Low-Level Reactive Languages

Jan Tobias Mühlberg

jantobias.muehlberg@cs.kuleuven.be
iMinds-DistriNet

PLaNES Reading Club, KU Leuven, 13th May 2015
Around 2010: Course on “Reactive Systems Design” for MSc in Software Engineering and Gas Turbine Control at York

- Focus on synchronous languages for reactive control systems
Motivation

Around 2010: Course on “Reactive Systems Design” for MSc in Software Engineering and Gas Turbine Control at York

- Focus on synchronous languages for reactive control systems
- Lectures: Mathematical foundations, Lustre, Esterel, Statecharts, compilation and design verification
Motivation

Around 2010: Course on “Reactive Systems Design” for MSc in Software Engineering and Gas Turbine Control at York

- Focus on synchronous languages for reactive control systems
- Lectures: Mathematical foundations, Lustre, Esterel, Statecharts, compilation and design verification
- Practicals: SCADE and Lego Mindstorms

- Graphical modelling of reactive systems using synchronous language
- Graphical debugging and efficient simulation
- Design Verifier – formal verification
- Generation of safe, efficient, small print production code (qual. DO-178B; cert. IEC 61508, EN 50128)
Motivation


- Graphical modelling of reactive systems using synchronous language
- Graphical debugging and efficient simulation
- Design Verifier – formal verification
- Generation of safe, efficient, small print production code (qual. DO-178B; cert. IEC 61508, EN 50128)

What are the new trends for RP in safety-critical systems?
To distinguish this from previous talks: Imperative languages, no distribution, deterministic w.r.t. timing, aiming at safety critical deployment & verification
This Talk

To distinguish this from previous talks: Imperative languages, no distribution, deterministic w.r.t. timing, aiming at safety critical deployment & verification

Outline

- Outline of synchronous languages
- Reactive C [Bou91]
- Synchronous C [vH09] (and SJ)
- PRET-C [ARGT14] (2009)

Overview & survey: [BCE⁺03] (focusing on Esterel, Lustre and Signal)
[BCC\textsuperscript{+}13] mentions Esterel, StateCharts, Lustre, LabVIEW, Simulink and others.

Overview & survey: [BCE\textsuperscript{+}03] (focusing on Esterel, Lustre and Signal)

\textbf{Properties}
Include specific/dedicated features for programming reactive controllers with real-time constraints:

- \textit{synchrony}
Synchronous Languages


Overview & survey: [BCE+03] (focusing on Esterel, Lustre and Signal)

**Properties**
Include specific/dedicated features for programming reactive controllers with real-time constraints:

- **synchrony**
- typically first-order
- concurrency
- determinism
Synchronous Languages

The Synchrony Hypothesis: Let $\Delta(f(x))$ denote the time to compute a reaction $f$ on inputs $x$. $\Delta(f(x))$ depends on (1) the implementation of $f$, (2) the target machine, and (3) the nature of $x$.

Problem: We wish to abstract $\Delta(f(x))$ to some $\delta$, but also require compositionality, i.e. if $f(x) = g(h(x))$, then $\Delta_f = \Delta_g + \Delta_h$. How can we obtain the required identity $\delta = \delta + \delta$?
The Synchrony Hypothesis: Let $\Delta(f(x))$ denote the time to compute a reaction $f$ on inputs $x$. $\Delta(f(x))$ depends on (1) the implementation of $f$, (2) the target machine, and (3) the nature of $x$.

Problem: We wish to abstract $\Delta(f(x))$ to some $\delta$, but also require compositionality, i.e. if $f(x) = g(h(x))$, then $\Delta_f = \Delta_g + \Delta_h$. How can we obtain the required identity $\delta = \delta + \delta$?

Solutions

1. $\delta = 0$ – **synchrony**, reactive control systems
2. $\delta = ?$ – asynchrony, interactive systems

Synchronous languages achieve separation of concerns: qualitative (logical) time versus of quantitative (physical) time.
Synchronous Languages

Reality

• Valid abstraction as long as $\delta_i \leq \Delta_i$
• This needs to be checked and verified for the implementation (worst-case execution time analysis, etc.)
• Two views of the system:
  • **External view**: Reactions are atomic
  • **Internal view**: Reactions are non-atomic
Synchronous Programming

... for Control Engineers in SCADE: ControlVehicle
Synchronous Programming: OperateMotor
Synchronous Programming: OperateMotor as SM

Speed: integer init ?BaseSpeed/3

{pre(?Speed) > ?SSMMaxSpeed}/Speed(?SSMMaxSpeed)

{pre(?Speed) < ?SSMMinSpeed}/Speed(?SSMMinSpeed)

