# Sixth International Workshop on Designing Correct Circuits

Vienna, 25–26 March 2006

A Satellite Event of the ETAPS 2006 group of conferences

## Participants' Proceedings

Edited by Mary Sheeran and Tom Melham

# Preface

This volume contains material provided by the speakers to accompany their presentations at the Sixth International Workshop on Designing Correct Circuits, held on the 25th and 26th of March 2006 in Vienna. The workshop is a satellite event of the ETAPS group of conferences. Previous workshops in the informal DCC series were held in Oxford (1990), Lyngby (1992), Båstad (1996), Grenoble (2002), and Barcelona (2004). Each of these meetings provided a stimulating occasion for academic and industrial researchers to get together for discussions and technical presentations, and the series as a whole has made a significant contribution to supporting our research community.

The 2006 DCC workshop again brings together researchers in formal methods for hardware design and verification. It will allow participants to learn about the current state of the art in formally based hardware verification and design and it is intended to further the debate about how more effective design and verification methods can be developed.

For some time now, research in hardware verification is being done industrial laboratories, as well as in universities. Industry is commonly focussed on relatively immediate verification goals, but also keeps our work grounded in practical engineering problems. To make progress on the longer-term problems in our field, academic and industrial researchers must continue to work together on the problems facing microprocessor and ASIC designers now but also into the future. A major aim of the DCC series of workshops has been to provide a congenial and relaxed venue for communication among researchers in our community. DCC 2006 attracted a very strong field of submissions, and we hope the selection the Programme Committee has made will keep the debate stimulating and productive. We look forward to two great days of presentations and discussion.

We wish to express our gratitude to the members of the Programme Committee for their work in selecting the presentations, and to all the speakers and participants for their contributions to Designing Correct Circuits.

Mary Sheeran and Tom Melham
March 2006

# Programme Committee

Dominique Borrione (TIMA, Grenoble University, France)
Elena Dubrova (KTH, Sweden)
Niklas Eén (Cadence Design Systems, USA)
Warren Hunt (UT Austin, USA)
Robert Jones (Intel Corporation, USA)
Wolfgang Kunz (TU Kaiserslautern, Germany)
Per Larsson-Edefors (Chalmers, Sweden)
Andrew Martin (IBM Research, USA)
Tom Melham (Oxford University, UK)
Johan Mårtensson (Jasper Design Automation, Sweden)
John O'Leary (Intel Corporation, USA)
Carl Pixley (Synopsys, USA)
Mary Sheeran (Chalmers, Sweden)
Satnam Singh (Microsoft Corporation, USA)
Joe Stoy (Bluespec, USA)
Jean Vuillemin (École Normale Supérieure, France)

*DCC 2006*

# Microprocessor Verification Based on Datapath Abstraction and Refinement

Zaher S. Andraus, Mark H. Liffiton, and Karem A. Sakallah

Department of Electrical Engineering and Computer Science
University of Michigan
Ann Arbor, MI 48109-2122

{zandrawi,liffiton,karem}@eecs.umich.edu

Counterexample Guided Abstraction Refinement (CEGAR for short) has been shown to be an effective paradigm in a variety of hardware and software verification scenarios. Originally pioneered by Kurshan [7], it has since been adopted by several researchers as a powerful means for coping with verification complexity. The wide-spread use of such a paradigm hinges, however, on the automation of its abstraction *and* refinement phases. Without automation, CEGAR requires laborious user intervention to choose the right abstractions and refinements based on a detailed understanding of the intricate interactions among the components of the design being verified. Clarke et al. [3], Jain et al. [5], and Dill et al. [2] have successfully demonstrated the automation of abstraction and refinement in the context of model checking for safety properties of hardware and software systems. In particular, these approaches create a smaller abstract transition system from the underlying concrete transition system and iteratively refine it with the spurious counterexamples produced by the model checker. The approaches in [3] and [5] are additionally based on the extraction of unsatisfiability explanations derived from the infeasible counterexamples to provide stronger refinement of the abstract model and to significantly reduce the number of refinement iterations.

All of these approaches are examples of *predicate abstraction* which essentially projects the concrete model onto a given set of relevant predicates to produce an abstraction suitable for model checking a given property. In contrast, we describe in [1] a methodology for *datapath abstraction* that is particularly suited for equivalence checking. In this approach, datapath components in behavioral Verilog models are automatically abstracted to uninterpreted functions and predicates while refinement is performed manually using the ACL2 theorem prover [6].

The use of (near)-minimal explanations of unsatisfiability forms the basis of another class of abstraction methods. These include the work of Gupta et al. [4] and McMillan et al. [8] who employ "proof analysis" techniques to create an abstraction from an unsatisfiable concrete bounded model checking (BMC) instance of a given depth.

In this talk we explore the application of CEGAR in the context of microprocessor correspondence checking. The approach is based on automatic datapath abstraction as in [1] augmented with automatic refinement using minimal unsatisfiable subset (MUS) extraction. One of our main conclusions is the necessity of basing refinement on the extraction of MUSes from both the abstract and concrete models. Additionally, refinement tends to converge faster when multiple MUSes are extracted in each iteration. Finally, localization and generalization of spurious counterexamples are also shown to be crucial for refinement to converge quickly. We will describe our implementation of these ideas in the Reveal system and discuss the effectiveness of the various refinement options in the verification of a few benchmarks.

**REFERENCES**

[1] Z. S. Andraus and K. A. Sakallah, "Automatic Abstraction of Verilog Models", In Proceedings of 41st Design Automation Conference 2004, pp. 218-223.

[2] S. Das and D. Dill, "Successive Approximation of Abstract Transition Relations" in 16th Annual IEEE Symposium on Logic in Computer Science (LICS) 2001.

[3] E. Clarke, O. Grumberg. S. Jha, Y. Lu and H. Veith, "Counterexample-Guided Abstraction Refinement," In CAV 2000, pp. 154-169.

[4] A. Gupta, M. Ganai, Z. Yang, and P. Ashar, "Iterative Abstraction Using SAT-based BMC with Proof Analysis." In Proc. of the International Conference on CAD, pp. 416-423, Nov. 2003.

[5] H. Jain, D. Kroening and E. Clarke, "Predicate Abstraction and Verification of Verilog," Technical Report CMU-CS-04-139.

[6] M. Kaufmann and J. Moore, "An Industrial Strength Theorem Prover for a Logic Based on Common Lisp." IEEE Transactions on Software Engineering 23(4), April 1997, pp. 203-213.

[7] R. Kurshan, "Computer-Aided Verification of Coordinating Processes: The Automata-Theoritic Approach," Princeton University Press, 1999.

[8] K. L. McMillan and N. Amla, "Automatic Abstraction without Counterexamples." In International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'03), pp. 2-17, Warsaw, Poland, April, 2003, LNCS 2619.

# An Implementation of Clock-Gating and Multi-Clocking in Esterel

Laurent Arditi, Gérard Berry, Marc Perreaut
Esterel Technologies
{Laurent.Arditi,Gerard.Berry,Marc.Perreaut}@esterel-technologies.com

Michael Kishinevsky
Intel Strategic CAD Labs
Michael.Kishinevsky@intel.com

Clock gating and multi-clocking are now common design techniques that are used for power reduction and for handling systems with different operational frequencies. They cannot be handled by Classic Esterel language and tools because the Classic Esterel is a single clock synchronous paradigm and Esterel compilers can generate single clock circuits only. To cover broader class of design needs, we propose to extend Esterel to other clocking schemes, including clock-gating and multi-clocking.

This extension must satisfy three major needs:

- enhance the scope of designs that can be modeled in Esterel,
- allows to generate different implementations depending on the final target: a single clock circuit (e.g. for compiling a specification into a basic FPGA), a circuit with clock-gating or an equivalent circuit without clock-gating, and a multi-clock circuit (e.g. for compiling to an ASIC). The choice of the implementation should be possible at compilation time, without requiring any change in the source model.
- provide support by all tools comprising the Esterel development framework: the graphical Esterel entry, software simulation and debug, embedded code generation, formal verification, optimization.

The core of the implementation for the clock-gating is based on a new Esterel instruction called `weak suspend`. This instruction freezes the control and data registers, while letting the combinational part computing the values as functions of inputs and the state. The effect of this instruction is similar to an effect of clock-gating on a hardware block. We developed an Esterel compiler which can generate RTL code (VHDL and Verilog) with the embedded clock-gating logic or with the regular equivalent logic to emulate functional behavior of clock-gating without the corresponding power saving.

The multi-clock design in the new Esterel compiler is based on the paradigm of Globally Asynchronous Locally Synchronous principle and is implemented using a few language extensions: multi-clock units, clock signals, clock gaters and clock multiplexers, and clock definition in module instantiations. The compiler can generate a truly modular and multi-clock RTL code, or mono-clock RTL code based on the `weak suspend` instruction. The latter compilation mode is also used for software simulation and formal verification. The trace equivalence of different forms of the design is guaranteed correct-by-construction.

We show a few classical multi-clock examples, including a dual-clock FIFO and a synchronizer based on a four-phase handshake protocol. A formal verification of this protocol is also discussed in detail.

# Using Lava and Wired for design exploration

Emil Axelsson, Koen Claessen, Mary Sheeran

Chalmers University of Technology

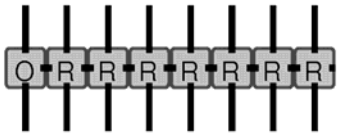
DCC 2006

## Design exploration

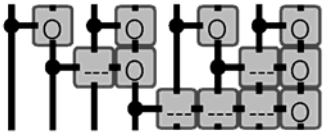
- Design exploration:
  - Comparing a set of designs in terms of non-functional properties (power, speed, area, manufacturability …)
- Function is the same, parameters to vary:
  - Word lengths
  - Netlist topology
  - Layout topology
  - Wiring topology

## Example – carry tree

- Serial (ripple carry)
  - ***O*(*n*)** speed/size




- Parallel (Sklansky)
  - ***O*(log *n*)** speed
    ***O*(*n*·log *n*)** size




## Non-functional properties

- Routing wires account for
  **≈75%** of path delays
  **≈50%** of the total power consumption
  in a typical high-performance design
- Design exploration requires **wire-aware** design methods

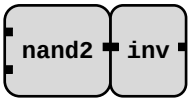
## Design exploration in Lava

- Functional style:

```
and2 (a,b) = inv (nand2 (a,b))

Main> simulate and2 (high,low)
low
```

- Layout combinator style:

```
and2 = nand2 ->- inv
```




## Design exploration in Lava (2)

- Functional style
  - Higher abstraction level, flexibility
  - Weak connection to the real hardware
  - Non-functional estimation with respect to gates only
- Layout combinator style
  - Lower abstraction level, less flexibility
  - Stronger connection to hardware
  - Limited reasoning about wires

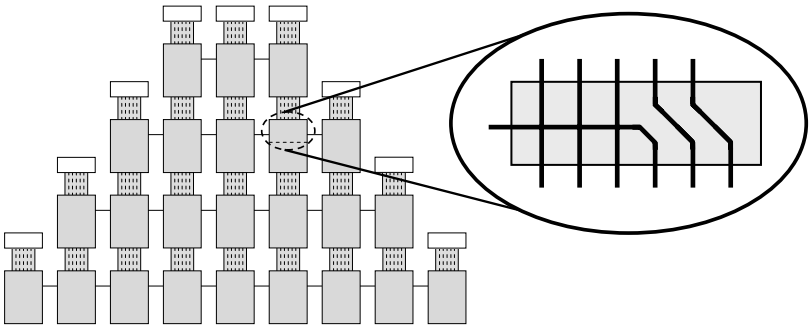
# Design exploration in Lava (3)

Simulation with non-standard signal interpretation (NSI) gives unit-delay estimation

```
rippAdder op = row op

input = (low, [(high,low),(high,high),(low,low)])

Main> simulate (rippAdder fullAdd) input
([high,low,high],low)

faNSI (a,(b,c)) = (d,d)
  where d = maximum [a,b,c] + 1

Main> (rippAdder faNSI) (0, [(0,0),(0,0),(0,0)])
([1,2,3],4)
```

# Design exploration in Lava (4)

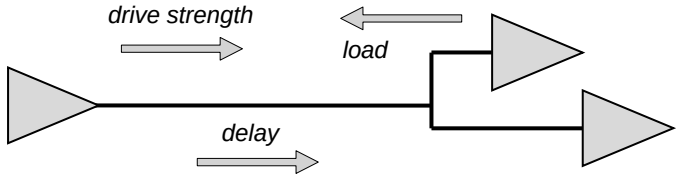
Clever circuit generators can adapt to non-functional properties



*M. Sheeran. Generating fast multipliers using clever circuits. FMCAD 2004.*

## Limitations in Lava

- Better modeling of wires needed
  - No built-in support for geometry
  - NSI in functional setting can only handle “forwards” properties (unit delay)
- Proper modeling of wire delay also needs to know about **load** (“bidirectional” properties)

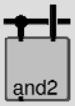


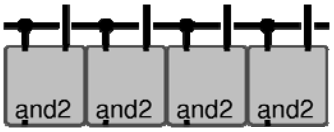

## Wired

- Models circuits as relational blocks with detailed geometry
- Wires are first-class circuits
- NSI in relational setting supports bidirectional properties
- **Allows wire-aware design exploration**

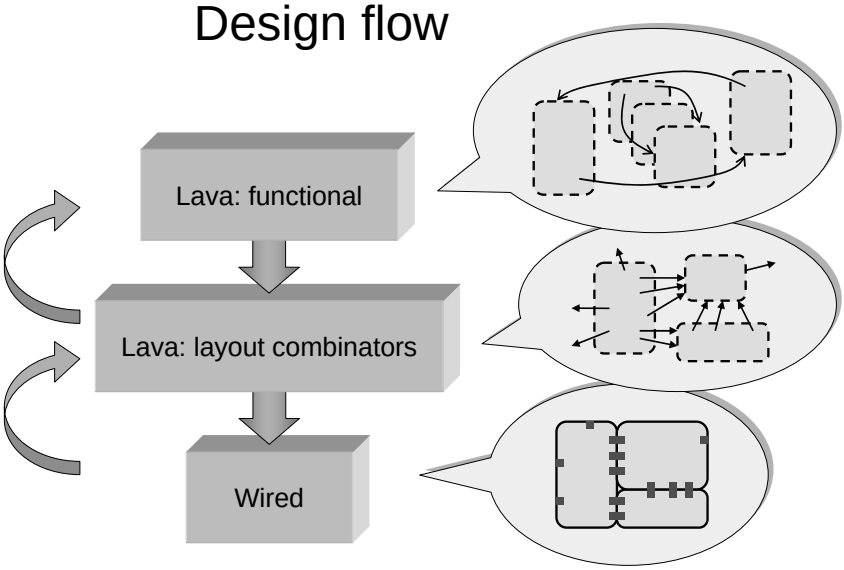
# Wired – example

```
bitMult1 = and2 *=* (crT0 *||* cro)


bitMult = row bitMult1

evalAndView (bitMult :: Description ())
            (XP, pl [XP,XP,XP,XP], XP,XP)
```




# Design flow



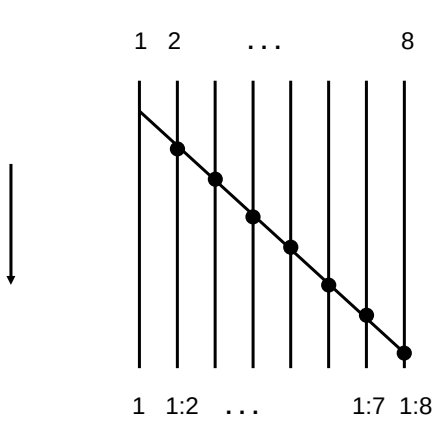
## Case study Prefix

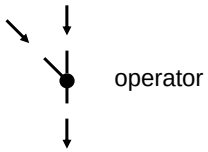
Given inputs
x1, x2, ... xn

Compute
x1, x1*x2, x1*x2*x3, . . . , x1*x2*...*xn

For * an associative (but not necessarily commutative) operator

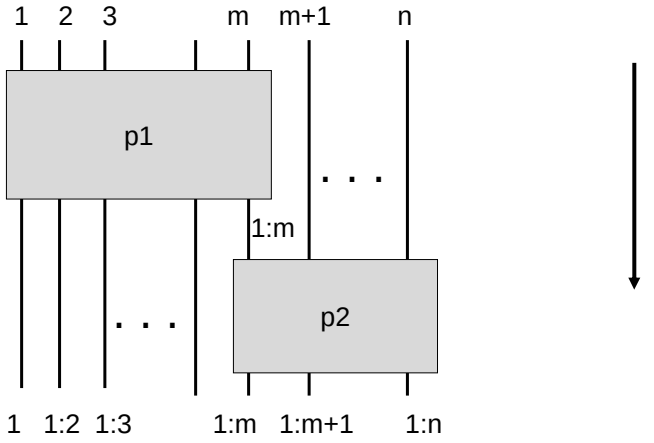
## Serial prefix




8 inputs
depth 7
size 7

## Composing prefix circuits



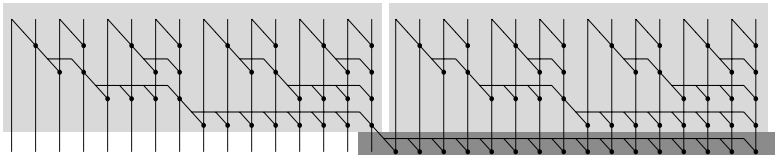

## Composition combinator

```
compose2 p1 p2 as = (init l1s) ++ r1s
  where
    (ls,rs) = splitAt t as
    l1s     = p1 ls
    r1s     = p2 ((last l1s): rs)
    t       = div (length as + 1) 2
```

## Serial prefix again

```
ripple op [a]    = [a]
ripple op [a,b]  = [a,op(a,b)]
ripple op as
    = compose2 (ripple op) (ripple op) as
```

## Sklansky



```
withEach op p (a:bs) = a:[op(a,b) | b <- p bs]

sklansky op [a] = [a]
sklansky op as
  = compose2 (sklansky op) (withEach op (sklansky op)) as
```

## Checking

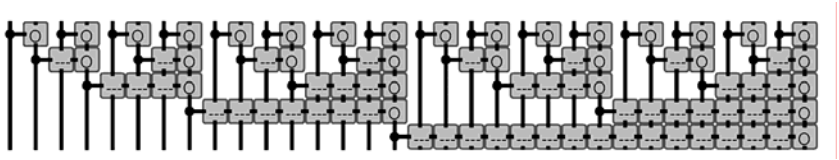
```
Main> simulate (sklansky plus) [1..9]
[1,3,6,10,15,21,28,36,45]
```

```
Main> sklansky append [[i] | i <- [1..6]]
[[1],[1,2],[1,2,3],[1,2,3,4],[1,2,3,4,5],[1,2,3,4,5,6]]
```

Pictures are also drawn by non-standard interpretation. Run the circuit and gather info. in a data type.

## Sklansky in Wired

```
sklansky 0   = rowN 1 (wyl 0)
sklansky dep = join *=~ (sub ~||~ sub)
  where
    sub  = sklansky (dep-1)
    join = (row wy1 ~||* wf) -||- (row d2 ~||* d)
```



## Limit fanout, number of operators, depth

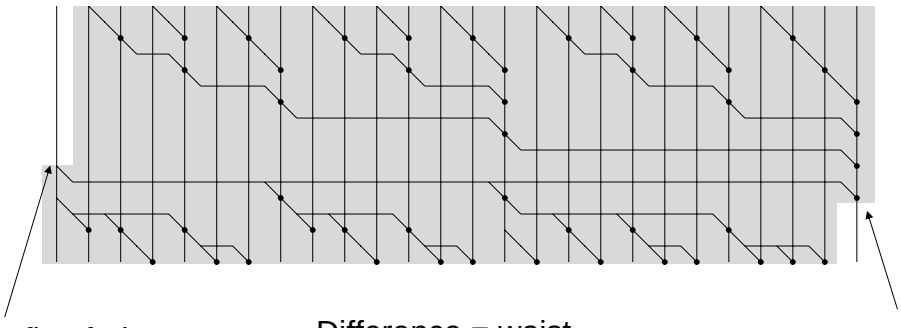
Produce only depth-size optimal circuits

depth+size = 2n-2 for n inputs

(serial prefix is DSO, but Sklansky is not)

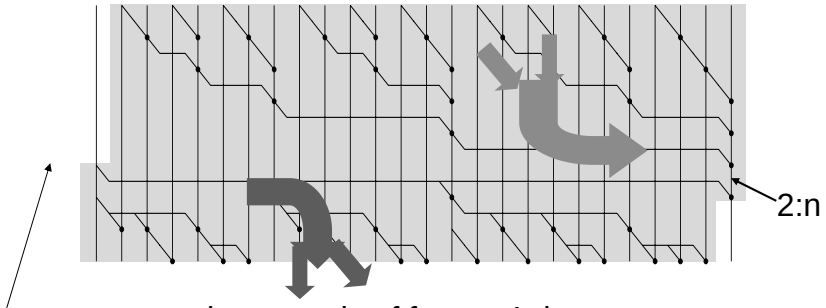
How? Design appropriate building blocks

## Slice (also parallel prefix)




waist + size = 2n-2 waist size optimal

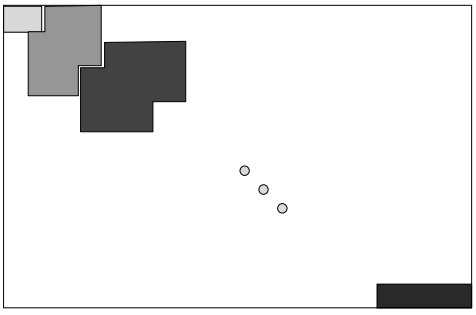
# Slice




each output is of form 1: i

Top tree spreads every input to last output
Bottom tree spreads first input to each output

# Slices

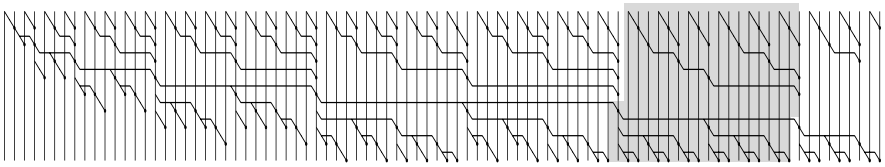


depth = d
= waist

Compose d such blocks

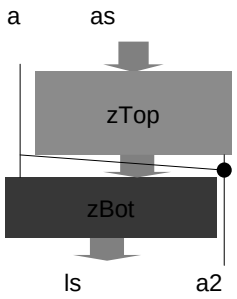
waist + size = 2n-2 (waist size optimal) =>
depth + size = 2n - 2 (depth size optimal)

# Slices (example)



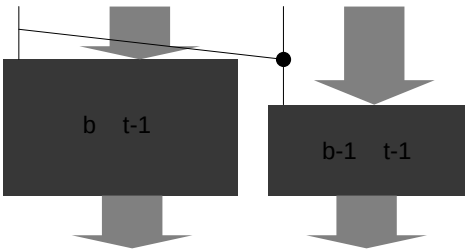
# Functional Lava

```
zslices d pop
    = composeKN [(zsliceW (d-i) i, zslice(d-i) i)|
                  i <- [0..d-1]]
  where
    zslice b t [a]     = [a]
    zslice b t (a:as)  = ls ++ [a2]
      where
        ms  = zTop b t as
        a2  = pop (a,last ms)
        ls  = zBot b t (a:init ms)


    ...

    fTop b 0 [a] = [a]
    ...
```



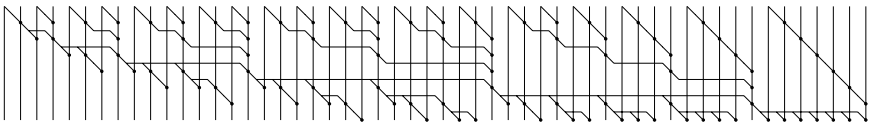
# Bottom tree

```
zBot b 0 [a] = [a]
zBot 0 t [a] = [a]
zBot b t as  = ls ++ rs
  where
    (l1s,r1s) = splitAt (ztW b (t-1)) as
    e2        = pop (head l1s,head r1s)
    ls        = zBot b (t-1) l1s
    rs        = zBot (b-1) (t-1) (e2:tail r1s)
```




Top tree defined similarly

# zslices 7



fanout = depth + 1

## Limit fanout

max fanout

```
fslices f d ps   = composeKN [..]
  where
    (wy,pop) = ps
    fslice b t (a:as)  = ls ++ [a2]
      where
        ms = fTop (f-1) b t as
        a2 = pop (a,last ms)
        ls = fBot (f-1) b t (a:init ms)
```

parameters

fanout at top of bottom tree (leaving one for waist)

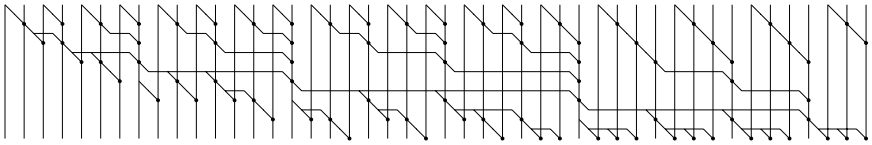
## Bottom tree

parameter to track fanout used up

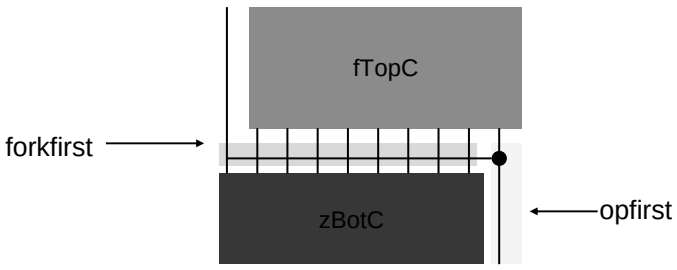
fanout only one => step down one level

```
fBot n b 0 [a] = [a]
fBot n 0 t [a] = [a]
fBot 1 b t as  = (fsT wy ->- fBot f (b-1) t) as
fBot n b t as  = ls ++ rs
  where
    (l1s,r1s) = splitAt (ftW (n-1) b (t-1)) as
    e2        = pop (head l1s,head r1s)
    ls        = fBot (n-1) b (t-1) l1s
    rs        = fBot f (b-1) (t-1)(e2:tail r1s)
```

fslices 4 7



## Next step: combinators

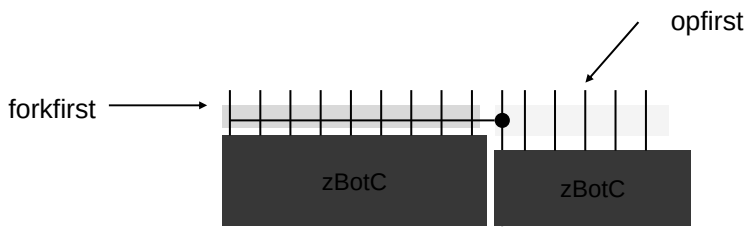



Slice

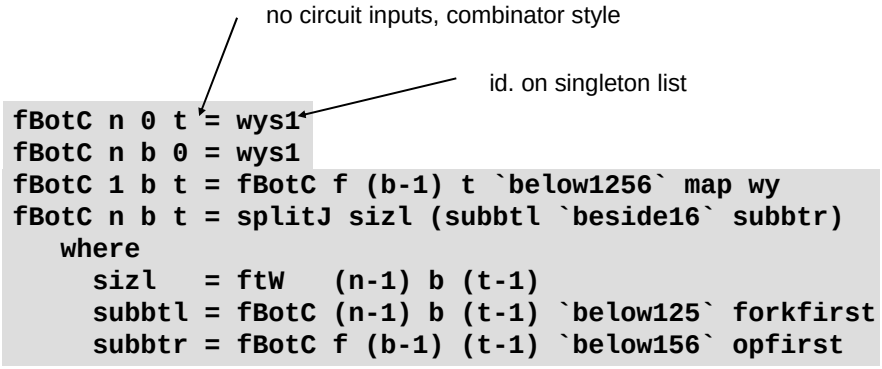
## Slice (combinators)

```
splitJ n p = splitAt n ->- p ->- append

fsliceC b t = splitJ 1 (tosnD  (fTopC (f-1) b t)) ->-
              splitJ sz (sbl `beside16` opfirst)
       where
         sz  = ftW  (f-1) b t
         sbl = fBotC (f-1) b t `below125` forkfirst
```

## Bottom tree



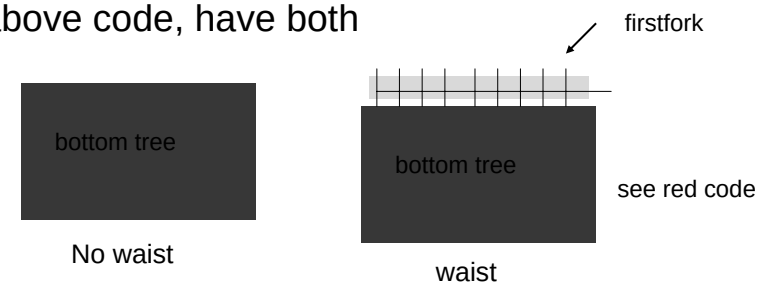
# Bottom tree combinator code




dIfferent forms of below and beside combinators, all wire crossings also tiles, no named wires, communication by abuttment

# Now think more about layout size

In above code, have both




To squeeze layout, should have same height (for same parameters).
Implement as two separate (mutually recursive) functions

## Slice

```
fsliceC1 b t = splitJ 1 (tosnD  (fTopC (f-1) b t)) ->-
               splitJ sz (sbl `beside16` opfirst)
       where
         sz  = ftW    (f-1) b t
         sbl = fBotCW (f-1) b t
```

waisted version

## No waist,  fbtcN

```
fBotCN n 0 t = wys1
fBotCN n b 0 = wys1
fBotCN 1 b t = fBotCN f (b-1) t `below1256` map wy
fBotCN n b t = splitJ szl (subbtl `beside16` subbtr)
   where
     szl    = ftW    (n-1) b (t-1)
     subbtl = fBotCW (n-1) b (t-1)
     subbtr = fTopCN f (b-1) (t-1) `below156` opfirst
```

calls waisted version  (and vice versa)

## Sanity check

```
sanity f d m
   =  and (zipWith (==)
           ((aslicesC f d m (id,copy,append))
               [[a]| a <- l])
           (tail (inits l)))
    where l = [1..m]
```

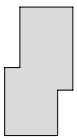
```
Main> sanity 4 8 57
True

Main> sanity 4 9 109
True
```

## Easy step to Wired

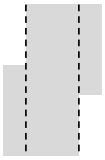
```
fBotCN n 0 t = single (wys 0)
fBotCN n b 0 = single (wys b)
fBotCN 1 b t = fBotCN f (b-1) t ~~=** forkFirst
fBotCN n b t = subbtl ~~||~~ subbtr
  where
    subbtl = fBotCW (n-1) b (t-1)
    subbtr = fBotCN f (b-1) (t-1) ~~=** opFirst
```

## Non-rectangular blocks

- Descriptions in Wired must be rectangular
- Slices are not!

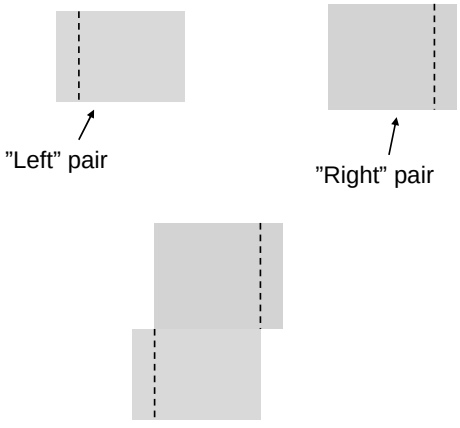


- Trick: represent each slice by three blocks (triple)



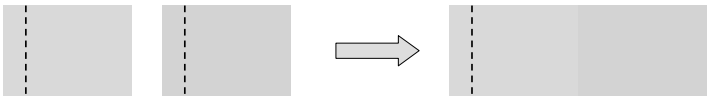
## Non-rectangular blocks

Construct triples from pairs

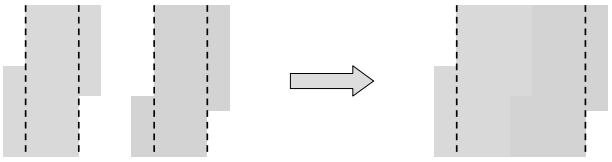


## New combinators

- Beside for pairs **(~~||~~):**



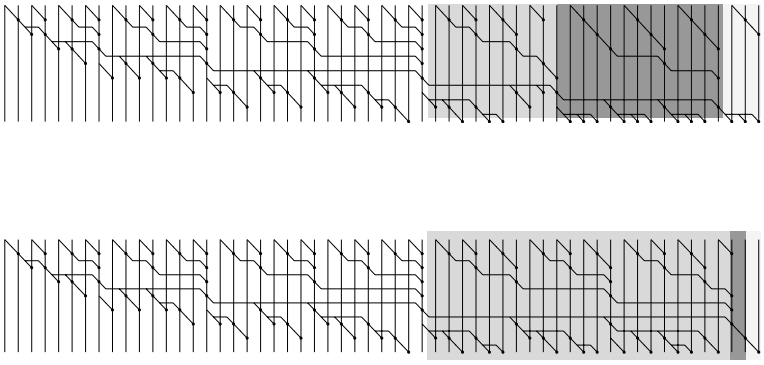
- Composing slices:



## Now add flexibility of size

- Return to Lava
- Add parameter for number of inputs and systematically crop individual slices by replacing some matching top and bottom trees by single wires
- Have experimented with cropping from right or in the middle
- Hoped cropping in middle would be better as it reduces length of longest wire on waist

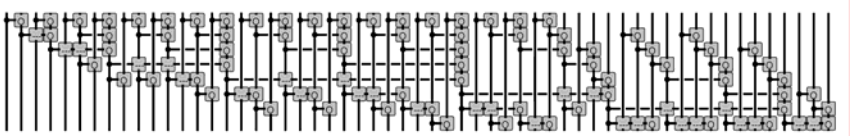
## Examples



57 inputs, cropped from slices 4 8, which has 72 inputs

## And then back to Wired

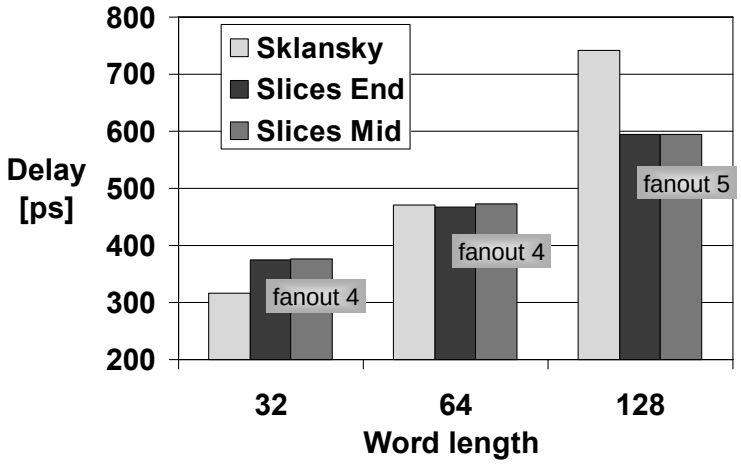
- Programming cleverness translates directly
- This is the point!



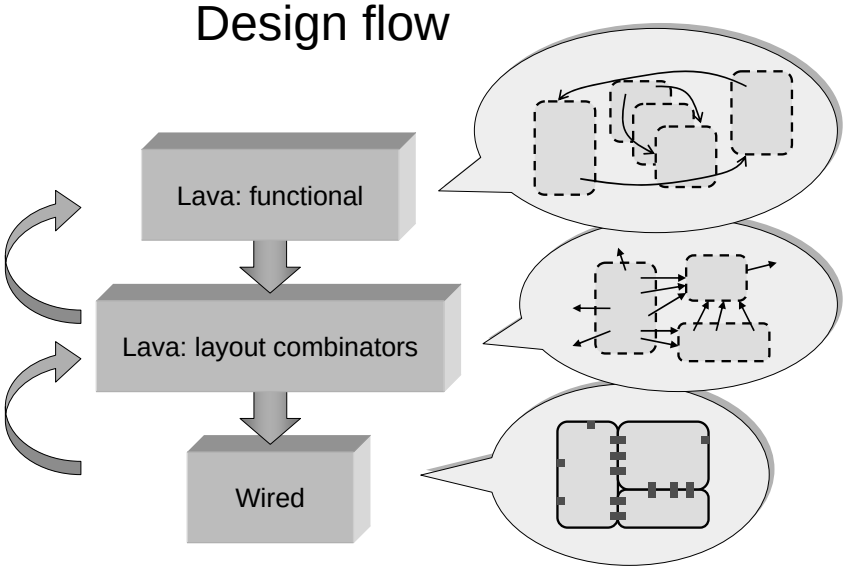
57 inputs, cropped (middle) from slices 4 8

## Design exploration by delay analysis

(Elmore for fictitious 100nm process, Huang&Ercegovac'97)




## Design flow



## Conclusion

- It works!
- Going back and forth between Lava and Wired really helps. Gives understanding and hence simpler solutions
- In the case study, we tried to go to Wird too early, and got over-complicated code (too many cases)
- We think we know what the steps are, but need to do more case studies
- Seem to need a new programming idiom in Lava, involving explicit calculations of numbers of inputs. Need to simplify this.

