

Writing output ports

Task t

\[ t \rightarrow \text{Start} \rightarrow \text{Suspend} \rightarrow \text{Resume} \rightarrow \text{Stop} \rightarrow t + T \]
Task Life Cycle

Example
**Runtime Environment**

- **E code**
- **Application object code**

- **execute**
- **calls**

- **Driver**
- **E machine**
- **Driver**

- **Platform**
- **Sensor**
- **Environment**
- **Actuator**

- **E machine** runs on **Platform**

**E-Code**

- **E-Code** controls the execution behavior

- **Call**: executes drivers
- **Schedule**: enqueues tasks
- **Future**: schedules a resume
- **Return**: exists the interpreter

```
lbl1:  call d [ t1 ]
       call d [ t2 ]
       schedule t1
       schedule t2
       future, 200, lbl2
       return

lbl2:  call d[ t2 ]
       schedule t2
       future, 200, lbl1
       return
```
Timing Specification

- Only allows periodic tasks.
- Defined by period and frequency.
- Each mode has a period.
- Each task has a frequency within the mode.

```plaintext
mode Flight () period 10ms
{
  actfreq 1 do Actuator (actuating);
  taskfreq 1 do Control (input);
  taskfreq 2 do Navigation (sensing);
}
```

Stopwatch Example

```plaintext
start Started {
  mode Started() period 3600 {
    actfreq 3600 do act_sec(a_sec_driver);
    taskfreq 3600 do comp_sec(sec_driver);
    actfreq 60 do act_tsec(a_tsec_driver);
    taskfreq 60 do comp_tsec(tsec_driver);
    actfreq 10 do act_min(a_min_driver);
    taskfreq 10 do comp_min(min_driver);
    actfreq 1 do act_tmin(a_tmin_driver);
    taskfreq 1 do comp_tmin(tmin_driver);
  }
}
```
Summary

<table>
<thead>
<tr>
<th>Name</th>
<th>Granularity</th>
<th>Task model</th>
<th>Type</th>
<th>Constr.</th>
<th>Err. handling</th>
</tr>
</thead>
<tbody>
<tr>
<td>PEARL-90</td>
<td>Task, per</td>
<td>per, sus</td>
<td>Spec.</td>
<td>Abs.&amp;rel.</td>
<td>No</td>
</tr>
<tr>
<td>Temp. scopes</td>
<td>Statement</td>
<td>off, per, dl</td>
<td>Spec.</td>
<td>Abs.&amp;rel.</td>
<td>Exceptions</td>
</tr>
<tr>
<td>ARTS kernel</td>
<td>Task, fun.</td>
<td>ph, off, per</td>
<td>Spec.</td>
<td>Rel.</td>
<td>No</td>
</tr>
<tr>
<td>RTSJ</td>
<td>Task</td>
<td>off, per, dl</td>
<td>Prgm.</td>
<td>Abs.&amp;rel.</td>
<td>Exceptions</td>
</tr>
<tr>
<td>Estrel</td>
<td>Statement</td>
<td>Reactive</td>
<td>Prgm.</td>
<td>-</td>
<td>No</td>
</tr>
<tr>
<td>Giotto</td>
<td>Task</td>
<td>per</td>
<td>Spec.</td>
<td>Rel.</td>
<td>No</td>
</tr>
</tbody>
</table>

a Also timing control for one-time execution (e.g., statement blocks) with offset.

b Although it is specification-based, it intertwines code and timing specification.

c Arts provides different temporal control elements for tasks and functions.

d Also offers deadlines for function calls.

Take Away Messages

- Timing constraints in programming languages are a topic since at least 1968.
  - What are the right abstractions? (Modules, tasks, statements)
  - What is the right notion of time? (Zero, continuous, discrete time)
  - Who checks timing constraints? (Offline, online)
  - How do you specify timing? (Specification-based vs. programming)
  - How to ensure timing constraints? (Verification, runtime checking, offline, online)
Summary

<table>
<thead>
<tr>
<th>Name</th>
<th>Abstraction level</th>
<th>Type</th>
<th>Guarantee</th>
<th>Enforcement</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>F/B Sys</td>
<td>Superloop</td>
<td>Prog.</td>
<td>None</td>
<td>None</td>
<td>Simple</td>
</tr>
<tr>
<td>Temporal Scopes</td>
<td>Statement level</td>
<td>Spec.</td>
<td>Impl.</td>
<td>Runtime</td>
<td>Exceptions</td>
</tr>
<tr>
<td>Esterel</td>
<td>Stmt.</td>
<td>Prog.</td>
<td>Exact</td>
<td>Compiler</td>
<td>Toolchain</td>
</tr>
<tr>
<td>PLC</td>
<td>Block</td>
<td>Spec.</td>
<td>Best eff.</td>
<td>Runtime</td>
<td>Commercial</td>
</tr>
<tr>
<td>TMO</td>
<td>Method</td>
<td>Spec.</td>
<td>Best eff.</td>
<td>Runtime</td>
<td></td>
</tr>
<tr>
<td>RTSJ</td>
<td>Thread</td>
<td>Prog.</td>
<td>Best eff.</td>
<td>Runtime</td>
<td>By popular demand</td>
</tr>
<tr>
<td>Giotto</td>
<td>Thread</td>
<td>Spec.</td>
<td>Exact (?)</td>
<td>By constr.</td>
<td>E-Code</td>
</tr>
</tbody>
</table>

Personal Note & Observations

- PLCs & Sequential Function Charts are a rock solid method, sold billion times, defeats many theoretic and academic models.
- Synchronous languages are about to become a huge industry-strength concept: Airbus uses SCADE.
- Temporal scopes present a general abstraction, but did not catch on.
- Simple, but effective solutions - or - a complete tool chain.
- Retrofitting does not work - it did not for security, it will not for RT systems.
Bibliography


Bibliography