The Specification and Execution of Heterogeneous Synchronous Reactive Systems

Stephen Edwards

Doctoral Qualifying Examination

December 11th, 1995

The problem

We want to describe large systems using a variety of languages.

LISP

Boolean Network

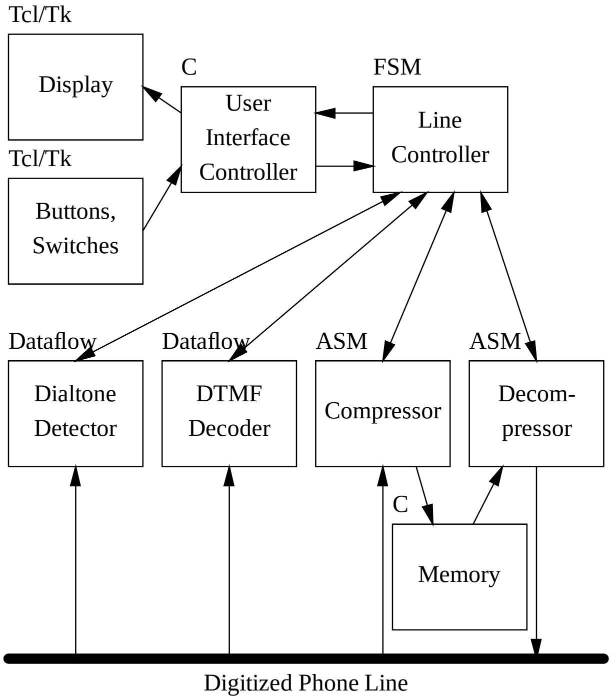
Dataflow

FSMs

C

How to connect them?

Example: Digital answering machine


Tcl/Tk

Display

C

User Interface Controller

FSM

Line Controller

Tcl/Tk

Buttons, Switches

Dataflow

Dialtone Detector

Dataflow

DTMF Decoder

ASM

Compressor

ASM

Decom- pressor

C

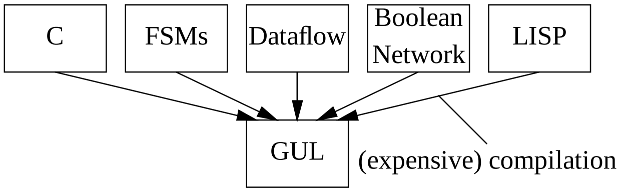
Memory

Digitized Phone Line

Two camps

- **Grand Unified Language**

  Translate everything into a single language:


C

FSMs

Dataflow

Boolean Network

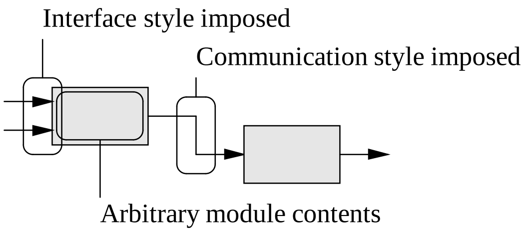
LISP

GUL

(expensive) compilation

- **Hierarchical Heterogeneity** (used here)

  Leave parts of the system abstract:


Interface style imposed

Communication style imposed

Arbitrary module contents

My proposal

Expected contributions:

- A mathematical framework for heterogeneously specifying an important class of systems (reactive) based on an existing communication scheme (synchronous semantics).
- A set of execution schemes (schedulers) for these specifications.
- An efficient implementation in an existing multi-language environment (Ptolemy).

Hypothesis: Synchronous semantics can be made heterogeneous and used effectively to describe reactive systems.