## Next

- Produce real layout from Wired
- Really merge Lava and Wired
- More analyses
- More case studies
- Adaptive circuits
- Study power consumption

# Reachability Analysis with QBF

Armin Biere

Institute for Formal Models and Verification
Johannes Kepler University Linz, Austria

Workshop Designing Correct Circuits

DCC'06

Vienna, Austria, March, 2006

## Model Checking



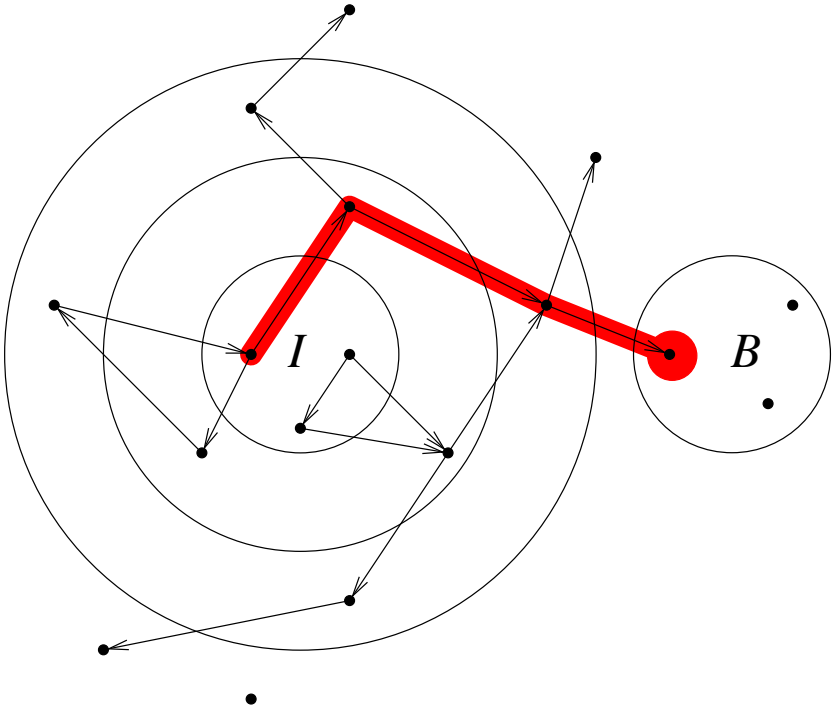
- explicit model checking [ClarkeEmerson'82], [Holzmann'91]
  - program presented symbolically (no transition matrix)
  - traversed state space represented explicitly
  - e.g. reached states are explicitly saved bit for bit in hash table

  $\Rightarrow$ State Explosion Problem (state space exponential in program size)

- symbolic model checking [McMillan Thesis'93], [CoudertMadre'89]
  - use symbolic representations for sets of states
  - originally with Binary Decision Diagrams [Bryant'86]
  - Bounded Model Checking using SAT [BiereCimattiClarkeZhu'99]

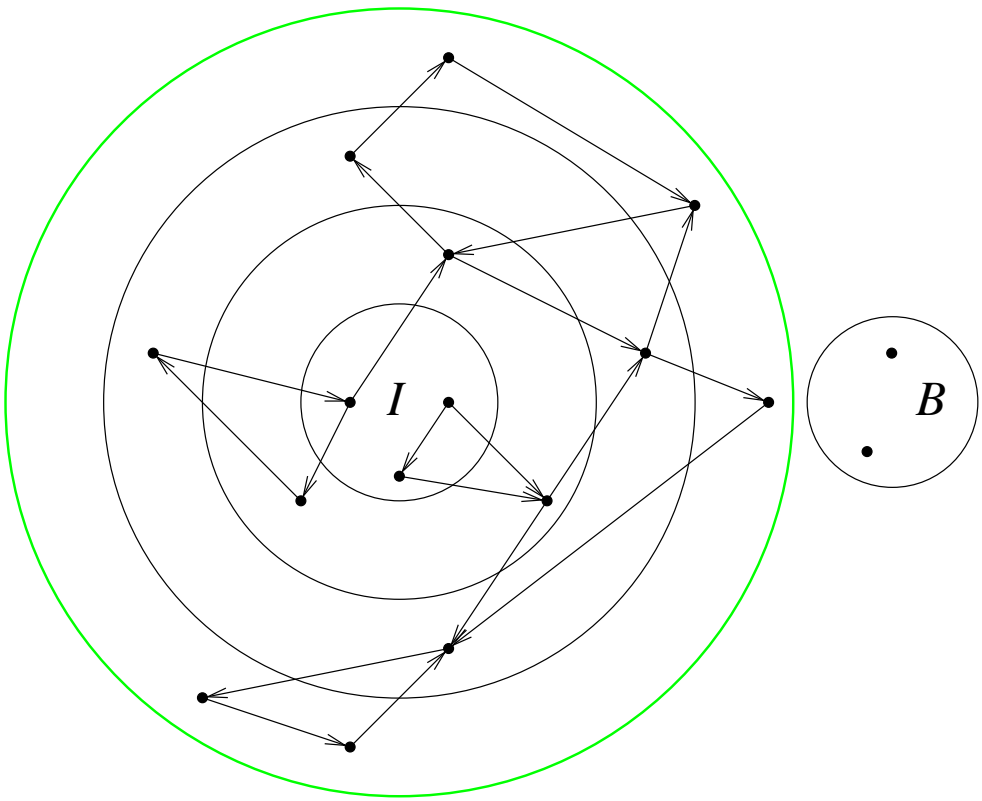
## Forward Fixpoint Algorithm: Bad State Reached






## Forward Fixpoint Algorithm: Termination, No Bad State Reachable





initial states $I$, transition relation $T$, bad states $B$

model-check$^{\mu}_{\text{forward}}$ $(I, T, B)$
  $S_C = \emptyset$; $S_N = I$;
  **while** $S_C \neq S_N$ **do**
    **if** $B \cap S_N \neq \emptyset$ **then**
      **return** "found error trace to bad states";
    $S_C = S_N$;
    $S_N = S_C \cup Img(S_C)$ ;
  **done**;
  **return** "no bad state reachable";

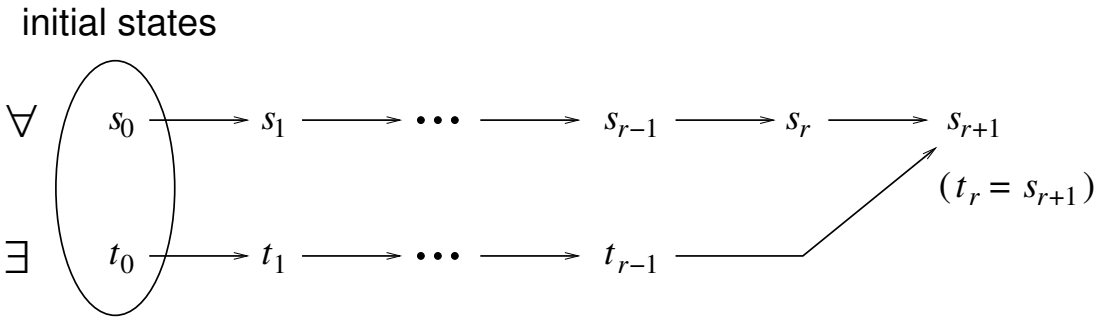
symbolic model checking represents set of states in this BFS symbolically



| | | |
|---|---|---|
| 0: continue? | $S^0_C \neq S^0_N$ | $\exists s_0[I(s_0)]$ |
| 0: terminate? | $S^0_C = S^0_N$ | $\forall s_0[\neg I(s_0)]$ |
| 0: bad state? | $B \cap S^0_N \neq \emptyset$ | $\exists s_0[I(s_0) \wedge B(s_0)]$ |
| 1: continue? | $S^1_C \neq S^1_N$ | $\exists s_0, s_1[I(s_0) \wedge T(s_0,s_1) \wedge \neg I(s_1)]$ |
| 1: terminate? | $S^1_C = S^1_N$ | $\forall s_0, s_1[I(s_0) \wedge T(s_0,s_1) \rightarrow I(s_1)]$ |
| 1: bad state? | $B \cap S^1_N \neq \emptyset$ | $\exists s_0, s_1[I(s_0) \wedge T(s_0,s_1) \wedge B(s_1)]$ |
| 2: continue? | $S^2_C \neq S^2_N$ | $\exists s_0, s_1, s2[I(s_0) \wedge T(s_0,s_1) \wedge T(s_1,s_2) \wedge \neg (I(s_2) \vee \exists t_0[I(t_0) \wedge T(t_0,s_2)])]$ |
| 2: terminate? | $S^2_C = S^2_N$ | $\forall s_0, s_1, s2[I(s_0) \wedge T(s_0,s_1) \wedge T(s_1,s_2) \rightarrow I(s_2) \vee \exists t_0[I(t_0) \wedge T(t_0,s_2)]]$ |
| 2: bad state? | $B \cap S^1_N \neq \emptyset$ | $\exists s_0, s_1, s_2[I(s_0) \wedge T(s_0,s_1) \wedge T(s_1,s_2) \wedge B(s_2)]$ |

$$\forall s_0,\ldots,s_{r+1}[\, I(s_0) \wedge \bigwedge_{i=0}^{r} T(s_i,s_{i+1}) \rightarrow$$

$$\exists t_0,\ldots,t_r[\, I(t_0) \wedge s_{r+1}=t_r \wedge \bigwedge_{i=0}^{r-1} (t_i=t_{i+1} \vee T(t_i,t_{i+1}))]]$$




(we allow $t_{i+1}$ to be identical to $t_i$ in the lower path)

**radius** is smallest $r$ for which formula is true



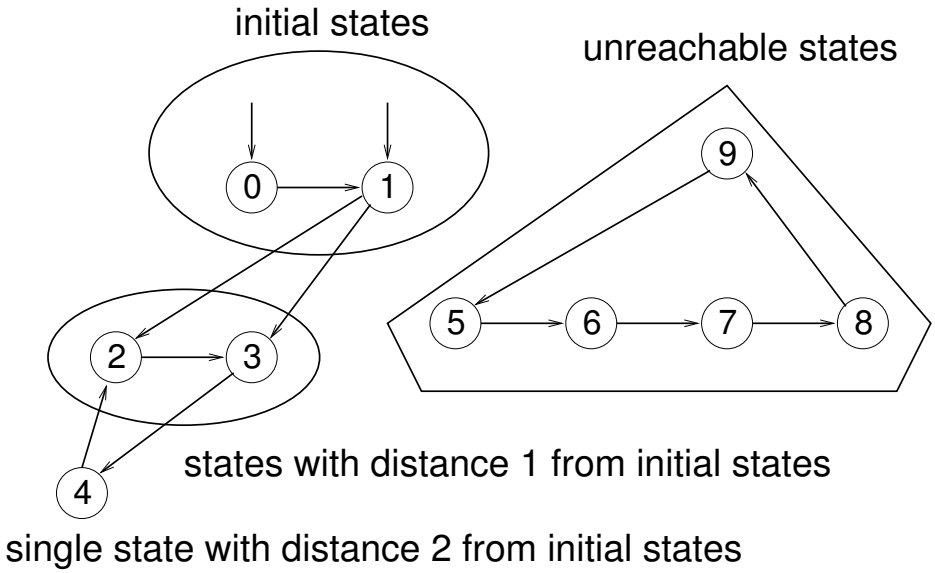
- propositional logic (SAT $\subseteq$ QBF)
  - constants $0,1$
  - operators $\wedge,\neg,\rightarrow,\leftrightarrow,\ldots$
  - variables $x,y,\ldots$ over boolean domain $\mathbb{B}=\{0,1\}$
- quantifiers over boolean variables
  - valid $\forall x[\exists y[x \leftrightarrow y]]$ (read $\leftrightarrow$ as $=$)
  - invalid $\exists x[\forall y[x \leftrightarrow y]]$

- semantics given as expansion of quantifiers

$$\exists x[f] \equiv f[0/x] \vee f[1/x] \qquad \forall x[f] \equiv f[0/x] \wedge f[1/x]$$

- expansion as translation from SAT to QBF is exponential
  - SAT problems have only existential quantifiers
  - expansion of universal quantifiers doubles formula size
- most likely no polynomial translation from SAT to QBF
  - otherwise PSPACE = NP





- checking $S_C = S_N$ in 2nd iteration results in QBF decision problem

$$\forall s_0, s_1, s2[I(s_0) \wedge T(s_0, s_1) \wedge T(s_1, s_2) \rightarrow I(s_2) \vee \exists t_0[I(t_0) \wedge T(t_0, s_2)]]$$

- not **eliminating quantifiers** results in QBF with one alternation
  - checking whether bad state is reached only needs SAT
  - number iterations bounded by radius $r = O(2^n)$

- so why not forget about termination and concentrate on bug finding?

  $\Rightarrow$ Bounded Model Checking (BMC)

## BMC Part of Fixpoint Algorithm

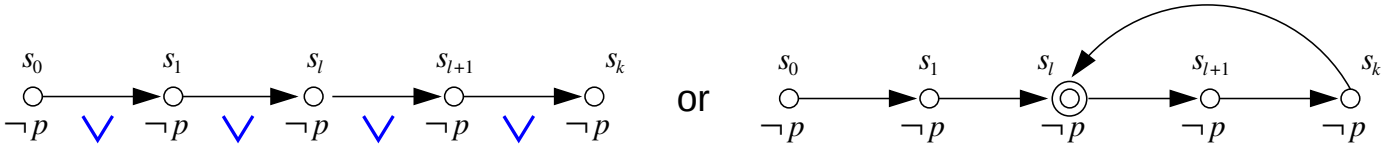


| | | |
|---|---|---|
| ~~0: continue?~~ | $S_C^0 \neq S_N^0$ | $\exists s_0[I(s_0)]$ |
| ~~0: terminate?~~ | $S_C^0 = S_N^0$ | $\forall s_0[\neg I(s_0)]$ |
| 0: bad state? | $B \cap S_N^0 \neq \emptyset$ | $\exists s_0[I(s_0) \wedge B(s_0)]$ |
| ~~1: continue?~~ | $S_C^1 \neq S_N^1$ | $\exists s_0, s_1[I(s_0) \wedge T(s_0, s_1) \wedge \neg I(s_1)]$ |
| ~~1: terminate?~~ | $S_C^1 = S_N^1$ | $\forall s_0, s_1[I(s_0) \wedge T(s_0, s_1) \rightarrow I(s_1)]$ |
| 1: bad state? | $B \cap S_N^1 \neq \emptyset$ | $\exists s_0, s_1[I(s_0) \wedge T(s_0, s_1) \wedge B(s_1)]$ |
| ~~2: continue?~~ | $S_C^2 \neq S_N^2$ | $\exists s_0, s_1, s2[I(s_0) \wedge T(s_0, s_1) \wedge T(s_1, s_2) \wedge \neg (I(s_2) \vee \exists t_0[I(t_0) \wedge T(t_0, s_2)])]$ |
| ~~2: terminate?~~ | $S_C^2 = S_N^2$ | $\forall s_0, s_1, s2[I(s_0) \wedge T(s_0, s_1) \wedge T(s_1, s_2) \rightarrow I(s_2) \vee \exists t_0[I(t_0) \wedge T(t_0, s_2)]]$ |
| 2: bad state? | $B \cap S_N^1 \neq \emptyset$ | $\exists s_0, s_1, s_2[I(s_0) \wedge T(s_0, s_1) \wedge T(s_1, s_2) \wedge B(s_2)]$ |

# Bounded Model Checking (BMC)

[BiereCimattiClarkeZhu TACAS'99]

- look only for counter example made of $k$ states (the bound)




- simple for safety properties $\mathbf{G}p$ (e.g. $p = \neg B$)

$$I(s_0) \wedge (\bigwedge_{i=0}^{k-1} T(s_i, s_{i+1})) \wedge \bigvee_{i=0}^{k} \neg p(s_i)$$

- harder for liveness properties $\mathbf{F}p$

$$I(s_0) \wedge (\bigwedge_{i=0}^{k-1} T(s_i, s_{i+1})) \wedge (\bigvee_{l=0}^{k} T(s_k, s_l)) \wedge \bigwedge_{i=0}^{k} \neg p(s_i)$$

# Bounded Model Checking State-of-the-Art



- increase in efficiency of SAT solvers [ZChaff,MiniSAT,SATelite]
- SAT more robust than BDDs in bug finding
  (shallow bugs are easily reached by explicit model checking or testing)
- better unbounded but still SAT based model checking algorithms
  - $k$-induction [SinghSheeranStålmarck'00]
  - interpolation [McMillan'03]
- 4th Intl. Workshop on Bounded Model Checking (BMC'06)
- other logics beside LTL and better encodings
  e.g. [LatvalaBiereHeljankoJuntilla'04]

## Induction



[SinghSheeranStålmarck'00]

- more specifically $k$-induction
  - does there exist $k$ such that the following formula is unsatisfiable

$$T(s_0,s_1) \wedge \cdots \wedge T(s_{k-1},s_k) \wedge B(s_k) \wedge \bigwedge_{0 \leq i < j \leq k} s_i \neq s_j$$

  - if UNSAT and $\neg$BMC($k$) then bad state unreachable
  - it is further possible to assume $\neg B(s_i)$ for all $i < k$
- backward version of initialized reoccurrence diameter
- $k = 0$ check whether $\neg B$ tautological (propositionally)
- $k = 1$ check whether $\neg B$ inductive for $T$

## Occurrence Diameter Explosion



- **diameter** longest shortest path between two states
- **occurrence diameter** longest *simple* path
  - simple = without reoccurring state
- occurrence diameter can be exponentially larger than diameter
  - $n$ bit register with load signal, initialized with zero
  - reoccurrence diameter $2^n - 1$
  - diameter 1
- applies to backward reoccurrence diameter and $k$ induction as well

### Transitive Closure

$$T^* \equiv T^{2^n}$$

(assuming $= \subseteq T$)

### Standard Linear Unfolding

$$T^{i+1}(s,t) \equiv \exists\, m[\, T^i(s,m) \wedge T(m,t)]$$

### Iterative Squaring via Copying

$$T^{2 \cdot i}(s,t) \equiv \exists\, m[\, T^i(s,m) \wedge T^i(m,t)]$$

### Non-Copying Iterative Squaring

$$T^{2 \cdot i}(s,t) \equiv \exists\, m[\forall c[\exists l,r[(c \rightarrow (l,r) = (s,m)) \wedge (\overline{c} \rightarrow (l,r) = (m,t)) \wedge T^i(l,r)]]]$$

## Hierarchy



- flat circuit model exponential in size of hierarchical model
  - $M_0$ has one signal resp. register
  - $M_{i+1}$ instantiates $M_i$ twice
  - $M_n$ has $2^n$ signals resp. registers
- model hierarchy/repetitions in QBF as in non-copying iterative squaring
  - $T$ interpreted as combinatorical circuit with inputs $s$, outputs $t$
- **conjecture:** [Savitch70] even applies to hierarchial descriptions

## Experiments

still work in progress

- bounded model checker for flat circuits with $k$ induction
- can also produce forward/backward diameter checking problems in QBF
- sofar instances have been quite challenging for current QBF solvers
- found some toy examples which can be checked much faster with QBF
  - for instance the $n$ bit register with load signal discussed before
- non-copying iterative squaring does not give any benefits (yet)



## DPLL for SAT and QBF

dpll-sat(*Assignment* S) [DavisLogemannLoveland62]

```
boolean-constraint-propagation()
if contains-empty-clause() then return false
if no-clause-left() then return true
v := next-unassigned-variable()
return dpll-sat(S ∪ {v ↦ false}) ∨ dpll-sat(S ∪ {v ↦ true})
```

dpll-qbf(*Assignment* S) [CadoliGiovanardiSchaerf98]

```
boolean-constraint-propagation()
if contains-empty-clause() then return false
if no-clause-left() then return true
v := next- outermost -unassigned-variable()
@ := is-existential(v) ? ∨ : ∧
return dpll-sat(S ∪ {v ↦ false}) @ dpll-sat(S ∪ {v ↦ true})
```

Why is QBF harder than SAT?

$$\models \quad \forall x \,.\, \exists y \,.\, (x \leftrightarrow y)$$

$$\not\models \quad \exists y \,.\, \forall x \,.\, (x \leftrightarrow y)$$

Decision order matters!



- most implementations DPLL alike: [Cadoli...98][Rintanen01]
  - **learning** was added [Giunchiglia...01] [Letz01] [ZhangMalik02]
  - top-down: split on variables from the outside to the inside
- multiple quantifier elimination procedures:
  - **enumeration** [PlaistedBiereZhu03] [McMillan02]
  - **expansion** [Aziz-Abdulla...00] [WilliamsBiere...00] [AyariBasin02]
  - bottom-up: eliminate variables from the inside to the outside
- **q-resolution** [KleineBüning...95], with **expansion** [Biere04]
- symbolic representations [PanVardi04] [Benedetti05] BDDs

- applications fuel interest in SAT
  - incredible capacity increase (last year: MiniSAT, SATelite)
  - SAT solver competition affiliated to SAT conference
  - SAT is becoming a core verification technology
- QBF is catching up
  - solvers are getting better (first competetive QBF evaluation)
  - new applications
  - richer structure

# "Easy" Parameterized Verification of Cross Clock Domain Protocols

Geoffrey M. Brown
Indiana University,
Bloomington
geobrown@cs.indiana.edu

Lee Pike*
Galois Connections,
leepike@galois.com

February 8, 2006

**Abstract**

This paper demonstrates how an off-the-shelf model checker that utilizes a Satisfiability Modulo Theories decision procedure and $k$-induction can be used for verification applications that have traditionally required special purpose hybrid model checkers and/or theorem provers. We present fully parameterized proofs of two types of protocols designed to cross synchronous boundaries: a simple data synchronization circuit and a serial communication protocol used in UARTs (8N1). The proofs were developed using the SAL model checker and its ICS decision procedures.

## 1 Introduction

This paper uses the bounded model checker and ICS decision procedures of SAL to develop fully parameterized *proofs* of two types of protocols designed to cross synchronous boundaries: a simple data synchronization circuit and a serial communication protocol, 8N1, used in UARTs.[1] Protocols such as these present challenging formal verification problems because their correctness requires reasoning about interacting time events. The proofs discussed in this paper are parameterized by expressing temporal constraints as a system of linear equations. The proofs are "easy" in that they require few proof steps. For example, we have previously presented a proof of the biphase mark protocol [1], which is structurally similar to, though simpler than, 8N1. Our biphase mark proof required 5 invariants, whereas a published proof using PVS required 37; our proof required 5 proof directives (the proof of each invariant is automated), whereas the PVS proof initially required more than 4000 proof directives [2]. Our proofs are quick to check – a few minutes computing time, while one published proof of biphase mark required

*The majority of this work was completed while this author was a member of the Formal Methods Group at the NASA Langley Research Center in Hampton, Virginia.

[1]The SAL specifications and proofs are available at `http://www.cs.indiana.edu/~lepike/pub_pages/dcc.html`.

five hours. Furthermore, our proofs identified a potential bug: in verifying the 8N1 decoder, we found a significant error in a published application note that incorrectly defines the relationship between various real time parameters which, if followed, would lead to unreliable operation [3].

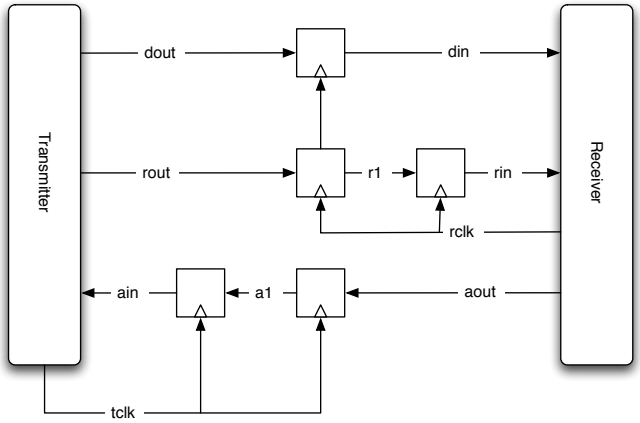



Figure 1: Synchronizer Circuit

The synchronizer circuit considered in this paper, illustrated in Figure 1, is constructed entirely of D-type flip-flops. The circuit, which is commonly used, allows a transmitter in one clock domain to reliably transmit data to a receiver in another clock domain irrespective of the relative frequencies of the clocks controlling the digital circuitry. This circuit allows the transmitter to send a bit (or in general a word) of data to the receiver through an exchange of "request" (`rout`, `rin`) and "acknowledgment" signals (`aout`, `ain`). A temporal illustration of the exchange between transmitter and receiver is presented in Figure 2. Each event initiated by the transmitter must propagate to the receiver and a response must be returned before the transmitter can initiate a new transfer. The protocol followed by the transmitter and receiver is a simple token passing protocol where the transmitter has the token and hence is allowed to modify its outputs only when `ain = rout`, and the receiver has the token and is allowed to read its input data `din` when `rin != aout`. For example, the transmitter sends data when `rout = ain` by setting `dout` to the value that it wishes to send and by changing the state of `rout`. Informally, the circuit satisfies a simple invariant:

$$rin \neq aout \Rightarrow din = dout \tag{1}$$

Although the protocol is trivial, there is a fundamental issue that greatly complicates the behavior of the circuit – metastability. The fact that the two clocks `rclk` and `tclk` are not synchronized and may run at arbitrary relative rates means that we cannot treat the flip-flops in the circuit as simple delay elements. In particular, the correct behavior of a flip-flop depends upon assumptions about when its input may change relative to its clock. Changes occurring too soon before a clock event are said to violate the "setup time" requirement of the flip-flop while changes occurring too soon after a clock event are said to violate the "hold time" requirement. Either violation may cause the flip-flop

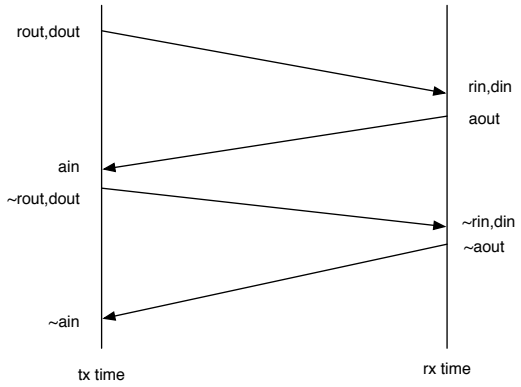



Figure 2: Synchronizer Circuit Timeline

to enter a metastable state in which its output is neither "one" nor "zero" and which may persist indefinitely. In practice, probabilistic bounds may be calculated which define how long a metastable state is likely to persist. The illustrated circuit assumes that the time between two events on a single clock is long enough to ensure that the metastability resolution time (plus setup time) is shorter that the clock period with sufficiently high probability. While there have been other proofs of this circuit, they did not model the effects of metastability [4, 5]. An alternative approach has been proposed and is evidently used in a commercial tool to reproduce synchronization bugs by introducing random one-clock jitter in cross domain signals [6, 7]. A fundamental difference between our work and those cited is that we explicitly model timing effects and rely upon clearly stated timing assumptions to verify the circuit.

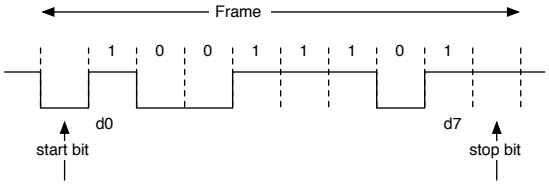



Figure 3: 8N1 Data Transmission

Metastability also is an issue in the behavior of the 8N1 implementation in which a receiver must sample a changing signal in order to determine the boundaries between valid data. To motivate the design of the 8N1 protocol, consider Figure 3 which illustrates the encoding scheme utilized by this protocol. In a synchronous circuit, the data and clock are typically transmitted as separate signals; however, this is not feasible in most communication systems (e.g., serial lines, Ethernet, SONET, Infrared) in which a single signal is transmitted. A general solution to this problem is to merge the clock and data information using a coding scheme. The clock is then recreated by synchronizing a local reference clock to the transitions in the received data. In 8N1 a transition is guaranteed to occur only at the beginning of each *frame*, a sequence of bits that includes a start bit, eight data bits, and a stop bit. Data bits are encoded by the identity function – a 1 is

a 1 and a 0 is a 0. Consequently, the clock can only be recovered once in each frame in which the eight data bits are transmitted.

Thus, the central design issue for a *data decoder* is reliably extracting a clock signal from the combined signal. Once the location of the clock events is known, extracting the data is relatively simple. Although the clock events have a known relationship to signal transitions, detecting these transitions precisely is usually impossible because of distortion in the signal around the transitions due to the transmission medium, clock jitter, and other effects. A fundamental assumption is that the transmitter and receiver of the data do not share a common time base and hence the estimation of clock events is affected by differences in the reference clocks used. Constant delay is largely irrelevant; however, transition time and variable delay (e.g., jitter) are not. Furthermore, differences in receiver and transmitter clock phase and frequency are significant. Any correctness proof of an 8N1 decoder must be valid over a range of parameters defining limits on jitter, transition time, frequency, and clock phase. Finally, any errors in detection can lead to metastable behavior as with the synchronization circuit.

The temporal proofs presented in this paper may be reproducible using specialized real-time verification tools such as Hytech, TReX and Parameterized Uppaal (we leave it as an open challenge to these respective communities to reproduce these models and proofs in the those tools) [8, 9, 10]. However, a key difference is that SAL is a general purpose model checking tool and the real time verification we performed utilized the standard decision procedures. Furthermore, the proofs are not restricted to finite data representations – in the case of the data synchronization circuit our proofs are valid for arbitrary integer data.

The remainder of the paper is organized as follows. In Section 2, we overview the language and proof technology of SAL. The modeling and verification of the synchronizer circuit is presented in Section 3. The model of the 8N1 protocol is presented in Section 4, and its verification is described in Section 5. In Section 6, we first describe how to derive error bounds on an operational model from a fully-parameterized one, and then we describe how this the operational model reveals errors in a published application note. We also mention future work.

# 2 Introduction to SAL

The protocols are specified and verified in the Symbolic Analysis Laboratory (SAL), developed by SRI, International [11]. SAL is a verification environment that includes symbolic and bounded model checkers, an interactive simulator, integrated decision procedures, and other tools.

SAL has a high-level modeling language for specifying transition systems. A transition system is specified by a *module*. A module consists of a set of state variables and guarded transitions. Of the enabled transitions, one is nondeterministically executed at a time. Modules can be composed both synchronously (`||`) and asynchronously (`[]`), and composed modules communicate via shared variables. In a synchronous composition, a transition from each module is simultaneously applied; a synchronous composition is deadlocked if either module has no enabled transition. In an asynchronous composition, an enabled transition from one of the modules is nondeterministically chosen to be

applied.

The language is typed, and predicate sub-typing is possible. Types can be both interpreted and uninterpreted, and base types include the reals, naturals, and booleans; array types, inductive data-types, and tuple types can be defined. Both interpreted and uninterpreted constants and functions can be specified. This is significant to the power of these models: the parameterized values are uninterpreted constants from some parameterized type.

Bounded model checkers are usually used to find counterexamples, but they can also be used to prove invariants by induction over the state space [12]. SAL supports *k-induction*, a generalization of the induction principle, that can prove some invariants that may not be strictly inductive. By incorporating a *satisfiability modulo theories* decision procedure, SAL can do $k$-induction proofs over infinite-state transition systems.[2]

Let $(S, I, \rightarrow)$ be a transition system where $S$ is a set of states, $I \subseteq S$ is a set of initial states, and $\rightarrow$ is a binary transition relation. If $k$ is a natural number, then a *k-trajectory* is a sequence of states $s_0 \rightarrow s_1 \rightarrow \ldots \rightarrow s_k$ (a 0-trajectory is a single state). Let $k$ be a natural number, and let $P$ be property. The $k$-induction principle is then defined as follows:

- *Base Case*: Show that for each $k$-trajectory $s_0 \rightarrow s_1 \rightarrow \ldots \rightarrow s_k$ such that $s_0 \in I$, $P(s_j)$ holds, for $0 \leq j < k$.
- *Induction Step*: Show that for all $k$-trajectories $s_0 \rightarrow s_1 \rightarrow \ldots \rightarrow s_k$, if $P(s_j)$ holds for $0 \leq j < k$, then $P(s_k)$ holds.

The principle is equivalent to the usual transition-system induction principle when $k = 1$. In SAL, the user specifies the depth at which to attempt an induction proof, but the attempt itself is automated. The main mode of user-guidance in the proof process is in iteratively building up inductive invariants. While arbitrary LTL safety formulas can be verified in SAL using $k$-induction, only state predicates may be used as lemmas in a $k$-induction proof. Lemmas strengthen the invariant. We have more to say about the proof methodology for $k$-induction in Section 5.

# 3 Modeling and Verification of the Synchronizer Circuit

In this section we use a simple synchronizer circuit to illustrate the various modeling techniques used in this paper through the creation of successively more accurate models of the synchronizer circuit utilizing the transition language of SAL. In order to make the problem slightly more interesting, we generalize the data transfered by the circuit (`din`, `dout`) to arbitrary integers. Our initial model for the system of Figure 1 consists of two asynchronous processes – a transmitter (`tx`) and a receiver (`rx`).

```
system : MODULE = rx [] tx;
```

[2]We use SRI's ICS decision procedure [13], the default SAT-solver and decision procedure in SAL, but others can be plugged in.

```
FF : MODULE = BEGIN
                INPUT  d : BOOLEAN
                OUTPUT q : BOOLEAN
              INITIALIZATION
                q = FALSE
              TRANSITION
                q' = d
              END;
```

Figure 4: Flip Flop

Thus, the transmitter and receiver execute in an interleaved fashion and at arbitrary rates; however, each is made up from several processes that are composed synchronously (i.e., operate in lock step). For example, the transmitter is composed of an "environment", which follows the basic protocol described above, and two instantiated flip-flops modules (described below) with their inputs and outputs suitably renamed.

```
tx : MODULE =  (   (RENAME d TO aout, q TO a1 IN FF)
               || (RENAME d TO a1, q TO ain IN FF)
               || tenv);
```

Our initial flip-flop model in Figure 4 has no provision for capturing timing constraints. Indeed, its behavior is simply an assignment that copies input `d` to output `q` without any reference to an underlying clock. Our models depend upon synchronous composition to force the flip-flops comprising the transmitter (and receiver) to execute in lock step.

As mentioned, the transmitter's environment, shown in Figure 5, is constrained to obey the underlying protocol. There are two subtle points in this definition – we allow the data transmitted to take any randomly selected integer value, and we allow the transmitter to "stutter" indefinitely when it is allowed to transmit a new value (stuttering is expressed by `guard -->` where `guard` is a boolean expression). The syntax `var IN range` defines a non-deterministic choice chosen from the set `range`. The infinite state model checker of SAL that enables our verification of timing constraints also enables verification with unbounded variables.

The receiver is similarly composed of an environment, flip-flops, and a data latch (the flipflop module in which the input and output variables are generalized to arbitrary integers).

```
rx : MODULE =
          ((RENAME d TO rout, q TO r1 IN FF)
        || (RENAME d TO r1, q TO rin IN FF)
        || (RENAME d TO dout, q TO din IN LATCH)
        || renv);
```

The receiver environment module non-deterministically stutters or echos `rin`.

```
tenv : MODULE = BEGIN
          INPUT ain : BOOLEAN
          OUTPUT rout : BOOLEAN
          OUTPUT dout : INTEGER
       INITIALIZATION
          dout IN { x : INTEGER | TRUE };
          rout = FALSE
       TRANSITION
       [  TRUE        -->
       [] rout = ain  --> rout' = NOT rout;
                          dout' IN {x : INTEGER | TRUE };
       ] END;
```

Figure 5: Transmitter's Environment

```
renv : MODULE =
  BEGIN
    INPUT rin : BOOLEAN
    OUTPUT aout : BOOLEAN
  INITIALIZATION
    aout = FALSE
  TRANSITION
    aout' IN {aout, rin}
  END;
```

The defined circuit can be verified by induction over the (infinite) state space using the bounded model checking capabilities of SAL. In its current form, this circuit requires only straight induction ($k = 1$) for verification. Because the circuit implements a token passing protocol, a token counting lemma like the one in Figure 6 is key to its verification. Here, a "token" exists where the input and output to a flip-flop differ or where the receiver or transmitter environments are enabled to receive or send a value respectively; the syntax is the LTL temporal logic where the `G` operator denotes that its argument holds in all states in a trajectory through the transition system. This lemma is used to prove the key theorem using simple induction:

```
Sync_Thm : THEOREM system |- G((rout /= ain) => (dout = din));
```

Not surprisingly, both l1 and Sync_Thm can be verified quickly by SAL; however, the model as given does not capture any of the flip-flop timing requirements nor does it model any of the negative effects due to violating these requirements. In the following, we present a model that captures some of these requirements and allows us to verify the circuit even in the face of failures to meet these requirements.

We begin by modeling clocks. The transmitter and receiver are each composed with a local clock that regulates when that component may execute. The system we are developing has the following form:

```
changing(i : BOOLEAN, o : BOOLEAN) : [0..1] =
      IF (i /= o) THEN 1 ELSE 0 ENDIF;

l1 : LEMMA system |- G(changing(rin, r1) +
                         changing(r1, rout) +
                         changing(rout,ain) +
                         changing(ain, a1) +
                         changing(a1, aout) +
                         changing(aout,NOT rin)
                         <= 1);
```

Figure 6: Counting Lemma

```
RPERIOD : { x : TIME | 0 < x};

rclock : MODULE = BEGIN
    INPUT  tclk : TIME
    OUTPUT rclk : TIME
    INITIALIZATION
        rclk IN { x : TIME | time(rclk,tclk) <= x }
    TRANSITION
        time(rclk,tclk) = rclk -->
           rclk ' IN { x : TIME | time(tclk,rclk) + RPERIOD <= x }
    END;
```

Figure 7: Receiver Clock

```
(rx || rclock) [] (tx || tclock)
```

The basic idea, described as *timeout automata* by Dutertre and Sorea, is that the progress of time is enforced cooperatively (but nondeterministically) [14, 15]. The receiver and transmitter have *timeouts*, `rclk` and `tclk`, that mark the real-time at which they will respectively make transitions (timeouts are always in the future and may be updated nondeterministically). Each respective module representing the receiver and transmitter is allowed to execute only if its timeout equals the value of `time(rclk, tclk)`, which is defined to be the minimum of all timeouts.

```
time(t1 : TIME, t2: TIME): TIME =
     IF t1 <= t2 THEN t1 ELSE t2 ENDIF;
```

The receiver clock is defined in Figure 7. The transmitter clock is identical except for signal and constant names. As might be expected, the proof for the untimed model continues to work without change for this timed model since the addition of the timeout modules can only restrict the possible behaviors of the system and hence does not effect the safety property we are interested in verifying.

```
FFnd : MODULE =
   BEGIN
     INPUT  d : BOOLEAN
     OUTPUT q : BOOLEAN
   INITIALIZATION
      q = FALSE
   TRANSITION
      q' IN {TRUE, FALSE}
   END;
```

Figure 8: Nondeterministic Flip Flop

```
tx2 : MODULE =  (  (RENAME d TO aout, q TO a1 IN FFnd)
                || (RENAME d TO a1,  q TO ain IN FF)
                || tclock || tenv);

rx2 : MODULE =  (  (RENAME d TO rout, q TO r1 IN FFnd)
                || (RENAME d TO r1, q TO rin IN FF)
                || (RENAME d TO dout, q TO din IN LATCHnd)
                || rclock || renv );
```

Figure 9: Transmitter and Receiver Modules

Our final refinement is to add a mechanism for defining timing constraints and for introducing behaviors that model the effect of violating these constraints. The approach we take is inspired by by a recent paper by Seshia et. al. describing the use of "Generalized Relative Timing "[16]. Briefly, we modify the described circuit elements to allow the aberrant behaviors that may arise due to violation of timing constraints and add "constraint" processes to regulate the conditions under which these aberrant behaviors may occur.

As mentioned previously, the behavior we wish to capture is due to metastability occurring when the inputs to a flip-flop do not satisfy timing requirements. The circuit design implicitly assumes that the period of the receiver and transmitter clocks are sufficiently long that metastability occurring at the beginning of a clock period will have been resolved prior to the next clock period. Thus, in the circuit described, the only signals which may exhibit metastability are `din`, `r1`, and `a1`. It is easy to demonstrate that the circuit will fail if this assumption is not met. Furthermore, the value of a signal after resolution of a metastable state is non-deterministic. We model this by replacing the key circuit elements with non-deterministic versions of the existing elements. For example, we define a non-deterministic flip-flop module in Figure 8. Similarly, we can define a non-deterministic latch which randomly selects its next output. The transmitter and receiver respectively are defined by appropriately renaming input and output variables, as shown in Figure 9.

```
Constraint [ stime : REAL ] : MODULE =
   BEGIN
     INPUT  dclk : TIME
     INPUT  qclk : TIME
     INPUT  d    : BOOLEAN
     INPUT  q    : BOOLEAN
     OUTPUT ts   : TIME
   INITIALIZATION
      ts = 0;
   TRANSITION
   [
      dclk /= dclk' AND (ts > time(dclk,qclk) OR q' = d) -->
   [] dclk = dclk' AND d /= d'  --> ts' = time(dclk,qclk) + stime
   [] dclk = dclk' AND d = d'   -->
   ]
END;
```

Figure 10: Constraint Module

Clearly, the circuit no longer satisfies its basic invariant. Our final step is to add processes that execute in parallel with the this system to constrain the outputs of the non-deterministic circuit elements. In particular, we assume that whenever `rout`, `aout`, or `dout` change state there is a settling period during which attempts to latch the new value will lead to metastability and hence a non-deterministic next state. As we shall show, the constraint processes that we add force the non-deterministic circuit elements to behave in a conventional manner outside these settling periods. The length of the settling period is implementation dependent and may be the result of a combination of factors such as signal propagation and circuit element setup time. In order to simplify the presentation, we have chosen to ignore hold time requirements. In practice, it is feasible to design flip-flops with zero-hold time requirements by inserting delays at the flip-flop input (at the cost of additional setup time). Furthermore, in an acyclic system such as 8N1 described in Section 4, one can simply shift the perspective of where the clock edge occurs to justify combining the setup and hold time requirements.

