Low-Level Reactive Languages

Jan Tobias Mühlberg

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

PLaNES Reading Club, KU Leuven, 13th May 2015

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

Motivation

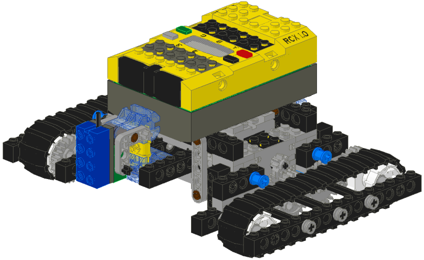
Around 2010: Course on “Reactive Systems Design” for MSc in Software Engineering and Gas Turbine Control at York

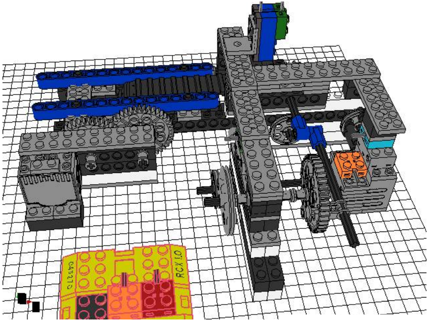
- Focus on synchronous languages for reactive control systems
- Lectures: Mathematical foundations, Lustre, Esterel, Statecharts, compilation and design verification

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





Motivation

SCADE: “The Standard for the Development of Safety-Critical Embedded Software in Aerospace & Defense, Rail Transportation, Energy and Heavy Equipment Industries” – `http://www.esterel-technologies.com/`

- 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

SCADE: "The Standard for the Development of Safety-Critical Embedded Software in Aerospace & Defense, Rail Transportation, Energy and Heavy Equipment Industries" – `http://www.esterel-technologies.com/`

- 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?

This Talk

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)

Synchronous Languages

[BCC$^+$13] mentions Esterel, StateCharts, Lustre, LabVIEW, Simulink and others.

Synchronous Languages

[BCC^{+}13] mentions Esterel, StateCharts, Lustre, LabVIEW, Simulink and others.

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

Synchronous Languages

[BCC$^+$13] mentions Esterel, StateCharts, Lustre, LabVIEW, Simulink and others.

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

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

- **synchrony**

Synchronous Languages

[BCC$^+$13] mentions Esterel, StateCharts, Lustre, LabVIEW, Simulink and others.

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 δ, 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 $\delta = \delta + \delta$?

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 δ, 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 $\delta = \delta + \delta$?

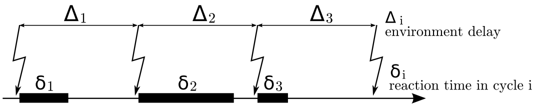
Solutions

(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.