Outline

- Introduction and Motivation
- Scope: Reactive Systems and Synchronous Semantics
- My Specification Scheme and its Mathematical Framework
- Execution Techniques
- Work to Date and Future Work

Scope: Reactive systems

[Harel, Pnueli 85]

- Maintain an ongoing dialog with their environment—listen, don't terminate
- *When* things happen as important as *what* happens
- Discrete-valued, time-varying
- Examples:
  - Systems with user interfaces
    - Digital watches
    - CD players
  - Real-time controllers
    - Anti-lock braking systems
    - Industrial process controllers

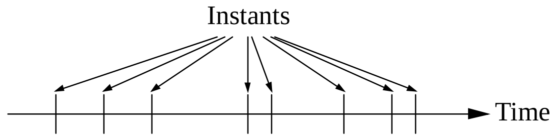
Many currently designed with ad-hoc techniques—difficult to do quickly and reliably

Synchronous semantics

[Berry, Halbwachs, Benveniste, et al.]

Basic idea: **Instantaneous Computation**

Induces a discrete model of time:

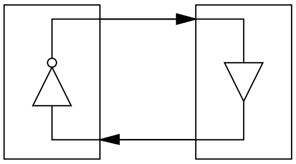


- **Rigorous:** Synthesis, verification made easier.

  Fewer states than asynchronous.
- **Decomposable:** Decomposes without affecting behavior, expressiveness.
- **Predictable:** Deterministic concurrency.
- **Buildable:** Make system faster than environment.

  Difficult to build systems with exact delays.

Cycles and zero delay



A contradictory specification!

A fundamental problem with zero delay

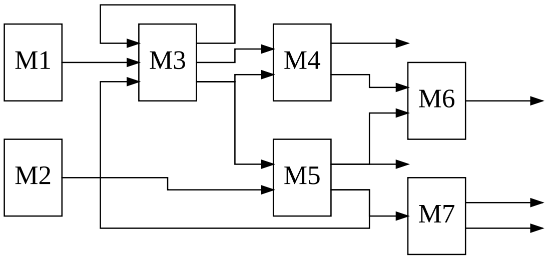
Existing Schemes	**Proposed Scheme**
check at compile time	check at run time
slow compilation	fast compilation
no heterogeneity	allows heterogeneity

Argument: Checking should not be necessary for compilation—it is a verification problem.

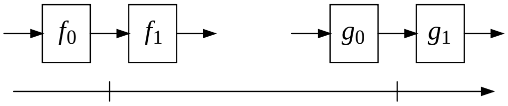
Outline

- Introduction and Motivation
- Scope: Reactive Systems and Synchronous Semantics
- My Specification Scheme and its Mathematical Framework
- Execution Techniques
- Work to Date and Future Work

My systems: Network of communicating modules



- Synchronous: zero-time computation, instants
- Cycles permitted
- Exactly one module drives each “wire”
- Each module computes a function in each instant
- Module functions may change between instants

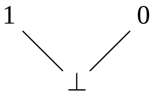


Wire values: Finite complete partial orders

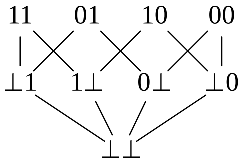
[Scott et al.]

A finite complete partial order (CPO): $(S, \sqsubseteq, \bot)$

- S: Finite set of values
- $\sqsubseteq$: binary relation (“approximates”) on S
  - Transitive: $x \sqsubseteq y$ and $y \sqsubseteq z$ implies $x \sqsubseteq z$
  - Antisymmetric: $x \sqsubseteq y$ and $y \sqsubseteq x$ implies $x = y$
  - Reflexive: $x \sqsubseteq x$
- $\bot \in S$: $\bot \sqsubseteq x$ for all $x \in S$



$\bot \sqsubseteq 0$, $\bot \sqsubseteq 1$

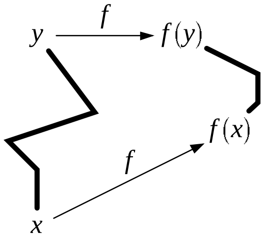


Pointwise extension

Modules: Monotonic functions

A monotonic function $f : S \to S$ has

$$x \sqsubseteq y \text{ implies } f(x) \sqsubseteq f(y)$$



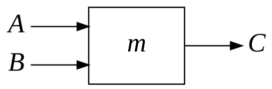
Intuition: Well-behaved functions:
more in ⇒ more out,
“doesn’t change its mind”