The system model, with the addition of the necessary constraints, has the form:

```
system : MODULE = (rx2 [] tx2) || constraints
```

Synchronous composition means that `rx2` and `tx2` can only execute when the necessary constraints are satisfied. Consider a flip-flop with input `d` and output `q`. We need a constraint module that monitors the `d` input for changes and constrains the `q` output to meet the requirements for "normal" behavior outside the settling period that follows a change, as shown in Figure 10 (note the module is a *parameterized module*; its parameter, `stime`, acts as a constant in the module).
Consider the following constraint module, with appropriately renamed input and output variables.

```
l0 : LEMMA system |- G((r1ts <= time(rclk,tclk) OR
                          (r1ts + TPERIOD - TSETTLE <= tclk)) AND
                        (d1ts <= time(rclk,tclk) OR
                          (d1ts + TPERIOD - TSETTLE <= tclk)) AND
                        (a1ts <= time(rclk,tclk) OR
                        (a1ts + RPERIOD - RSETTLE <= rclk)) AND
                        (a1ts <= time(rclk,tclk) + RSETTLE) AND
                        (d1ts <= time(rclk,tclk) + TSETTLE) AND
                        (r1ts <= time(rclk,tclk) + TSETTLE ) AND
                        (time(rclk,tclk) <= rclk) AND
                        (time(rclk,tclk) <= tclk));
```

Figure 11: Lemma `l0`

```
(RENAME d TO rout, q to r1, dclk TO rclk, qclk TO tclk, ts TO r1ts IN
  Constraint[TSETTLE])
```

Whenever `rout` changes value and `rclk` preserves its value (i.e., `tx2` executes), the local timer `r1ts` is set to a value equal to the current time plus the settling constant `TSETTLE`. Whenever `rclk` changes value (i.e., `rx2` takes a step) either `r1` is assigned `rout` or the local timer must be active. Finally, if neither condition occurs, the constraint module allows `tx2` to execute. To constrain the three possible sources of non-deterministic behavior, there are three constraint modules with the local timers `r1ts`, `a1ts`, and `d1ts` monitoring changes on `rout`, `aout`, and `dout`, respectively.

The three constraint modules utilize two settling constants `TSETTLE` (for `rout` and `dout`) and `RSETTLE` (for `aout`). In verifying the circuit, we found that correct behavior depends on establishing a relationship between settling times and clock periods. In particular, the settling time of the transmitter must be less than the clock period of the receiver (and vice versa). Violating these assumptions has the effect of "injecting" additional tokens into the circuit whenever metastability occurs. Thus, we performed verification under the following assumptions.

```
TSETTLE  : { x : TIME | 0 <= x AND x < RPERIOD AND x < TPERIOD };
RSETTLE  : { x : TIME | 0 <= x AND x < RPERIOD AND x < TPERIOD };
```

With the changes described above, verification of the circuit behavior is more challenging, requiring $k$-induction over a modified token counting lemma and an additional helper lemma. To make the $k$-induction proof technique feasible, it is helpful to constrain the state space whenever possible. Hence, we developed the lemma shown in Figure 11 to assert certain obvious facts about system timing.

It was necessary to augment the counting lemma with additional addends to account for the possible spontaneous creation of tokens due to metastability, as shown in Figure 12.

Lemmas l0 and l1 can be proved at depth 1 (straight induction) with l1 using l0 as an assumption. The main theorem, Sync_Thm, can be verified at depth 3 using l0 and l1 as assumptions.

```
l1 : LEMMA system |- G(changing(rout, r1) +
                       changing(r1, rin) +
                       changing(rin,aout) +
                       changing(aout, a1) +
                       changing(a1,ain) +
                       changing(ain,NOT rout) +
                       if (rout=r1 AND rclk < r1ts ) THEN
                           1 ELSE 0 ENDIF +
                       if (aout=a1 AND tclk < a1ts) THEN
                           1 ELSE 0 ENDIF
                      <= 1);
```

Figure 12: Lemma `l1`

# 4 Modeling the 8N1 Protocol

In this section we discuss the model of the 8N1 protocol – its proof is deferred to Section 5. We model the protocol using two processes asynchronously composed – a transmitter (`tx`) and a receiver (`rx`). The general arrangement of the two major modules is illustrated in Figure 13. [3]

```
system : MODULE = rx [] tx;
```

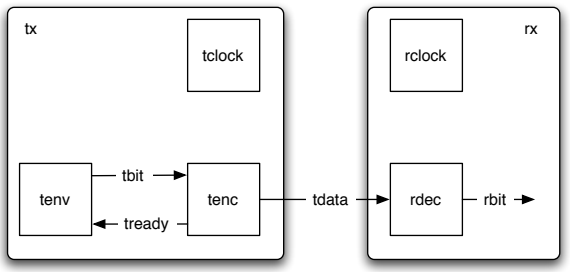



Figure 13: System Block Diagram

As with the synchronizer circuit of Section 3, the transmitter and receiver each have a local clock module to manage their timeout. Recall that time is advanced whenever the module with the minimum timeout value executes and that the current time is always equal to the minimum timeout.

In addition to its local clock (`tclock`), the transmitter consists of an encoder (`tenc`) that implements the basic protocol, and an environment (`tenv`) that generates the data to be transmitted. These modules are synchronously composed.

```
tx : MODULE = tclock || tenc || tenv;
```

[3] Not shown are the shared variables used by the clock modules to compute the global "time".

Similarly, the receiver consists of its local clock (`rclock`) and and a decoder (`rdec`) that implements the protocol.

```
rx : MODULE = rdec || rclock;
```

The system is defined by the asynchronous composition of the transmitter and receiver which are then composed synchronously with a "constraint" module that models uncertainty in signal propagation as well as timing constraints. For the moment, we postpone discussion of the constraint module.

```
system : MODULE = (rx [] tx) || constraint;
```

The clock and environment modules for the transmitter are illustrated in Figure 14. The environment determines when new input data should be generated and is regulated by `tenc`. Whenever `tready` is true, a random boolean datum is selected; otherwise the old datum is preserved.

The timing model for the transmitter is similar to that for the synchronizer circuit. We assume an arbitrary clock period consisting of a settling phase (`TSETTLE`) and a stable phase (`TSTABLE`). The settling phase captures both setup requirements for the receiver as well as propagation delay. We will assume that reading the output of the transmitter `tdata` during the settling phase yields a non-deterministic result. As with the synchronizer, we assume that the receiver is implemented in such a manner that any metastability is resolved within the minimum clock period of the receiver. `TSETTLE` and `TSTABLE` are uninterpreted constants; however they are parameterized, which allows us to verify the model for any combination of settling time and receiver clock error (described subsequently). The transmitter settling time can be used to capture the effects of jitter and dispersion in data transmission as well as jitter in the transmitter's clock. In the case of the settling period, the model can be viewed as less deterministic than an actual implementation which might reach stable transmission values sooner. This means we verify the model under more pessimistic conditions than an actual implementation would face. As with the synchronization circuit, we do not actually model non-boolean values, rather we model a receiver that detects random values for signals that are not stable (as determined by the separate "constraint" module).

The transmitter encoder is defined as a simple state machine – state 0 corresponds to the start bit, states 1-8 correspond to the 8 data bit transmission states, and state 9 is the stop state. The encoder model is illustrated in Figure 15. Notice that the model allows the transmitter to stutter at state 9 indefinitely. The output `tdata` is either current value of `tbit` (states 1-8), FALSE (state 0), or TRUE (state 9).

The receiver clock is more complicated than the transmitter because of the manner in which a UART is implemented. Consider Figure 3. There may be an arbitrary "idle" period between frames during which the signal is high (TRUE). The behavior of a UART receiver is to "scan" for the high-to-low transition that marks the beginning of a frame. Once this transition is detected, the receiver predicts, based upon its local time reference, the middle of the 8 data and 1 stop bit times. There are two different intervals used for this prediction – the time between the detected "start" transition and the middle of the first data bit and the "period" between successive data samples. In an implementation, the bit period is generally an integer multiple of the scan time and the start interval is 1.5

```
TPERIOD : { x : TIME | 0 < x};
TSETTLE : { x : TIME | 0 <= x AND x < TPERIOD};

% function to compute current time

time(t1 : TIME, t2 : TIME) : TIME = IF t1 <= t2 THEN t1 ELSE t2 ENDIF;

tclock : MODULE =
 BEGIN
    INPUT  rclk   : TIME
    OUTPUT tclk   : TIME

 INITIALIZATION

    tclk  IN {x : TIME | 0 <= x AND x <= TSTABLE};

 TRANSITION
  [ tclk = time(tclk, rclk) -->  tclk' = tclk + TPERIOD;]
 END;

tenv : MODULE =
  BEGIN
    INPUT   tready : BOOLEAN
    OUTPUT  tbit   : BOOLEAN

  TRANSITION
   [
      tready --> tbit' IN {TRUE, FALSE}
   [] ELSE  -->
   ]
  END;
```

Figure 14: Transmitter Environment and Clock

times the bit period. Generally the bit time of the receiver is approximately that of the transmitter; however, in practice jitter and frequency errors mean that each measurement interval is subject to error. In our model we associate all errors with the receiver and assume that the transmitter runs at a constant rate.

The various receiver clock periods are expressed in terms of linear equations that define lower and upper bounds for "SCAN", "START", and "PERIOD". The details of these equations can be viewed as part of the proof – we verify the protocol subject to these bounds – and are postponed to Section 5. The receiver clock along with the various is illustrated in Figure 16. The specific timeout interval depends upon the state of the decoder; i.e., whether the decoder is scanning, sampling the first data bit, or sampling subsequent data bits.

```
tenc : MODULE =
  BEGIN
    OUTPUT  tdata  : BOOLEAN
    OUTPUT  tstate : [0..9]
    OUTPUT  tready : BOOLEAN
    INPUT   tbit   : BOOLEAN
  INITIALIZATION
     tdata  = TRUE;
     tstate = 9;
  DEFINITION
     tready =  tstate < 8
  TRANSITION
   [
       tstate = 9 -->
    [] tstate = 9 --> tdata' = FALSE;
                      tstate' = 0;
    [] tstate < 9 --> tdata' =  (tbit' OR tstate = 8);
                      tstate' = tstate + 1;
   ]
  END;
```

Figure 15: Transmitter Encoder

The decoder is illustrated in Figure 17. There are three transitions – the first two model the non-deterministic choice that occurs when scanning for the start bit and a third models sampling the data bits. The receiver has 10 states (numbered [0..9]) where the 8 data bits are received in states 0-7, the stop bit is received in state 8, and scanning for a new start bit occurs in state 9.

As with the synchronizer, the value of the bit read is always chosen non-deterministically, though the next state may depend upon the specific choice. Furthermore, the choice is constrained by a separate module that determines when the sampled value should reflect the input (`tdata`) and when the sampled value may be random. The constraint module is also presented in Figure 17. The only significant difference between this and the constraint modules used with the synchronizer is the extra output `stable` which is used in developing the proof.

```
timeout (min : TIME, max : TIME) : [TIME -> BOOLEAN]  =
               { x : TIME | min <= x AND x <=  max};

rclock : MODULE  =
  BEGIN
    INPUT tclk   : TIME
    INPUT rstate : [0..9]
    OUTPUT rclk  : TIME
  INITIALIZATION
    rclk IN { x : TIME | 0 <= x AND x < RSCANMAX };
  TRANSITION
   [
    rclk = time(rclk, tclk) -->
         rclk' IN IF (rstate' = 9) THEN
                    timeout(rclk + RSCANMIN, rclk + RSCANMAX)
                  ELSIF (rstate' = 0) THEN
                     timeout(rclk + RSTARTMIN, rclk + RSTARTMAX)
                  ELSE
                     timeout(rclk + RPERIODMIN, rclk + RPERIODMAX)
                  ENDIF;
   ]
   END;
```

Figure 16: Receiver Clock

# 5 Verification of the 8N1 Protocol

Our main goal is to prove that the 8N1 decoder reliably extracts the data from the signal it receives.

```
Uart_Thm :  THEOREM system |- G(rstate < 9 AND
                                rstate > 0 AND
                                rclk >= tclk => ((tstate = rstate)  AND
                                                (rbit = tbit)));
```

Briefly, the theorem states that immediately after the receiver executes each of its 8 bit receive states (0..7), the received bit is equal to the currently transmitted bit. This interpretation of the theorem depends upon the knowledge that the states of the transmitter and receiver obey the following sequence. This sequence is verified with theorem t0 to be discussed subsequently.

$$(tstate, rstate) = (9,9), (0,9), (0,0), (1,0), (1,1), ...(9,8), (9,9) \tag{2}$$

As mentioned previously, an important component of the proof is the set of bounds on the various time constants utilized in the decoder model. We derived the bounds by assuming worst case (minimum or maximum) and then determining how temporal

```
rdec : MODULE =
  BEGIN
    INPUT  tdata  : BOOLEAN
    OUTPUT rstate : [0..9]
    OUTPUT rbit   : BOOLEAN
  INITIALIZATION
    rbit  = TRUE;
    rstate = 9;
  TRANSITION
   [
      rstate = 9  -->  rbit' = TRUE
   [] rstate = 9  -->  rbit' = FALSE;
                        rstate' = 0
   [] rstate /= 9 -->  rbit' IN {FALSE, TRUE};
                        rstate' = rstate + 1
   ]
  END;

constraint : MODULE =
  BEGIN
    INPUT  tclk    : TIME
    INPUT  rclk    : TIME
    INPUT  rbit    : BOOLEAN
    INPUT  tdata   : BOOLEAN
    OUTPUT stable  : BOOLEAN
    LOCAL changing : BOOLEAN
  DEFINITION
    stable = (NOT changing OR (tclk - rclk < TSTABLE));
  INITIALIZATION
    changing = FALSE
  TRANSITION
   [
      rclk' /= rclk AND (stable => rbit' = tdata) -->
   [] tclk' /= tclk --> changing' = (tdata' /= tdata)
   ]
  END;
```

Figure 17: Receiver Decoder and Constraint Module

errors accumulate by the 10th bit time (the stop bit). Informally, the correct behavior of the protocol requires that all samples other than the initial scan fall during the "stable" portion of the transmitter clock. We derived these bounds by considering the execution sequence described and with the knowledge that the correct behavior of the receiver requires that in receiver states 0..8, we require the clock events fall during the "stable" period of the transmitter. Consider the case of the "scan" operation. In order to detect the start bit, we must guarantee that the receiver sample `tdata` with a period that is no longer that the stable period – if the interval were longer, then the start bit might might

be missed because two successive samples by the receiver fall outside the stable interval.

```
RSCANMIN : { x : TIME | 0 < x };
RSCANMAX : { x : TIME | RSCANMIN <= x AND x < TSTABLE };
```

Once the start bit is detected, the receiver waits for a "start" time before reading the first data bit. Reading this data bit must fall in the stable region for transmitter state 1.

```
RSTARTMIN : { x : TIME | TPERIOD + TSETTLE < x };
RSTARTMAX : { x : TIME | RSTARTMIN <= x AND
                 TSETTLE + RSCANMAX + x < 2 * TPERIOD };
```

In subsequent states the receiver clock error accumulates. Thus, the constraint on the receiver "period" depends upon the accumulated error at the point of sampling the stop bit.

```
RPERIODMIN : { x : TIME | 9 * TPERIOD + TSETTLE < RSTARTMIN + 8 * x };
RPERIODMAX : { x : TIME | RPERIODMIN <= x AND
                 TSETTLE + RSCANMAX + RSTARTMAX + 8 * x < 10 * TPERIOD };
```

The proofs of t0 and Uart_Thm require supporting lemmas. In general, when a $k$-induction proof attempt fails, two options are available to the user: the proof can be attempted at a greater depth, or supporting lemmas can be added to restrict the state-space. A $k$-induction proof attempt is automated, but if the attempt is not successful for a sufficiently small $k$ (i.e., the attempt takes too long or too much memory), additional invariants are necessary to reduce the necessary proof depth. The user must formulate the supporting invariants manually, but their construction is facilitated by the counterexamples returned by SAL for failed proof attempts. If the property is indeed invariant, the counterexample is a trajectory that fails the induction step but lies outside the set of reachable states, and the state-space can be appropriately constrained by an auxiliary lemma based on the counterexample. The following lemmas are built by examining the counterexamples returned from proof attempts for the main theorem and the successive intermediary lemmas.

Once it is determined what property the states fail to have that makes them unreachable, this property can be stated (and proved) as an additional predicate. This predicate is used as a lemma to support the proof original of the original property. The following lemmas capture some simple facts about the relationships between the two clocks. Of these, l1, is the least obvious and was derived along with theorem t0 in order to reduce the required induction depth. Each of these lemmas is inductive and hence can be proved at depth 1.

```
l1 : LEMMA system |- G(tclk <= (rclk + TPERIOD) OR stable);

l2 : LEMMA system |- G(rclk <= tclk + RSTARTMAX OR
                       rclk <= tclk + RSCANMAX OR
                       rclk <= tclk + RPERIODMAX);
```

```
t0 : THEOREM system |- G(
    % idle
     ((rstate =  9) AND
      (tstate = 9) AND
      (tdata AND rbit) AND
      stable AND
      (rclk - tclk <= RSCANMAX))
 OR % start bit sent, not detected
     ((rstate = 9) AND
      (tstate = 0) AND
      (NOT tdata AND rbit) AND
      (rclk - tclk <= RSCANMAX - TSTABLE))
 OR % --- unwind all the other cases
    rec_states(8, tstate, rstate, tdata, rbit, rclk, tclk, stable));
```

The key part of our proof of 8N1 is an invariant that describes the relationship between the transmitter and receiver. We must relate them both temporally and with respect to their discrete state (e.g., `tstate` with `rstate` and `tdata` with `rbit`). The number of and the complexity of the supporting lemmas necessary to prove the main results is significantly reduced by proving a *disjunctive invariant* [17]. A disjunctive invariant has the form $\bigvee_{i \in I} P_i$ where each $P_i$ is a state predicate (predicates $P_i$ and $P_j$ need not be disjoint for $i \neq j$). Disjunctive invariants are easier to generate iteratively than conjunctive invariants. If a disjunctive invariant fails to cover the reachable states, additional disjuncts can be incrementally added to it (in a conjunctive invariant, additional conjunctions must hold in all the reachable states). Although this is a general proof technique, it is particularly easy to build a disjunctive invariant in SAL. The counterexamples SAL returns can be used to iteratively weaken the disjunction until it is invariant.

Theorem t0 has 20 disjuncts corresponding to the 20 unique states in equation 2. Of the disjuncts, 18 follow a simple pattern and are defined in SAL with a recursive function. The following defines theorem t0.

The recursively defined disjuncts use the following pattern for $n = 0..8$.

In general, each disjunct defines the control state (`tstate` and `rstate`), the constraints on the data signals if any, and describes the relative difference between `tclk` and `rclk`. A bug in ICS which involved multiplication of uninterpreted constants required a workaround in which we defined multiplication recursively. This theorem can be proved at depth 3, while the main theorem (Uart_Thm) can then be proved at depth 2 with t0 as a lemma.

# 6 Discussion

Our proof of the 8N1 protocol is verified with respect to bounds on the various timing constants. In a practical implementation, the receiver scan period is defined relative to the nominal transmitter bit period and the receiver start and bit periods are integer multiples of this. What an implementor ultimately cares about is the the trade off between settling time (in general due to signal dispersion over a given transmission medium) and frequency

```
 ((tstate = n + 1) AND
  (rstate = n) AND
  (rclk - tclk <=
     mult(n, RPERIODMAX) - mult(n+1, TPERIOD) + RMAX - TSTABLE) AND
  (rclk - tclk >=
     mult(n, RPERIODMIN) - mult(n+1, TPERIOD) + RSAMPMIN - TPERIOD))
OR
 ((tstate = n) AND
   (rstate = n) AND
    stable AND
   (tdata = rbit) AND
   (rclk - tclk <=
      mult(n, RPERIODMAX) - mult(n, TPERIOD) + RMAX - TSTABLE) AND
   (rclk - tclk >=
      mult(n, RPERIODMIN) - mult(n+1, TPERIOD) + RSAMPMIN));
```

```
 RSTARTMAX : TIME = TSTART * (1 + ERROR);
 RSTARTMIN : TIME = TSTART * (1 - ERROR);
 RSCANMAX : TIME = 1 + ERROR;
 RSCANMIN : TIME = 1 - ERROR;
 RPERIODMAX : TIME = TPERIOD * (1 + ERROR);
 RPERIODMIN : TIME = TPERIOD * (1 - ERROR);
```

Figure 18: Receiver Parameters Defined with respect to Error

error.

In the following, we show how the bounds that we have verified can be used to derive error and settling time bounds in a form that is more convenient for a protocol implementer. These derived bounds are somewhat more restrictive than what we have verified since we require the maximum allowable frequency error to be symmetric about the nominal frequency. As before, let `TPERIOD` be the nominal period duration. We introduce another uninterpreted constant in the operational model representing the nominal duration the receiver waits for the start bit ("START").

```
TSTART : TIME;
```

Now, let `ERROR` be an uninterpreted constant from `TIME`, and then the constants in Figure 10 are defined in terms of `ERROR`. By replacing these defined terms in the parameterization of the types in Sec 5, we compute the bound on the error. For example, `RSTARTMAX` is an uninterpreted constant from the following parameterized type:

```
RSTARTMAX : { x : TIME | RSTARTMIN <= x AND
                         TSETTLE + RSCANMAX + x < 2 * TPERIOD };
```

Replacing `RSTARTMIN` and `RSCANMAX` by their definitions from Figure 18, we get

```
RSTARTMAX : { x : TIME | TSTART * (1 - ERROR) <= x AND
                         TSETTLE + 1 + ERROR + x < 2 * TPERIOD };
```

By replacing each term with its definition, the type parameters are defined completely in terms of TPERIOD, TSETTLE, and ERROR. Isolating ERROR in the system of inequalities gives bounds on ERROR. For the 8N1 protocol, ERROR is thus parameterized as follows:

```
ERROR : { x : TIME | 0 <= x AND
          (9 * TPERIOD + TSETTLE <
              8 * TPERIOD * (1-x) + TSTART * (1-x)) AND
          ((8 * TPERIOD * (1+x) + TSTART * (1+x) + (1+x) + TSETTLE) <
              10 * TPERIOD) };
```

This derived model can be verified using the same invariants proved at the same depth as in the verification described in Section 5.

As mentioned in Section 1, we discovered significant errors in the analysis in an application note for UARTs [3]. For TPERIOD = 16 and TSTART = 23, the authors suggest that if TSTABLE is TPERIOD/2 (they call this the "nasty" scenario), then a frequency error of $\pm 2\%$ is permissible. In fact, even with zero frequency mismatch, the stable period is too short – if we assume "infinitely" fast sampling, it is possible to show that the settling time must be less than 50% of TPERIOD. In other words, the type parameterizing ERROR is empty when TSTABLE is TPERIOD/2 (this can be shown using SAL or by a simple calculation). With our choice of time constants, the longest settling time must be less than 7 (43.75%). In reading the article, it becomes clear that the authors neglected the temporal error introduced by sampling the start bit. They describe a "normal" scenario with TSETTLE = TPERIOD/4 and assert that a frequency error of $\pm 3.3\%$ is permissible. As our derivation above illustrates, the frequency error in this case is limited to $\pm 3/151 \approx \pm 1.9\%$.

This paper describes the use of SAL to model and verify a data synchronization circuit and the 8N1 protocol. We show, by example, how models of these can be refined in the language of SAL to capture timing constraints and environmental effects such as metastability and settling. Future work includes extending this framework to other cross domain protocols as well as developing the theory for refinement.

# Acknowledgments

We thank Leonardo de Moura, John Rushby, and anonymous reviewers for a recent paper [1] for their suggestions and corrections.

# References

[1] Geoffrey M. Brown and Lee Pike. Easy parameterized verification of biphase mark and 8N1 protocols. In *The Proceedings of the 12th International Conference on Tools and the Construction of Algorithms (TACAS'06)*, 2006. To appear. Available at http://www.cs.indiana.edu/~lepike/pub_pages/bmp.html.

[2] F. W. Vaandrager and A. L. de Groot. Analysis of a Biphase Mark Protocol with Uppaal and PVS. Technical Report NIII-R0455, Nijmegen Institute for Computing and Information Science, 2004.

[3] Maxim Integrated Products, Inc. *Determining Clock Accuracy Requirements for UART Communications*, June 2003. Available at `http://www.maxim-ic.com/appnotes.cfm/appnote_number/2141`.

[4] Tsachy Kapschitz and Ran Ginosar. Formal verification of synchronizers. In *CHARME 2005 – to appear*, 2005.

[5] Tsacky Kapschitz, Ran Ginosar, and Richard Newton. Verifying synchronization in multi-clock domain SoC. In *DVCon 2004*, 2004.

[6] Tai Ly, Neil Hand, and Chris Ka-Kei Kwok. Formally verifiying clock domain crossing jitter using assertion-based verification. In *DVCon 2004*, 2004.

[7] Karen Yorav, Sagi Katz, and Ron Kiper. Reproducing synchronization bugs with model checking. In *CHARME*, pages 98–103, 2001.

[8] T. Henzinger, J. Preussig, and H. Wong-Toi. Some lessons from the Hytech experience. In *Proceedings of the 40th Annual Conference on Decision and Control*, pages 2887–2892, 2001.

[9] Aurore Annichini, Ahmed Bouajjani, and Mihaela Sighireanu. TReX: A tool for reachability analysis of complex systems. In *Computer-Aided Verification, CAV'01*, pages 368–372, London, UK, 2001. Springer-Verlag.

[10] F. W. Vaandrager and A. L. de Groot. Analysis of a biphase mark protocol with Uppaal and PVS. Technical Report NIII-R0445, Radboud University Nijmegen, 2004.

[11] Leonardo de Moura, Sam Owre, Harald Rueß, John Rushby, N. Shankar, Maria Sorea, and Ashish Tiwari. SAL 2. In *Computer-Aided Verification, CAV'04*, volume 3114 of *LNCS*, pages 496–500, Boston, MA, July 2004. Springer-Verlag.

[12] Leonardo de Moura, Harald Rueß, and Maria Sorea. Bounded model checking and induction: From refutation to verification. In *Computer-Aided Verification, CAV'03*, volume 2725 of *LNCS*, 2003.

[13] Leonardo de Moura, Sam Owre, Harald Ruess, John Rushby, and N. Shankar. The ICS decision procedures for embedded deduction. In *2nd International Joint Conference on Automated Reasoning (IJCAR)*, volume 3097 of *LNCS*, pages 218–222, Cork, Ireland, July 2004. Springer-Verlag.

[14] Bruno Dutertre and Maria Sorea. Timed systems in SAL. Technical Report SRI-SDL-04-03, SRI International, 2004.

[15] Bruno Dutertre and Maria Sorea. Modeling and verification of a fault-tolerant real-time startup protocol using calendar automata. In *FORMATS/FTRTFT*, pages 199–214, 2004.

[16] Sanjit A. Seshia, Randal E. Bryant, and Kenneth S. Stevens. Modeling and verifying circuits using generalized relative timing. In *ASYNC*, pages 98–108, 2005.

[17] John Rushby. Verification diagrams revisited: Disjunctive invariants for easy verification. In *Computer-Aided Verification, CAV'00*, volume 1855 of *LNCS*, pages 508–520, Chicago, IL, July 2000. Springer-Verlag.

# A Coverage Analysis for Safety Property Lists

Talk Abstract

Koen Lindström Claessen
Chalmers University of Technology
koen@chalmers.se

## 1 Background

In property-based verification, a natural question that often arises is 'Have we specified enough properties?' Simulation-based *coverage* notions do not help us to answer this question. Therefore, there exist notions of coverage in formal verification, where it is checked how much of the design under verification is actually needed in the formal proof [1]. A disadvantage of these methods is that they include the actual design in the coverage analysis, which makes the complexity of the analysis dependent on the size of the design, and which also implies that the coverage analysis has to be re-done every time the design changes.

We present a complement to existing notions of property coverage that is design-independent. The idea is simple: Given a list of safety properties, and a set of output signals from the design, our analysis checks if there exists a "forgotten case": a trace where there exists a point in time where a particular output signal is not constrained by the properties. In other words, given a trace where the values of all other signals and the values of all other points in time are known, and given that the list of properties holds, it is still not known what the value of that particular output at that particular point in time should be.

The analysis works for all typical specification logics in which safety properties can be expressed. Here, we use a simple variant of LTL.

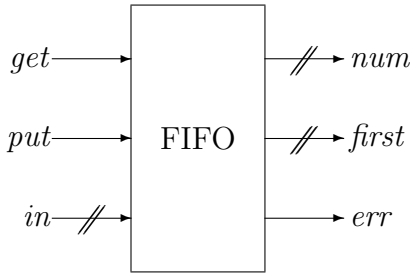



Figure 1: A simple FIFO interface

# 2 An Example

Consider the following specification of a simple FIFO. As depicted in figure 1, the input signals are *get*, *put* and a vector *in*, and the output signals are *err* and vectors *fst* and *num*. For simplicity, we specify that putting takes priority over getting. When we try to put something in a full FIFO, or get something from an empty FIFO, the signal *err* becomes **1** for one clock cycle. The output *fst* always indicates the first element of the FIFO, and the output *num* indicates the number of elements currently in the FIFO (maximum $n$). An initial attempt to create a list of safety properties formalizing the above description might look as follows.

$$
\begin{array}{ll}
\Box(put = \mathbf{1} \wedge num = n & \Rightarrow \ \mathsf{next}\ err = \mathbf{1}) \\
\Box(put = \mathbf{1} \wedge num < n & \Rightarrow \ \mathsf{next}\ num = num + 1 \\
 & \wedge\ \mathsf{next}\ err = \mathbf{0}) \\
\Box(put = \mathbf{1} \wedge num = 0 & \Rightarrow \ \mathsf{next}\ fst = in) \\
\Box(put = \mathbf{1} \wedge 0 < num < n & \Rightarrow \ \mathsf{next}\ fst = fst) \\
\Box(get = \mathbf{1} \wedge put = \mathbf{0} \wedge num = 0 & \Rightarrow \ \mathsf{next}\ err = \mathbf{1}) \\
\Box(get = \mathbf{1} \wedge put = \mathbf{0} \wedge 0 < num & \Rightarrow \ \mathsf{next}\ num = num - 1 \\
 & \wedge\ \mathsf{next}\ err = \mathbf{0})
\end{array}
$$

We can now analyze this list of properties using the proposed analysis, in order to discover forgotten cases in our specification. Note that we are only analyzing the list of properties, not the design. At this stage, having the design ready for formal verification is not neccessary.
When we ask the analysis about the property coverage of output *err*, it immediately replies that *err* is not constrained by the properties at time point 1, as the following trace shows:

| | |
|---|---|
| *get* | **0** |
| *put* | **0** |
| *in* | 0 |
| *num* | 0 |
| *fst* | 0 |
| *err* | ? |

No matter what the value of the ? in the trace, the property list is still fulfilled. Indeed we should have added a property $err = \mathbf{0}$ at time 0. After this, the analysis complains about *err* being unconstrained when we do not put or get something from the FIFO:

| | | |
|---|---|---|
| *get* | **0** | **0** |
| *put* | **0** | **0** |
| *in* | 0 | **0** |
| *num* | 0 | 0 |
| *fst* | 0 | 0 |
| *err* | **0** | ? |

And indeed, we should have added a property that says that errors do not occur when we do not change the contents of the FIFO:

$$\square(get = \mathbf{0} \wedge put = \mathbf{0} \Rightarrow \mathsf{next}\ err = \mathbf{0}).$$

Now, the analysis is happy about *err*.
Next, we analyze the output *num*. We find out that *num* is not constrained in the first point in time either:

| | |
|---|---|
| *get* | **0** |
| *put* | **0** |
| *in* | 0 |
| *num* | ? |
| *fst* | 0 |
| *err* | **0** |

This counter example leads to us adding the property $num = 0$. Next, the analysis complains about *num* being unconstrained when we do not put or get:

| | | |
|---|---|---|
| *get* | **0** | **0** |
| *put* | **0** | **0** |
| *in* | 0 | **0** |
| *num* | 0 | ? |
| *fst* | 0 | 0 |
| *err* | **0** | **0** |

This is easily fixed by adding the property:

$$\Box((get = \mathbf{0} \wedge put = \mathbf{0}) \Rightarrow \mathsf{next}\ num = num).$$

However, the analysis is still not happy. It reports:

| | | |
|---|---|---|
| *get* | **1** | **0** |
| *put* | **1** | **0** |
| *in* | 0 | **0** |
| *num* | 0 | ? |
| *fst* | 0 | 0 |
| *err* | **0** | **1** |

In other words, when an error occurs, it is not specified what should happen with *num*. We fix this by adapting the last property we added thus:

$$\Box((get = \mathbf{0} \wedge put = \mathbf{0}) \vee err = \mathbf{1} \Rightarrow \mathsf{next}\ num = num).$$

Finally, the analysis is happy about the output *num*.

# 3 Free signals

Dealing with the signal *fst*, there appear to be two problems. Firstly, it is not always the case that we want to specify what the value of a signal is in all cases. For example, when the FIFO is empty, we would like to leave *fst* unspecified, since there is no first value in the FIFO. At the moment, the analysis would simply complain about this case, making it rather useless in this case.
Secondly, sometimes it is hard or impossible to completely formally specify the exact behavior of a particular signal in a temporal logic, and as a specifier one wants to be able to take the pragmatic decision of not specifying the behaviour completely. Again, the analysis would immediately find holes in

the specification, holes which have deliberatly been put there. In the case of the signal *fst*, formally specifying the exact FIFO behavior for general $n$ is impossible in a limited logic like LTL.
One solution to this problem is simply not to use the analysis on the output *free*. This has an obvious drawback, namely that we will not be able to find *real* forgotten cases, as opposed to the intended forgotten cases that we already know about. So, we would like to argue for another solution, namely one where the specifier explicitly indicates what in what cases an output is allowed to be underconstrained. We therefore introduce a new construct $\mathsf{free}x$ to the specification logic, that can be used to express that the output $x$ is allowed to be unconstrained. As a logical construct, $\mathsf{free}x$ is simply true, but to the analysis, it is a way to suppress complaints about the signal $x$.
For example, our analysis complains about the output *fst* being unconstrained in the beginning, when the FIFO is empty. This can be remedied by adding the following property:

$$\Box(num = 0 \Rightarrow \mathsf{free}\ fst).$$

The above property explicitly expresses the unconstrainedness of *fst* in the case when $num = 0$.
Our analysis also complains about the output *fst* being unconstrained when we put two elements in the FIFO, and get an element out once[1]:

| | | | | |
|---|---|---|---|---|
| *get* | **0** | **0** | **1** | **0** |
| *put* | **1** | **1** | **0** | **0** |
| *in* | 17 | 5 | 0 | 0 |
| *num* | 0 | 1 | 2 | 1 |
| *fst* | 0 | 17 | 17 | ? |
| *err* | **0** | **0** | **0** | **0** |

And indeed, we have not said anything about this particular case. If the specifier decides not to specify the exact FIFO behavior, this can be fixed by adding $\mathsf{next}\ \mathsf{free}\ fst$ to the right-hand side of the last property:

$$\begin{aligned}\Box(get = \mathbf{1} \wedge put = \mathbf{0} \wedge 0 < num \quad &\Rightarrow \mathsf{next}\ num = num - 1\\ &\wedge \mathsf{next}\ err = \mathbf{0}\\ &\wedge \mathsf{next}\ \mathsf{free}\ fst)\end{aligned}$$

[1]The actual counter example that was generated was slightly edited for presentational reasons.

In other words, when we remove an element from the FIFO, we are not quite sure what the new first element is going to be.
Even now, our analysis still reports a forgotten case, namely when we do not put or get at all, *fst* is unconstrained:

| | | |
|---|---|---|
| *get* | **0** | **0** |
| *put* | **0** | **0** |
| *in* | 0 | **0** |
| *num* | 0 | 0 |
| *fst* | 0 | ? |
| *err* | **0** | **0** |

We solve this by adding next $fst = fst$ to the right-hand side of the second property we added when we analyzed *num*:

$$\square((get = \mathbf{0} \wedge put = \mathbf{0}) \vee err = \mathbf{1} \Rightarrow \mathsf{next}\ num = num\ \wedge\ \mathsf{next}\ fst = fst).$$

After adding this property, the analysis is happy.

# 4 Implementation

The implementation of our analysis is quite straightforward. We first build the *safety property observer* belonging to the property list [4]. A safety property observer is sometimes also called "checker circuit"; a circuit that has as inputs all signals appearing in the properties, and that has only one output "OK", that is always high if and only if the properties hold. Observers can be constructed automatically for formulas in many logics [5, 6, 2].
Then, we build a circuit using two copies of the observer, a *main* copy and a *shadow*. The signals that the two observers are observing are the same, except for the output signal we are analyzing, which we call *out* for the main copy and *out'* for the shadow copy.
Now, we can ask a standard LTL model checker (such as SMV [3]) to look for traces where both observers say OK, but where the values of *out* and *out'* differ at exactly one point. This indicates a forgotten case, because the property lists hold for both variants of the discovered trace.
The size of the analyzed circuit is linear in the size of the property observer.

# 5 Discussion

There are basically three reasons for not fully specifying the behaviour of all output signals: (1) The output is supposed to be underconstrained in the specification; (2) By choice, the specifier has decided to leave the output underconstrained; (3) The specifier has forgotten a case. We argue for an analysis that can discover the 3rd case, by forcing the specifier to explicity document in the properties if cases (1) or (2) are meant. We believe this leads to specifications of higher quality, which in turns leads to more dependable verification results. The analysis can be used in both simulation-based and formal property verification.

# References


[1] Hana Chockler, Orna Kupferman, Robert P. Kurshan, and Moshe Y. Vardi. A practical approach to coverage in model checking. In *Computer Aided Verification (CAV)*, 2001.

[2] IBM. FoCs – Formal Checkers, 2002. http://www.haifa.il.ibm.com/-projects/verification/focs/.

[3] K. McMillan. The SMV model checker, 2002. http://www-cad.eecs.berkeley.edu/~kenmcmil/smv/.

[4] F. Lagnier N. Halbwachs and P. Raymond. Synchronous observers and the verification of reactive systems. In *Third Int. Conf. on Algebraic Methodology and Software Technology (AMAST'93)*, 1993.

[5] J.-C. Fernandez N. Halbwachs and A. Bouajjanni. An executable temporal logic to express safety properties and its connection with the language lustre. In *Sixth International Symp. on Lucid and Intensional Programming (ISLIP'93)*, 1993.

