Modelling with Streams in Daisy
and
The SchemEngine Project

Steven D Johnson
Indiana University Computer Science Department
System Design Methods Laboratory
sjohnson@cs.indiana.edu
www.cs.indiana.edu/~sjohnson

Eric Jeschke
Computer Science Department
The University of Hawaii, Hilo
jeschke@hawaii.edu
uhhach.uhh.hawaii.edu/~jeschke/
Outline

I. Background and context

II. Modeling with streams in Daisy
   A. Lazy CONS demand-oriented computation, stream I/O, concurrency.
   B. Stream systems.
   C. Some modeling techniques.
   D. Distributed extensions.

III. The SchemEngine Project
   A. Design derivation
   B. Previous studies, Schemachine
   C. VLISP [Guttman, Ramsdell, Wand, L&SC 95]
   D. SchemEngine objectives in integrated formal analysis
   E. Toward an integrated codesign environment
I. Background and context, 1975–now
Background (languages)

75-80  Functional programming languages
       Fridman & Wise, “CONS should not evaluate its arguments”
       Applicative programming for systems
       Extensions for indeterminacy, set: \([\alpha \ \beta \ \cdots]\)
       Suspending construction model [continuations, engines in Scheme]
       Semantics ?!

80-85  Daisy/DSI in Unix
       Stream-based I/O From concurrency to parallelism
       O’Donnell, programming environments, hardware models

85-90  Parallel DSI [Jeschke95]
       Language-driven architecture \(\Rightarrow\) design derivation

90-95  Bounded speculation
       Windows on the data space
       Daisy/DSI for small-scale MIMD?

95-00  Distributed demand propagation
       HW models as case studies
Background (methods)

75-80  Functional programming methods
       Prosser & Winkel, structured digital design, ASMs
80-85  Compiler derivation [Wand]
       Combinator factorization ⇒ “machine”
       Stream systems and synchronous hardware
       Formalized synthesis
85-90  Digital design derivation
       Functional/algebraic formalism
       GC-PLD, GC-VLSI, SECD
90-95  DDD ⇒ · · · [Bose, Tuna, Rath, W.Hunt]
       Heterogeneous reasoning
       FM8502, FM9001-DDD, Schemachine
       DDD vis-a-vis PVS, coinductive types [Minor]
95-00  Tools
       Behavior tables
       etc.
II. Modeling with streams in Daisy

Animation is a key aspect of functional formalism.

- Suspending CONS, demand oriented computation.
- List (stream) representation of I/O
- Concurrency construct, set
- Windowing support
Suspending CONS

Delays? No.
Futures? No.
Engines? Almost.
Demand driven computation? No.
Demand oriented computation \( \cdots \)
bounded speculation

DSI:
- Heap based symbolic multiprocessing
- Transparent process management
Processes as streams

\[ Zs: \begin{array}{l}
\text{input } N; \\
\text{cycle} \\
\{ \text{output } N; \\
N := N + 1 \\
\} 
\end{array} \]

\[ F: n = [n ! F: \text{inc}: n] \quad N = [0 ! (\text{map:inc}): N] \]
Stream (i.e. lazy-list) based I/O

I/O synchronization and suspension coercion use the same synchronization mechanism (e.g. a presence bit)

keyboard events

console prompts

stream of characters

scnis

stream of tokens

prsis

stream of s-expressions

evlst

stream of values

prsos

stream of tokens

scnos

stream of characters

screen

display events
NOT = (map:not)
OR = (mapxps:or)

RSFF = \([S R]\).
rec
  \[Qh = [0 ! OR:[S NOT:Ql]]\]
  \[Ql = [1 ! OR:[S NOT:Qh]]\]
  \([Qh Ql]\)

bit-filter = \([C ! Cs]\).
  if:[ same?:[C "0"] [0 ! bit-filter:Cs]
    same?:[C "1"] [1 ! bit-filter:Cs]
    bit-filter:Cs
  ]

xps:
  RSFF:[bit-filter:console:"\[NL\] S: "
    bit-filter:console:"\[NL\] R: "]
& xps:
RSFF:[bit-filter:console:"NL S:"
    bit-filter:console:"NL R:"
]

```
[ S: 0 0 0 0 0 0 0 0 1 0 ]
[ R: 0 0 0 1 1 1 0 0 0 0 ]
[ S: 0 0 0 1 1 1 0 0 0 0 ]
[ R: 0 0 0 0 0 0 0 0 0 0 ]
```