#SpeedRight(?Speed)

#SpeedRight(pre(?Speed))

CalcSpeed/Speed(?BaseSpeed)

Turn

Drive

SensorRight/

<1>

<4>

not SensorRight/

<2>

<3>

not SensorLeft/

Brake

SensorLeft/

<5>

#BrakeRight, SpeedRight(?SSMMinSpeed)
**Synchronous Programming**: Compilation & Execution

**Event Driven**

Initialise Memory
for each input event do
  Compute Outputs
  Update Memory
end

e.g. Esterel

**Sample Driven**

Initialise Memory
for each clock tick do
  Read Inputs
  Compute Outputs
  Update Memory
end

e.g. Lustre
Synchronous Programming

Design Verification

- SensorLight_1
- SensorLight_3
- ControlVehicle
- OutputRangesProperty

OutputRangesProperty
Synchronous Programming

Design Verification

---

SensorLight_1

SensorLight_3

ControlVeror

Motor_A_Speed
Motor_A_Direction
Motor_C_Speed
Motor_C_Direction

MotorRanges
MotorRanges

OutputRanges Property

---

12 /32
13th May 2015
Low-Level Reactive Languages
Synchronous Programming

Design Verification

SensorLight_1

SensorLight_3

Control

Max Speed
Speed
Min Speed
Motors Brake
Direction
Motors Reverse

MotorOkay

Logic Diagram
Reactive C

Extends C with parallelism, exceptions and reactive statements.

Semantics of RC extensions is based directly on Esterel: parallelism is evaluated deterministically with no run-time concurrency.

Embedding of RC in C is done by preprocessor. Compiler enforces deadlock freedom for reactive statements.
An Example: Time, Signals and Parallelism

```c
// Reactive C [Bou91]

signal SYNC, REQ, OK, NOK, ALARM;

rproc req_handler() {
  every (present(SYNC)) {
    await (present(REQ));
    emit (OK);
    stop;
    every (present(REQ))
    emit (NOK);
  }
}

rproc alarm_handler() {
  loop {
    watching {
      await (present(SYNC));
      emit (ALARM);
    } timeout await(present(SYNC));
    stop;
  }
}

rproc sync_req_handler() {
  par
  exec req_handler();
  exec alarm_handler();
}
```
<table>
<thead>
<tr>
<th></th>
<th>RC</th>
<th>Esterel</th>
</tr>
</thead>
<tbody>
<tr>
<td>par printf(&quot;1&quot;); printf(&quot;2&quot;);</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>par printf(&quot;1&quot;); printf(&quot;2&quot;); present S else emit S end</td>
<td>RC: valid</td>
<td>Esterel: invalid: causality cycle!</td>
</tr>
<tr>
<td></td>
<td>RC</td>
<td>Esterel</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-----------------------------------------</td>
<td>----------------------------------------------</td>
</tr>
<tr>
<td>par</td>
<td>printf(&quot;1&quot;);</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>printf(&quot;2&quot;);</td>
<td>12</td>
</tr>
<tr>
<td>present $S$</td>
<td>valid</td>
<td>invalid:</td>
</tr>
<tr>
<td>else emit $S$ end</td>
<td></td>
<td>causality cycle!</td>
</tr>
<tr>
<td>present $S_1$</td>
<td>can be implemented with run-time checks</td>
<td>valid:</td>
</tr>
<tr>
<td>then emit $S_2$ end</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Reactive C [Bou91]

<table>
<thead>
<tr>
<th>RC</th>
<th>Esterel</th>
</tr>
</thead>
</table>
| `par`<br>`printf("1");`
`printf("2");`<br>`present S`<br>`else emit S end`<br>`present S1`<br>`then emit S2 end ||` emit S1; present S2`<br>`then emit S3 end` | `12`<br>`valid`<br>`can be implemented with run-time checks` | `12`<br>`invalid:`<br>`causality cycle!`<br>`valid:`<br>`instantaneous dialogue` |

#### Data Types

- **RC**: Signals, primitive types, structured data
- **Esterel**: Signals and numeric values
## Reactive C [Bou91]

<table>
<thead>
<tr>
<th></th>
<th>RC</th>
<th>Esterel</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>par</code></td>
<td>printf(&quot;1&quot;); printf(&quot;2&quot;);</td>
<td>12</td>
</tr>
<tr>
<td><code>present S</code></td>
<td><code>valid</code></td>
<td>invalid: causality cycle!</td>
</tr>
<tr>
<td><code>else emit S end</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>present S1</code></td>
<td>can be implemented with run-time checks</td>
<td>valid: instantaneous dialogue</td>
</tr>
<tr>
<td><code>then emit S2 end</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>emit S1; present S2</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>then emit S3 end</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Data Types
- Signals, primitive types, structured data
- Dynamic