[6] P. Raymond. Recognizing regular expressions by means of dataflows networks. In *23rd International Colloquium on Automata, Languages, and Programming (ICALP'96)*, LNCS 1099. Springer Verlag, 1996.

# Is Feature-Oriented Verification Useful for Hardware?*

Kathi Fisler
WPI Dept. of Computer Science
kfisler@cs.wpi.edu

Shriram Krishnamurthi
Computer Science Dept., Brown University
sk@cs.brown.edu

**Abstract**

Many in the software and programming languages communities have been exploring design modularizations based on features rather than physical components. Features engender a modular verification methodology that avoids many of the challenges that underlie conventional theories of modular verification, such as design and property decomposition. We present an overview of features and discuss our prior work in exploiting features for software verification. We hope this paper will spark discussion about whether features have a meaningful role in designing hardware for verifiability.

## 1 Introduction

Verifying large systems automatically often requires techniques for identifying design fragments that can be analyzed tractably. Because isolating such fragments is challenging, verifiers frequently rely on the modular structure of the design for guidance. Unfortunately, the portions of designs that impact properties can span several modules. As a result, the verification engineer either has to decompose the properties around the design modules or apply more sophisticated decomposition methods that do not directly exploit the modular structure. This seems a lost opportunity, and raises a question: are there modularizations that enable designers to naturally express more of their knowledge that matters for verification?

Modules in hardware description languages generally correspond to physical subcomponents of a system (such as the CPU or RAM). Aligning design modules with physical components seems natural, particularly in the hardware domain in which the end result is actual chips. Over the last decade, however, many researchers in software engineering and programming languages have

*This work is partially supported by the U.S. National Science Foundation grants CCR-0132659, CCR-0305834 and CCR-0305950.

explored modules that encapsulate user-defined *features* rather than fragments of implementations [2, 9, 17, 24, 25]. Intuitively, a feature is a piece of system functionality that is meaningful to an end user (an identifiable piece of functionality that an end user would pay for). A single system is a composition of the features that the end user wants. The large number of systems definable from a common set of features form a *product-line*. In a hardware context, an appropriate analog for an end user might be the overall system architect, who wants a chip to implement a particular set of algorithms or optimizations.

Feature-oriented modules are attractive for verification because properties often describe user-identifiable traits of a system. Many properties align with small sets of features. This alignment reduces, and often eliminates, the need for property decomposition. Features also support incremental reasoning about designs as they evolve. They suggest a two-stage verification methodology in which properties are first checked against individual features to determine constraints that the feature places on the rest of the system. As features are composed into products, the constraints are checked using lightweight analysis techniques. The constraints enable incremental verification and amortize verification costs over many products built from the same core features.

The first author began exploring feature-based verification after hearing a talk by Ken McMillan in 1998 describing his verification of a hardware implementation of Tomosulo's algorithm [23]. McMillan effectively decomposed the implementation around some of the key dataflows through the architecture. This decomposition isolated the parts of the design that affected key properties of the algorithm, enabling them to be verified efficiently using a combination of abstraction and other model-reduction techniques. The fragments had the spirit of features as were being described in the software community, but were not identified as such at the design level. Feature-based constructs are uncommon, but not new, in hardware specification languages. The **extend** construct in Verisity's *e* language was also motivated by the work on features from the software community [13]. To the best of our knowledge, however, the Verisity team has not exploited this modularization for verification. The question then is whether feature-based decompositions could help capture complicated manual decompositions such as McMillan's.

Over the last five years, we have been developing theories of incremental and modular model checking for feature-oriented systems expressed as state machines [5, 18, 21]. Our work has shown that features induce a form of module composition that lies between purely sequential and purely parallel composition [10]. Furthermore, modular verification in this framework is best viewed as a combination of constraint generation and constraint solving, rather than as compositions of results from straightforward model checking [5]. Our work to date is largely theoretical but has been prototyped (with implementation) against some actual software designs.

The position paper has two goals: first, to give an overview of the benefits, assumptions, and challenges of feature-oriented modeling and verification; second, to spark discussion as to whether this style has a meaningful role for hardware. Key questions include (1) the extent to which features are useful for large-scale organization of hardware designs, (2) how hardware design flows might exploit the opportunities for incremental verification that features enable, and (3) how well feature-based decompositions align with challenging hardware verification tasks. As this paper is more of an overview than a presentation of new results, the presentation is informal, with references to other published papers containing the formal details.

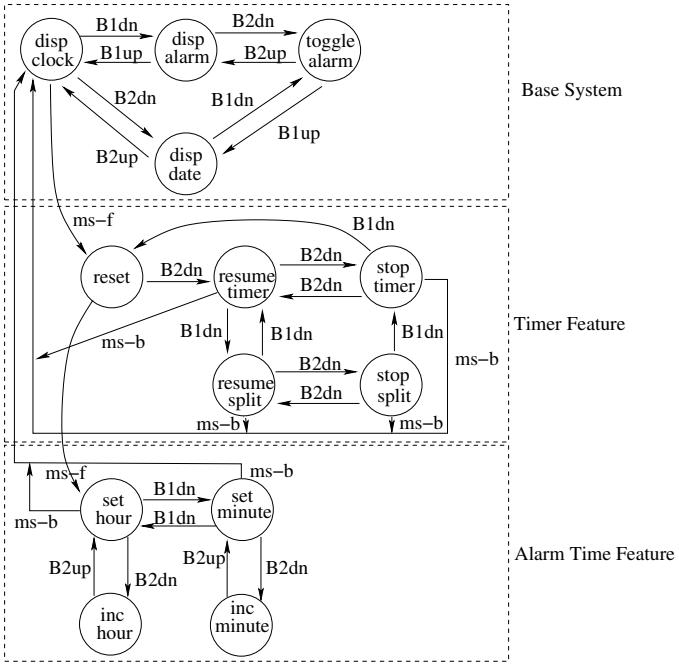



Figure 1: A feature-oriented design for a sportswatch.

## 2 Modularizing Systems by Features

We motivate the structure of feature-oriented systems through two small yet illustrative examples. Every design has a *base system* containing its core functionality. Features extend the base system with additional functionality. Both the base system and the features consist of state machines that get composed into larger state machines in a particular way. As the base and features are indistinguishable at the level of formal models, the rest of the paper views the base as just another feature (but one that happens to be included in every system).

The simplest feature-oriented systems use a single state machine for each feature (including the base). Figure 1 shows the design of a stopwatch expressed in terms of features.[1] The base contains four display nodes: clock display, alarm time display, date display, and an alarm status display that supports toggling the alarm status. One feature adds a timer which the user can reset, resume, and stop; this feature also supports a split timer for capturing time instantaneously. Another feature enables setting the alarm time. The watch is controlled through two buttons (B1 and B2) and a mode switch that can be in the forward (ms-f) or back (ms-b) positions.

In this example, each state in the overall design belongs to some feature. Most of the transitions fall within features, but some connect features. The latter transitions are added when features are composed into larger systems. Each feature has a *construction interface* indicating which states

[1]This example is due to Jia Liu, graduate student at UT Austin.

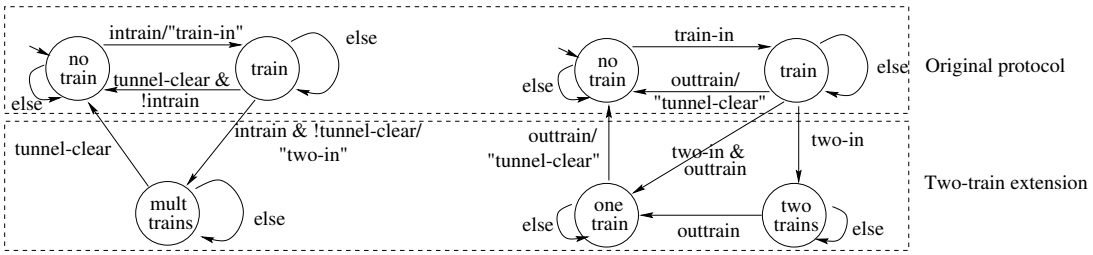



Figure 2: A feature-oriented design for a track-operator communication protocol.

---

can have transitions to and from other features (the figure implicitly identifies these states). When a designer composes two features (or adds a feature to an existing system), he indicates which of the interface states should be used as the source and sink from each piece. Composition entails inserting edges between the chosen states. In the stopwatch example, the interface for the base system uses *dispclock* as both the source and the sink of transitions that connect new features, while the timer feature uses *reset* as its *entry* state and the other four states as its *exit* states back to the overall system. All of the transitions that span the base and the timer feature connect to the interface states. Once the timer feature is added, the designer can change the construction interface for the composed system to transfer control out at the *reset* state and in at the *dispclock* state. The alarm time feature attaches at this revised interface.

This example suggests that features compose sequentially, albeit potentially creating cycles as they direct control flow back to other features. They do indeed, though the composition model becomes more complicated for features that span multiple state machines, as the next example illustrates.

Figure 2 shows an example of a communications protocol expressed as features over multiple state machines. This protocol, taken from Holzmann's book [14], governs communication between operators at either end of a long train tunnel covering a one-way track. The two state machines model the human operators on either end of the tunnel. Unable to see one another, the operators communicate messages about the status of the tunnel. In the original protocol (the base system), the operators communicate when trains are entering and exiting the tunnel. The inbound operator sends a *train-in* message to the outbound operator when a train enters the tunnel. The outbound operator sends a *train-clear* message to the inbound operator when a train exits the tunnel. The base system consists of the protocol for exchanging these two messages.

Although the protocol was designed to prevent two trains from ever being in the tunnel simultaneously, an accident occurred when a second train entered the tunnel (in the same direction as the first train) before the first one left; although the inbound operator suspected the problem, the communication protocol was too weak to convey the situation to the outbound operator. One solution is to add messages to the protocol that convey this information accurately. The extension (feature) adds a *two-in* message from the inbound to the outbound operator; it also adds states to both operator machines so that the outbound operator does not send the *train-clear* message until both trains have left the tunnel.

As the figure shows, features that span multiple state machines contain a fragment for each

state machine in the overall system (features may omit fragments for state machines that they do not affect; they may also introduce new state machines in the general model). Each fragment connects to its corresponding fragment in another feature through interface states as in the single state machine case. Thus, connection interfaces remain at the level of the individual state machines. The global state machine for this design is the cross-product of the compositions of the individual state machine fragments from the feature. This definition appears to lose the sequential composition model used for single features. Furthermore, it does not suggest how we get a cross-product state machine for an individual feature. We address both issues shortly.

# 3 Verifying Systems by Features

Features enable two related styles of verification. First, global system properties can be established incrementally as new features are added to the system. Second, properties can be proven of an individual feature and then shown to hold after the feature is composed into an existing system. In both cases, property preservation checks should traverse only the new portions of the state space: the new feature for incremental verification, and (portions of) the larger system for feature-specific properties.

Our work supports these styles of verification through the following tasks:

1. Proving a CTL property of an individual feature or composition of features. This requires building a single state machine representing the state space of an individual feature.

2. Deriving a set of interface constraints for a feature that are sufficient to preserve a particular property after composition (the *preservation constraints*).

3. Proving that a feature satisfies the preservation constraints of another feature (or existing system).

These activities correspond to a kind of modular verification, where the features are modules. As in standard approaches to modular verification, we are interested in proving properties of modules and in preserving those properties upon composition with other modules. Our work differs from standard modular verification because features use a different composition semantics than the purely sequential or purely parallel models that underlies other theories of modular verification. The motivation for modular reasoning is also different in our context. Since features largely compose sequentially, modular reasoning has less impact on tractability than under parallel composition. The main benefit of modularity in our context comes from amortizing verification effort across different systems that are built from the same features. Modularity in this view is more about design methodology than verification methodology, though we propose exploiting the fruits of the former to simplify the latter.

Intuitively, items 2 and 3 perform modular model checking assuming sequential composition. Item 2 essentially caches the subformula labels ascribed to interface states during CTL model checking (we use CTL instead of LTL because the algorithm for the former associates subformulas

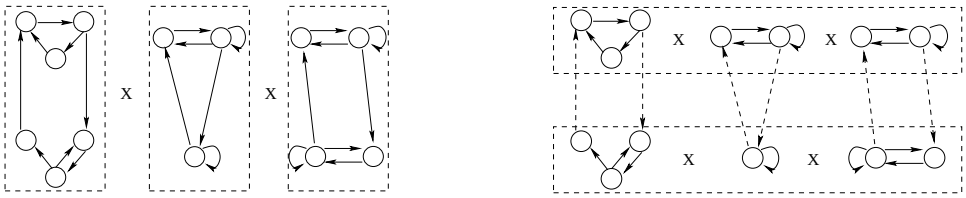



Figure 3: Two approaches to constructing composed systems.

with states). Item 3 uses model checking to prove that a new feature preserves the cached subformula labels at the interface states to which it connects. Our existing papers describe these steps in more detail [10, 19]; Section 4 explains some limitations of this intuitive model.

Item 1 is the interesting step for this paper, because it requires a single state machine for each feature which is not readily available in the multiple-machine model. Consider the feature extension in the tunnel protocol, which consists of the two state machines in the lower dashed box in Figure 2. We cannot simply form the cross-product of these machines using their interface states as the initial states because the two machines might not reach these interface states simultaneously. For example, the inbound operator may notice the second train before the outbound operator has registered that there is a train in the tunnel. The initial states of the feature's cross-product therefore depends on the synchronization of the interface states in the other system to which the new feature connects. We can compute this information in the form of the subgraph of the existing system that contains its interface states; this subgraph in turn provides the initial states for constructing the cross-product of the new feature. The details of this construction appear in an earlier paper [10].

Interface subgraphs allow us to compose features in a mostly-sequential manner (sans the limited interleaving of behavior across features in the interface subgraph). Figure 3 shows two views of a feature-oriented system, one in which cross-products are taken at the level of the composed machines (left) and one in which cross-products are taken at the level of features, which are then composed sequentially (right). The model on the right, which our verification methodology exploits, shows that the composition model underlying feature-based systems is *quasi-sequential*, a sequential composition (between features) over well-scoped parallel compositions (within features).

The quasi-sequential model is reasonable when features operate largely independently of one another, an assumption that holds in some feature-oriented systems [2] but not all [15]. If the features themselves, rather than their components, operate in parallel, the interface subgraphs could possibly induce state explosion. More experience with feature-oriented designs is needed to determine the extent of this problem in practice.

# 4 Challenges and Status

We have outlined a model of feature-oriented designs and a methodology for verifying systems incrementally and for verifying features in isolation. As one would expect, this approach faces several challenges and open problems:

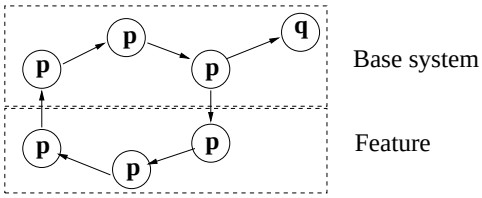



Figure 4: How composition can induce cyclic reasoning.

## Cyclic Reasoning

Modular reasoning theories must handle dependencies between modules that lead to cyclic reasoning. In the context of features, cyclic reasoning arises when adding a feature creates a new cycle in the global state space that visits states from both the existing system and the new feature. Figure 4 shows a simple case in which a naïve sequential model checker might incorrectly determine that $A[pUq]$ holds after adding a new feature to the system. This formula labels all states in the (fragment of the) base system shown. When the feature is added, a naïve sequential composition algorithm might assume the formula is true at the point of reentry to the base and use that to (incorrectly) infer that the formula is true at all states in the base. Such situations can be handled correctly by verifying additional properties about discharging eventualities on paths between interface states and altering the modular verification algorithm slightly based on the results of those checks [18].

## Handling Environment Models

Verifying the train communication protocol requires an *environment model* of the trains that can enter and exit the tunnel. The model, which generates the signals *intrain* and *outtrain* used in the protocol machine, must encode constraints such as "no train can exit the tunnel before it enters the tunnel". Environment models always compose in parallel with the systems that consume their outputs; this is true for feature-oriented systems as well. Imagine that we want to verify a property about a new feature $F$ in isolation, and that $F$ references signals defined in the environment model. We therefore must take the cross product of the environment model with the feature prior to verification, but what is the initial state of the environment model in this cross product? The initial state of the environment model synchronizes with the initial state of the global composed system. The feature in the train protocol is only reached once a train is already in the tunnel, however, which is not true at the initial state of the environment model.

This scenario demonstrates that verifying features in isolation requires ways to determine the state of the environment at the point when features might be invoked. We proposed a preliminary approach to this problem in the context of the train protocol [10], but much more work needs to be done on a wider range of case studies. This problem is similar to that of generating a testing harness for a feature-oriented design [27].

## Lifting Feature Properties to the Entire System

When global system properties are verified incrementally as new features are added to the system, our methodology guarantees that those properties are true at the initial state of the final composed system (assuming that initial state is part of the base). Our modular methodology guarantees that properties proven of individual features remain true once the feature is composed with other features, but this merely proves that the feature property is true at the initial state of the feature, not of the global system. In practice, we often want to lift properties proven of individual features to the initial state of the global system. A naïve way to handle this is to incrementally verify a system relative to all desired properties of individual features, but this doesn't exploit the benefit of features, namely that each feature contains the majority of the system detail needed to verify its properties. When lifting properties of a feature $F$ to the system level, other features often contribute nothing more than paths from the initial state to $F$. Dominguez exploits this observation to lift feature properties without verifying those properties against all features [7]. Shmuel Katz at the Technion is also working on modular approaches to proving properties introduced by individual features.[2]

## Interactions Between Features

A property that holds of one feature in isolation may not hold in a global system due to interactions between the features. Some interactions are desirable (such as when the operation of one feature takes priority over another, thus changing the conditions under which the first feature is invoked); others are not. Feature interaction is an established problem with an extensive research literature [16]. Verification methodologies based on features must be able to detect when features have interacted and provide ways to reason about the conditions under which features may interact safely. Our core methodology using sequential model checking handles the former. Handling the latter amounts to reasoning about the (sequentially-composed) environments in which a feature can operate without violating its properties.

Asking designers to attach environment models to each feature individually is onerous. We have therefore developed initial techniques to automatically generate the environment constraints that will preserve specific properties of individual features [5]. This approach relies on a two phases–constraint generation on individual features and constraint discharge when features are composed–to verify global system properties. Our experience suggests that model checking, with its binary output decision, may not be sufficiently fine-grained to analyze features in the face of potential feature interactions.

## Using Features to Repair Failed System Properties

We have discussed how to preserve properties of systems and features as features are composed into larger systems. Sometimes, features are introduced to make a system satisfy a property that does not hold without the feature. The tunnel protocol provides an example: the base system

[2] Personal communication; results not yet published.

alone is not safe if two trains get into the tunnel, but the feature repairs the protocol to satisfy this property. Modular approaches for determining that features repair failed properties is an open area of research.

### Sharing Code Across Features

The features model presented in Section 2 assumes that each feature module is self-contained. Real systems, however, may need to share support code across features, either in the form of code libraries (software) or shared devices (hardware). We are not aware of models of features that enable sharing of common infrastructure. Example systems that utilize such sharing would help drive research into this issue.

## 5 Related Work

Many researchers have proposed system decompositions that resemble features in spirit; these include layers [3], collaborations [24], aspects [17] and units [9]. A brief sampling of successful designs in this vein includes a military command-and-control scenario simulator [2], a programming environment [8], network protocols and database systems [3, 4, 22], and verification tools [11, 26].

Laster and Grumberg proposed the first algorithm for modular model checking under sequential composition [20]. This work aims at decomposing systems specified as a single, monolithic, state machine; in particular, it lacks a design framework, such as features, that drives the decomposition of the system. Other work uses hierarchical state machines [1] and StateCharts [6] to guide the decomposition, but the resulting systems are still monolithic. In contrast, our work is designed to support systems that are conceived and built incrementally as combinations of features. Features can be developed in isolation of one another, without knowledge of which other features will be included in the final system. This basis and our handling of multiple state machines per feature distinguish our approach from these others.

The Horus/Ensemble project [22] builds network protocols through compositions of features. They have discussed feature-oriented verification, but our work differs from theirs in several ways: first, they do not appear to verify features in isolation from one another; second, their work concentrated on a particular set of features, while we are proposing a general framework for this style of verification; third, their work uses theorem proving rather than model checking. We have done a preliminary extension of our work to theorem proving [12].

## 6 Summary

This paper has given an overview of feature-oriented design and verification. Features are promising for verification because they naturally decompose designs into fragments that align with properties. This simplifies modular verification without requiring designers to decompose properties around the modular structure. Exploiting this promise, however, requires methodologies and models for capturing realistic hardware designs as sets of features. We hope this paper will spark dis-

cussion about the possible role of features in capturing designs and the extent to which problems such as feature interaction arise in a hardware context.

# References


[1] Alur, R. and M. Yannakakis. Model checking of hierarchical state machines. In *Symposium on the Foundations of Software Engineering*, pages 175–188, 1998.

[2] Batory, D., C. Johnson, B. MacDonald and D. von Heeder. Achieving extensibility through product-lines and domain-specific languages: A case study. *ACM Transactions on Software Engineering and Methodology*, 11(2):191–214, April 2002.

[3] Batory, D. and S. O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology*, 1(4):355–398, October 1992.

[4] Biagioni, E., R. Harper, P. Lee and B. G. Milnes. Signatures for a network protocol stack: A systems application of Standard ML. In *ACM Symposium on Lisp and Functional Programming*, 1994.

[5] Blundell, C., K. Fisler, S. Krishnamurthi and P. Van Hentenryck. Parameterized interfaces for open system verification of product lines. In *IEEE International Symposium on Automated Software Engineering*, pages 258–267, September 2004.

[6] Clarke, E. M. and W. Heinle. Modular translation of Statecharts to SMV. Technical Report CMU-CS-00-XXX, Carnegie Mellon University School of Computer Science, August 2000.

[7] Dominguez, A. L. J. Verification of DFC call protocol correctness criteria. Master's thesis, University of Waterloo Department of Computer Science, 2005.

[8] Findler, R. B., J. Clements, C. Flanagan, M. Flatt, S. Krishnamurthi, P. Steckler and M. Felleisen. DrScheme: A programming environment for Scheme. *Journal of Functional Programming*, 12(2):159–182, 2002.

[9] Findler, R. B. and M. Flatt. Modular object-oriented programming with units and mixins. In *ACM SIGPLAN International Conference on Functional Programming*, pages 94–104, 1998.

[10] Fisler, K. and S. Krishnamurthi. Modular verification of collaboration-based software designs. In *Symposium on the Foundations of Software Engineering*, pages 152–163. ACM Press, September 2001.

[11] Fisler, K., S. Krishnamurthi and K. E. Gray. Implementing extensible theorem provers. In *International Conference on Theorem Proving in Higher-Order Logic: Emerging Trends*, Research Report, INRIA Sophia Antipolis, September 1999.

[12] Fisler, K. and B. G. Roberts. A case study in using ACL2 for feature-oriented verification. In *Proceedings of the ACL2 Workshop*, November 2004.

[13] Hollander, Y., M. Morley and A. Noy. The *e* language: A fresh separation of concerns. In *Proceedings of TOOLS Europe*, March 2001.

[14] Holzmann, G. *Design and Validation of Computer Protocols*. Prentice-Hall, 1991.

[15] Jackson, M. and P. Zave. Distributed feature composition: A virtual architecture for telecommunications services. *IEEE Transactions on Software Engineering*, 24(10):831–847, October 1998.

[16] Keck, D. O. and P. J. Kuehn. The feature and service interaction problem in telecommunications systems: A survey. *IEEE Transactions on Software Engineering*, 24(10):779–796, October 1998.

[17] Kiczales, G., J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier and J. Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming*, pages 220–242, 1997.

[18] Krishnamurthi, S. and K. Fisler. Foundations of incremental aspect model-checking. *ACM Transactions on Software Engineering and Methodology*, Accepted pending minor edits, 2005. Preliminary versions appeared in other cited works [10, 19].

[19] Krishnamurthi, S., K. Fisler and M. Greenberg. Verifying aspect advice modularly. In *ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, pages 137–146, November 2004.

[20] Laster, K. and O. Grumberg. Modular model checking of software. In *Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 1998.

[21] Li, H., S. Krishnamurthi and K. Fisler. Modular verification of open features through three-valued model checking. *Journal of Automated Software Engineering*, 12(3):349–382, July 2005.

[22] Liu, X., C. Kreitz, R. van Renesse, J. Hickey, M. Hayden, K. Birman and R. Constable. Building reliable, high-performance communication systems from components. In *Symposium on Operation System Principles*, pages 80–92. ACM Press, 1999.

[23] McMillan, K. L. Verification of an implementation of Tomasulo's algorithm by compositional model checking. In *International Conference on Computer-Aided Verification*, 1998.

[24] Mezini, M. and K. Lieberherr. Adaptive plug-and-play components for evolutionary software development. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications*, pages 97–116, October 1998.

[25] Ossher, H. and P. Tarr. Multi-dimensional separation of concerns in hyperspace. Technical Report RC 21452(96717), IBM, April 1999.

[26] Stirewalt, K. and L. Dillon. A component-based approach to building formal-analysis tools. In *International Conference on Software Engineering*, 2001.

[27] Xie, T. and J. Zhao. A framework and tool supports for generating test inputs of AspectJ programs. In *Proc. 5th International Conference on Aspect-Oriented Software Development (AOSD 2006)*, March 2006.

# Another Dimension to High Level Synthesis: Verification

Malay K. Ganai
Aarti Gupta

NEC Labs America
Princeton, NJ, USA

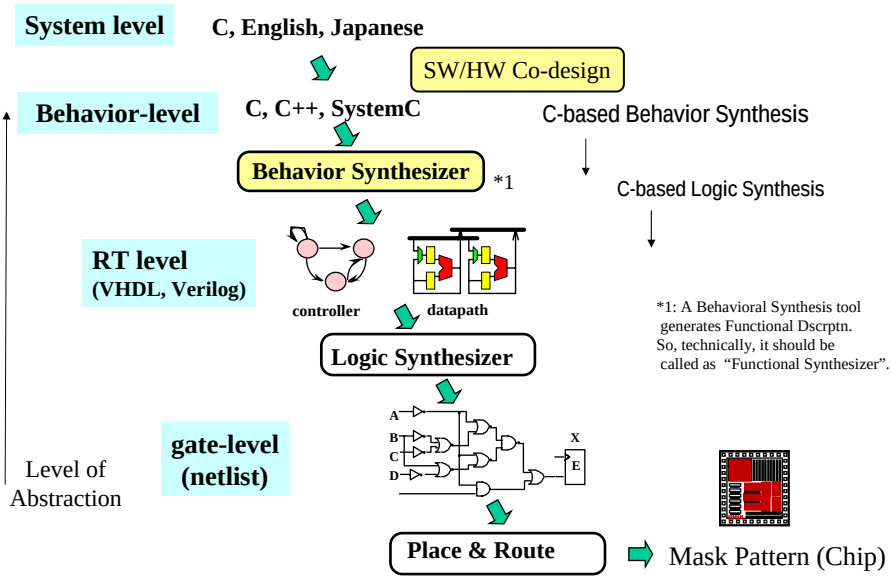
Akira Mukaiyama
Kazutoshi Wakabayshi

Central Research Lab
NEC, Tokyo, Japan

## Outline

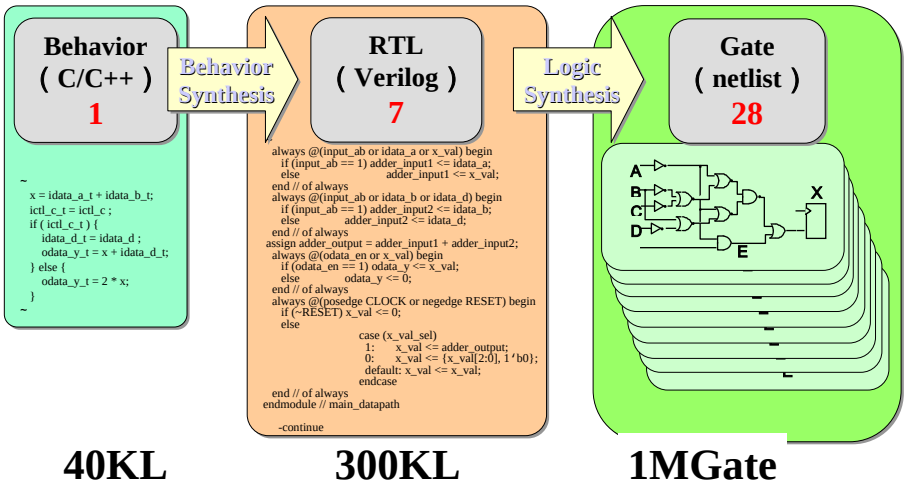
- Introduction
  - High level synthesis
  - Verification techniques (strengths and weaknesses)
- Parameters controlling HLS
  - Area, performance and power trade-off
  - Impact on verification
- Case Study
  - Experimentation with *Cyber* and *DiVer* Industrial Tools
- Paradigm Shift: Synthesis for Verification
  - Generate verification "aware" model
  - Maximize the benefit of current verification techniques
  - Open discussion

# Design Flow & Levels of Abstraction






# Why Upper Level leads to High Productivity? - Less Design Size



# What is *"Behavior Synthesis"*

Behavior in C

```
char A,B,C,D;
char E,F;
main(){
char X;
X = A + B;
E = X * D;
F = (B + C) * X;
}
```

**8 Lines**

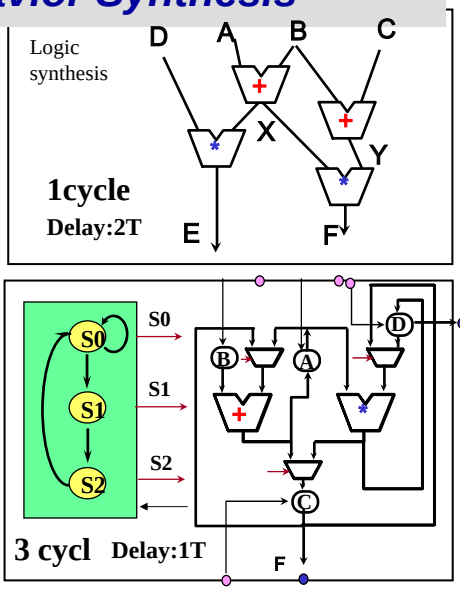
+ : 2
* : 2

constraints

**RTL**

+ : 1
* : 1

**100 lines**



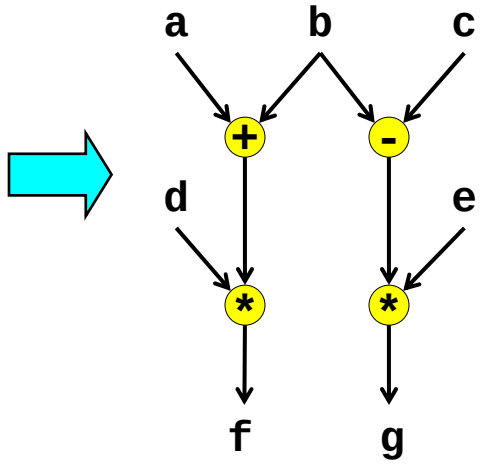



# Steps of Behavioral Synthesis

1. Create behavior description into DFG
2. Source code level optimization
3. Allocate
4. Scheduling
5. Binding
6. Generation of Controller
7. Module Generation

## 1: *Create* DFG

```
f = (a+b)*d;
g = (b-c)*e;
```





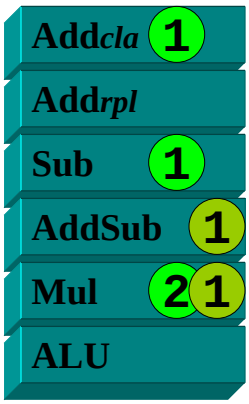

## 2: *Source Code Optimization*

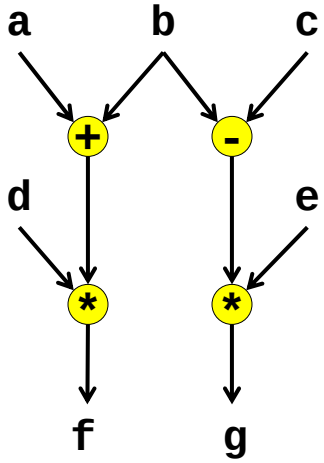
- Extract common sub-expression
  - a=(x+y)*z; b=x+y-z; t=x+y; a=t*z;b=t-z;
- Multi-dimensional array to one-dimensional array
  - a2[4][5]=>a1[20],x=a2[i][j]=>x=a1[i*5+j]=>a1[i<<2+i+j]
- Constant propagation
  - a*2=> a::0; a=1+b, b=2 => a=3
- Automatic Bit adjustment (automatic cast )

  a(4bit) = b(2bit) => a = 00::b or a=b(0)::b(0)::b;
- Loop unrolling, Loop folding, Pipelining
- Function inline expansion
- Dead Code Elimination
- Tree balancing (for extracting parallelism)

## 3: *Allocate* set of usable FUs

(#of FUs, bit width, multi-function, etc)



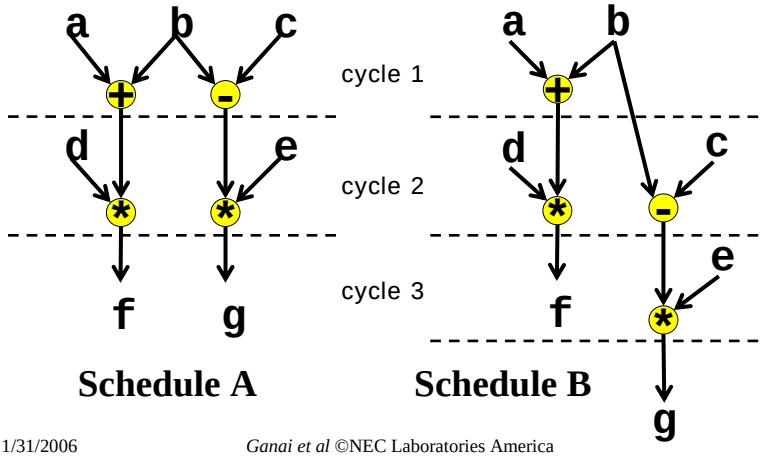





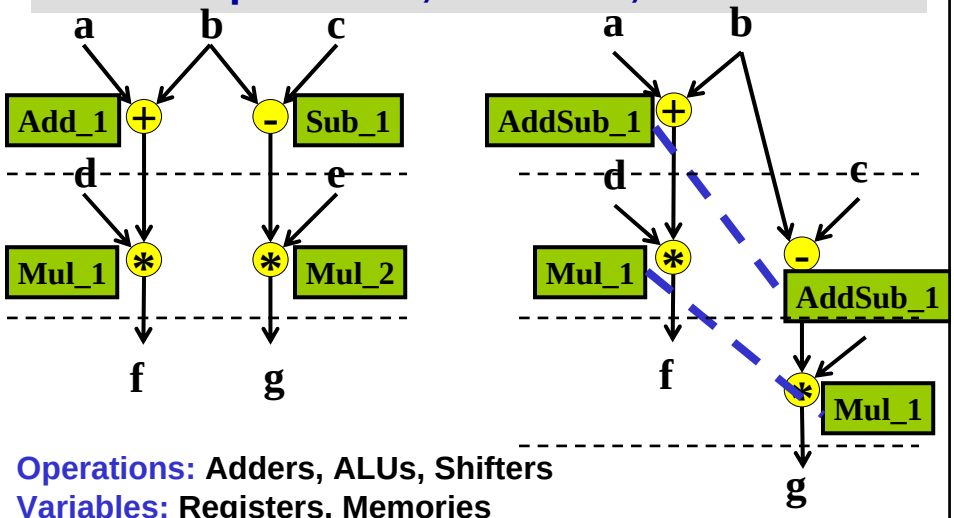

## 4: *Schedule* an operation to a cycle

**Area constraint:** Extract parallelism, speculation, operation chaining, pipeline FUs

**Time Constraint:** Extract conditional mutual exclusion



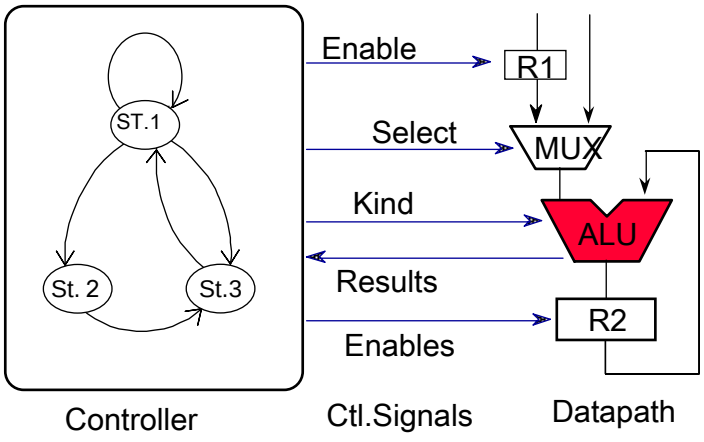
## 5: *Bind* operations, variables, data transfer




**Operations:** Adders, ALUs, Shifters
**Variables:** Registers, Memories
**Data Transfers:** Wire, MUX, Bus

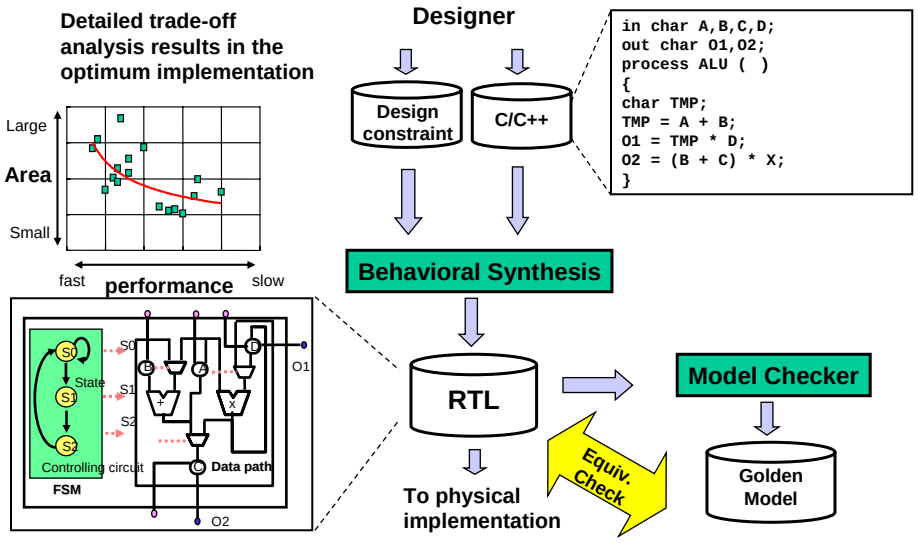


## 6: *Generation of Controller: FSMD*



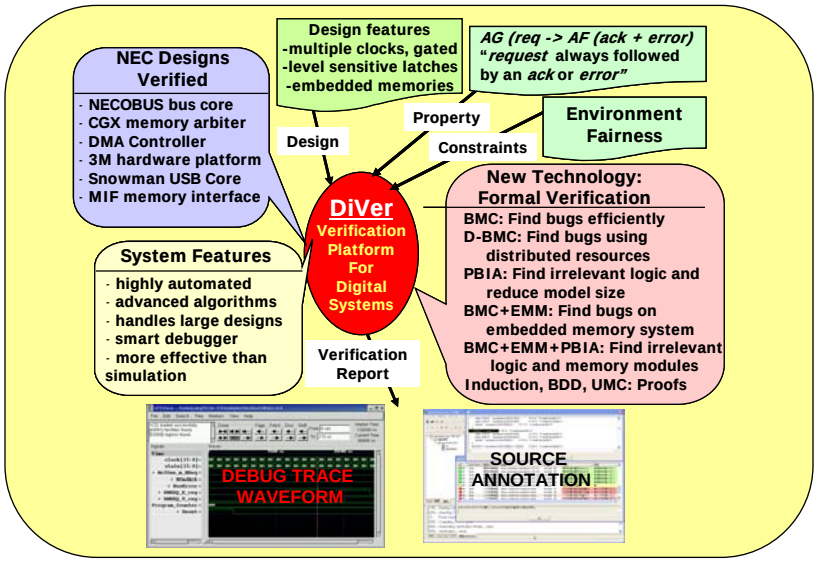

Each Behavior Synthesis tool has their own target architecture.
FSMD is a natural architecture for Behavior Synthesis.
Some tools generate Micro-code based architecture or more processor oriented architecture. Synthesis algorithms are somehow different for such cases from the method in this tutorial.