![Diagram of RSFF circuit](diagram.png)
**Widget devices:**

wndi: $name \rightarrow char^*$

wndo: $[name, char^*] \rightarrow []$

*Also:* file i/o,
socket i/o$^*$,
exec i/o$^*$,...

wndo: $[ "[Qh Ql]\"$  
prsos:  
scnos:  
xps:  
RSFF: [ bitfilter:wndi:"S"  
bitfilter:wndi:"R" ]]

Concurrency

*Implicit* through bounded speculation

*Explicit* through (constructs such as) set

• Notation: expression $A$, computation $\alpha$, result $a$.

• set: $[A \ B \ C]$ constructs list object $L = [\alpha \ \beta \ \gamma]$

• $L$ becomes manifest as $[a \ b \ c]$, or $[b \ a \ c]$, or $[b \ c \ a]$, etc.

• There is an imprecise operational relationship with computational effort.

• *Semantics (?!)*
DFF = \[a b c e f g].
[CLK D].
rec
A = [a ! AND:[D NOT:B]]
B = [b ! AND:[CLK
AND:[NOT:A
NOT:C]]]
C = [c ! AND:[CLK
NOT:E]]
E = [e ! AND:[NOT:C
NOT:A]]
F = [f ! OR:[C
NOT:G]]
G = [g ! OR:[NOT:F
B]]
in
[CLK D A B C E F G [NL *]]
let
[CLK D A B C E Ql Qh NLs] = (DFF:[2 2 2 2 1 0]):
[ f-t:wndi:"CLK"
  f-t:wndi:"D"
]
in
head:
set:
[ consume:wndi:"DFF model"
  wndo:["A_B_C_D_E" s-p-x:[NLs A B C D E]]
  wndo:["F=Qh" s-p-x:[NLs Qh]]
  wndo:["G=Ql" s-p-x:[NLs Ql]]
]

consume = \Cs. if:[ nil?:Cs [] consume:tail:Cs ]
Asynchronous interactions

The pseudo-function \( \text{gif} \) ("guarded" if) takes a list of guarded expressions, \([ \ldots [g_k \ v_k] \ldots ]\) and returns one of the values, \(v_j\), whose guard is \(true\).

| FAIL: a distinguished value |
| guard:[Bool Value] --> Value + %FAIL% |
| gif:{[Bool , Value] ... [Bool, Value]} --> Value + %FAIL% |

def gif = \Guards.
rec
    GUARD = \[Test Result].
    if Test then [Result] else FAIL

LOOP = \Guards = [G ! Gs].
if       nil?:Guards
    then  FAIL
else if fail?:G
    then  LOOP:Gs
else  head:G
in
    LOOP: set:(map:GUARD):Guards
\( Ce = \text{rec} \)
\[
Se = \{ [Xh \rightarrow Xt] = Xs \rightarrow [Yh \rightarrow Yt] = Ys \}.
\]
\[
gif:\{ \text{[present?}:Xh \text{ Sy}([Xt \ Ys]]) \rightarrow [present?}:Yh \text{ Sx}([Xs \ Yt]) \}
\]
\[
Sx = \{ [Xh \rightarrow Xt] = Xs \rightarrow [Yh \rightarrow Yt] = Ys \}.
\]
\[
gif:\{ \text{[present?}:Xh \text{ Xh} \rightarrow \text{Se}([Xt \ Ys])] \rightarrow [present?}:Yh \text{ Sx}([Xs \ Yt]) \}
\]
\[
SY = \{ [Xh \rightarrow Xt] = Xs \rightarrow [Yh \rightarrow Yt] = Ys \}.
\]
\[
gif:\{ \text{[present?}:Xh \text{ Se}([Xt \ Ys])] \rightarrow [present?}:Yh \rightarrow [Yh \rightarrow \text{Se}([Xs \ Yt])] \}
\]
in Se
Interaction refinements

\[
\text{wndo: [}
\quad \text{"OUT"}
\quad \text{interleave_prompts: [}
\quad \quad \quad \quad \text{[NL ! wndi:'IN']}
\quad \quad \quad \quad \text{ints:1]}]
\text{]}
\]

\[
\text{rec}
\quad \text{Cs = wnd: ["IN"}
\quad \quad \quad \quad \text{insert_prompts [NL ! Cs]}
\quad \quad \quad \quad \text{ints:0]}]
\quad \text{in}
\quad \quad \text{wndo: ["OUT" Cs]}
\]