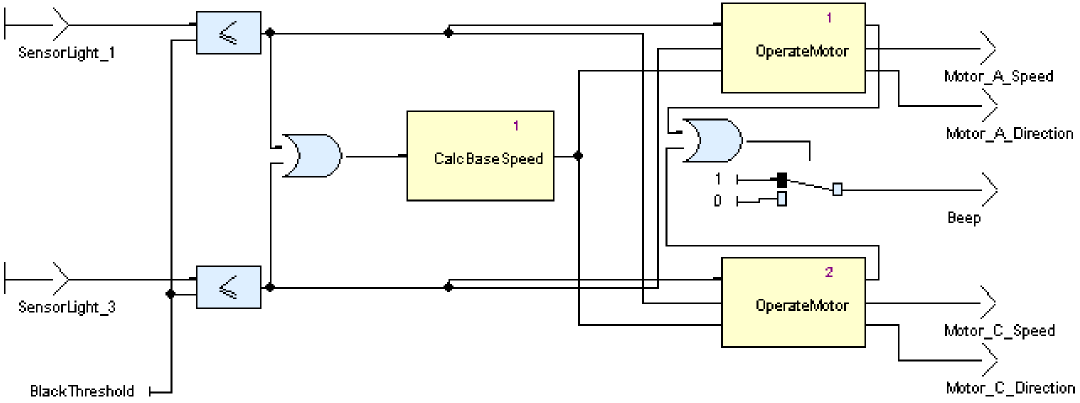
Reality



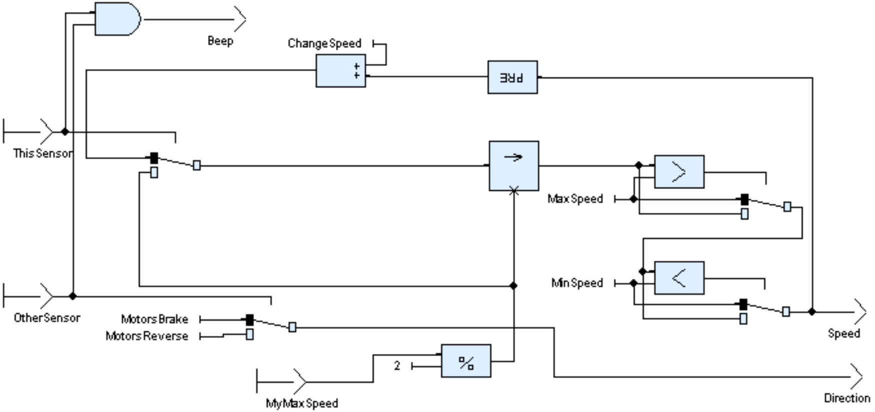
- Valid abstraction as long as $\delta_i \leq \Delta_i$
- This needs to be checked and verified for the implementation (worst-case execution time analysis, etc.)
- Two views of the system:
  - **External view**: Reactions are atomic
  - **Internal view**: Reactions are non-atomic

Synchronous Programming

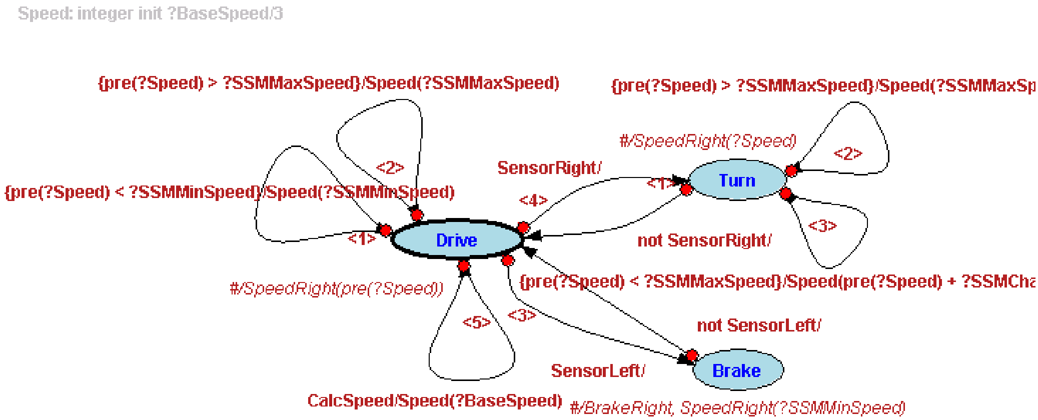
... for Control Engineers in SCADE: `ControlVehicle`