# Current Synthesis + Verification flow

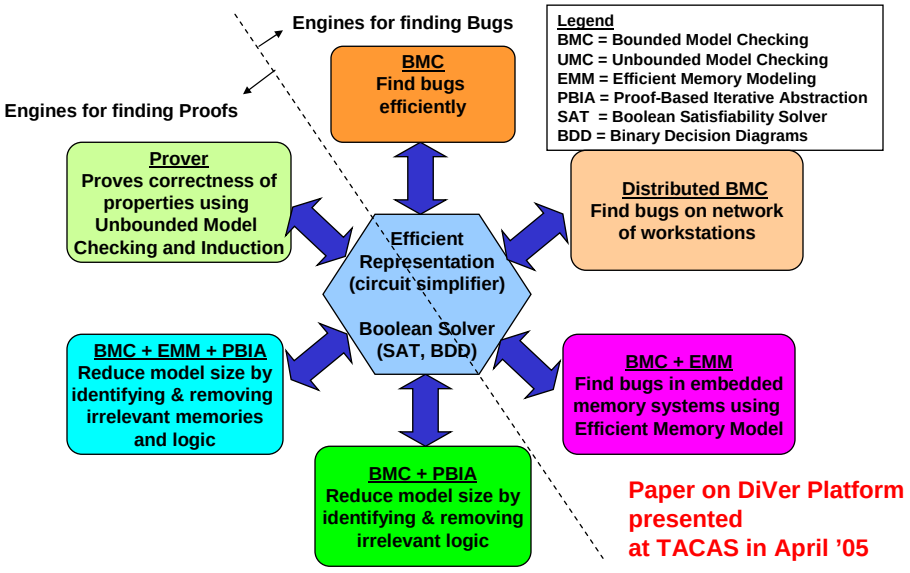





# DiVer: Overview



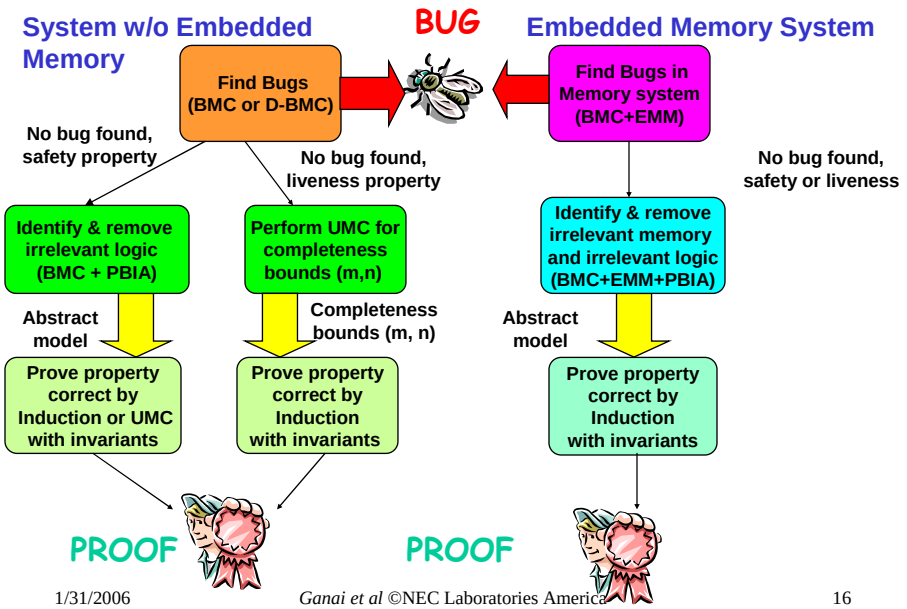
# Wheel of Verification Engines




Paper on DiVer Platform presented at TACAS in April '05



# Verification Flows



# Hybrid SAT

– Ganai et al, DAC'02

**Problem Representation**

- Gate Clauses: typically short, maintained as 2-input gate
- Learned clauses: typically large, maintained as CNF

- **Deduction Engine – Hybrid BCP**
  - Circuit-based BCP on gate clauses using *fast table lookup*
  - CNF-based BCP on learnt clauses using *lazy update*
- **Decision Engine**
  - Use of circuit-based information, CNF Heuristics
- **Diagnostic Engine**
  - Record both clauses and gate nodes as reasons for conflict



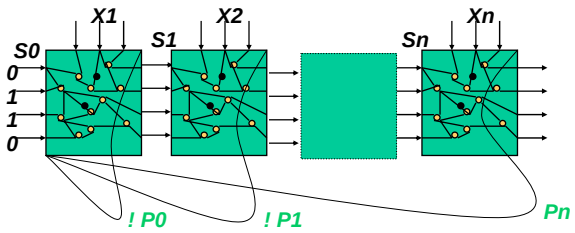
# Strengths/Weaknesses: SAT Solver

- **Strengths**
  - Good at finding a satisfying solution
  - Search scales well with problem size
  - Incremental learning: Low overhead and improves subsequent solves drastically
  - Performs well on disjoint-partitioned sub-structures
  - Matured and well-known heuristics exist
  - Hybrid SAT: advantageous over CNF/Circuit SAT solvers
- **Weakness**
  - Muxes are detrimental
  - Not good for problems arising from optimized circuits with heavy sharing of registers and function units
  - Not good at enumerating solutions

# Bounded Model Checking

- Unroll Circuit for bounded length
- Translate problem into a Boolean formula
- Use SAT solver to check for a witness

– Biere et al DAC'00




- Dynamic Circuit Transformation
  - Propagation of initial state constants
  - Subgraph isomorphism detection and removal
- Reuse of Learned Property Constraints
- Time-based Partitioning
  - Smaller sub problems, incremental formulation
  - Incremental Hybrid SAT Solver

*state-of-the-art features*

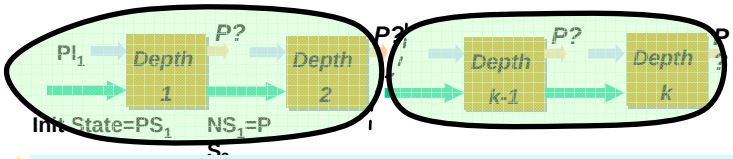
– Ganai et al, DAC'02, VLSI'02, DAC'05



# SAT-based Distributed BMC

**As unroll depth k increases, memory requirement can exceed the limit of a single server !**

– Ganai et al CHARME'03




Partition unrolled circuit and Distribute SAT check

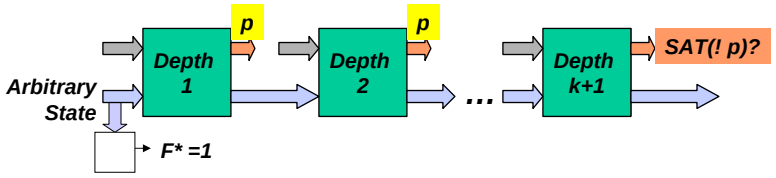
- Ubiquitous Ethernet LAN
  - Standard, Cheap
- Network of Workstations (NOW)
  - Easily available, Idle mostly (esp. Manager's)
- BMC problem provides natural disjoint partitioning
  - Parallelizing partition specific tasks like BCP

# BMC/UMC Proof for G(p) with Invariants

– Gupta et al CAV'03
– Sheeran et al FMCAD'00

**K-depth Inductive Step:**

- If *Unsat(!p_k)*, then property is true




Additional constraint *F** on the arbitrary starting state

- *F** is overapproximated forward reachable states
- Provides an induction invariant
- Frequently allows induction proof to succeed



# Strengths/Weaknesses: SAT-based BMC

## Strengths

- Finds shortest length counter-example efficiently
- Successive problems overlap, allowing incremental formulation and incremental learning for SAT
- Provides natural disjoint partitioning of problems; allowing distributed BMC to overcome memory requirements
- Reached states are not explicitly stored
- Insensitive to number of registers, search scales well
- State-of-the-art improvisations exist, well matured technology

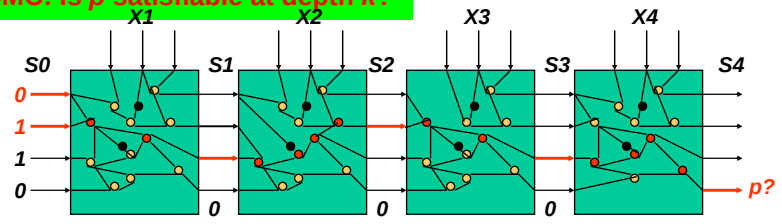
## Weaknesses

- Problem size grows linearly with unroll depth, computation can only worse
- Sensitive to the size of the model
- Incomplete, stopping criteria is difficult to determine
- Not good when model is heavily sequentialized (few events per cycle); longer search depths

# BMC with SAT Proof Analysis

– McMillan et al TACAS'03
– Gupta et al ICCAD'03, VLSI'05

**BMC: Is *p* satisfiable at depth *k*?**




- **Suppose no counterexample at depth *k***
  - **Derive an unsatisfiable core *R(k)* using SAT solver**

- **Intuition for Abstraction**
  - **Abstract model with *R(k)* implies no counterexample at depth *k***
  - **The abstract model may be correct for k' > k, maybe all k'**
  - **Typically *R(k)* is much smaller than entire design**



# Strengths/Weaknesses: Proof-based Abstraction

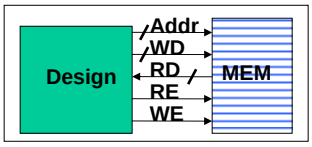
- **Strengths**
  - **Identifying and removing irrelevant logic**
  - **Identifies irrelevant memory and ports**
  - **Can be applied iteratively**
  - **Often leads to proof or deeper bounded proof**
  - **Reduces the refinements requirements by eliminating all counter-example for a bounded depth**

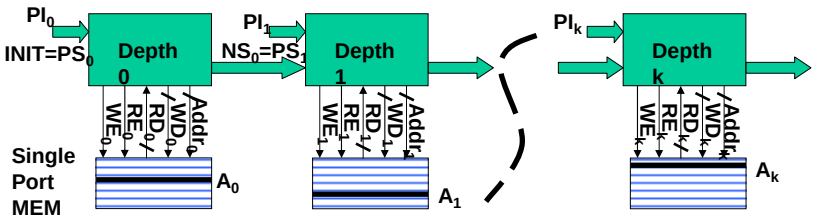
- **Weaknesses**
  - **Sharing of functional unit is detrimental, increases unsatisfiable core size**
  - **Other weakness similar to SAT-based BMC with or without EMM**

# Efficient Memory Model Approach

## Observation: (1 Mem, 1 Port)

- At most one address valid per cycle
- At most one mem write per cycle




– Ganai et al CAV'04, DATE'05

## EMM: Remove memories, but add constraints lazily

- Forwarding semantics is maintained
- Exclusivity of a read-write pair is captured explicitly
- Constraints are represented efficiently



# Strengths/Weaknesses: SAT-based BMC+EMM

## Strengths

- EMM is sound and complete, memory semantics is preserved
- No examination or translation of design module is required
- Memory is not modeled explicitly; reduces design size significantly
- Exclusivity of read-write pair is captured; reduces muxes
- Constraints grow lazily (quadratically) with unrolling
- Supports multiple memories and ports; can be exploited to reduce the sequential depth of the design
- Good for properties dependent on fewer memory accesses
- Works well with arbitrary or uniform memory initialization

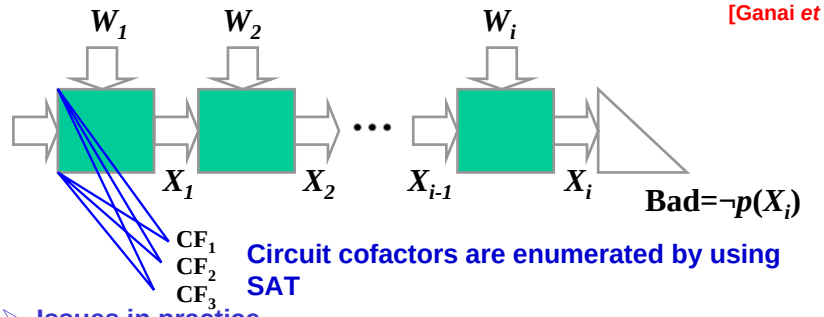
## Weaknesses

- Large memory accesses is detrimental
- Non-uniform memory initialization increases memory cycles
- Other weaknesses similar to SAT-based BMC

## SAT-based Unbounded Model Checking

- Symbolic backward traversal using unrolled TR

[Ganai *et al.* 04]




Circuit cofactors are enumerated by using SAT

- Issues in practice
  - State sets (represented as circuit cofactors) may blow up
  - Performance is not as good as SAT-based BMC (search for bugs), which avoids computation of state sets
- Complementary to BDD-based UMC for deriving proofs



## Strengths/Weaknesses: UMC

- Strengths
  - Efficient SAT solution enumerations
  - Uses efficient representation for states
  - Several low-overhead heuristics to enlarge solution states
  - Efficient pre-image computation, uses unrolled transition relation
  - Effective when primary inputs are large
  - Effective when the backward diameter is small
- Weaknesses
  - Not good for forward image computation
  - Not effective in quantifying out relational variables

## Parameters Controlling HLS (1/4)

- **Increase use of Functional Units, Registers**
  - **Area:** Increases
  - **Performance:** Can increase by reduction in control steps
  - **Power:** Increases due to increase in output capacitance, leakage current
- **Reuse of Functional Units, registers**
  - **Area:** Decreases, assuming mux area is small
  - **Performance:** Decreases due to increase in control steps
  - **Power:** Increases due to increased switching activities



## Parameters Controlling HLS (2/4)

- **Reduction in Control Steps**
  - **Area**: Increases due to increase use of FUs
  - **Performance:** Increases if frequency of control clock is unchanged
  - **Power:** Decreases due to decrease in switching activity. For a fixed throughput, frequency of control clock and hence, supply voltage can be reduced to allow power reduction
- **Speculative Execution**
  - **Area:** Increase if more FUs are required for speculative branch
  - **Performance:** Increases as control steps reduces
  - **Power:** Increases due to increase switching, speculative execution and reuse of FUs

## Parameters Controlling HLS (3/4)

- **Pipelines**
  - **Area:** Increase due to increase use of FUs
  - **Performance:** Increases through puts
  - **Power:** Increase due reuse of Regs, switching.

- **Parallelization (conditional mutual exclusion)**
  - **Area:** Decrease by reuse of FUs
  - **Performance:** Increases as c-steps reduces
  - **Power:** Increase due to increase switching, reuse of FUs



## Parameters Controlling HLS (4/4)

- **Operations reuse**
  - **Area**: Decreases due to fewer FUs
  - **Performance:** Decreases if critical length increases
  - **Power:** Decreases due to less switching

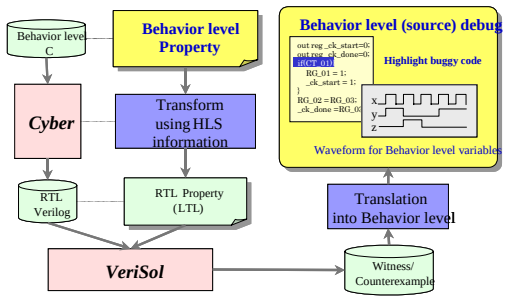
- **Increase of Clock freq (f) and supply voltage (V)**
  - **Area:** Increases due to use of faster but larger FUs
  - **Performance:** Increases if control steps does not change; otherwise, can affect throughput
  - **Power:** Increases $\alpha$ $(f . V^{2})$

# Impact of HLS parameters on Verification

- **Re-use of FUs, Regs**
  - **Adversely affects due to use of muxes**
- **Increase use of FUs, Regs**
  - **Not good in general. But better than re-use.**
- **Reduction in control steps**
  - **Definitely good. Reduces the sequential depth**
- **Speculative execution/ Parallelization**
  - **Good if FUs are not re-used. Reduces the sequential depth.**
- **Pipelines**
  - **Increases verification complexity without adding any functional behavior. FUs re-used and is not good.**
- **Operation re-use**
  - **Good in general. Reduces number of operations.**
- **Clock/Voltage change**
  - **Selection of FUs affects verification complexity**



# Cyber Work Bench: C-based Design Flow




- **Cyber Work Bench (CWB)**
  - **Developed by CRL (Wakabayashi-san) for C-based design**
  - **Generates property monitors automatically for VeriSol**
  - **Provides source-level debugging based on bugs found by VeriSol**
  - **Provides a seamless integration, with look-and-feel of a single tool**
- **VeriSol is a key feature of NEC's C-based design flow**
  - **Provides verification of RTL designs**

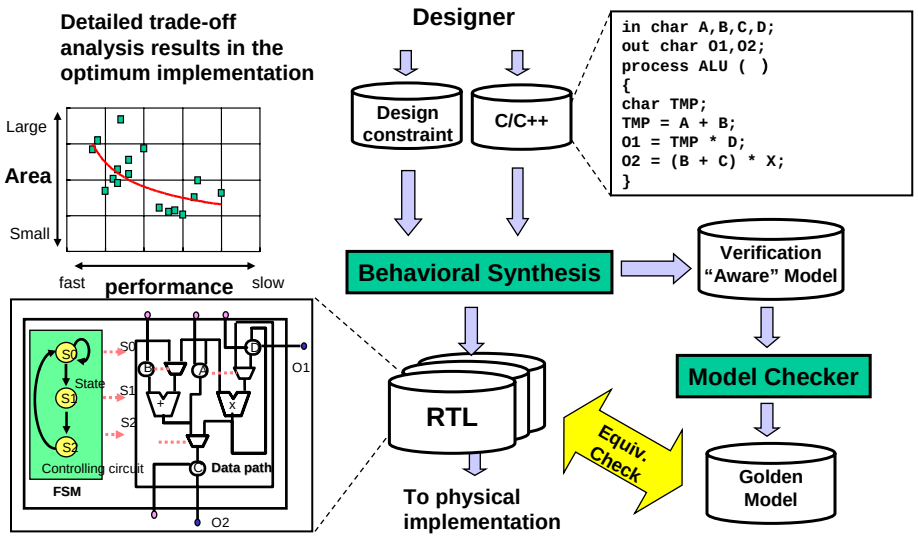
# Case Study

**FiFo: Address Width=7 Data Width=7**
**Design: Data-in <= wptr**
**Property 1: F(full_flag);// FIFO_LENGTH=24**
**Property 2: F(RAM[rptr]+50 == wptr);// FIFO_LENGTH=128**

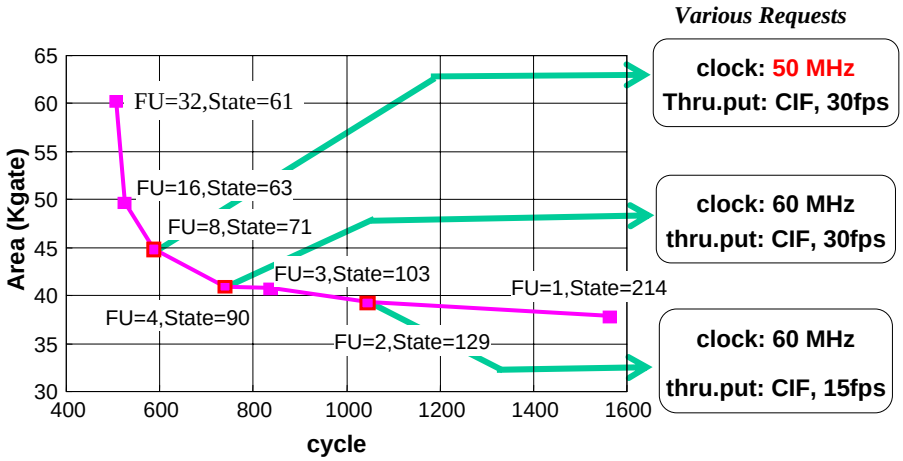
| C-step (1 RST) | #FU | MEM/REG/ SLICE | PRP id | #MUX | #REG | Verification |
|---|---|---|---|---|---|---|
| 1+1R | 2+2 | REG | 1 | 891 | 1165 | 243s, D=25 |
| 1+1R | 2+2 | SLICE | 1 | 93 | 109 | 135s, D=25 |
| 3+1R | 2 | REG | 2 | 4398 | 5748 | 143s, D=150 |
| 3+1R | 5 | REG | 2 | 4341 | 5748 | 52s, D=150 |
| 1+1R | 3+2 | REG | 2 | 4314 | 5741 | 64s, D=52 |
| 3+1R | 2 | MEM:R1,W1 | 2 | 177 | 116 | 33s, D=150 EMM |
| 3+1R | 5 | MEM: R1,W1 | 2 | 126 | 116 | 30s, D=150 EMM |
| 1+1R | 3+2 | MEM:R2,W1 | 2 | 93 | 109 | 10s, D=51 EMM |



# Paradigm Shift: Synthesis for Verification



# Various RTLs can be checked against Golden Reference Model (GRM)




Various circuits for IDCT: #. of REGs, FUs, States



# Synthesis for Verification "aware" Model

- No Re-use of FUs, Regs
- Minimize the use of muxes, sharing
- Reduce control steps
- Speculative execution/ Parallelization: No re-use of FUs
- Pipelines: avoid
- Select "verification friendly" FUs
- Operation reuse
- Slice statements using data flow analysis as source code optimization
- Support "assume" and "assert" in the language
- Use external memories instead of register arrays to take advantage of EMM modeling
- …..(open discussion forum)

# Thank you!

# Applications of the **DE2** Language

Warren A. Hunt, Jr. and Erik Reeber

February 16, 2006

**Abstract**

We have developed a formal verification approach that permits the mechanical verification of circuit generators and hardware optimization procedures, as well as existing hardware designs. Our approach is based on deeply embedding the **DE2** HDL into the ACL2 logic [3]; we use the ACL2 theorem-proving system to verify the circuit generators. During circuit generation, a circuit generator may generate circuits based on variety of non-functional criteria. For example, a circuit generator may produce different structural circuit descriptions depending on wire lengths, circuit primitives, target technology, and circuit topology.

In this paper, we show how we have applied the **DE2** system to a simple circuit generator—the n-bit ripple-carry adder. We then show how we have applied the **DE2** system to the verification of components of the TRIPS microprocessor design.

## 1 Introduction

We have developed a hardware description language, **DE2**, which has a number of features that make it suitable for the verification of modern hardware designs. **DE2** has a simple semantics and includes capabilities for specifying and verifying non-functional properties, circuit generators, and hardware optimization programs.

Our verification system is based on the deep embedding of **DE2** within the ACL2 logic and theorem prover. Furthermore, we have built a fully automatic SAT-based proof engine that can verify invariants of machines

designed in **DE2**. This SAT-based proof engine involves an extension to the ACL2 theorem-proving system so that it can use external SAT solvers.

In this paper, we discuss related work in Section 2. We provide some background on the ACL2 theorem prover, the **DE2** language, and our verification system, in Section 3. Next, in Section 4, we show how to apply our system to the verification of a ripple-carry adder. In Section 5, we show how we apply our system to the verification of a communication protocol used in the TRIPS processor.

# 2 Related Work

This work builds on our previous work with the **DE2** language [3], as well as our previous work with the verification of the FM9001 microprocessor [8]. In our earlier work, we only employed theorem-proving techniques, but our current effort also permits the use of SAT and BDD based techniques. In addition, our current approach to verifying circuit generators permits a circuit generator to make choices based on non-functional criteria. For example, a circuit generator may produce different structural circuit descriptions depending on wire lengths, circuit primitives, target technology, and circuit topology.

This work is similar in spirit to work by the functional language community to generate regular circuits using functional programs. For instance, the WIRED language has been used to improve performance of multipliers by incorporating layout information into the design of circuit generators [1].

Many model-checkers, and other automated verification tools, verify FSM properties automatically. UCLID, for example, uses SAT solvers to verify high-level FSMs with uninterpreted function symbols [5]. Another example is the FORTE tool, which has been used at Intel to verify components of processor designs [2].

# 3 Background

## 3.1 The ACL2 Theorem Prover

ACL2 stands for A Computation Logic for Applicative Common Lisp. The ACL2 language is a functional subset of Common Lisp. For a thorough description of ACL2 see Kaufmann, Manolios, and Moore's book [4].

```
(defun concatn (n a b)
  (if (zp n)
      b
    (cons (car a)
          (concatn (- n 1) (cdr a) b))))

(defun uandn (n a)
  (if (zp n)
      t
    (if (car a)
        (uandn (- n 1) (cdr a))
      nil)))

(defun bequiv (a b)
  (if a b (not b)))

(defthm example-thm
  (implies (and (not (zp x))
                (not (zp y)))
           (bequiv (uandn (+ x y) (concatn x a b))
                   (and (uandn x a) (uandn y b)))))
```

Figure 1: ACL2 Definitions and a Bit-Vector Concatenation Theorem

Figure 1 illustrates several ACL2 definitions. Here, function `concatn` concatenates two bit vectors, `uandn` returns the conjunction of the bits in a bit vector. The ACL2 function `bequiv` determines whether two ACL2 values represent the same Boolean value. We also make use of the built-in ACL2 function `(zp n)`, which returns `nil` if `n` is a positive integer and `t` otherwise.

The functions `uandn` and `concatn` are defined recursively. In order for such definitional axioms to be added to the ACL2 theory, one must first prove that the definition terminates for all inputs. In this case, the proof follows from the fact that the function argument `n` decreases on every recursive call.

Figure 1 also illustrates an ACL2 theorem. This theorem states that the `unary-and` of the concatenation of two bit vectors is equivalent to the conjunction of the `unary-and` of each individual bit vector.

## 3.2 The DE2 Evaluator

The semantic evaluation of a **DE2** design proceeds by binding actual (evaluated) parameters (both the inputs and the current state) to the formal parameters of the module to be evaluated; this in turn causes the evaluation of each submodule. This evaluation process is recursively repeated until a primitive module is encountered. This recursive-descent/ascent part of the evaluation can be thought of as performing all of the "wiring"; values are "routed" to appropriate modules and results are collected and passed along to other modules or become primary outputs. Finally, to evaluate a primitive, a specific primitive evaluator is then called after binding the necessary arguments. This set of definitions is composed of four (two groups of) functions (given below), and these functions contain an argument that permits different primitive evaluators to be used.

The following four functions completely define the evaluation of a netlist of modules, no matter which type of primitive evaluation is specified. The functions presented in this section constitute the entire definition of the simulator for the **DE2** language. This definition is small enough to allow us to reason with it mechanically, yet it is rich enough to permit the definition of a variety of evaluators. The `se` function evaluates a module and returns its outputs as a function of its inputs and its internal state. The `de` function evaluates a module and returns its next state; this state will be structurally identical to the module's current state, but with updated values. Both `se` and `de` have sibling functions, `se-occ` and `de-occ` respectively, that iterate through each sub-module referenced in the body of a module definition. We

present the `se` and `de` evaluator functions to make clear the importance we place on making the definition compact.

The `se` and `de` functions both have a `flg` argument that permits the selection of a specific primitive evaluator. The `fn` argument identifies the name of a module to evaluate; its definition should be found in the `netlist`. The `ins` and `st` arguments provide the primary inputs and the current state of the `fn` module. The `params` argument allows for parametrized modules; that is, it is possible to define modules with wire and state sizes that are determined by this parameter. The `env` argument permits configuration or test information to be passed deep into the evaluation process.

The `se-occ` function evaluates each occurrence and returns an environment that includes values for all internal signals. The `se` function returns a list of outputs by filtering the desired outputs from this environment. To compute the outputs as functions of the inputs, only a single pass is required.

```
(defun se (flg fn params ins st env netlist)
  (if (consp fn)
      ;; Primitive Evaluation.
      (cdr (flg-eval-lambda-expr flg fn params ins env))
    ;; Evaluate submodules.
    (let ((module (assoc-eq fn netlist)))
      (if (atom module)
          nil
        (let-names
         (m-params m-ins m-outs m-sts m-occs)
         (m-body module)
         (let*
           ((new-env     (add-pairlist m-params params nil))
            (new-env     (add-pairlist (strip-cars m-ins)
                                       (flg-eval-list flg ins env)
                                       new-env))
            (new-env     (add-pairlist m-sts
                                       (flg-eval-expr flg st env)
                                       new-env))
            (new-netlist (delete-assoc-eq-netlist fn netlist)))
           (assoc-eq-list-vals
            (strip-cars m-outs)
            (se-occ flg m-occs new-env new-netlist)))))))))

(defun se-occ (flg occs env netlist)
```

```
(if (atom occs)  ;; Any more occurrences?
    env
  ;; Evaluate specific occurrence.
  (let-names
   (o-name o-outs o-call o-ins)
   (car occs)
   (se-occ flg (cdr occs)
           (add-pairlist
            (o-outs-names o-outs)
            (flg-eval-list
             flg (parse-output-list
                  o-outs
                  (se flg (o-call-fn o-call)
                      (flg-eval-list flg
                                     (o-call-params o-call)
                                     env)
                      o-ins o-name env netlist))
             env)
            env)
           netlist))))
```

Similarly, the functions `de` and `de-occ` perform the next-state computation for a module's evaluation; given values for the primary inputs and a structured state argument, these two functions compute the next state of a specified module. This result state is structured isomorphically to its input (internal) state. Note that the definition of `de` contains a reference to the function `se-occ`; this reference computes the value of all internal signals for the module whose next state is being computed. This call to `se-occ` represents the first of two passes through a module description when `DE` is computing the next state.

```
(defun de (flg fn params ins st env netlist)
  (if (consp fn)
      (car (flg-eval-lambda-expr flg fn params ins env))
    (let ((module (assoc-eq fn netlist)))
      (if (atom module)
          nil
        (let-names
         (m-params m-ins m-sts m-occs) (m-body module)
         (let*
           ((new-env     (add-pairlist m-params params nil))
```

```
            (new-env     (add-pairlist (strip-cars m-ins)
                                       (flg-eval-list flg ins env)
                                       new-env))
            (new-env     (add-pairlist m-sts
                                       (flg-eval-expr flg st env)
                                       new-env))
            (new-netlist (delete-assoc-eq-netlist fn netlist))
            (new-env     (se-occ flg m-occs new-env new-netlist)))
           (assoc-eq-list-vals
            m-sts
            (de-occ flg m-occs new-env new-netlist))))))))

(defun de-occ (flg occs env netlist)
  (if (atom occs)
      env
    (let-names
     (o-name o-call o-ins) (car occs)
     (de-occ flg (cdr occs)
             (cons
              (cons
               o-name
               (de flg (o-call-fn o-call)
                   (flg-eval-list flg (o-call-params o-call) env)
                   o-ins o-name env netlist))
              env)
             netlist))))
```

This completes the entire definition of the **DE2** evaluation semantics. This clique of functions is used for all different evaluators; the specific kind of evaluation is determined by the `flg` input. We have proved a number of lemmas that help to automate the analysis of **DE2** modules. These lemmas allow us to hierarchically verify FSMs represented as **DE2** modules. We have also defined simple functions that use `de` and `se` to simulate a **DE2** design through any number of cycles.

An important aspect of this semantics is its brevity. Furthermore, since we specify our semantics in the formal language of the ACL2 theorem prover, we can mechanically and hierarchically verify properties about any system defined using the **DE2** language.

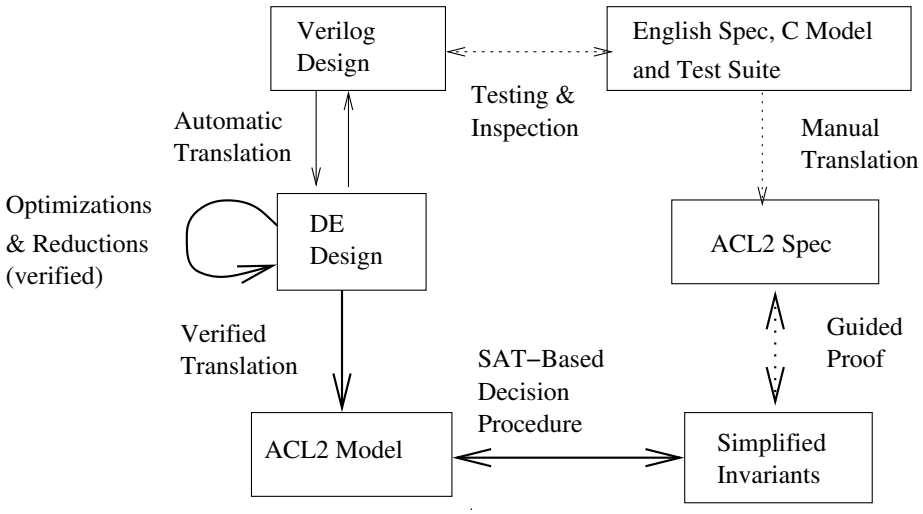



Figure 2: An overview of the **DE2** verification system

## 3.3 The Verification System

Having an evaluator for **DE2** written in ACL2 enables many forms of verification. In Figure 2, we illustrate our verification system, which is built around the **DE2** language.

We typically use the **DE2** verification system to verify Verilog designs. These designs are denoted in the upper left of Figure 2. Currently, our subset of Verilog includes arrays of wires (bit vectors), instantiations of modules, assignment statements, and some basic primitives (e.g. `&`, `?:` and `|`). We also allow the instantiation of memory (array) modules and vendor-defined primitives.

We have built a translator that translates a Verilog description into an equivalent **DE2** description. Our translator parses the Verilog source text into a Lisp expression, and then an ACL2 program converts this Lisp expression into a **DE2** description.

We have also built a translator that converts a **DE2** netlist into a cycle-accurate ACL2 model. This translator also provides an ACL2 proof that the **DE2** description is equivalent to the mechanical produced ACL2 model. The process of translating a **DE2** description into its corresponding ACL2 model includes a partial cone-of-influence reduction; an ACL2 function is created for each module's output and parts of the initial design which are irrelevant to that output are removed. The **DE2** to ACL2 translator allows

us to enjoy both the advantages of a shallow embedding (e.g. straightforward verification) and the advantages of a deep embedding (e.g. syntax resembling Verilog).

We start with an informal specification of the design in the form of English documents, charts, graphs, C-models, and test code which is represented in the upper right of Figure 2. This information is converted manually into a formal ACL2 specification. Using the ACL2 theorem prover, these specifications are simplified into a number of invariants and equivalence properties. If these properties are simple enough to be proven by our SAT-based decision procedure, we prove them automatically; otherwise, we simplify such conjectures using the ACL2 theorem prover until we can successfully appeal to some automated decision procedure.

We also use our system to verify sets of **DE2** descriptions. This is accomplished by writing ACL2 functions that generate **DE2** descriptions, and then proving that these functions always produce circuits that satisfy their ACL2 specifications.

Since **DE2** descriptions are represented as ACL2 constants, functions that transform **DE2** descriptions can be verified using the ACL2 theorem prover. By converting from Verilog to **DE2** and from **DE2** to back into Verilog, we can use **DE2** as an intermediate language to perform verified optimizations. Another use of this feature involves performing reductions or optimizations on **DE2** specifications prior to verification. For example, one can use a decision procedure to determine that two **DE2** circuits are equivalent and then use this fact to avoid verifying properties of a less cleanly structured description.

We can also build static analysis tools, such as extended type checkers, in **DE2** by using annotations. In **DE2**, annotations are first-class objects (i.e. annotations are not embedded in comments). Such type checkers, since they are written in ACL2, can be analyzed and can also assist in the verification of **DE2** descriptions. Furthermore, annotations can be used to embed information into a **DE2** description to assist with synthesis or other post-processing tools.

# 4 Ripple-Carry Adder Generator Verification

In this section we present a definition of a simple parametrized ripple-carry adder to show how the **DE2** verification system is applied to verify circuit

generators. The following two ACL2 functions generate the **DE2** definition of the top-level module of the ripple-carry adder:

```
(defun generate-ripple-occs (n)
  (if (zp n)
      nil
    (append (generate-ripple-occs (1- n))
            `((,(de-make-n-name 'carry n)
               ((q ,(1- n) ,(1- n)) (carry ,n ,n))
               (full-adder)
               ((g x ,(1- n) ,(1- n)) (g y ,(1- n) ,(1- n))
                (g carry ,(1- n) ,(1- n))))))))

;; Make an n-bit ripple-carry adder
(defun generate-ripple-carry (n)
  `(,(de-make-n-name 'ripple-carry n)
    (type module)
    (params )
    (outs (q ,n) (c_out 1))
    (ins (x ,n) (y ,n) (c_in 1))
    (sts )
    (wires (carry ,(1+ n)))
    (occs
     (carry_0 ((carry 0 0)) (bufn 1) ((g c_in 0 0)))
     . ,(append (generate-ripple-occs n)
                `((carry_out ((c_out 0 0))
                             (bufn 1)
                             ((g carry ,n ,n))))))))
```

The function `generate-ripple-occs` creates the occurrence list by recursively laying down one full-adder for each output bit. The function `generate-ripple-carry` then uses this occurrence list to create the top-level ripple-carry adder definition. For example, the following is the four bit ripple-carry adder produced by `(generate-ripple-carry 4)`:

```
(RIPPLE-CARRY_4
 (TYPE MODULE)
 (PARAMS)
 (OUTS (Q 4) (C_OUT 1))
 (INS (X 4) (Y 4) (C_IN 1))
```

```
(STS)
(WIRES (CARRY 5))
(OCCS (CARRY_0 ((CARRY 0 0))
               (BUFN 1)
               ((G C_IN 0 0)))
      (CARRY_1 ((Q 0 0) (CARRY 1 1))
               (FULL-ADDER)
               ((G X 0 0) (G Y 0 0) (G CARRY 0 0)))
      (CARRY_2 ((Q 1 1) (CARRY 2 2))
               (FULL-ADDER)
               ((G X 1 1) (G Y 1 1) (G CARRY 1 1)))
      (CARRY_3 ((Q 2 2) (CARRY 3 3))
               (FULL-ADDER)
               ((G X 2 2) (G Y 2 2) (G CARRY 2 2)))
      (CARRY_4 ((Q 3 3) (CARRY 4 4))
               (FULL-ADDER)
               ((G X 3 3) (G Y 3 3) (G CARRY 3 3)))
      (CARRY_OUT ((C_OUT 0 0))
                 (BUFN 1)
                 ((G CARRY 4 4)))))
```

We next define a ripple-carry adder in ACL2 which follows the same structure as the one defined in **DE2**. The following is the top-level definition of the ACL2 ripple-carry adder and the main theorem we prove about it:

```
(defun acl2-ripple-adder (n x y c_in)
  (if (zp n)
      (list nil (get-sublist c_in 0 0))
    (let* ((adder_1b
            (acl2-full-adder (get-sublist x 0 0)
                             (get-sublist y 0 0)
                             (get-sublist c_in 0 0)))
           (sub_adder (acl2-ripple-adder (1- n)
                                         (nth-cdr 1 x)
                                         (nth-cdr 1 y)
                                         (cadr adder_1b))))

      (list (append-n 1 (car adder_1b) (car sub_adder))
            (append-n 1 c_in (cadr sub_adder))))))
```

```
(defthm acl2-ripple-adder-adds
  (implies
   (and (equal n (len a))
        (equal (len b) (len a)))
   (equal (v-to-nat
           (car (acl2-ripple-adder n a b
                                   (list (bool-fix c_in)))))
          (mod-2-n (+ (if c_in 1 0)
                      (v-to-nat a)
                      (v-to-nat b))
                   n))))
```

The above theorem states that the ACL2 functional definition of the ripple-carry adder implements modular addition, as defined by ACL2's addition axioms. We prove this theorem by making use of ACL2's induction and simplification proof engines, as well as the library of lemmas that has been created to assist ACL2 users in the verification of arithmetic properties.