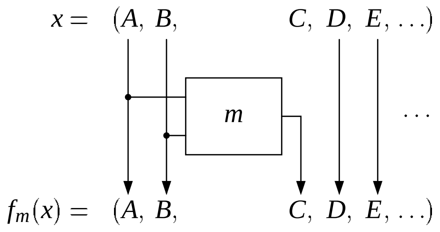
If f and g monotonic, so is $f \circ g$.

Extending module functions

The input and output to each module is the vector of all wires in the system.

However, a module only examines its inputs, only modifies its outputs.

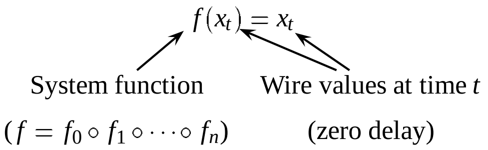




⇒ Input and output domains are the same

Behavior in an instant: The least fixed point

Why a fixed point?



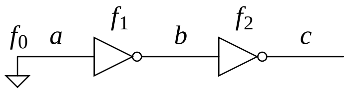
$f(x_t) = x_t$

System function

$(f = f_0 \circ f_1 \circ \cdots \circ f_n)$

Wire values at time t

(zero delay)



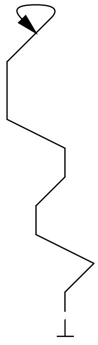
$$\begin{aligned}
(a,b,c) &= (\bot,\bot,\bot)\\
f_0(\bot,\bot,\bot) &= (0,\bot,\bot)\\
f_1(0,\bot,\bot) &= (0,1,\bot)\\
f_2(0,1,\bot) &= (0,1,0)\\
f_2(f_1(f_0(0,1,0))) &= (0,1,0)
\end{aligned}$$

Unique least fixed point theorem

[well-known]

Theorem: A monotonic function on a finite complete partial order has a unique least fixed point.

$$\begin{array}{rcll}
\bot & \sqsubseteq & f(\bot) & \text{(definition of } \bot)\\
f(\bot) & \sqsubseteq & f(f(\bot)) & (f \text{ is monotonic})\\
f(f(\bot)) & \sqsubseteq & f(f(f(\bot))) &
\end{array}$$



Behavior: least fixed point of a monotonic function on a finite CPO

Implications:

- unique
- always defined
- quickly computed
- heterogeneous
  (only care about monotonicity)

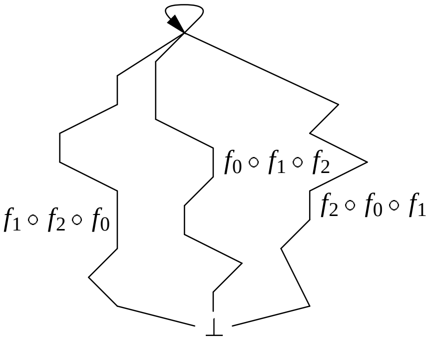
Order-invariance theorem

[Murthy, Edwards 95]

Theorem: The least fixed point is the same for all composition orders of these functions.

Proof. (technical) Consequence of “one wire,” “one driver” rule.

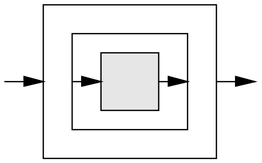
Implication: Behavior independent of module evaluation order—optimize for speed, code size, etc.