Synchronous Programming: `OperateMotor`



Synchronous Programming: `OperateMotor` as SM



Synchronous Programming: Compilation & Execution

Event Driven

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

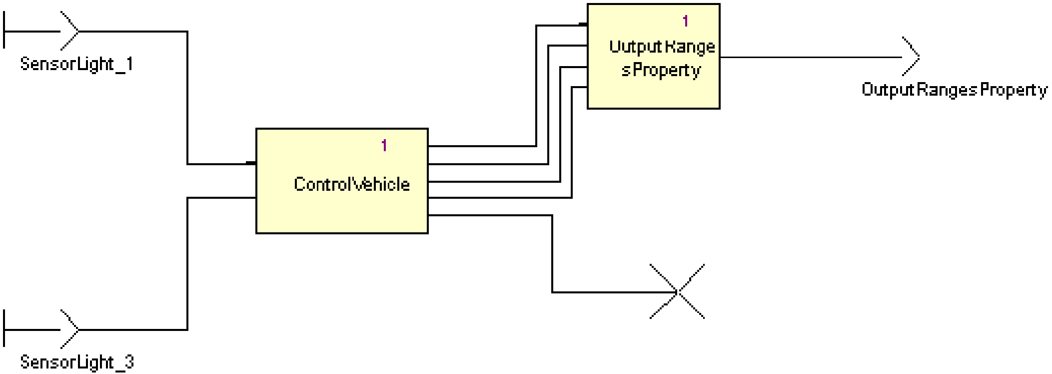
e.g. Esterel

Sample Driven

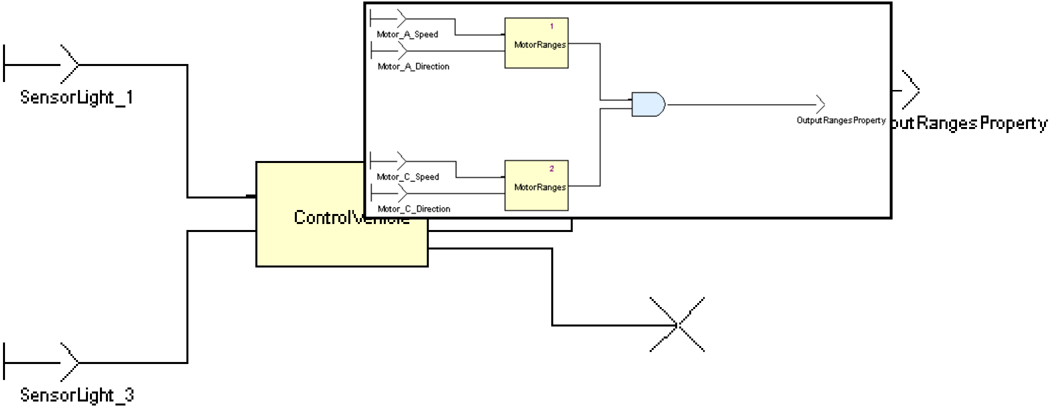
```
Initialise Memory
for each clock tick do
  Read Inputs
  Compute Outputs
  Update Memory
end
```

e.g. Lustre

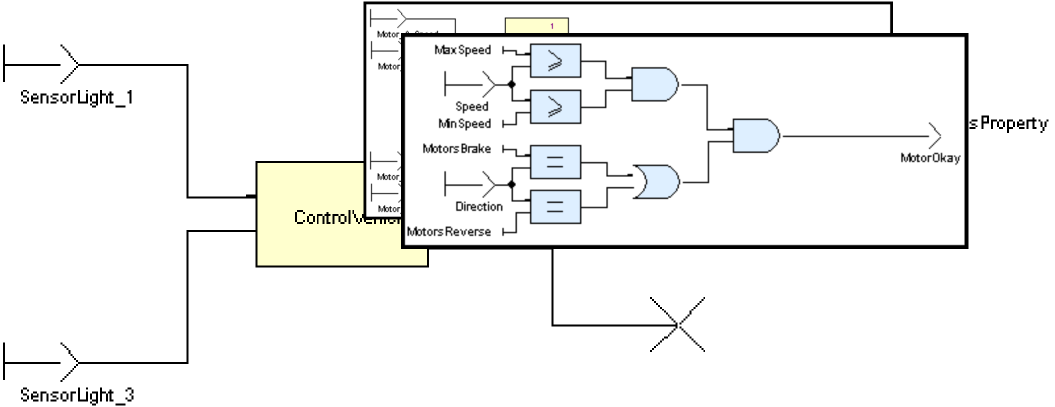
Design Verification



Design Verification



Design Verification



Reactive C

Reactive C [Bou91]

Frederic Boussinot, 1991.

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

```
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();
}
```

Reactive C [Bou91]

	RC	Esterel

`par`
`  printf("1");`
`  printf("2");`
	`12`	`12`

Reactive C [Bou91]

	RC	Esterel

`par`
`  printf("1");`
`  printf("2");`
	`12`	`12`

`present S` `  else emit S end`	**valid**	**invalid:** causality cycle!

Reactive C [Bou91]

	RC	Esterel

`par`
`  printf("1");`
`  printf("2");`
	`12`	`12`

`present S` `  else emit S end`	**valid**	**invalid:** causality cycle!

`present S1`
`  then emit S2 end ||`
`emit S1; present S2`
`  then emit S3 end`
	can be implemented with run-time checks	**valid:**
		instantaneous dialogue

Reactive C [Bou91]

	RC	Esterel

`par`
`  printf("1");`
`  printf("2");`
	12	12

`present S` `  else emit S end`	**valid**	**invalid:** causality cycle!

`present S1`
`  then emit S2 end ||`
`emit S1; present S2`
`  then emit S3 end`
	can be implemented with run-time checks	**valid:**
		instantaneous dialogue

Data Types	Signals, primitive types, structured data	Signals and numeric values

Reactive C [Bou91]

	RC	**Esterel**

`par`
`  printf("1");`
`  printf("2");`
	`12`	`12`

`present S` `  else emit S end`	**valid**	**invalid:** causality cycle!