Next we verify the theorem below:

```
(defthm generate-ripple-se-rewrite
  (implies
   (and (not (zp n))
        (generate-ripple-carry-& n netlist))
   (equal
    (se 'bvev
        (de-make-n-name 'ripple-carry n)
        params ins st env netlist)
    (let ((x (get-value 'bvev ins env))
          (y (get-value 'bvev (cdr ins) env))
          (c_in (get-sublist (get-value 'bvev
                                        (cddr ins)
                                        env)
                             0
                             0)))
      (list (car (acl2-ripple-adder n x y c_in))
            (get-sublist (cadr (acl2-ripple-adder n
                                                  x
                                                  y
                                                  c_in))
                         n
                         n))))))
```

This theorem states that, given certain conditions, the **DE2** ripple-carry adder produces the same result as the ACL2 ripple-carry adder. The hypotheses of the theorem are that the number of bits is a positive integer and that the ripple-carry adder modules occurs in the given netlist, along with its submodules. This theorem is proven using ACL2's induction proof engine, which we use to show that each occurrence produced by a recursive step of `generate-ripple-occs` corresponds to a recursive step in `acl2-ripple-adder`.

Once we have verified `generate-ripple-se-rewrite`, we can prove the final theorem below:

```
(defthm generate-ripple-se-adds
  (implies
   (and (not (zp n))
        (generate-ripple-carry-& n netlist)
        (equal (len (get-value 'bvev ins env)) n)
        (equal (len (get-value 'bvev (cdr ins) env)) n))
   (equal
    (v-to-nat (car (se 'bvev
                       (de-make-n-name 'ripple-carry n)
                       params ins st env netlist)))
    (let ((x (get-value 'bvev ins env))
          (y (get-value 'bvev (cdr ins) env))
          (c_in (get-sublist (get-value 'bvev (cddr ins) env)
                             0
                             0)))
      (mod-2-n (+ (if (car c_in) 1 0)
                  (v-to-nat x)
                  (v-to-nat y))
               n)))))
```

This theorem states that if the `n`-bit, ripple-carry adder module is in the netlist, along with its submodules, and the first two inputs are `n` bit, bit vectors, then the natural number representation of the output of the ripple-carry adder is equal to the modular addition of its inputs.

Note we proved this theorem entirely using the standard ACL2 theorem proving techniques, without the use of SAT solvers or BDDs. That is because we completed this proof before our SAT-based proof engine was fully in place. In the next section we will show how we are verifying next-generation hardware using a mixture of SAT-solving and theorem proving.

# 5 Verifying TRIPS Processor Components

We are using our verification system to verify components of the TRIPS processor. The TRIPS processor is a prototype of a next-generation processor that has been designed at the University of Texas [7] and being built by IBM. One novel aspect of the TRIPS processor is that its memory is divided into four pieces; each piece has its own memory control tile, with its own cache and Load Store Queue (LSQ). We plan to verify the LSQ design, based on the design described in Sethumadhavan et. al., [6], using our verification system. In this section, we present our verification of a part of the LSQ that manages communication with other LSQs.

We first use our verification system, mentioned in Section 3.3, to "compile" the Verilog design that implements the LSQ communication protocol into a **DE2** module. We then used our automatic translation engine to compile the **DE2** description into an ACL2 model and prove their equivalence relative to the **DE2** semantics.

## 5.1 Verification of the Exception Protocol

One reason that the LSQ units must communicate is to conglomerate exceptions generated in various tiles into a single mask. Figure 3 presents an overview of the protocol that conglomerates exceptions. Each tile receives a four-bit input denoting the exception generated this cycle—a three-bit address plus a one-bit enable signal. The exceptions are decoded into an eight-bit mask, that each tile passes to the tile above it. Exceptions are removed when the instruction that generated the exception is flushed. The schematic of the design that implements this protocol is shown in Figure 4.

To verify the multi-tile design in Figure 3, we prove that it is equivalent to the single-tile design in Figure 5. This equivalence is broken into the following two properties:

```
(defthm exception-safety
 (implies
  (and (integerp tao)
       (<= 0 tao)
       (Tth-inputs-goodp tao input-list))
  (submaskp
   8
   (out-udt_miss_ordering_exceptions
```

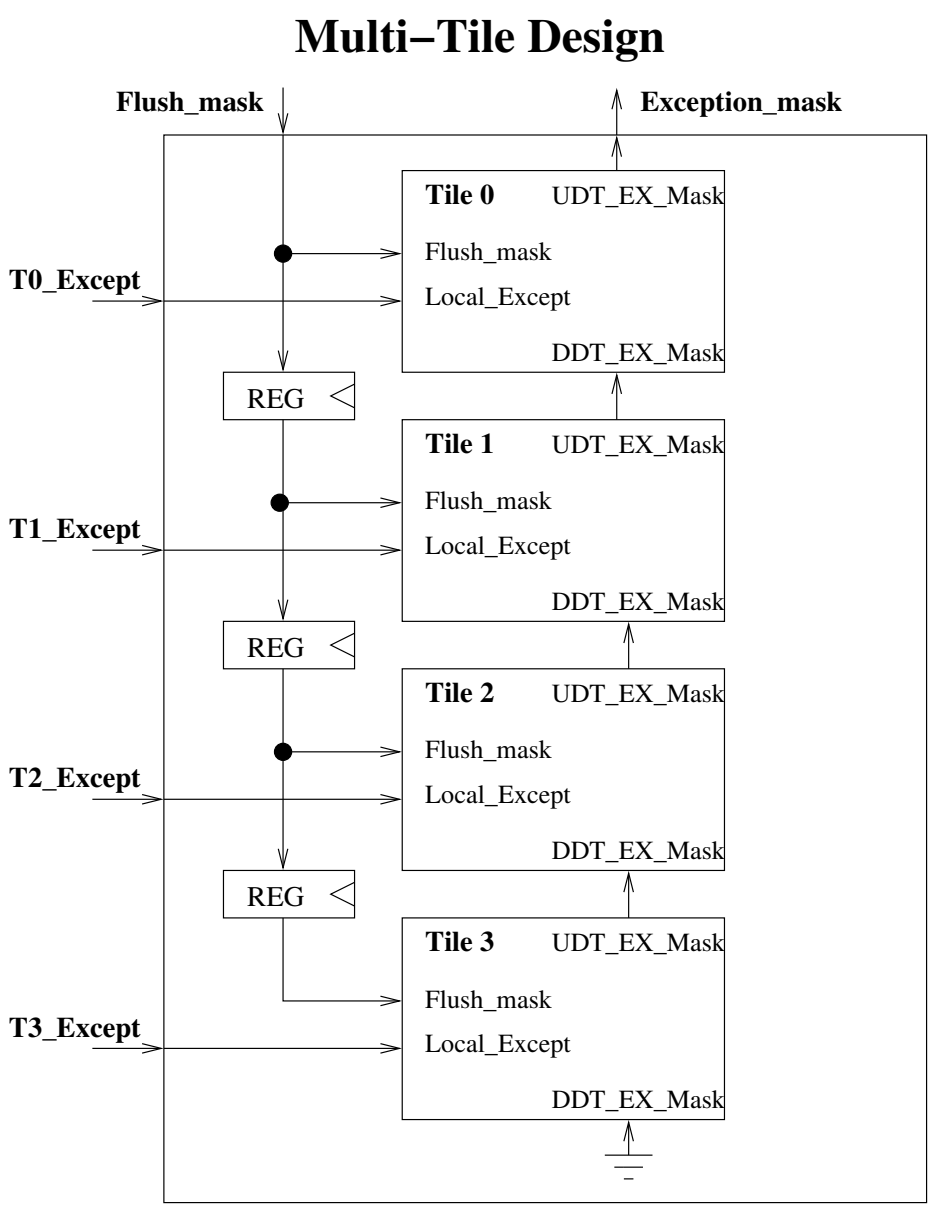



Figure 3: An overview of the four tile exception protocol design.

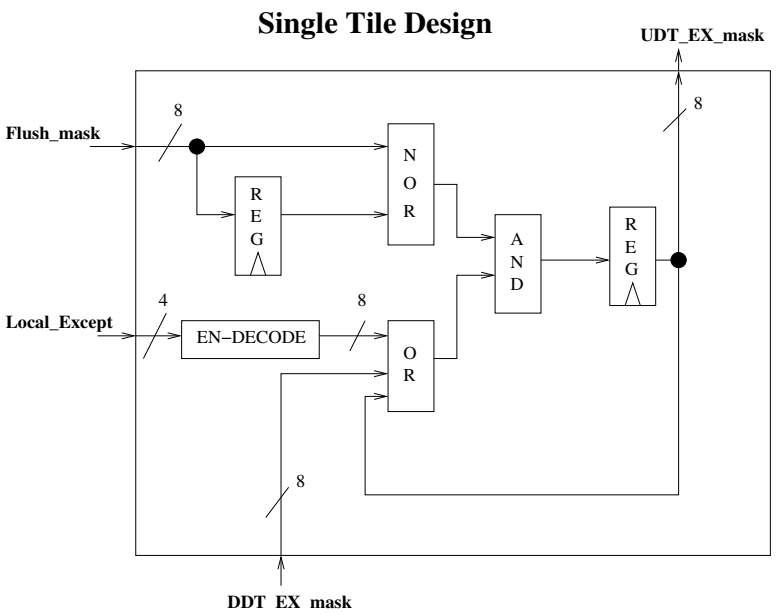



Figure 4: A look into the internals of a tile within the exception protocol.

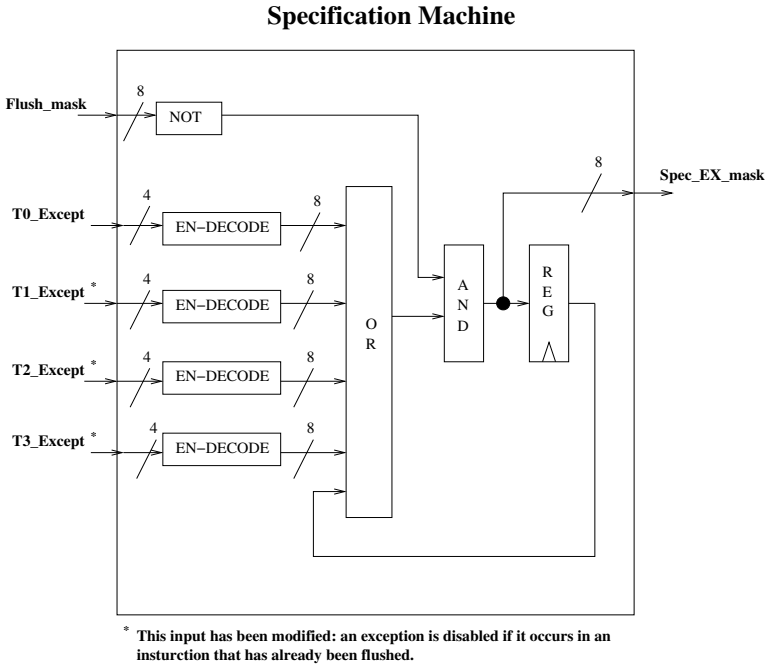



Figure 5: A simplified machine that produces the exception mask.

```
    *t0*
    (Tth-internal-state tao input-list)
    (nth tao input-list))
   (spec-miss_ordering
    (Tth-spec-state tao input-list)
    (nth tao input-list)))))

(defthm exception-liveness
 (implies
  (and (integerp tao)
       (<= 3 tao)
       (Tth-inputs-goodp tao input-list))
  (submaskp
   8
   (bv-or
    8
    (recent-flushes 3 tao *t0* input-list)
    (spec-miss_ordering
     (Tth-spec-state (- tao 3) input-list)
     (nth (- tao 3) input-list)))
   (out-udt_miss_ordering_exceptions
    *t0*
    (Tth-internal-state tao input-list)
    (nth tao input-list)))))
```

The first property proves that, for any cycle number `tao`, assuming good inputs, the exception mask generated by tile zero is a subset of the exception mask generated by the single-tile machine. The second property proves that the exception mask generated by the single tile machine is a subset of the combination of the exception mask generated by tile zero and the last three flush masks. In effect, these properties prove that our multi-tile exception design only produces exceptions produced by the specification and eventually produces all exceptions produced by the specification.

We prove these properties by reducing them to the proof of an invariant; we prove these invariants through a mixture of theorem proving and SAT solving. The following example illustrates the type of lemma that we prove with SAT. This lemma is proven by telling ACL2 to automatically call the SAT-based proof engine once its simplification rules reach a fix point.

```
(defthm sub-of-spec-mask-t0
```

```
(implies
 (and
  (equiv-bvp
   8
   (in-ddt_miss_ordering_exceptions *t0* ins)
   (internal-st-udt_miss_ordering *t1* internal-state))
  (equiv-bvp
   8
   (in-flush_mask *t0* ins)
   (internal-st-flush_mask *t1* internal-state))
  (sub-of-spec-mask-tile *t0* spec-st internal-state)
  (sub-of-spec-mask-tile *t1* spec-st internal-state))
 (sub-of-spec-mask-tile
  *t0*
  (update-spec-st spec-st internal-state ins)
  (update-internal-state internal-state ins))))
```

## 5.2 Verification of an Arrived-Store Protocol

The LSQ units also communicate to create a mask of arrived stores; these are used to generate exceptions, wake deferred loads, and detect completion. Figure6 presents an overview of the arrived-store-mask protocol. This protocol is more complex than the exception protocol, because tiles send information to both the tile above and the tile below them. Also, since the arrived store mask is 256 bits, the whole mask is never sent. Instead up to three, nine-bit store signals are sent to each neighboring tile, informing each neighbor of all the new stores it has received in the last cycle.

We used the same methodology to verify the arrived-store-mask protocol as we used to verify the exception-mask protocol. We first define a single-tile design that produces the store mask. This design is shown in Figure 7. Next, we prove the equivalence of the single-tile and multi-tile designs using the following two theorems. Note that these theorems prove an equivalence over all tiles, whereas the exception mask equivalence only dealt with tile zero.

```
(defthm arrived-safety
 (implies
  (and (integerp tao)
       (<= 0 tao)
       (Tth-inputs-goodp tao input-list))
```

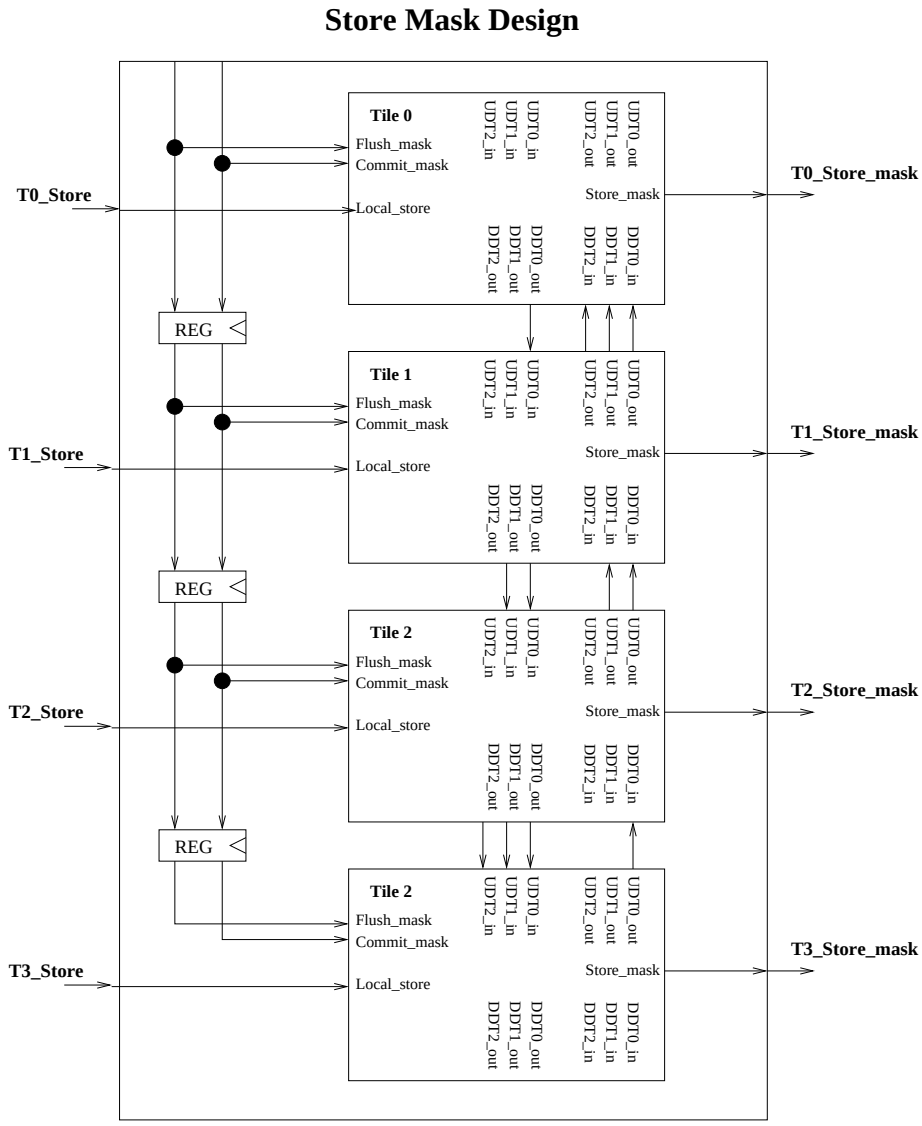



Figure 6: An overview of the protocol for generating the mask of arrived stores. Note that the tile inputs that are unconnected are either grounded or known to always be low.

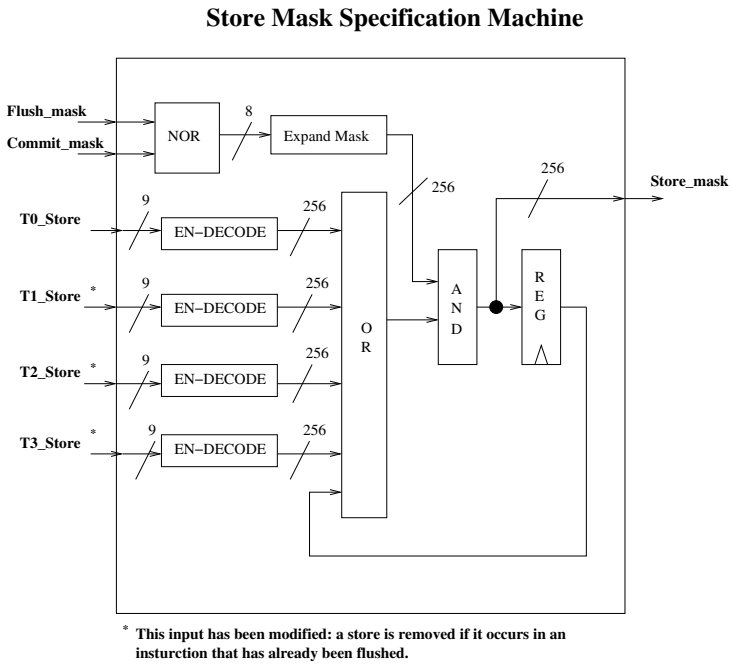



Figure 7: A simplified machine that produces the mask of arrived stores.

```
 (submaskp
  8
  (out-arrived_mask
   tile
   (Tth-internal-state tao input-list)
   (nth (- tao 3) input-list))
  (spec-arrived_mask
   (Tth-spec-state tao input-list)
   (nth tao input-list)))))

(defthm arrived-liveness
 (implies
  (and (integerp tao)
       (<= 3 tao)
       (Tth-inputs-goodp tao input-list))
  (submaskp
   8
   (bv-or
```

```
 8
 (expand-mask 8 256 (recent-flushes 3 tao tile input-list))
 (bv-or
  8
  (expand-mask 8 256 (recent-commits 3 tao tile input-list))
  (spec-arrived_mask
   (Tth-spec-state (- tao 3) input-list)
   (nth (- tao 3) input-list))))
(out-arrived_mask
 tile
 (Tth-internal-state tao input-list)
 (nth tao input-list)))))
```

# 6 Conclusion

The verification of an automatically generated circuit description usually involves verifying the netlist post-synthesis. Through our ripple-carry adder example, we have shown how we can verify the correctness of the circuit generators directly, thus obviating the need to verify the resultant circuit descriptions.

To aid our verification effort, we have combined the complementary techniques of theorem proving and SAT solving. We show the usefulness of this combination through the verification of a Verilog implementation of a communication protocol used in the TRIPS processor.

An extension of our approach is to show how circuit generators can be used within the verification of the TRIPS processor. Rather than partition memory into four pieces, one could design a TRIPS processor with memory partitioned into a parametrized number of pieces. This type of verification fits well into the modular nature of the TRIPS processor design and showcases the advantages of the **DE2** language. Furthermore, this verification effort will allow us to explore the applications and limitations of fully automated verification techniques, like SAT, when used to verify large circuit generation designs.

Moving beyond circuit generators, there are many other potential applications for the **DE2** verification system. For example, we can use the **DE2** language to verify hardware optimization programs and non-functional properties. The flexibility of the **DE2** language and the ACL2 theorem proving system provides the opportunity to verify many types of applications, many

of which are rarely, if ever, been formally verified.

# References


[1] Emil Axelsson, Koen Claessen, and Mary Sheeran. Wired: Wire-Aware Circuit Design. In *Correct Hardware Design and Verification Methods (CHARME 2005)*, volume 3725 of *Lecture Notes in Computer Science*, pages 5–19. Springer, 2005.

[2] Robert B. Jones, John W. O'Leary, Carl-Johan H. Seger, Mark Aagaard, and Thomas F. Melham. Practical Formal Verification in Microprocessor Design. *IEEE Design & Test of Computers*, 18(4):16–25, 2001.

[3] Warren A. Hunt Jr. and Erik Reeber. Formalization of the DE2 Language. In *Correct Hardware Design and Verification Methods (CHARME 2005)*, volume 3725 of *Lecture Notes in Computer Science*, pages 20–34. Springer, 2005.

[4] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. *Computer Aided Reasoning: An Approach*. Kluwer Academic, 2000.

[5] Shuvendu K. Lahiri and Randal E. Bryant. Deductive verification of advanced out-of-order microprocessors. In *Computer Aided Verification, 15th International Conference (CAV 2003)*, volume 2725 of *Lecture Notes in Computer Science*, pages 341–353. Springer, 2003.

[6] Simha Sethumadhavan, Rajagopalan Desikan, Doug Burger, Charles R. Moore, and Stephen W. Keckler. Scalable hardware memory disambiguation for high ilp processors. In *Proceedings of the 36th Annual International Symposium on Microarchitecture (MICRO 36)*, pages 399–410. ACM/IEEE, 2003.

[7] Tera-op Reliable Intelligently adaptive Processing System, www.cs.utexas.edu/users/cart/trips.

[8] Warren A. Hunt, Jr. and Bishop C. Brock. A Formal HDL and its Use in the FM9001 Verification. In *Mechanized Reasoning and Hardware Design*, pages 35–47, Upper Saddle River, NJ, USA, 1992. Prentice-Hall, Inc.

# *Evolution and Impact of a Large Industrial Proof*

Robert B. Jones
Strategic CAD Labs
Intel Corporation
Hillsboro, OR, USA
robert.b.jones@intel.com

Noppanunt Utamaphethai
Low Power Technologies Group
Intel Corporation
Austin, Texas, USA
noppanunt.utamaphethai@intel.com

The Intel® IA-32 instruction-set architecture includes several hundred opcodes of varying length [3]. Certain instructions have optional bytes that specify register modes, memory modes, and address offsets. Instructions vary in length from one to twelve bytes. An additional complication arises from *prefix bytes* that can change the semantics and even length of the subsequent instruction.

Decoding the IA-32 instruction-set in a high-frequency pipeline is challenging. Recent processor implementations divide the decoding process into separate activities; the first is an *instruction-length decoder* (ILD) that marks instruction boundaries.

This talk at DCC 2006 will overview the evolution and impact of a formal proof about the ILD. The proof has evolved as it has been applied to multiple microprocessor designs over almost a decade. The proof has detected bugs in almost every design it has been applied to, and has largely replaced simulation-based validation of ILD functionality on some projects. This talk will consider the evolution of the proof in the face of hardware changes and additions to the instruction set. Technical details of an early version of the proof have been published previously [1, 2, 4].

We have learned several important lessons about specification and verification during the evolution of the ILD proof.

- Formal specifications should avoid implementation details when possible. Current ILD implementations are significantly different from the first ILD pipeline that was verified. Writing the formal specification to reason about the ILD as a "black box" has been an important aspect of applying it on multiple hardware designs.

- Proofs need to be amenable to variations in proof complexity induced by design changes. We found that certain hardware changes made the BDDs underlying the proof simpler. On the other hand, changes usually made the proof BDDs more complex. Managing this complexity was one of the main challenges as the proof evolved. With the benefit of hindsight, we can see techniques that would have made complexity management easier.

- Certain classes of specifications must be written in an extensible way. It was fortunate that the original ILD specification was extensible. In the years since the original specification was created, multiple features have been added to Intel microprocessors that required new instructions. More recently, the introduction of a 64-bit mode to the Intel architecture resulted in extensive additions to the instruction set–and to its formal specification.

The ILD proof has been very successful and a wide range of bugs have been found over the proof's lifetime. Some might have escaped simulation-based validation, and, as far as we know, the formal proof has not missed any bugs in its targeted area. We will highlight examples of bugs and their underlying causes. As the proof has matured, the ability to apply it early in the design process has proven particularly useful.

## References


[1] M. D. Aagaard, R. B. Jones, and C.-J. H. Seger. Combining theorem proving and trajectory evaluation in an industrial environment. In *Design Automation Conference (DAC)*, pages 538–541. ACM Press, June 1998.

[2] M. D. Aagaard, R. B. Jones, and C.-J. H. Seger. Formal verification using parametric representations of Boolean constraints. In *Design Automation Conference (DAC)*, pages 402–407. ACM Press, June 1999.

[3] *IA-32 Intel® Architecture Software Developer's Manual, Volumes 2A and 2B: Instruction Set Reference*. Intel Corporation, September 2005. Document numbers 253666 and 254667. Available at `http://www.intel.com`.

[4] R. B. Jones. *Symbolic Simulation Methods for Industrial Formal Verification*. Kluwer Academic Publishers, 2002.

# Synchronous Elastic Networks

Sava Krstić[1], Jordi Cortadella[2], Mike Kishinevsky[1], and John O'Leary[1]

[1] Strategic CAD Labs, Intel Corporation, Hillsboro, Oregon, USA
[2] Universitat Politècnica de Catalunya, Barcelona, Spain

**Abstract.** We formally define a class of networks obeying a protocol that tolerates any variability in the latency of the components. We study behavioral properties of these networks and prove fundamental compositionality results. The paper contributes to bridging the gap between the theory of latency-insensitive systems and the correct implementation of efficient control structures for such systems.

## 1 Introduction

The conventional abstract model for a synchronous circuit is a machine that reads inputs and writes outputs at every cycle. The outputs at cycle $i$ are produced according to a calculation that depends on the inputs at cycles $0, \ldots, i$. Computations and data transfers are assumed to take zero delay.

*Latency-insensitive design* [2] aims to relax this model by elasticizing the time dimension and so decoupling the cycles from the calculations of the circuit. It enables the design of circuits tolerant to any discrete variation (in the number of cycles) of the computation and communication delays. With this modular approach, the functionality of the system only depends on the functionality of its components and not on their timing characteristics.

The motivation for latency-insensitive design comes from the difficulties with timing and communication in nanoscale technologies. The number of cycles required to transmit data from a sender to a receiver is governed by the distance between them, and often cannot be accurately known until the chip layout is generated late in the design process. Traditional design approaches require fixing the communication latencies up front, and these are difficult to amend when layout information finally becomes available. Elastic circuits offer a solution to this problem. In addition, their modularity promises novel methods for microarchitectural design that can use variable-latency components and tolerate static and dynamic changes in communication latencies, while—unlike asynchronous circuits—still employing standard synchronous design tools and methods.

The recent paper [4] presents a simple elastic protocol, called SELF (Synchronous Elastic Flow) and describes methods for efficient implementation of elastic systems and for conversion of regular synchronous designs into elastic form. Inspired by the original work on the latency-insensitive design [2], SELF also differs from it in ways that render the theory developed in [2] hardly applicable.

In this paper we give theoretical foundations of SELF. The paper is self-contained, but for lack of space all proofs are omitted. The interested reader can find them in the accompanying technical report [7].

**Note:** This paper has been submitted to a conference. The extended version [7] contains an appendix with complete proofs and auxiliary material, including a brief overview of SELF.

### 1.1 Overview

Figure 1(a) depicts the timing behavior of a conventional synchronous adder that reads input and produces output data at every cycle (boxes represent cycles). In this adder, the $i$-th output value is produced at the $i$-th cycle. Figure 1(b) depicts a related behavior of an elastic adder—a synchronous circuit too—in which data transfer occurs in some cycles and not in others. We refer to the transferred data items as *tokens* and we say that idle cycles contain *bubbles*.

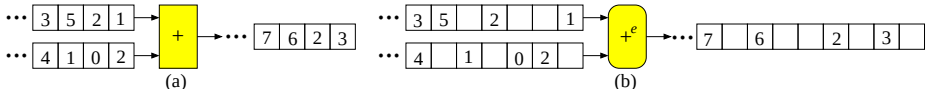



**Fig. 1.** (a) Conventional synchronous adder, (b) Synchronous elastic adder.

Put succinctly, elasticization decouples cycle count from token count. In a conventional synchronous circuit, the $i$-th token of a wire is transmitted at the $i$-th cycle, whereas in a synchronous elastic circuit the $i$-th token is transmitted at some cycle $k \geq i$.

Turning a conventional synchronous adder into a synchronous elastic adder requires a communication discipline that differentiates idle from non-idle cycles (bubbles from tokens). In SELF, this is implemented by a pair of single-bit control wires: *Valid* and *Stop*. Every input or output wire $X$ in a synchronous component is associated to a *channel* in the elastic version of the same component. The channel is a triple of wires $\langle X, \mathsf{valid}_X, \mathsf{stop}_X \rangle$, with $X$ carrying the data and the other two wires implementing the control bits, as shown in Figure 2(b). A token is transferred on this channel when $\mathsf{valid}_X = 1$ and $\mathsf{stop}_X = 0$: the sender sends valid data and the receiver is ready to accept it. Additional constraints that guarantee correct elastic behavior are given in Section 3. There we define precisely what it means for a circuit $A^{\mathrm{e}}$ to be an elastization of a given circuit $A$. In particular, our definition implies liveness: $A^{\mathrm{e}}$ produces infinite streams of tokens if its environment produces infinite streams of tokens at the input channels and is ready to accept infinite streams at the output channels.

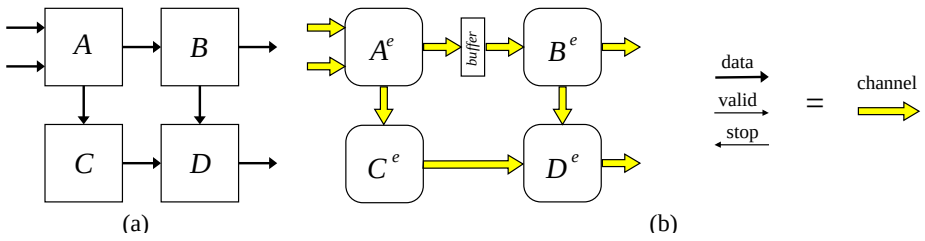



**Fig. 2.** A synchronous network (a) and its elastic counterpart (b).

Suppose $\mathcal{N}$ is a network of standard (non-elastic) components, as in Figure 2(a). Suppose we then take elasticizations of these standard components and join their channels accordingly, as in Figure 2(b), ignoring the buffer. Will the resulting network $\mathcal{N}^{\mathrm{e}}$ be an elasticization of $\mathcal{N}$? Will it be elastic at all? These fundamental questions are answered by Theorem 4 of Section 4, which is the main result of the paper. The answers are "yes", provided a certain graph $\Delta^{\mathrm{e}}(\mathcal{N}^{\mathrm{e}})$ associated with $\mathcal{N}^{\mathrm{e}}$ is acyclic. This graph captures the information about paths inside elastic systems that contain no tokens—analogous to combinational paths in ordinary systems. Importantly, $\Delta^{\mathrm{e}}(\mathcal{N}^{\mathrm{e}})$ can be constructed using only local information (the "sequentiality interfaces") of the individual elastic components.

Since elastic networks tolerate any variability in the latency of the components, empty FIFO buffers can be inserted in any channel, as shown in Figure 2(b), without changing the functional behavior of the network. This practically important fact is proved as a consequence of Theorem 4.

Synchronous circuits are modeled in this paper as stream transformers, called *machines*. This well-known technique (see [8] and references therein) appears to be quite underdeveloped. Our rather lengthy preliminary Section 2 elaborates the necessary theory of networks of machines, culminating with a surprisingly novel combinational loop therorem (Theorem 1).

Figure 3 illustrates Theorem 1 and, by analogy, Theorem 4 as well. It relies on the formalization of the notion of combinational dependence at the level of input-output wire pairs. Each input-output pair of a machine is either *sequential* or not, and the set of sequential pairs provides a machine's "sequentiality interface". When several machines are put together into a network $\mathcal{N}$, their sequentiality interfaces define the graph $\Delta(\mathcal{N})$, the acyclicity of which is a test for the network to be a legitimate machine itself.

Elasticizations of ordinary circuits are not uniquely defined. On the other hand, for every elastic machine $A$ there is a unique standard machine, denoted $A^{\intercal}$, that corresponds to it. We do not discuss any specific elasticization procedures in this paper, but state our results in the form that only involves elastic machines and their unique standard counterparts. This makes the results potentially applicable to multiple elasticization procedures.

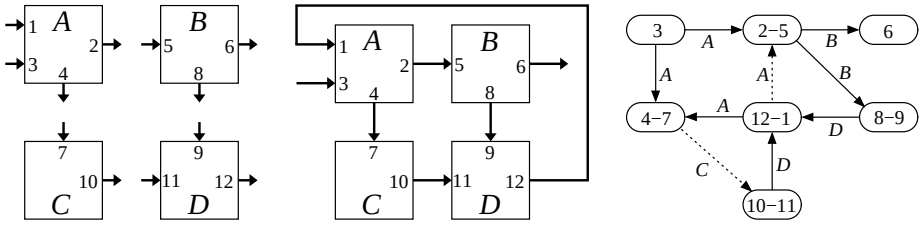



**Fig. 3.** A network $\mathcal{N}$ (middle) and its acyclic dependency graph $\Delta(\mathcal{N})$ (right). The nodes of $\Delta(\mathcal{N})$ are wires; internal wires get two labels. The arcs of $\Delta(\mathcal{N})$ are *non-sequential* input-output wire pairs of component circuits. Dotted arcs indicate that (1,2) and (7,10) are sequential pairs for $A$ and $C$ resp.; they are not part of $\Delta(\mathcal{N})$.

### 1.2 Related Work

Carloni et al. [2] pioneered a theory of latency-insensitive systems based on their notion of *patient systems*. We could not rely on this theory for proving properties of SELF since it does not cover systems with combinational propagation of signals, an important class for most practical applications. In addition, the papers [2] and its companion [3] do not specify a particular implementation protocol, nor the properties required for its correctness. Our recovery of the protocol specification based on [3] and the private communication with the author proved that SELF cannot be covered by the theory of patient systems and requires a separate theory.

Suhaib et al. [11] revisited and generalized Carloni's elasticization procedure, validating its correctness by a simulation method based on model checking.

Lee et al. [9] study *causality interfaces* (pairwise input-output dependencies) and are "interested in existence and uniqueness of the behavior of feedback composition", but do not go as far as deriving a combinational loop theorem.

In their work on design of interlock pipelines [6], Jacobson et al. use a protocol equivalent to SELF, without explicitly specifying it.

## 2 Circuits as Stream Functions

In this section we introduce *machines* as a mathematical abstraction of *circuits without combinational cycles*. For simplicity, this abstraction implicitly assumes that all sequential elements inside the circuit are *initialized*. Extending to partially initialized systems appears to be trivial. While there is a large body of work studying circuits or equivalent objects with *good (e.g. constructive [1]) combinational cycles* and their composition (e.g. [5]), we deliberately restrict consideration to the fully acyclic objects, since neither logic synthesis nor timing analysis can properly treat circuits with combinational cycles.

Most of the effort in this section goes into establishing modularity conditions guaranteeing that a system obtained as a network of machines (the feedback construction in particular) is a machine itself.

### 2.1 Streams

A *stream over* $A$ is an infinite sequence whose elements belong to the set $A$. The first element of a stream $a$ is referred to by $a[0]$, the second by $a[1]$, etc. For example, the equation $a[i] = 3i + 1$ describes the stream $(1, 4, 7, \ldots)$.

The set of all streams will be denoted $A^\infty$. Occassionally we will need to consider finite sequences too; the set of all, finite or infinite, sequences over $A$ is denoted $A^\omega$.

We will write $a \sim_k b$ to indicate that the streams $a$ and $b$ have a common prefix of length $k$. The equivalence relations $\sim_0, \sim_1, \sim_2, \ldots$ are progressively finer and have trivial intersection. Thus, to prove two sequences $a$ and $b$ are equal, it suffices to show $a \sim_k b$ holds for every $k$. Note also that $a \sim_0 b$ holds for every $a$ and $b$.

We will use the equivalence relations $\sim_k$ to express properties of systems and machines viewed as multivariate stream functions. All these properties will be derived from the following two basic properties of single-variable stream functions $f\colon A^\infty \to B^\infty$.

$$\begin{array}{ll} \textit{causality:} & \forall a, b \in A^\infty.\ \forall k \geq 0.\ \ a \sim_k b \Rightarrow f(a) \sim_k f(b) \\ \textit{contraction:} & \forall a, b \in A^\infty.\ \forall k \geq 0.\ \ a \sim_k b \Rightarrow f(a) \sim_{k+1} f(b) \end{array}$$

Informally, $f$ is causal if (for every $a$) the first $k$ elements of $f(a)$ are determined by the first $k$ elements of $a$, and $f$ is contractive if the first $k$ elements of $f(a)$ are determined by the first $k-1$ elements of $a$.