### Process Management
- Signals and numeric values
- Static
### Reactive C [Bou91]

<table>
<thead>
<tr>
<th></th>
<th>RC</th>
<th>Esterel</th>
</tr>
</thead>
<tbody>
<tr>
<td>par</td>
<td></td>
<td>12</td>
</tr>
<tr>
<td>printf(&quot;1&quot;);</td>
<td></td>
<td>12</td>
</tr>
<tr>
<td>printf(&quot;2&quot;);</td>
<td></td>
<td>invalid: causality cycle!</td>
</tr>
<tr>
<td>present S</td>
<td>valid</td>
<td></td>
</tr>
<tr>
<td>else emit S end</td>
<td>can be implemented</td>
<td>valid: instantaneous dialogue</td>
</tr>
<tr>
<td>present S1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>then emit S2 end</td>
<td></td>
<td></td>
</tr>
<tr>
<td>emit S1; present S2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>then emit S3 end</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Data Types**

- Signals, primitive types, structured data
- Signals and numeric values

**Process Management**

- Dynamic
- Static

**Compilation and Execution**

- Compiled directly
- Automaton → validation → code
Synchronous C
Synchronous C [vH09] (and SJ)


Based on Statecharts [Har87] (sequential reactive control flow & visual syntax)
SyncCharts [And95] (synchronous semantics)

Light-weight approach to embed deterministic reactive control flow constructs into widely used programming languages (C and Java).

Fairly small number of primitives suffices to cover all of SyncCharts.

Multi-threaded, priority-based approach inspired by synchronous reactive processing – where it required special HW & special compiler.
Idea: Cooperative thread scheduling at application level  
Problem: High-level languages do not provide access to program counter  
Solution: Explicit labelling of continuation points  
  - Expressed as program labels or switch cases  
  - Each thread maintains a coarse program counter that points to continuation point  

Furthermore:  
  - Synchronous model of time, threads execute ticks in lock-step  
  - Shared address space, broadcast communication via ordinary variables or signals  
  - Dynamic priorities, may switch control back and forth within tick
## SC Thread Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TICKSTART(^*(\text{init}, p))</td>
<td>Start (initial) tick, assign main thread priority (p).</td>
</tr>
<tr>
<td>TICKEND</td>
<td>Return true (1) iff there is still an enabled thread.</td>
</tr>
<tr>
<td>PAUSE(^*)</td>
<td>Deactivate current thread for this tick.</td>
</tr>
<tr>
<td>TERM(^*)</td>
<td>Terminate current thread.</td>
</tr>
<tr>
<td>ABORT</td>
<td>Abort descendant threads.</td>
</tr>
<tr>
<td>TRANS((l))</td>
<td>Shorthand for ABORT; GOTO((l)).</td>
</tr>
<tr>
<td>SUSPEND(^*(\text{cond}))</td>
<td>Suspend (pause) thread + descendants if (\text{cond}) holds.</td>
</tr>
<tr>
<td>FORK((l, p))</td>
<td>Create a thread with start address (l) and priority (p).</td>
</tr>
<tr>
<td>FORKE(^*(l))</td>
<td>Finalize FORK, resume at (l).</td>
</tr>
<tr>
<td>JOINELSE(^*++(l_{\text{else}}))</td>
<td>If descendant threads have terminated normally, proceed; else pause, jump to (l_{\text{else}}).</td>
</tr>
<tr>
<td>JOIN(^*++)</td>
<td>Waits for descendant threads to terminated normally.</td>
</tr>
<tr>
<td>PRIOR(^*++(p))</td>
<td>Set current thread priority to (p).</td>
</tr>
</tbody>
</table>

\(^*\) possible thread dispatcher call
\(^+\) automatically generates continuation label
Recall: Producer-Consumer-Observer in SC

1 int tick ( int isInit )
2 {
3 static int BUF, fd, i, j, k = 0, tmp, arr[8];
4
5 TICKSTART(isInit, 1);
6
7 PCO:
8 FORK(Producer, 3);
9 FORK(Consumer, 2);
10 FORK(Observer);

Producer:
12 for (i = 0; ; i++) {
13  PAUSE;
14  BUF = i;
15 }

Consumer:
17 for (j = 0; j < 8; j++)
18  arr[j] = 0;
19 for (j = 0; ; j++) {
20   PAUSE;
21   tmp = BUF;
22   arr[j % 8] = tmp;
23 }