**Left Diagram:**

- **IN:** abc def ghi
  - jkl mno pqr
  - stu vwx yzA

- **OUT:**
  - 1: abc def ghi
  - 2: jkl mno pqr
  - 3: stu vwx yzA
  - 4:

**Right Diagram:**

- **IN:**
  - 0: abcde
  - 6: fghij
  - 12: klmno

- **OUT:**
  - abcde
  - fghij
  - klmno
Refinements

source = \[D Vs CLKs].
gif:[
    [ present?:head:Vs
        [head:Vs ! source:[head:Vs tail:Vs CLKs]]
    [present?:head:CLKs
        [D ! source:[D Vs tail:CLKs]]
    ]
]

rec
    CLK = wndi:"CLK"
    INA = source:["?" wndi:"INA" CLK]
    INB = source:["?" wndi-lg:"INB" CLK]
in
    wndo:["OUTAB" s-p-x:[INA INB NLs]]
Modeling systems with Daisy, conclusions

Daisy/DSI is not a production language. It is an experimental vehicle. A central motivation of the work has been to explore implications of demand-oriented computation on architecture. We are currently looking at distributed modeling and demand propagation across networks. We would like to contribute to a modeling methodology based on functional expressions and streams. One topic of interest is the proper abstraction of devices (given that they have effects).

- **output**: a (unit?) function that
  - merges all streams it is applied to?
- **input**: an (nullary?) function that splits on demand?
- Is bidirectionality basic (sockets, dialogues [O’Donnell])?
III. The SchemEngine Project

Objectives [Johnson, IUTR 544]

- Advancing design derivation
- System-level formal analysis
- From semantics to hardware

- Heterogeneous reasoning
- Embedded applications (!?)
- Foundations for high-confidence
DDD studies in language-driven architecture

www.cs.indiana.edu/hmg/

- Garbage collectors in PLDs, VLSI, FPGAs [Boyer 86-90]
- SECD computer [Wehrmeister 89]
- Schemachine [Burger 94]

The bigger picture

- Compiler correctness [Wand, Clinger 80-85]
- Compiler *derivation* [Wand 80-85]
- Scheme based methodology and pedagogy
- Codesign tools
• CPU, GC, ALLOC, INIT derived with DDD
• CPU is naive
• Memory system tuned to GC
• PLDs, mux-based FPGAs, DRAM simms.
• Codesign using Scheme and Logic Engine
  – Spec. models
  – Derived models
  – Staging to hardware
Relationship to VLISP [Guttman, Ramsdell, Wand, L&SC 95]

Scheme Semantics

Compiler

Machine

Machine + heap

Target architecture

System model

System factorization

Data refinement

Combinator factorization

(formalized synthesis)

Hardware implementation
Pending issues

CPU: The architecture is naive; it needs retiming. The specification could be at a much higher level, closer to VLISP/EoPL.

GC: Non-interference verification at the system, algorithm verification. Investigation of caching and incremental variants.
**Bus timing:** Redesign needed for current memory architectures. Could model checking have found the killer bug?

Large

\[
\begin{align*}
\text{read}(\text{NEW}, a) & \parallel \text{write}(\text{OLD}, b, c) \\
\text{write}(\text{NEW}, a, b) & \parallel \text{read}(\text{OLD}, c) \\
\text{read}(\text{NEW}, a) & \parallel \text{mark-next}(\text{OLD}) \\
\text{write}(\text{NEW}, a, b) & \parallel \text{mark-next}(\text{OLD})
\end{align*}
\]
Synchronization: Hierarchical clocking was used.

Other pending issues

- Retargetting Schemachine (Virtex, SDRAM?), cores
- Behavior table case study
- Interface abstraction
- Software factorization
- Integrating **modeling**, derivation, and co-design
- Close with VLISP
- Embedded resource management
- ...Java (?!)