**Lemma 1.** *If $f\colon A^\infty \to A^\infty$ is contractive, then it has a unique fixpoint.*

*Remark 1.* One can define the *distance* $d(a, b)$ between sequences $a$ and $b$ to be $1/2^k$, where $k$ is the length of the largest common prefix of $a$ and $b$. This gives the sets $A^\infty$ and $A^\omega$ the structure of complete metric spaces and Lemma 1 is an instance of Banach Fixed Point Theorem. See [8] for more details and references about the metric semantics of systems. We choose not to use the metric space terminology in this paper since all "metric reasoning" we need can be as easily done with equivalence relations $\sim_k$ instead. See [10] for principles of reasoning with such "converging equivalence relations" in more general contexts.

### 2.2 Systems

Suppose $W$ is a set of typed *wires*; all we know about an individual wire $w$ is a set $\mathsf{type}(w)$ associated to it. A *$W$-behavior* is a function $\sigma$ that associates a stream $\sigma.w \in \mathsf{type}(w)^\infty$ to each wire $w \in W$. The set of all $W$-behaviors will be denoted $[\![W]\!]$. Slightly abusing the notation, we will also write $[\![w]\!]$ for the set $\mathsf{type}(w)^\infty$. Notice that the equivalence relations $\sim_k$ extend naturally from streams to behaviors:

$$\sigma \sim_k \sigma' \quad \text{iff} \quad \forall w \in W.\ \ \sigma.w \sim_k \sigma'.w$$

Notice also that a $W$-behavior $\sigma$ can be seen as a single stream $(\sigma[0], \sigma[1], \ldots)$ of *$W$-states*, where a state is an assignment of a value in $\mathsf{type}(w)$ to each wire $w$.

**Definition 1.** *A* $W$-system *is a subset of* $[\![W]\!]$.

*Example 1.* A circuit that at each clock cycle receives an integer as input and returns the sum of all previously received inputs is described by the $W$-system $\mathcal{S}$, where $W$ consist of two wires $u, v$ of type $\mathbb{Z}$, and $\mathcal{S}$ consists of all stream pairs $(a, b) \in \mathbb{Z}^\infty \times \mathbb{Z}^\infty$ such that $b[0] = 0$ and $b[n] = a[0] + \cdots + a[n-1]$ for $n > 0$. Each stream pair $(a, b)$ represents a behavior $\sigma$ such that $\sigma.u = a$ and $\sigma.v = b$.

We will use wires as typed variables in formulas meant to describe system properties. The formulas are built using ordinary mathematical and logical notation, enhanced with temporal operators *next*, *always*, and *eventually*, denoted respectively by $(\_)^+, \mathsf{G}, \mathsf{F}$. For example, the system $\mathcal{S}$ in Example 1 is characterized by the property $v = 0 \wedge \mathsf{G}\,(v^+ = v + u)$. Also, one has $\mathcal{S} \models \mathsf{F}\,\mathsf{G}\,(u > 0) \Rightarrow \mathsf{F}\,\mathsf{G}\,(v > 1000)$, where $\models$ is used to denote that a formula is true of a system.

### 2.3 Operations on Systems

If $W' \subseteq W$, there is an obvious projection map $\sigma \mapsto \sigma \downarrow W' \colon [\![W]\!] \to [\![W']\!]$. These projections are all one needs for the definition of the following two basic operations on systems.

**Definition 2.** (a) *If $\mathcal{S}$ is a $W$-system and $W' \subseteq W$, then* hiding *$W'$ in $\mathcal{S}$ produces a $(W - W')$-system* $\mathsf{hide}_{W'}(\mathcal{S})$ *defined by*

$$\tau \in \mathsf{hide}_{W'}(\mathcal{S}) \quad \textit{iff} \quad \exists \sigma \in \mathcal{S}.\ \tau = \sigma \downarrow (W - W').$$

(b) *The* composition *of a $W_1$-system $\mathcal{S}_1$ and a $W_2$-system $\mathcal{S}_2$ is a $(W_1 \cup W_2)$-system $\mathcal{S}_1 \sqcup \mathcal{S}_2$ defined by*

$$\sigma \in \mathcal{S}_1 \sqcup \mathcal{S}_2 \quad \textit{iff} \quad \sigma \downarrow W_1 \in \mathcal{S}_1 \wedge \sigma \downarrow W_2 \in \mathcal{S}_2.$$

If $W$ and $W'$ are disjoint wire sets, $\sigma \in [\![W]\!]$, and $\tau \in [\![W']\!]$, then there is a unique behavior $\vartheta \in [\![W \cup W']\!]$ such that $\sigma = \vartheta \downarrow W$ and $\tau = \vartheta \downarrow W'$. This "product" of behaviors will be written as $\vartheta = \sigma * \tau$. (If $W$ is the empty set, then $[\![W]\!]$ has one element—a "trivial behavior" that is also a multiplicative unit for the product operation $*$.) We will also use the notation $[u \mapsto a, v \mapsto b, \ldots]$ for the $\{u, v, \ldots\}$-behavior $\sigma$ such that $\sigma.u = a$, $\sigma.v = b$, etc.

Hiding and composition suffice to define complex networks of systems. To model identification of wires, we use simple *connection systems*: by definition, $\mathsf{Conn}(u, v)$ is the $\{u, v\}$-system consisting of all behaviors $\sigma$ such that $\sigma.u = \sigma.v$.

Now if $\mathcal{S}_1, \ldots, \mathcal{S}_m$ are given systems and $u_1, \ldots, u_n, v_1, \ldots, v_n$ are some of their wires, the network obtained from these systems by identifying each wire $u_i$ with the corresponding wire $v_i$ (of equal type) is the system

$$\begin{aligned}&\langle \mathcal{S}_1, \ldots, \mathcal{S}_m \,|\, u_1 = v_2, \ldots, u_n = v_n \rangle \\ &= \mathsf{hide}_{\{u_1,\ldots,u_n,v_1,\ldots,v_n\}}(\mathcal{S}_1 \sqcup \cdots \sqcup \mathcal{S}_m \sqcup \mathsf{Conn}(u_1, v_1) \sqcup \cdots \sqcup \mathsf{Conn}(u_n, v_n))\end{aligned}$$

The simplest case ($m = n = 1$) of networks is the construction

$$\langle \mathcal{S} \,|\, u = v \rangle = \mathsf{hide}_{\{u,v\}}(\mathcal{S} \sqcup \mathsf{Conn}(u, v)),$$

used for a *feedback* definition in Section 2.5. A behavior $\sigma$ belongs to $\langle \mathcal{S} \,|\, u = v \rangle$ if and only if $\sigma * [u \mapsto a, v \mapsto a] \in \mathcal{S}$ for some $a \in [\![u]\!]$.

### 2.4 Machines

Suppose $I$ and $O$ are disjoint sets of wires, called *inputs* and *outputs*, correspondingly. By definition, an $(I, O)$-*system* is just an $(I \cup O)$-system. Consider the following properties of an $(I, O)$-system $\mathcal{S}$.

| | |
|---|---|
| *deterministic:* | $\forall \omega, \omega' \in \mathcal{S}.\ \omega \downarrow I = \omega' \downarrow I \ \Rightarrow\ \omega \downarrow O = \omega' \downarrow O$ |
| *functional:* | $\forall \sigma \in [\![I]\!].\, \exists! \tau \in [\![O]\!].\, \sigma * \tau \in \mathcal{S}$ |
| *causal:* | $\forall \omega, \omega' \in \mathcal{S}. \forall k \geq 0.\ \omega \downarrow I \sim_k \omega' \downarrow I \ \Rightarrow\ \omega \downarrow O \sim_k \omega' \downarrow O$ |

Clearly, functionality implies determinism. Conversely, a deterministic system is functional if and only if it accepts all inputs. Note also that causality implies determinism: if $\omega \downarrow I = \omega' \downarrow I$, then $\omega \downarrow I \sim_k \omega' \downarrow I$ holds for every $k$, so $\omega \downarrow O \sim_k \omega' \downarrow O$ holds for every $k$ too, so $\omega \downarrow O = \omega' \downarrow O$.

**Definition 3.** *An* $(I, O)$-machine *is an* $(I, O)$*-system that is both functional and causal.*

A functional system $\mathcal{S}$ uniquely determines and is determined by the function $F\colon [\![I]\!] \to [\![O]\!]$ such that $F(\sigma) = \tau$ holds if and only if $\sigma * \tau \in \mathcal{S}$. The causality condition for such $\mathcal{S}$ can be also written as follows:

$$\forall \sigma, \sigma' \in [\![I]\!].\, \forall k \geq 0.\ \ \sigma \sim_k \sigma' \Rightarrow F(\sigma) \sim_k F(\sigma').$$

The system in Example 1 is a machine if we regard $u$ as an input wire and $v$ as an output wire. The same is true of the system $\mathsf{Conn}(u, v)$: its associated function $F$ is the identity function.

### 2.5 Feedback on Machines

We will use the term *feedback* for the system $\langle \mathcal{S} \,|\, u = v \rangle$ as mentioned in Section 2.3 when $\mathcal{S}$ is a machine and the wires $u$ and $v$ of the same type are an input and output of $\mathcal{S}$ respectively. Our concern now is to understand under what conditions the feedback produces a machine.

To fix the notation, assume $\mathcal{S}$ is an $(I, O)$-machine given by $F\colon [\![I]\!] \to [\![O]\!]$, with wires $u \in I$, $v \in O$ of the same type $A$. By the note at the end of Section 2.3, we have that for every $\sigma \in [\![I - \{u\}]\!]$ and $\tau \in [\![O - \{v\}]\!]$,

$$\sigma * \tau \in \langle \mathcal{S} \,|\, u = v \rangle \ \text{ iff } \ \exists a \in A^\infty.\, F(\sigma * [u \mapsto a]) = \tau * [v \mapsto a]),$$

so $\langle \mathcal{S} \,|\, u = v \rangle$ is functional when the function $F^\sigma_{uv} : A^\infty \to A^\infty$ defined by $F^\sigma_{uv}(a) = F(\sigma * [u \mapsto a]).v$ has a unique fixpoint. By Lemma 1, this is guaranteed if $F^\sigma_{uv}$ is contractive. The following definition introduces the key concept of sequentiality that formalizes the intutive notion that there is no combinational dependence of a given output wire on a given input wire. Sequentiality of the pair $(u, v)$ easily implies contractivity of $F^\sigma_{uv}$ for all $\sigma$.

**Definition 4.** *The pair* $(u, v)$ *is* sequential *for* $\mathcal{S}$ *if for every* $k \geq 0$*, every* $\sigma, \sigma' \in [\![I - \{u\}]\!]$*, and every* $a, a' \in [\![u]\!]$ *one has*

$$\sigma \sim_{k+1} \sigma' \wedge a \sim_k a' \ \Rightarrow\ F(\sigma * [u \mapsto a]).v \sim_{k+1} F(\sigma' * [u \mapsto a']).v$$

**Lemma 2 (Feedback).** *If* $(u, v)$ *is a sequential input-output pair for a machine* $\mathcal{S}$*, then the feedback system* $\langle \mathcal{S} \,|\, u = v \rangle$ *is a machine too.*

*Example 2.* Consider the system $\mathcal{S}$ with $I = \{u, v\}$, $O = \{w, z\}$, specified by equations

$$w = u \oplus ((0)\#v) \qquad z = v \oplus v,$$

where all wires have type $\mathbb{Z}$, the symbol $\oplus$ denotes the componentwise sum of streams, and $\#$ denotes concatenation. Since $z$ does not depend on $u$, the pair $(u, z)$ is sequential. The pair $(v, w)$ is also sequential since to compute a prefix of $w$ it suffices to know (a prefix of the same size of $u$ and) a prefix of smaller size of $v$. The remaining two input-output pairs $(u, w)$ and $(v, z)$ are not sequential.

To find the machine $\langle \mathcal{S} \,|\, v = w \rangle$, we need to solve the equation $v = u \oplus ((0)\#v)$ for $v$. For each $u = (a_0, a_1, a_2, \ldots)$, the equation has a unique solution $v = \hat{u} = (a_0, a_0 + a_1, a_0 + a_1 + a_2, \ldots)$. Substituting the solution into $z = v \oplus v$, we obtain a description of $\langle \mathcal{S} \,|\, v = w \rangle$ by a single equation that relates its input and output: $z = \hat{u} \oplus \hat{u}$. The other feedback $\langle \mathcal{S} \,|\, u = z \rangle$ is easier to calculate; it is given by equation $w = v \oplus v \oplus ((0)\#v)$.

### 2.6 Networks of Machines and the Combinational Loop Theorem

Consider a network $\mathcal{N} = \langle \mathcal{S}_1, \ldots, \mathcal{S}_m \,|\, u_1 = v_1, \ldots, u_n = v_n \rangle$, where $\mathcal{S}_1, \ldots, \mathcal{S}_m$ are machines with disjoint wire sets and the pairs $(u_1, v_1), \ldots, (u_n, v_n)$ involve $n$ distinct input wires $u_i$ and $n$ distinct output wires $v_i$. (There is no assumption that $u_i, v_i$ belong to the same machine $\mathcal{S}_j$.) Our goal is to understand under what conditions the system $\mathcal{N}$ is a machine.

Note that $\mathcal{N} = \langle \mathcal{S} \,|\, u_1 = v_2, \ldots, u_n = v_n \rangle$, where $\mathcal{S} = \mathcal{S}_1 \sqcup \cdots \sqcup \mathcal{S}_m$. It is easy to check that an input-output pair $(u, v)$ of $\mathcal{S}$ is sequential if either (1) $(u, v)$ is sequential for some $\mathcal{S}_i$, or (2) $u$ and $v$ belong to different machines. Thus, the information about sequentiality of input-output pairs of the "parallel composition" machine $\mathcal{S}$ is readily available from the sequentiality information about the component machines $\mathcal{S}_i$, and our problem boils down to determining when a multiple feedback operation performed on a single machine results in a system that is itself a machine.

Simultaneous feedback specified by a set of two or more input-output pairs of a machine does not necessarlily produce a machine even if all pairs involved are sequential. Indeed, in Example 2 we had a system $\mathcal{S}$ with two sequential pairs $(u, z)$ and $(v, w)$, but $(u, z)$ ceases to be sequential for $\langle \mathcal{S} \,|\, v = w \rangle$. Indeed, if $z$ and $u$ are related by $z = \hat{u} \oplus \hat{u}$, then knowing a prefix of length $k$ of $z$ requires knowing the prefix of the same length of $u$; a shorter one would not suffice.

To ensure that a multiple feedback construction produces a machine, it is necessary that, in addition to the wire pairs to be identified, sufficiently many other input-output pairs are also sequential. A precise formulation for a *double* feedback is given by a version of the Bekić Lemma: for the system $\langle \mathcal{S} \,|\, u = w, v = z \rangle$ to be a machine, it suffices that *three* pairs of wires be sequential—$(u, w)$, $(v, z)$, and one of $(u, z)$, $(v, w)$. This non-trivial auxiliary result is needed for the proof of Theorem 1 below, and is a special case of it.

Given an $(I, O)$-machine $\mathcal{S}$, let its *dependency graph* $\Delta(\mathcal{S})$ have the vertex set $I \cup O$ and directed edges that go from $u$ to $v$ for each pair $(u, v) \in I \times O$ that is *not* sequential. For a network system $\mathcal{N} = \langle \mathcal{S}_1, \ldots, \mathcal{S}_m \,|\, u_1 = v_1, \ldots, u_n = v_n \rangle$, its graph $\Delta(\mathcal{N})$ is then defined as the direct sum of graphs $\Delta(\mathcal{S}_1), \ldots, \Delta(\mathcal{S}_m)$ with each vertex $u_i$ $(1 \le i \le n)$ identified with the corresponding vertex $v_i$ (Figure 3).

**Theorem 1 (Combinational Loop Theorem).** *The network system $\mathcal{N}$ is a machine if the graph $\Delta(\mathcal{N})$ is acyclic.*

## 3 Elastic Machines

In this section we give the definition of elastic machines. Its four parts—input-output structure, persistence conditions, liveness conditions, and the transfer determinism condition—are covered by Definitions 5-8 below.

### 3.1 Input-output Structure, Channels, and Transfer

We assume that the set of wires is partitioned into *ordinary*, *valid*, and *stop* wires, so that for each ordinary wire $X$ there exist associated wires $\mathsf{valid}_X$ and $\mathsf{stop}_X$ of boolean type. (In actual circuit implementations, $\mathsf{valid}_X$ and $\mathsf{stop}_X$ need not be physical wires; it suffices that they be appropriately encoded.)

**Definition 5.** *Let $I, O$ be disjoint sets of ordinary wires. An $[I,O]$-*system *is an $(I',O')$-machine, where $I' = I \cup \{\mathsf{valid}_X \,|\, X \in I\} \cup \{\mathsf{stop}_Y \,|\, Y \in O\}$ and $O' = O \cup \{\mathsf{valid}_Y \,|\, Y \in O\} \cup \{\mathsf{stop}_X \,|\, X \in I\}$.*

The triples $\langle X, \mathsf{valid}_X, \mathsf{stop}_X\rangle$ $(X \in I)$ and $\langle Y, \mathsf{valid}_Y, \mathsf{stop}_Y\rangle$ $(Y \in O)$ are to be thought of as *elastic input and output channels* of the system.

Let $\mathsf{transfer}_Z$ be a shorthand for $\mathsf{valid}_Z \wedge \neg\mathsf{stop}_Z$ and say that *transfer along $Z$ occurs in a state $s$* if $s \models \mathsf{transfer}_Z$. Given a behavior $\sigma = (\sigma[0], \sigma[1], \sigma[2], \ldots)$ of an $[I,O]$-system and $Z \in I \cup O$, let $\sigma_Z$ be the sequence (perhaps finite!) obtained from $\sigma.Z = (\sigma[0].Z, \sigma[1].Z, \sigma[2].Z, \ldots)$ by deleting all entries $\sigma[i].Z$ such that transfer along $Z$ does not occur in $\sigma[i]$. The *transfer behavior* $\sigma^\intercal$ associated with $\sigma$ is then defined by $\sigma^\intercal.Z = \sigma_Z$. If all sequences $\sigma_Z$ are infinite, then $\sigma^\intercal$ is an $(I \cup O)$-behavior; in general, however, we only have $\sigma_Z \in \mathsf{type}(Z)^\omega$.

For each wire $Z$ of an $[I,O]$-system $\mathcal{S}$ we introduce an auxiliary *transfer counter variable* $\mathsf{tct}_Z$ of type $\mathbb{Z}$. The counters serve for expressing system properties related to transfer. By definition, $\mathsf{tct}_Z$ is equal to the number of states that precede the current state and in which transfer along $Z$ has occurred. That is, for every behavior $\sigma$ of $\mathcal{S}$, we have $\sigma.\mathsf{tct}_Z = (t_0, t_1, \ldots)$, where $t_k$ is the number of indices $i$ such that $i < k$ and transfer along $Z$ occurs in $\sigma[i]$. Note that the sequence $\sigma.\mathsf{tct}_Z$ is non-decreasing and begins with $t_0 = 0$.

The notation $\mathsf{min\_tct}_S$, for any subset $S$ of $I \cup O$ will be used to denote the smallest of the numbers $\mathsf{tct}_Z$, where $Z \in S$.

### 3.2 Definition of Elasticity

An elastic component, when ready to communicate over an output channel must remain ready until the transfer takes place.

**Definition 6.** *The* persistence conditions *for an $[I,O]$-system $\mathcal{S}$ are given by*

$$\mathcal{S} \models \mathsf{G}\,(\mathsf{valid}_Y \wedge \mathsf{stop}_Y \Rightarrow (\mathsf{valid}_Y)^+), \quad \textit{for every } Y \in O. \tag{1}$$

The most useful consequence of persistence is the "handshake lemma":

$$\mathcal{S} \models \mathsf{G}\,\mathsf{F}\,\mathsf{valid}_Y \wedge \mathsf{G}\,\mathsf{F}\,\neg\mathsf{stop}_Y \;\Rightarrow\; \mathsf{G}\,\mathsf{F}\,\mathsf{transfer}_Y$$

Liveness of an elastic component is expressed in terms of token count: if all input channels have seen $k$ transfers and there is an output channel that has seen less, then the communication on output channels with the minimum amount of transfer must be eventually offered. The following definition formalizes this, together with a similar commitment to eventual readiness on input channels.

**Definition 7.** *The* liveness conditions *for an $[I, O]$-system are given by*

$$\mathcal{S} \models \mathsf{G}\,(\mathsf{min\_tct}_O \geq \mathsf{tct}_Y \wedge \mathsf{min\_tct}_I > \mathsf{tct}_Y \Rightarrow \mathsf{F}\,\mathsf{valid}_Y), \text{ for every } Y \in O \quad (2)$$

$$\mathcal{S} \models \mathsf{G}\,(\mathsf{min\_tct}_{I\cup O} \geq \mathsf{tct}_X \Rightarrow \mathsf{F}\,\neg\mathsf{stop}_X), \text{ for every } X \in I \quad (3)$$

In practice, elastic components will satisfy simpler (but stronger) liveness properties; e.g. remove $\mathsf{min\_tct}_O \geq \mathsf{tct}_Y$ from (2) and replace $\mathsf{min\_tct}_{I\cup O} \geq \mathsf{tct}_X$ with $\mathsf{min\_tct}_O \geq \mathsf{tct}_X$ in (3). However, a composition of such components, while satisfying (2) and (3), may not satify the stronger versions of these conditions.

Consider single-channel $[I, O]$-systems satisfying the persistence and liveness conditions: an *elastic consumer* is a $[\{Z\}, \emptyset]$-system $\mathcal{C}$ satisfying (4) below; similarly, an *elastic producer* is a $[\emptyset, \{Z\}]$-system $\mathcal{P}$ satisfying (5) and (6).

$$\mathcal{C} \models \mathsf{G}\,\mathsf{F}\,\neg\mathsf{stop}_Z \quad (4)$$

$$\mathcal{P} \models \mathsf{G}\,(\mathsf{valid}_Z \wedge \mathsf{stop}_Z \Rightarrow (\mathsf{valid}_Z)^+) \quad (5)$$

$$\mathcal{P} \models \mathsf{G}\,\mathsf{F}\,\mathsf{valid}_Z \quad (6)$$

Let $\mathcal{C}_Z$ be the $\{Z, \mathsf{valid}_Z, \mathsf{stop}_Z\}$-system characterized by condition (4)—the largest (in the sense of behavior inclusion) of the systems satisfying this condition. Similarly, let $\mathcal{P}_Z$ be the $\{Z, \mathsf{valid}_Z, \mathsf{stop}_Z\}$-system characterized by properties (5) and (6). Finally, define *the $[I, O]$-elastic environment* to be the system

$$\mathsf{Env}_{I,O} = \bigsqcup_{X\in I} \mathcal{P}_X \sqcup \bigsqcup_{Y\in O} \mathcal{C}_Y.$$

Note that $\mathsf{Env}_{I,O}$ is only a system; it is not functional and so is not a machine.

When a system satisfying the persistence and liveness conditions (1-3) is coupled with a matching elastic environment, the transfer on all data wires never comes to a stall:

**Lemma 3 (Liveness).** *If $\mathcal{S}$ satisfies (1-3), then for every behavior $\omega$ of $\mathcal{S} \sqcup \mathsf{Env}_{I,O}$, all the component sequences of the transfer behavior $\omega^\mathsf{T}$ are infinite.*

As an immediate consequence of Liveness Lemma, if $\mathcal{S}$ satisfies (1-3), then

$$\mathcal{S}^\mathsf{T} = \{\omega^\mathsf{T} \mid \omega \in \mathcal{S} \sqcup \mathsf{Env}_{I,O}\}$$

is a well-defined $(I, O)$-system.

**Definition 8.** *An $[I, O]$-system $\mathcal{S}$ is an* $[I, O]$-elastic machine *if it satisfies the properties (1-3) and the associated system $\mathcal{S}^\mathsf{T}$ is deterministic.*

The liveness conditions (2,3) are visibly related to causality at the transfer level: $k$ transfers on the input channels imply $k$ transfers on the output channels in the cooperating environment. Thus, it is not surprising that the determinism postulated in Definition 8 suffices to derive the causality of $\mathcal{S}^\mathsf{T}$:

**Theorem 2.** *If $\mathcal{S}$ is an $[I,O]$-elastic machine, then $\mathcal{S}^\mathsf{T}$ is an $(I,O)$-machine.*

In the situation of Definition 8, we say that $\mathcal{S}$ is an *elasticization* of $\mathcal{S}^\mathsf{T}$ and that $\mathcal{S}^\mathsf{T}$ is the *transfer machine* of $\mathcal{S}$.

## 4 Elastic Networks

An *elastic network* $\mathcal{N}$ is given by a set of elastic machines $\mathcal{S}_1,\ldots,\mathcal{S}_m$ with no shared wires, together with a set of channel pairs $(X_1,Y_1),\ldots,(X_n,Y_n)$, where the $X_i$ are $n$ distinct input channels and the $Y_i$ are $n$ distinct output channels. As a network of standard machines, the elastic network $\mathcal{N}$ is defined by

$$\mathcal{N} = \langle \mathcal{S}_1,\ldots,\mathcal{S}_m \,|\, X_i = Y_i, \mathsf{valid}_{X_i} = \mathsf{valid}_{Y_i}, \mathsf{stop}_{X_i} = \mathsf{stop}_{Y_i}\ (1 \le i \le n)\rangle,$$

for which we will use the shorter notation

$$\mathcal{N} = \langle\!\langle \mathcal{S}_1,\ldots,\mathcal{S}_m \,\|\, X_1 = Y_1,\ldots,X_n = Y_n \rangle\!\rangle.$$

We will define a graph that encodes the sequentiality information about the network $\mathcal{N}$ and prove in Theorem 4 that acyclicity of that graph implies that $\mathcal{N}$ is an elastic machine and that $\mathcal{N}^\mathsf{T} = \langle \mathcal{S}_1^\mathsf{T},\ldots,\mathcal{S}_m^\mathsf{T} \,|\, X_1 = Y_1,\ldots,X_n = Y_n\rangle$.

### 4.1 Elastic Feedback

*Elastic feedback* is a simple case of elastic network:

$$\langle\!\langle \mathcal{S} \,\|\, P = Q \rangle\!\rangle = \langle \mathcal{S} \,|\, P = Q, \mathsf{valid}_P = \mathsf{valid}_Q, \mathsf{stop}_P = \mathsf{stop}_Q\rangle.$$

**Definition 9.** *Suppose $\mathcal{S}$ is an elastic machine. An input-output channel pair $(P,Q)$ will be called* sequential *for $\mathcal{S}$ if*

$$\mathcal{S} \models \mathsf{G}\,(\mathsf{min\_tct}_{I\cup O} \ge \mathsf{tct}_Q \wedge \mathsf{min\_tct}_{I-\{P\}} > \mathsf{tct}_Q \Rightarrow \mathsf{F}\,\mathsf{valid}_Q). \tag{7}$$

Condition (7) is a strengthening of the liveness condition (2) for channel $Q$. It expresses a degree of independence of the output channel $Q$ from the input channel $P$; e.g., the first token at $Q$ need not wait for the arrival of the first token at $P$. This independence can be achieved in the system by storing some tokens inside, between these two channels. Note that (7) does not guarantee that connecting channels $P$ and $Q$ would not introduce ordinary combinational cycles. Therefore the acyclicity condition in the following theorem is required to ensure (by Theorem 1) that the elastic feedback, viewed as an ordinary network, is a machine.

**Theorem 3.** *Let $\mathcal{S}$ be an elastic machine and $\mathcal{F}$ the elastic feedback system $\langle\!\langle \mathcal{S} \,\|\, P = Q \rangle\!\rangle$. If the channel pair $(P,Q)$ is sequential for $\mathcal{S}$, then: (a) the wire pair $(P,Q)$ is sequential for $\mathcal{S}^\mathsf{T}$. If, in addition, $\Delta(\mathcal{F})$ is acyclic, then: (b) $\mathcal{F}$ is an elastic machine, and (c) $\mathcal{F}^\mathsf{T} = \langle \mathcal{S}^\mathsf{T} \,|\, P = Q\rangle$.*

### 4.2 Main Theorems

Sequentiality of two channel pairs $(P,Q),(P',Q)$ of an elastic machine does not imply their "simultaneous sequentiality"

$$\mathcal{S} \models \mathsf{G}\,(\mathsf{min\_tct}_{I\cup O} \geq \mathsf{tct}_Q \wedge \mathsf{min\_tct}_{I-\{P,P'\}} \geq \mathsf{tct}_Q \Rightarrow \mathsf{F}\,\mathsf{valid}_Q).$$

This deviates from the situation with ordinary machines, where the analogous property holds and is instrumental in the proof of Combinational Loop Theorem.

To justify multiple feedback on elastic machines, we have thus to postulate that simultaneous sequentiality is true where required. Specifically, we demand that elastic machines come with simultaneous sequentiality information: If $\mathcal{S}$ is an $[I,O]$-elastic machine, then for every $Y \in O$ a set $\delta(Y) \subseteq I$ is given so that

$$\mathcal{S} \models \mathsf{G}\,(\mathsf{min\_tct}_{I\cup O} \geq \mathsf{tct}_Q \wedge \mathsf{min\_tct}_{I-\delta(Q)} > \mathsf{tct}_Q \Rightarrow \mathsf{F}\,\mathsf{valid}_Q). \tag{8}$$

Note that if $P \in \delta(Q)$, then the pair $(P,Q)$ is sequential, but the converse is not implied. A function $\delta\colon O \to 2^I$ with the property (8) will be called a *sequentiality interface* for $\mathcal{S}$.

For an $[I,O]$-elastic machine $\mathcal{S}$ with a sequentiality interface $\delta$, we define $\Delta^{\mathrm{e}}(\mathcal{S},\delta)$ to be the graph with the vertex set $I \cup O$ and directed edges $(X,Y)$ where $X \notin \delta(Y)$. By Theorem 3(a), $\Delta^{\mathrm{e}}(\mathcal{S},\delta)$ contains $\Delta(\mathcal{S}^{\mathsf{T}})$ as a subgraph.

Given an elastic network $\mathcal{N} = \langle\!\langle \mathcal{S}_1,\ldots,\mathcal{S}_m \,\|\, X_1 = Y_1,\ldots,X_n = Y_n\rangle\!\rangle$, where each $\mathcal{S}_i$ comes equipped with a sequentiality interface $\delta_i$, its graph $\Delta^{\mathrm{e}}(\mathcal{N})$ is by definition the direct sum of graphs $\Delta^{\mathrm{e}}(\mathcal{S}_1,\delta_1),\ldots,\Delta^{\mathrm{e}}(\mathcal{S}_m,\delta_m)$ with each vertex $X_i$ $(1 \leq i \leq n)$ identified with the corresponding vertex $Y_i$.

**Theorem 4.** *If the graphs $\Delta(\mathcal{N})$ and $\Delta^{\mathrm{e}}(\mathcal{N})$ are acyclic, then the network system $\mathcal{N}$ is an elastic machine, the corresponding non-elastic system $\bar{\mathcal{N}} = \langle \mathcal{S}_1^{\mathsf{T}},\ldots,\mathcal{S}_m^{\mathsf{T}} \,|\, X_1 = Y_1,\ldots,X_n = Y_n\rangle$ is a machine, and $\mathcal{N}^{\mathsf{T}} = \bar{\mathcal{N}}$.*

As in Theorem 3, acyclicity of $\Delta(\mathcal{N})$ is needed to ensure (by Theorem 1) that $\mathcal{N}$ defines a machine. Elasticization procedures (e.g. [4]) will typically produce elastic components with enough sequential input-output wire pairs, so that $\Delta(\mathcal{N})$ will be acyclic as soon as $\Delta^{\mathrm{e}}(\mathcal{N})$ is acyclic.

Note, however, that cycles in $\Delta^{\mathrm{e}}(\mathcal{N})$ need not correspond to combinational cycles in $\mathcal{N}$ seen as an ordinary network, since empty buffers with sequential elements cutting the combinational feedbacks may be inserted into $\mathcal{N}$. Even though non-combinational in the ordinary sense, these cycles contain no tokens and therefore no progress along them can be made.

Theorem 4 impies that insertion of empty elastic buffers does not affect the basic functionality of an elastic network, as illustrated in Figure 2(b).

**Definition 10.** *An* empty elastic buffer *is an elastic machine $\mathcal{S}$ such that $\mathcal{S}^{\mathsf{T}} = \mathsf{Conn}(X,Y)$ for some $X,Y$.*

**Theorem 5 (Buffer Insertion Theorem).** *Let $\mathcal{B}$ be an empty elastic buffer with channels $X,Y$. Let $\mathcal{N} = \langle\!\langle \mathcal{S}_1,\ldots,\mathcal{S}_m \,\|\, X_1 = Y_1,\ldots,X_n = Y_n\rangle\!\rangle$ and $\mathcal{M} = \langle\!\langle \mathcal{B},\mathcal{S}_1,\ldots,\mathcal{S}_m \,\|\, X = Y_1, X_1 = Y, X_2 = Y_2,\ldots,X_n = Y_n\rangle\!\rangle$. If $\Delta(\mathcal{N})$, $\Delta(\mathcal{M})$, and $\Delta^{\mathrm{e}}(\mathcal{N})$ are acyclic, then $\mathcal{M}$ is an elastic machine, and $\mathcal{M}^{\mathsf{T}} = \mathcal{N}^{\mathsf{T}}$.*

The precise relationship between graphs $\Delta(\mathcal{M})$ and $\Delta(\mathcal{N})$ can be easily described. In practice they are at the same time acyclic or not, as a consequence of sequentiality of sufficiently many input-output wire pairs of $\mathcal{B}$.

## 5 Conclusion

We have presented a theory of elastic machines that gives an easy-to-check condition for the compositional theorem of the form "an elasticization of a network of ordinary components is equivalent to the network of components' elasticizations". Verification of a particular SELF implementation, such as in [4], is reduced to proving that conditions of Definition 8 are satisfied for all elastic components used, and that the graph $\Delta^{\mathrm{e}}(\mathcal{N}^{\mathrm{e}})$ is acyclic for every network $\mathcal{N}$ to which the elasticization is applied. While the definition of the graphs $\Delta^{\mathrm{e}}$ may appear complex because of the sequentiality interfaces involved, it should be noted that the elasticization procedures, e.g. [4], are reasonably expected to completely preserve sequentiality: a channel $P$ belongs to $\delta(Q)$ if the wire-pair $(P, Q)$ is sequential in the original non-elastic machine. This ensures $\Delta^{\mathrm{e}}(\mathcal{N}^{\mathrm{e}}) = \Delta(\mathcal{N})$ and so testing for sequentiality is done at the level of ordinary networks.

Future work will be focused on proving correctness of particular elasticization methods, on techniques for mechanical verification of elasticity, and on extending the theory to more advanced SELF protocols.

## References


1. G. Berry. *The Constructive Semantics of Pure Esterel*. Draft book, available at `www.esterel.org`, version 3, 1999.
2. L. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli. Theory of latency-insensitive design. *IEEE Tr. on CAD*, 20(9):1059–1076, 2001.
3. L. P. Carloni and A. L. Sangiovanni-Vincentelli. Coping with latency in SoC design. *IEEE Micro, Special Issue on Systems on Chip*, 22(5):12, 2002.
4. J. Cortadella, M. Kishinevsky, and B. Grundmann. SELF: Specification and design of a synchronous elastic architecture for DSM systems. *TAU* 2006 (to appear). Available at `www.lsi.upc.edu/~jordicf/gavina/BIB/reports/self_tr.pdf`.
5. S. A. Edwards and E. A. Lee. The semantics and execution of a synchronous block-diagram language. *Sci. Comput. Program.*, 48(1):21–42, 2003.
6. H. M. Jacobson et al. Synchronous interlocked pipelines. In *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, pp. 3–12, 2002.
7. S. Krstić, J. Cortadella, M. Kishinevsky, and J. O'Leary. Synchronous elastic networks. 2006. Available at `www.lsi.upc.edu/~jordicf/gavina/BIB/reports/elastic_nets.pdf`.
8. E. A. Lee and A. Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Tr. on CAD*, 17(12):1217–1229, 1998.
9. E. A. Lee, H. Zheng, and Y. Zhou. Causality interfaces and compositional causality analysis. Invited paper in *Foundations of Interface Technologies (FIT 2005)*.
10. J. Matthews. Recursive function definition over coinductive types. In *Proc. 12th Int. Conf. on Theorem Proving in Higher Order Logics*, pp. 73–90, 1999.
11. S. Suhaib et al. Presentation and formal verification of a family of protocols for latency insensitive design. TR 2005-02, FERMAT, Virginia Tech, 2005.