Observer:
25 for ( ; ; ) {
26   PAUSE;
27   fd = BUF;
28   k++;
29 }
30
31 TICKEND;
32 }
Synchronous C [vH09] (and SJ)

Producer-Consumer-Observer with Preemption in SC

```c
int tick ( int isInit )
{
    static int BUF, fd, i, j,
    k = 0, tmp, arr [8];

    TICKSTART(isInit, 1);

    Producer:
    for ( i = 0; ; i++)
    {
        BUF = i;
        PAUSE;
    }

    Consumer:
    for ( j = 0; j < 8; j++)
    {arr [j] = 0;
     for ( j = 0; ; j++)
     {
        tmp = BUF;
        arr [j % 8] = tmp;
        PAUSE;
    }

    Observer:
    for ( ; ; )
    {
        fd = BUF;
        k++;
        PAUSE;
    }

    Parent:
    while (1) {
        if (k == 20)
            TRANS(Done);
        if (BUF == 10)
            TRANS(PCO);
        PAUSE;
    }

    Done:
    TERM;
    TICKEND;
}
```
PRET-C
“Precision Timed C”, Sidharta Anadlam et al., 2009.

Synchronous extension of C; compiler provides worst-case reaction time analysis and allows mapping of logical time to physical time.

Offers safe, C-based shared memory communications between concurrent threads. Concurrency is logical, execution is sequential.

Minimal extensions to C, implemented as macros.

Only language with quantitative evaluation: generated code is generally more efficient than Esterel.
### C Language Extensions

<table>
<thead>
<tr>
<th>Statement</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ReactiveInput I</td>
<td>declares I as a reactive input coming from the environment</td>
</tr>
<tr>
<td>ReactiveOutput O</td>
<td>declares O as a reactive output emitted to the environment</td>
</tr>
<tr>
<td>PAR(T1, ..., Tn)</td>
<td>synchronously executes in parallel the n threads Ti, with higher priority of Ti over Ti+1</td>
</tr>
<tr>
<td>EOT</td>
<td>marks the end of a tick (local or global depending on its position)</td>
</tr>
<tr>
<td>[weak] abort P when pre C</td>
<td>immediately kills P when C is true in the previous instant</td>
</tr>
</tbody>
</table>
Restrictions:

- Pointers and dynamic memory allocation are disallowed.
- All loops must have at least one EOT in their body.
- All function calls have to be non-recursive.
- Jumps via goto are not allowed to cross logical instants (i.e. EOT).
Summary
## Summary

<table>
<thead>
<tr>
<th></th>
<th>Esterel</th>
<th>RC</th>
<th>SC</th>
<th>PRET-C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Commutativity of $</td>
<td></td>
<td>$</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Communication</td>
<td>signals</td>
<td>signals &amp; variables</td>
<td>variables</td>
<td>variables</td>
</tr>
<tr>
<td>Instantaneous dialogue</td>
<td>yes</td>
<td>yes/no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Signals/variable values/instants</td>
<td>single</td>
<td>multiple</td>
<td>multiple</td>
<td>multiple</td>
</tr>
<tr>
<td>Types of aborts</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Types of suspend</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>Traps</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Non-causal programs</td>
<td>possible</td>
<td>possible</td>
<td>not possible</td>
<td>not possible</td>
</tr>
<tr>
<td>Dynamic processes</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Compilation</td>
<td>complex</td>
<td>macro exp. resolve $</td>
<td></td>
<td>$ cycle det.</td>
</tr>
</tbody>
</table>
The original synchronous languages were designed for safety-critical reactive control systems: determinism and support verification.

Embedding of synchronous constructs in general-purpose programming languages appears to be less adequate for safety-critical applications. Yet, Esterel programs also need to interact with OS and drivers.

There are many (mostly syntactic) variants of the languages discussed here. Many semantical extensions being proposed.

There are many alternative approaches: ECL (Esterel C), Jester (Java Esterel), etc.

**Suggestion**
There is real-time FRP [WTH01]. Anyone?
Thank you! Questions?
C. André.  
SyncCharts: A visual representation of reactive behaviors.  
_Rapport de recherche tr95-52, Université de Nice-Sophia Antipolis, 1995._

S. Andalam, P. S. Roop, A. Girault, and C. Traulsen.  
A predictable framework for safety-critical embedded systems.  

A survey on reactive programming.  

A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone.  
The synchronous languages 12 years later.  

F. Boussinot.  
Reactive C: An extension of C to program reactive systems.  

D. Harel.  
Statecharts: A visual formalism for complex systems.  
R. von Hanxleden.
SyncCharts in C: A proposal for light-weight, deterministic concurrency.

Z. Wan, W. Taha, and P. Hudak.
Real-time FRP.