`present S1`
`  then emit S2 end ||`
`emit S1; present S2`
`  then emit S3 end`
	can be implemented with run-time checks	**valid:**
		instantaneous dialogue

Data Types	Signals, primitive types, structured data	Signals and numeric values
Process Management	dynamic	static

Reactive C [Bou91]

	RC	Esterel
`par printf("1"); printf("2");`	`12`	`12`
`present S else emit S end`	**valid**	**invalid:** causality cycle!
`present S1 then emit S2 end || emit S1; present S2 then emit S3 end`	can be implemented with run-time checks	**valid:** instantaneous dialogue
Data Types	Signals, primitive types, structured data	Signals and numeric values
Process Management	dynamic	static
Compilation and Execution	compiled directly	automaton $\rightarrow$ validation $\rightarrow$ code

Synchronous C

Synchronous C [vH09] (and SJ)

Reinhard von Hanxleden, 2009.

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.

Synchronous C [vH09] (and SJ)

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

Synchronous C [vH09] (and SJ)

SC Thread Operators

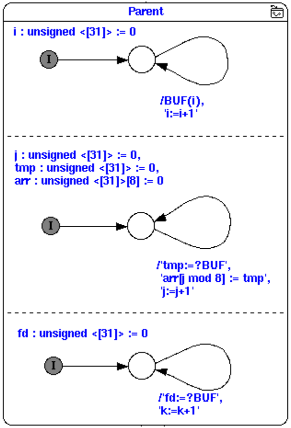
TICKSTART*(*init*, *p*)	Start (initial) tick, assign main thread priority *p*.
TICKEND	Return true (1) iff there is still an enabled thread.
PAUSE*+	Deactivate current thread for this tick.
TERM*	Terminate current thread.
ABORT	Abort descendant threads.
TRANS(*l*)	Shorthand for ABORT; GOTO(*l*).
SUSPEND*(*cond*)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

* possible thread dispatcher call

\+ automatically generates continuation label

Producer-Consumer-Observer in SC

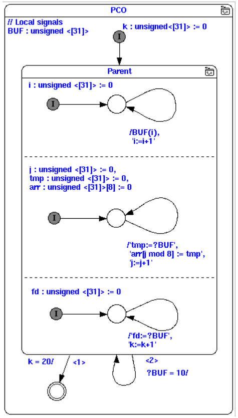
```
int tick(int isInit )
{
  static int BUF, fd, i, j,
    k = 0, tmp, arr[8];

  TICKSTART(isInit, 1);

 PCO:
  FORK(Producer, 3);
  FORK(Consumer, 2);
  FORKE(Observer);

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

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

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

  TICKEND;
}
```



Producer-Consumer-Observer with Preemption in SC

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

  TICKSTART(isInit, 1);

 PCO:
  FORK(Producer, 4);
  FORK(Consumer, 3);
  FORK(Observer, 2);
  FORKE(Parent);
```

```
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

PRET-C [ARGT14]

"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.

PRET-C [ARGT14]

C Language Extensions

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

PRET-C [ARGT14]

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

	Esterel	RC	SC	PRET-C
Commutativity of ||	yes	no	no	no
Communication	signals	signals & variables	variables	variables
Instantaneous dialogue	yes	yes/no	no	no
Signals/variable values/ ... instants	single	multiple	multiple	multiple
Types of aborts	4	4	2	2
Types of suspend	4	4	4	2
Traps	yes	yes	no	no
Non-causal programs	possible	possible	not possible	not possible
Dynamic processes	no	yes	no	no
Compilation	complex	macro exp. resolve || cycle det.	???	macro exp. WCRT

Summary

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!

Thank you! Questions?

References I

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.
IEEE Trans. Comput., 63(7):1600–1612, 2014.

E. Bainomugisha, A. L. Carreton, T. v. Cutsem, S. Mostinckx, and W. d. Meuter.
A survey on reactive programming.
ACM Comput. Surv., 45(4):52:1–52:34, 2013.

A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone.
The synchronous languages 12 years later.
Proceedings of the IEEE, 91(1):64–83, Jan 2003.

F. Boussinot.
Reactive C: An extension of C to program reactive systems.
Softw. Pract. Exper., 21(4):401–428, 1991.

D. Harel.
Statecharts: A visual formalism for complex systems.
Science of Computer Programming, 8(3):231 – 274, 1987.

References II

R. von Hanxleden.
SyncCharts in C: A proposal for light-weight, deterministic concurrency.
In *Proceedings of the Seventh ACM International Conference on Embedded Software*, EMSOFT '09, pp. 225–234, New York, NY, USA, 2009. ACM.

Z. Wan, W. Taha, and P. Hudak.
Real-time FRP.
In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming*, ICFP '01, pp. 146–156, New York, NY, USA, 2001. ACM.