Interfacing with other languages

Original problem: Using multiple languages

One solution: Build a generic module interface

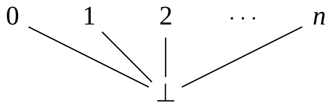


Outside:
A strict
monotonic
function

Inside:
Simple
“function call”
semantics

- Need a complete partial order

  Solution: Build a flat CPO:



- Need a monotonic function

  Solution: Make the foreign function strict:

$$f(\ldots,\perp,\ldots) = \perp$$

Outline

- Introduction and Motivation
- Scope: Reactive Systems and Synchronous Semantics
- My Specification Scheme and its Mathematical Framework
- Execution Techniques
- Work to Date and Future Work

Implementation

Problem: In each instant, find the least fixed point.

Solution: (follows from proof of fixed point theorem)

$$\perp \sqsubseteq f(\perp) \sqsubseteq f(f(\perp)) \sqsubseteq \cdots \sqsubseteq \text{LFP} = \text{LFP} = \cdots$$

For each instant,

1. Start with all wires at $\perp$
2. Evaluate all module functions (in some order)
3. If any change their outputs, repeat Step 2

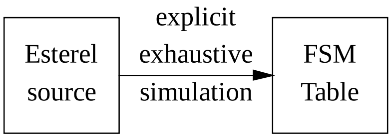
Challenge: Reduce the number of function evaluations.

Order-invariance ensures same result for all orderings.

Other Execution Schemes

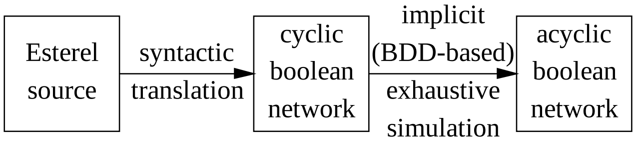
Esterel V3 Compiler: Tabular FSM
[Berry et al. 88]

Recall results from a table at run time.



Esterel V4 Compiler: Boolean Network
[Berry, Shiple, Malik et al. 94]

Simulate a boolean network at run time.



Execution Schemes Compared

Execution Scheme	Heterogeneous	Compilation Time	Executable Size	Execution Speed
Tabular FSM	no	exp.	exp.	const.
Boolean Network	no	exp.	poly.	poly.
Convergent Iteration	yes	poly.	poly.	poly.

My scheme

No checking for contradications

Scheduling

Possible objectives

- Minimize execution time or code size

Possible approaches

- Fully static scheduling

  Determine evaluation order once at compile-time.
- Fully dynamic scheduling

  Determine evaluation order at run-time.

Possible techniques

- Clustering (e.g., [Buck 93])
- Weak Topological Ordering [Bourdoncle 93]
- Strong Component Decomposition [Buhl et al. 93]
- Minimum feedback arc set (NP-complete)

Outline

- Introduction and Motivation
- Scope: Reactive Systems and Synchronous Semantics
- My Specification Scheme and its Mathematical Framework
- Execution Techniques
- Work to Date and Future Work

Work to date

- **Proof of concept**:

  Wrote a compiler for synchronous language Esterel with simpleminded scheduler

lines	297	467	619
V3 Compilation (m:s)	0:52	4:43	15:57
My Compilation (m:s)	0:02	0:03	0:03
V3 Executable (K)	870	3700	12200
My Executable (K)	64	80	96
V3 Execution Time (s)	2.8	4.8	6.6
My Execution Time (s)	2.3	2.6	3.2

- **Foundation for future work**:

  A mathematical framework based on finite complete partial orders and monotonic functions.

  - unique solution always exists
  - can be evaluated different ways

Future work

- Extend and polish the mathematical framework
- Implement this scheme as a domain in Ptolemy
  - Write a simple-minded reference scheduler
  - Create primitive modules
  - Devise foreign module interface(s)
- Work on scheduling schemes
  - Find an exact algorithm for the optimal schedule (probably NP-complete)
  - Devise heuristics for approaching the optimum

Conclusion

- A heterogeneous approach to reactive systems based on synchronous semantics
- Expected contributions:
  1. A mathematical framework for describing reactive systems using synchronous semantics
  2. A set of scheduling algorithms for efficient execution
  3. A practical implementation of these
- Proof-of-concept compiler created
- Mathematical framework created