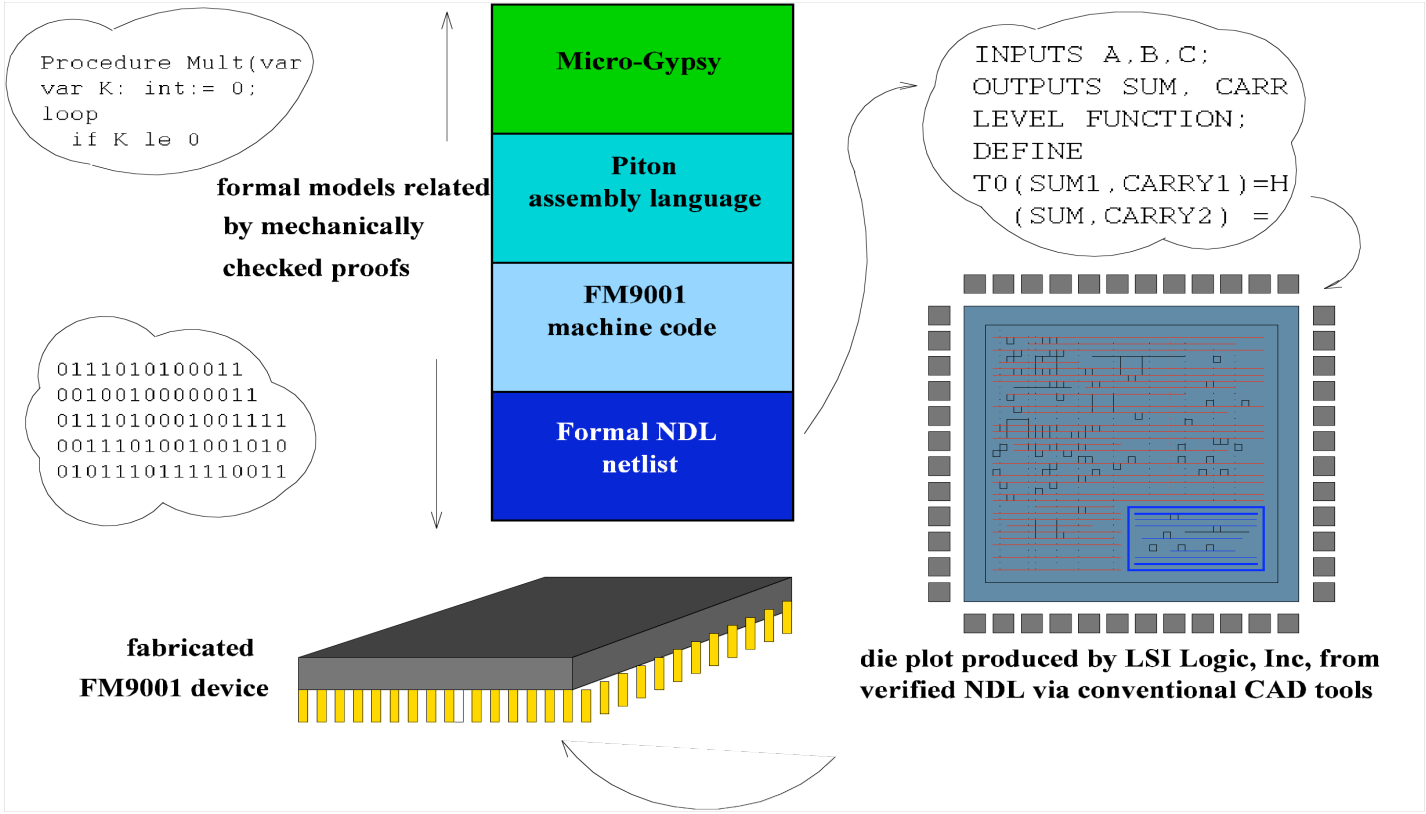
# Automating the Verification of RTL-Level Pipelined Machines

Panagiotis (Pete) Manolios
College of Computing
Georgia Institute of Technology
Supported by: NSF grants 0429924, 0417413, 0438871

DCC Workshop Vienna, Austria March 25-26, 2006

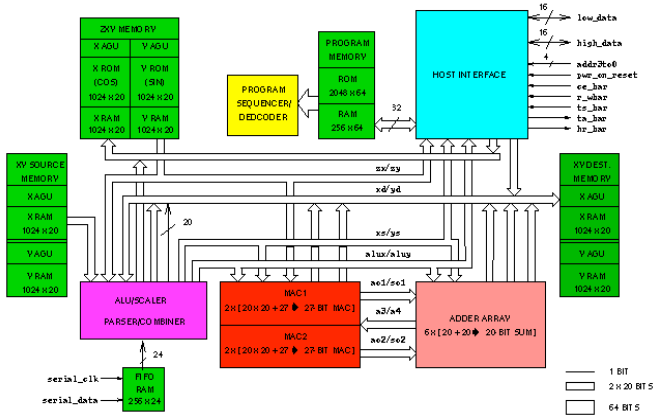
# Hardware Verification with ACL2

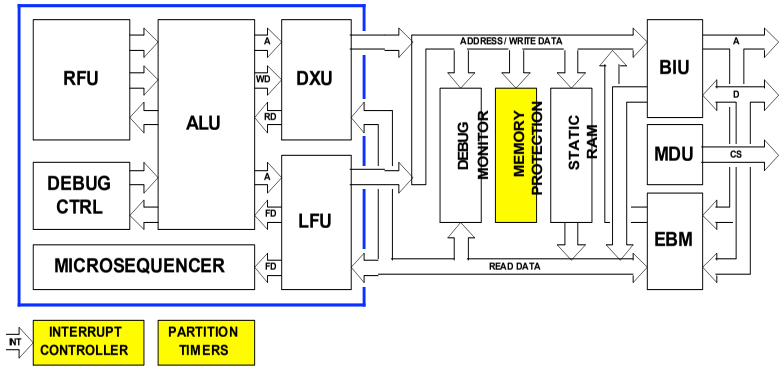
Verification system used to prove some of the largest and most complicated theorems ever proved about commercially designed systems.



# Hardware Verification with ACL2

- Motorola CAP DSP.
  - Bit/cycle-accurate model.
  - Run fasters than SPW model.
  - Proved correctness of pipeline hazard detection in microcode.
  - Verified microcode programs.
- Rockwell Collins AAMP7.
  - MILS EAL-7 certification from NSA for their crypto processor.
  - Verified separation kernel.
- Rockwell Collins JEM1.
- AMD Floating Point, … .



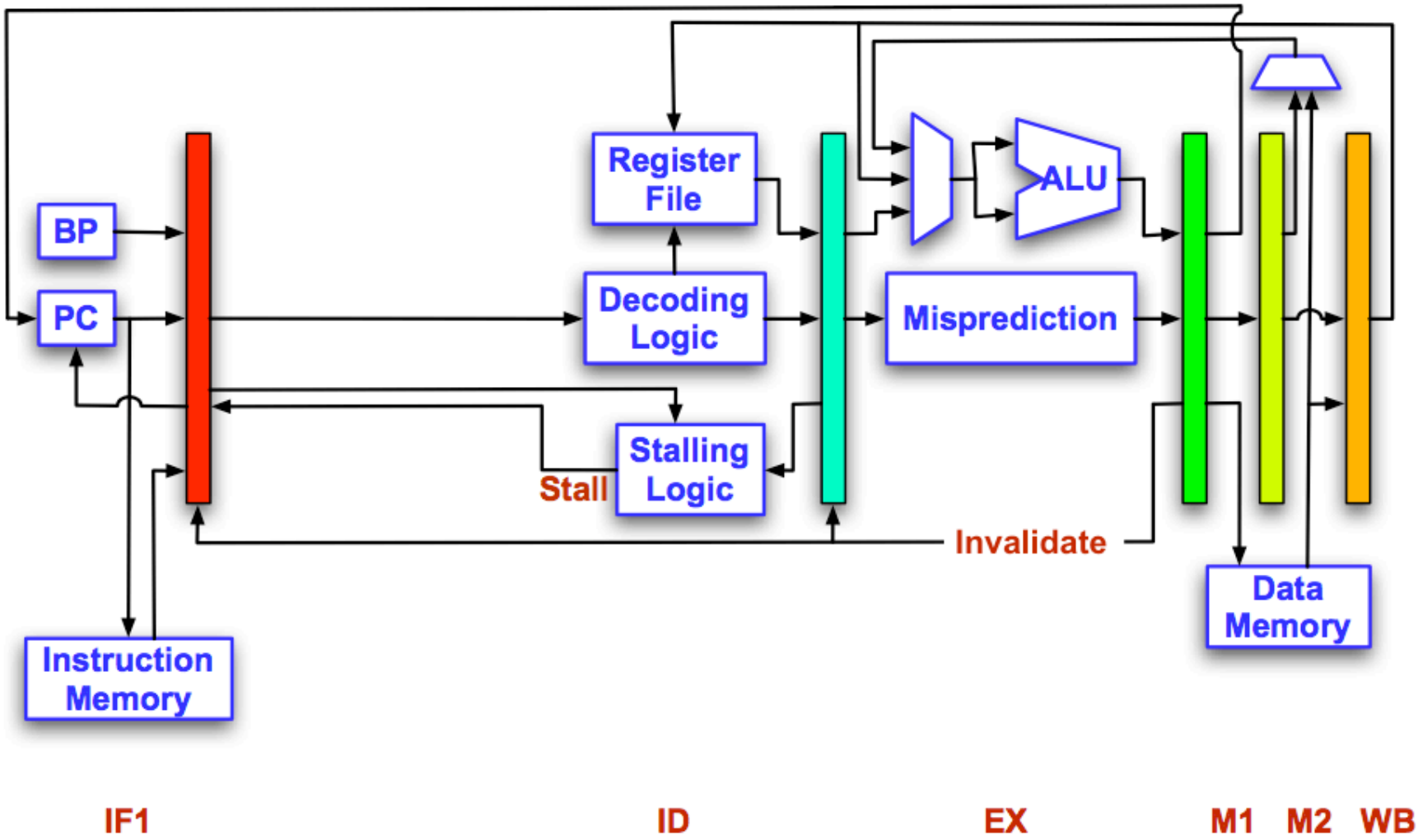
# Pipelined Machine Verification with UCLID

| Model | CNF Vars | CNF Clauses | UCLID [**sec**] | | | ACL2 [**days**] |
|---|---|---|---|---|---|---|
| | | | UCLID | Siege | Total | |
| 1 | 5,285 | 15,457 | 1 | 2 | 3 | 16 |
| 2 | 5,285 | 15,457 | 1 | 2 | 3 | ***16*** |
| 3 | 12,495 | 36,925 | 3 | 29 | 32 | ***165*** |
| 4 | 23,913 | 70,693 | 5 | 300 | 305 | ***1,575*** |
| 5 | 24,149 | 71,350 | 5 | 233 | 238 | ***1,230*** |
| 6 | 24,478 | 72,322 | 6 | 263 | 269 | ***1,390*** |
| 7 | 53,441 | 159,010 | 15 | 160 | 175 | ***904*** |
| 8 | 71,184 | 211,723 | 16 | 187 | 203 | ***1,049*** |

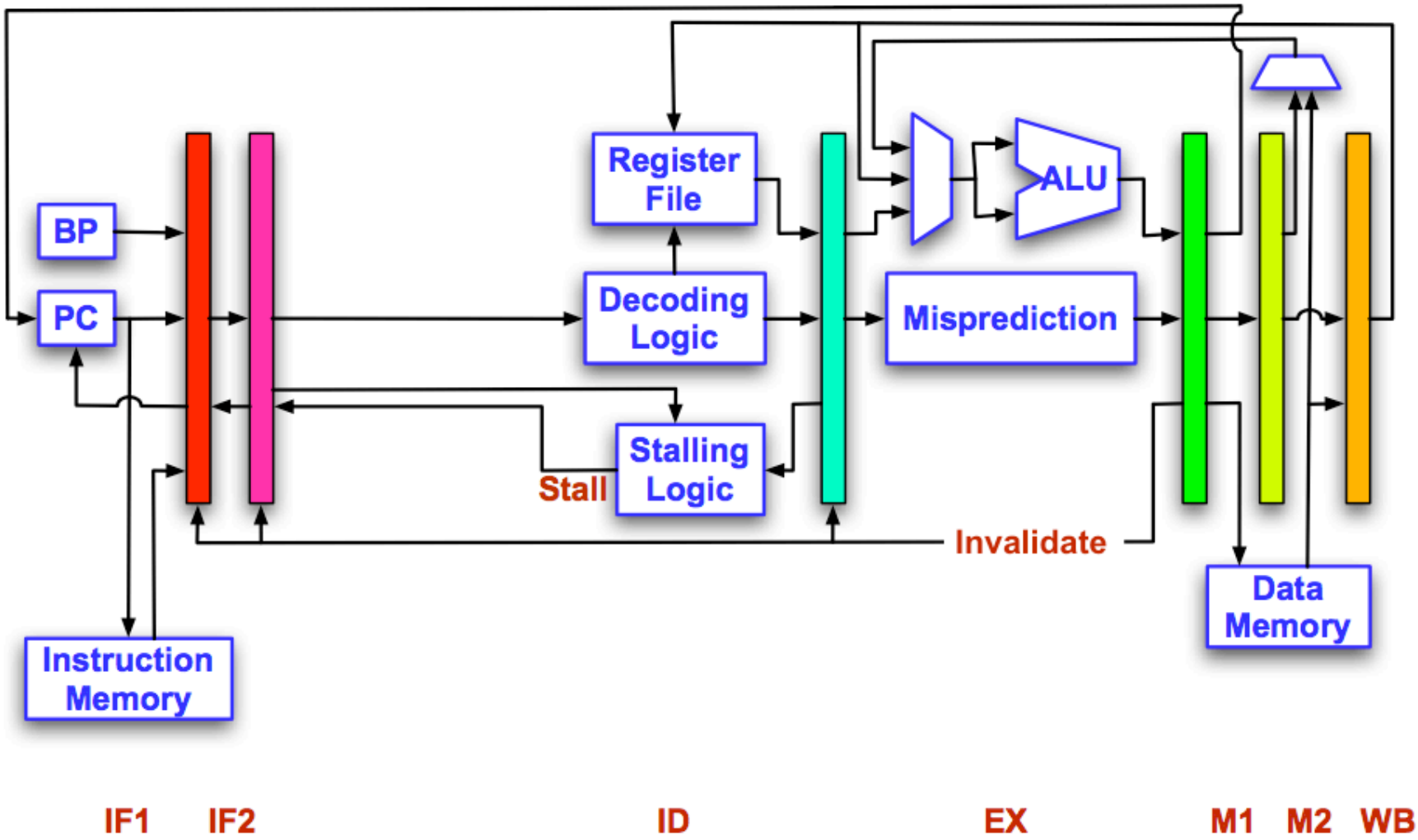
# Limitations of UCLID/Decision Procedures

- Correctness statement not expressible in CLU.
  - “Core theorem” is expressible.
- Term-level Modeling.
  - Datapath, decoder, etc. abstracted away.
  - Only small subset of instruction set implemented.
  - Restricted modeling language: no modules.
  - Restricted logic: forces us to add extra state & control logic.
  - Far from bit-level or executable.
  - No way to reason about programs: have no semantics.
  - Not clear how to relate to RTL, bit-level designs.
- Scaling issues arise as machine complexity increases.

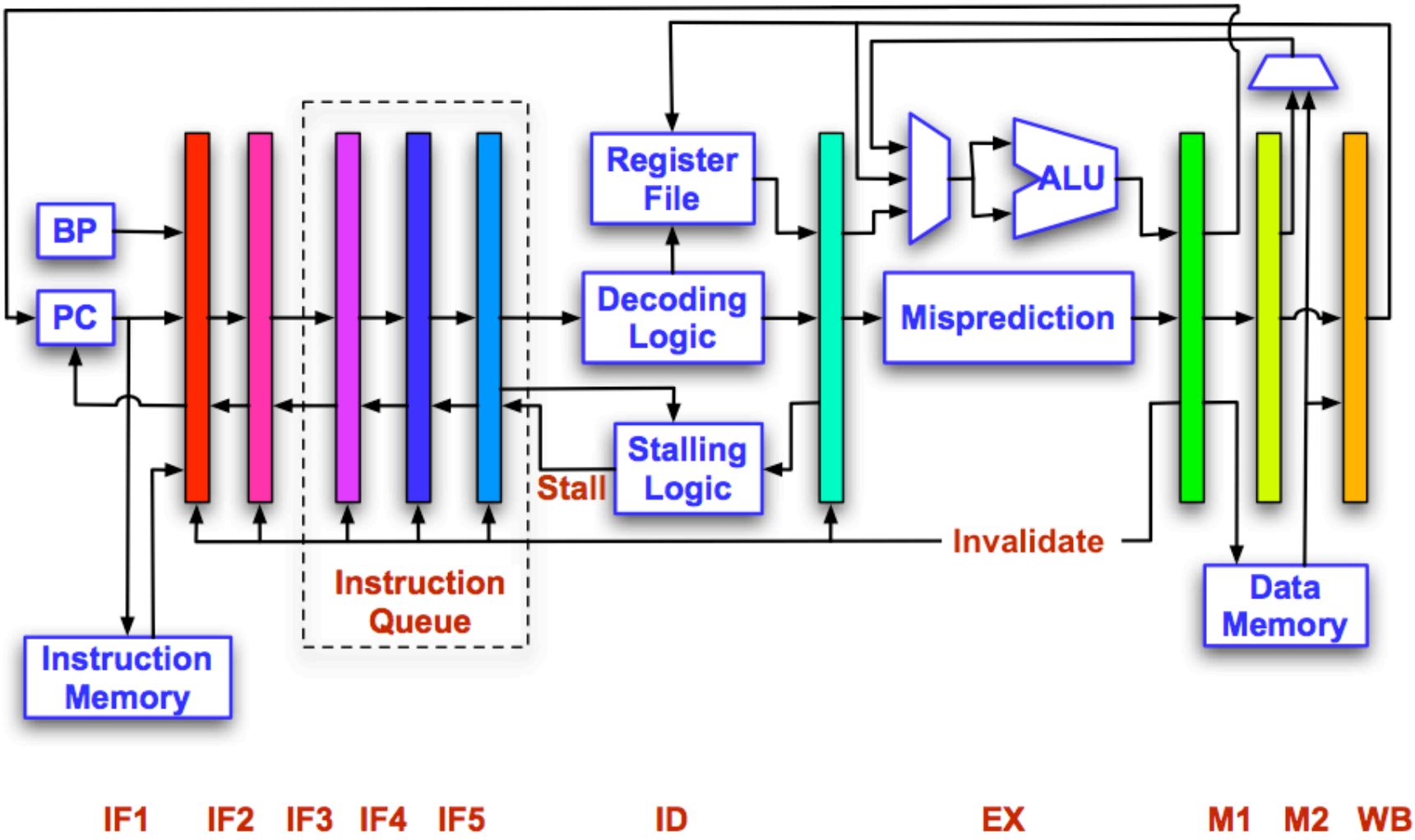
# Pipelined Machine M6



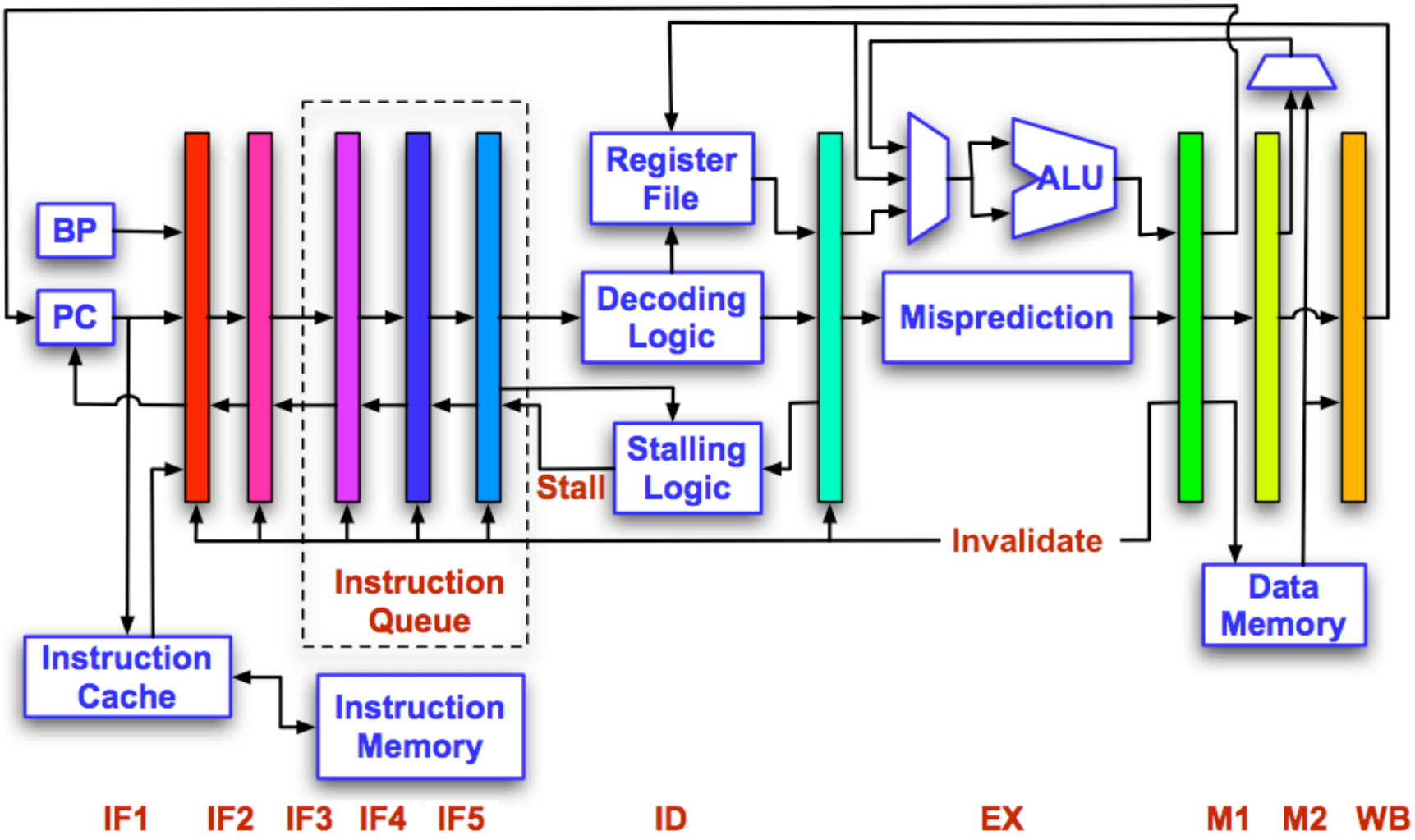
# Pipelined Machine M7



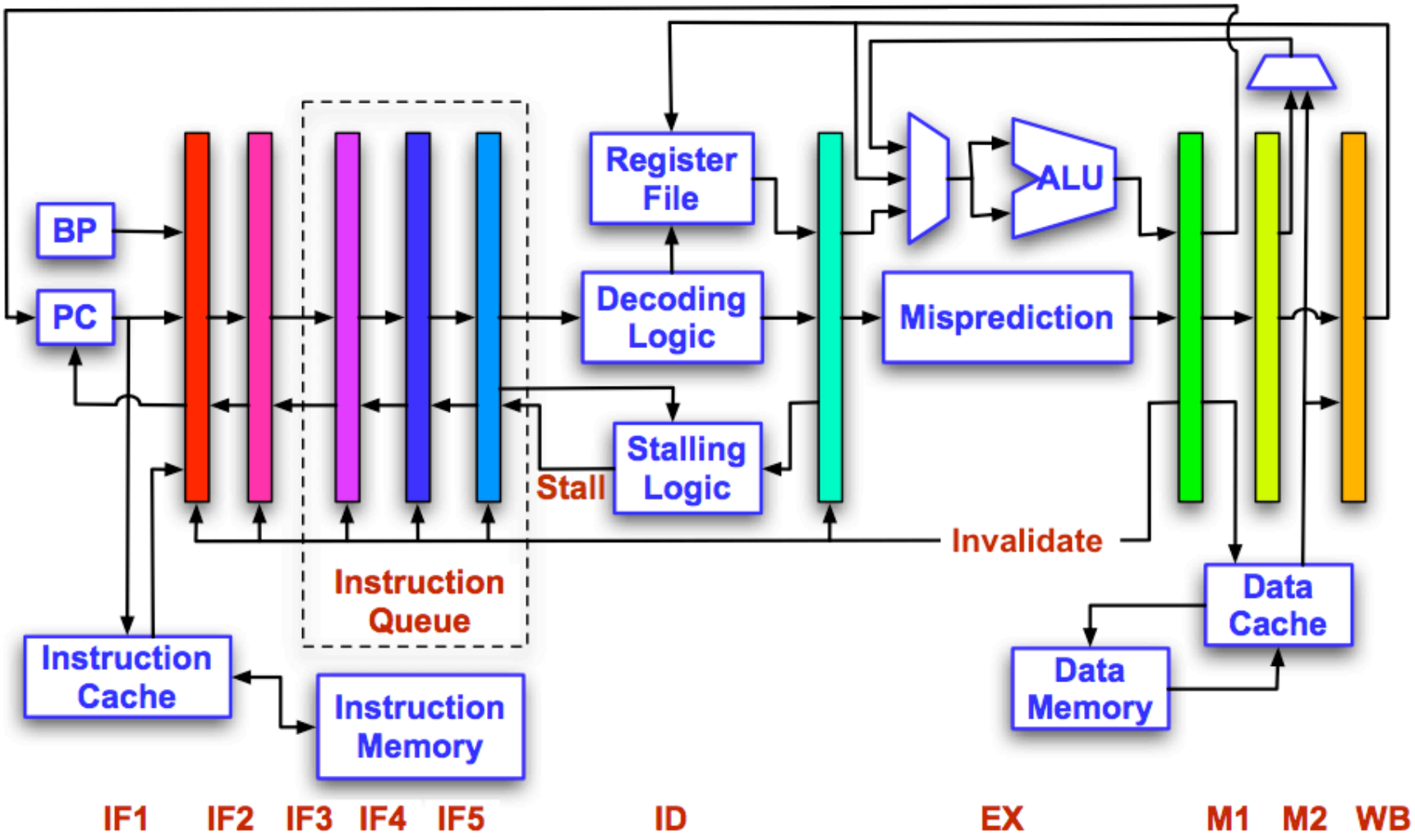
# Pipelined Machine M10



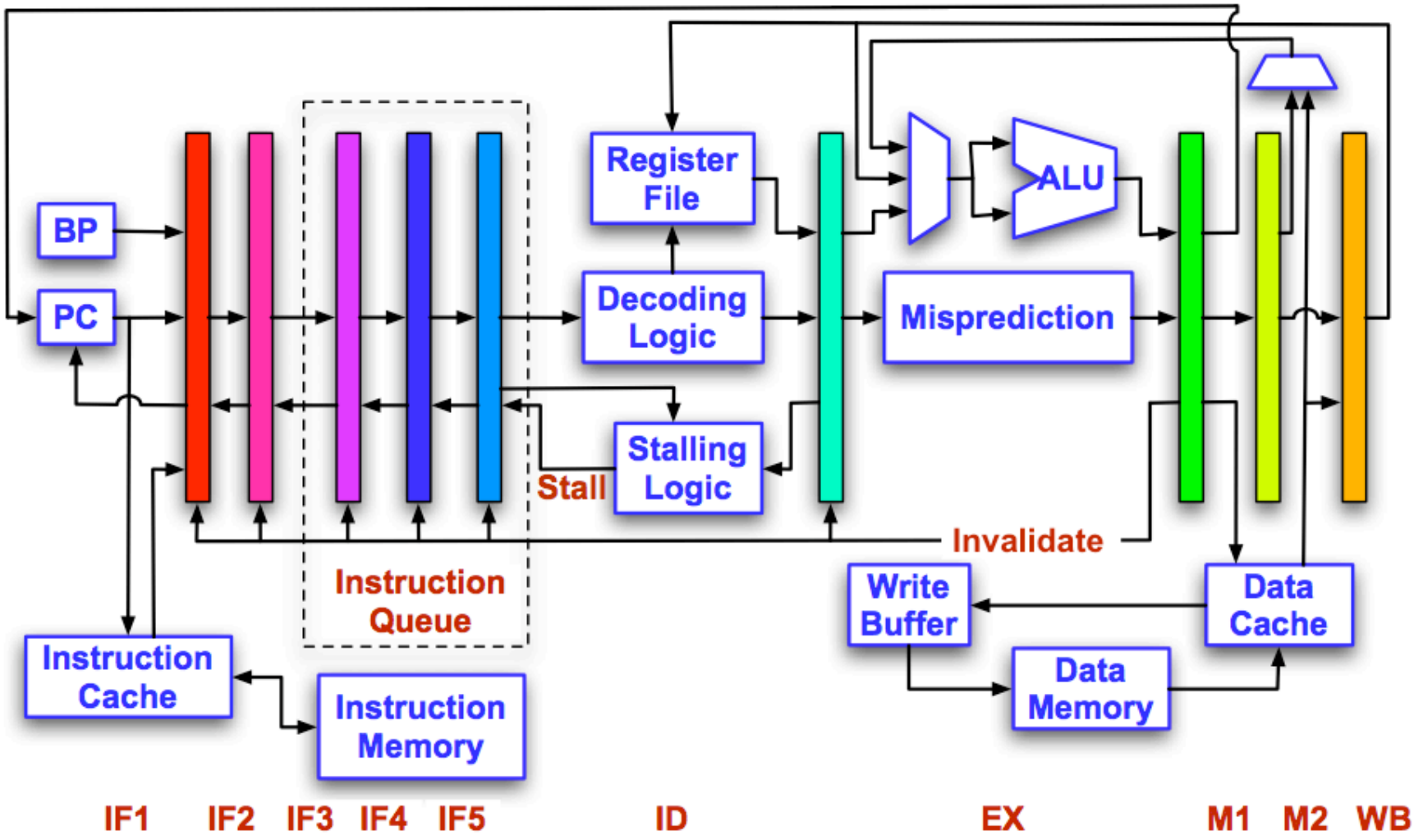
# Pipelined Machine M10I



# Pipelined Machine M10ID



# Pipelined Machine M10IDW



# Monolithic Verification

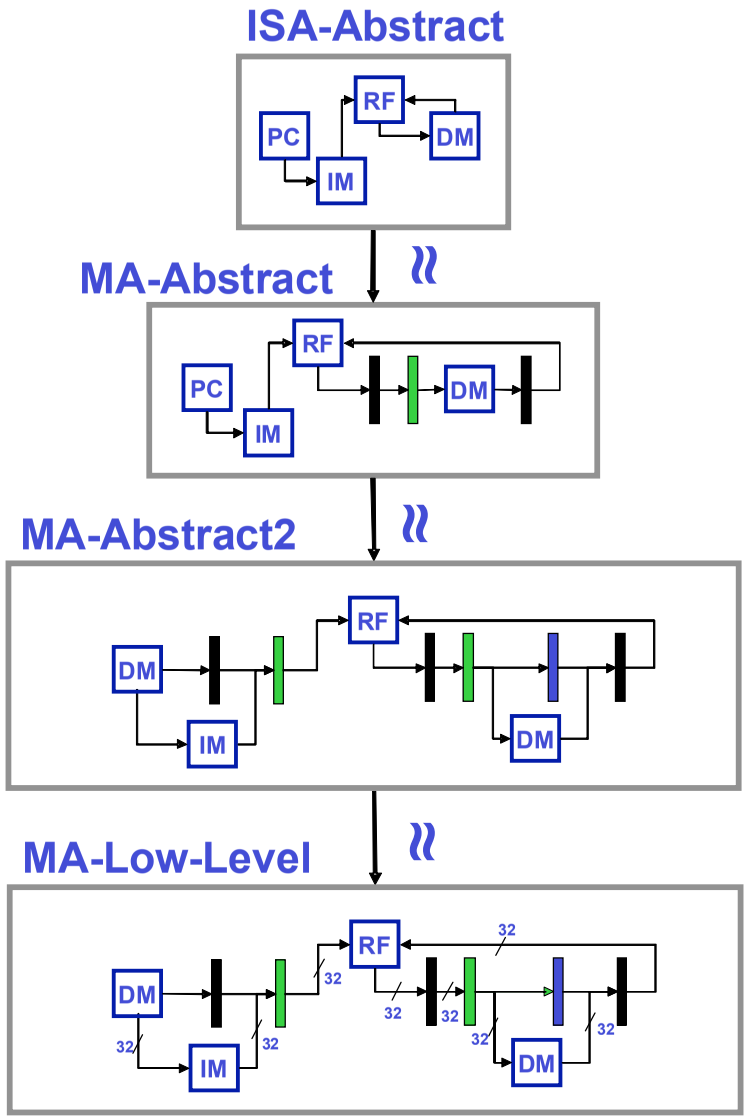
- We modeled the machines using UCLID.
- UCLID compiles to SAT; we used Siege.
- Results: exponential increase in verification time.

| Pipelined Machine | CNF | | Verification Times (sec) | | |
|---|---|---|---|---|---|
| | Vars | Clauses | UCLID | Siege | Total |
| M6 | 28,256 | 83,725 | 8 | 10 | 18 |
| M7 | 53,165 | 158,182 | 15 | 150 | 165 |
| M8 | 95,092 | 283,465 | 25 | 766 | 791 |
| M9 | 144,045 | 429,973 | 41 | 2,436 | 2,477 |
| M10 | 198,375 | 592,660 | 55 | 6,762 | 6,817 |
| M10I | 293,862 | 876,820 | 92 | 8,641 | 8,733 |
| M10ID | 580,355 | 1,730,704 | 244 | Fail | NA |
| M10IDW | 690,598 | 2,060,557 | 297 | Fail | NA |

# Overview

- Refinement.
- Refinement Maps.
- Compositional Reasoning.
- Counterexamples.
- Combining ACL2 & UCLID.
- Future Work.

# Refinement, the Picture




- Formal connection between different abstraction levels.
- Accounts for stuttering.
- Refinement maps.
- Developed general theory.
- Simplified classic results.
- Preservation of safety and liveness (CTL* \ X).
- Compositional.
- Avoid “leaky abstractions.”

# Automating Refinement

- STB refinement follows from the "Core Theorem":

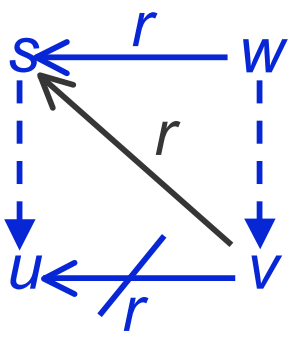
$\langle \forall w \in \text{MA} ::$

$s = r.w \ \wedge \ u = \text{ISA-step}(s) \ \wedge$

$v = \text{MA-step}(w) \ \wedge \ u \neq r.v$

$\Rightarrow$

$s = r.v \ \wedge \ rank.v < rank.w \rangle$




*rank.v < rank.w*

- Theorem is expressible in CLU, a decidable logic.
- Note that MA-step, ISA-step, *r*, and *rank* are complex.

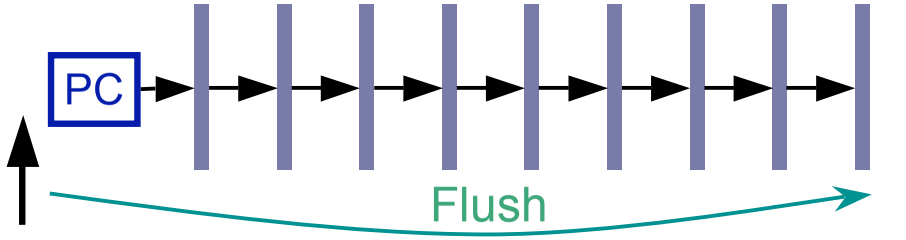
# Refinement Maps & Ranks

- Flushing refinement map:
  - Finish all partially executed instructions.
  - MA-step is used to define flushing.
  - For M10IDW, 14 symbolic simulations required.
  - Consistency invariants required for write-through caches.
- Rank function:
  - Number of steps to fetch an instruction that eventually completes.
- Both can be defined automatically.

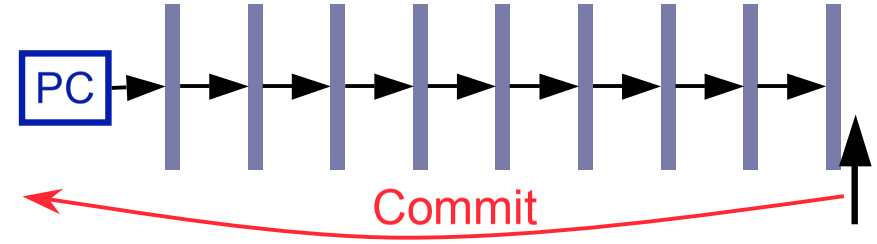
# The Refinement Map Factor

- The refinement maps used can have a drastic impact on verification times.
  - It is possible to attain orders of magnitude improvements in verification times.
  - Can enable the verification of machines that are too complex to otherwise automatically verify.
- Beyond flushing.
  - Commitment (FMCAD 00, DATE 04).
  - GFP (Memocode 05).
  - Intermediate maps (DATE 05).
  - Collapsed flushing (DATE 06).

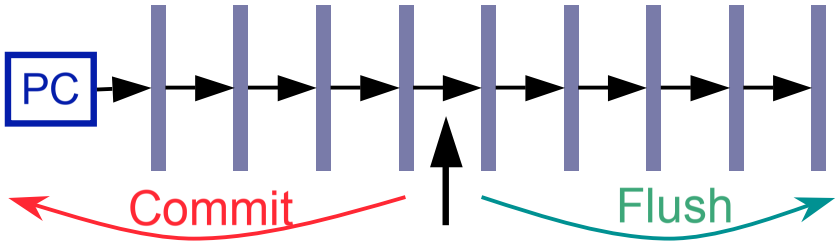
# Intermediate Refinement Maps




IR0: Flush 9 latches,
(Flushing)



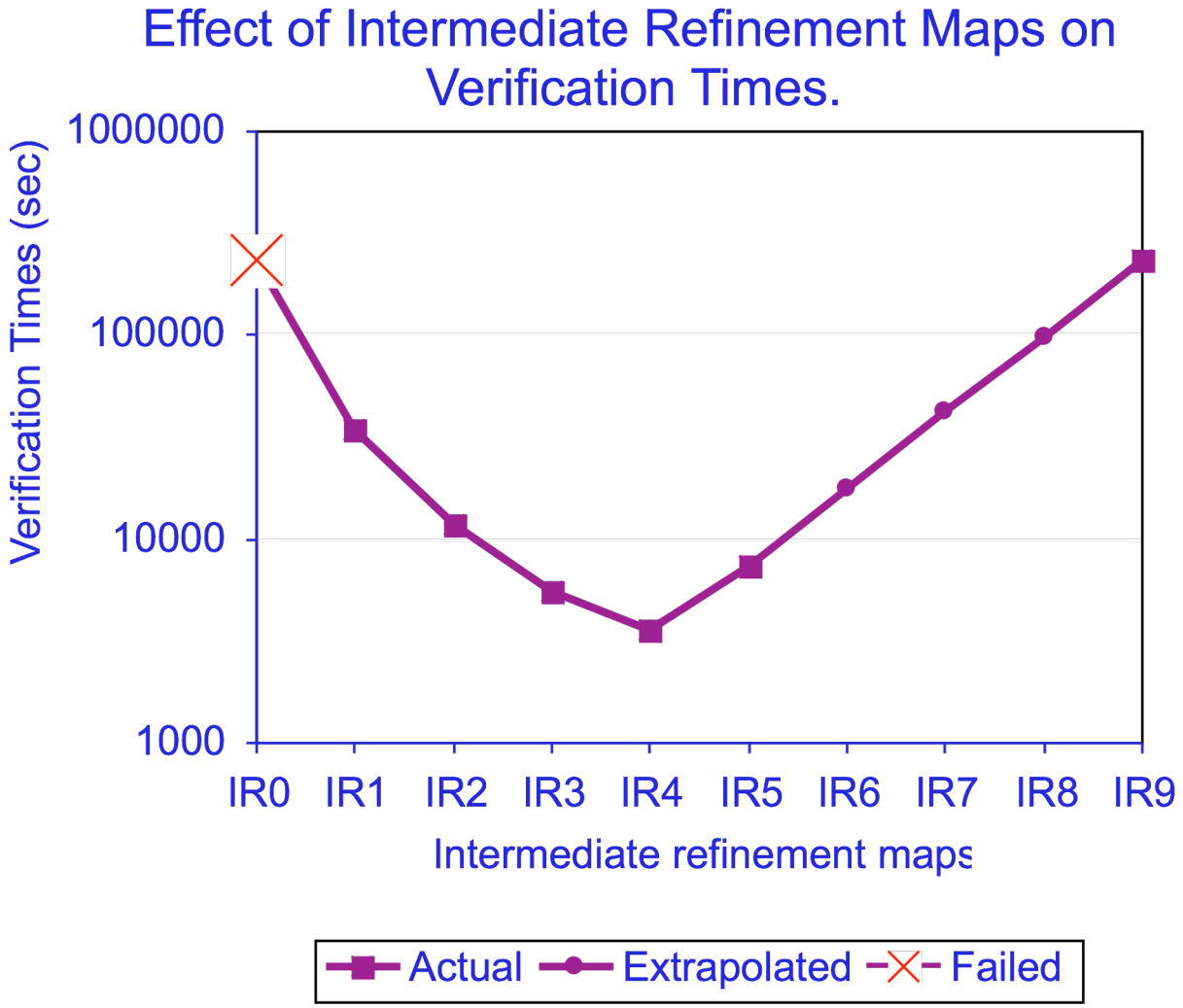

IR9: Commit 9 latches,
(Commitment)




IR4: Commit first 4 latches,
Flush last 5 latches

- Verification time is exponential in the pipeline "complexity": $O(2^c)$.
- For intermediate refinement maps:
  - Complexity of flushing and commitment parts is c/2.
  - Resulting verification time is $O(2^{c/2})$.

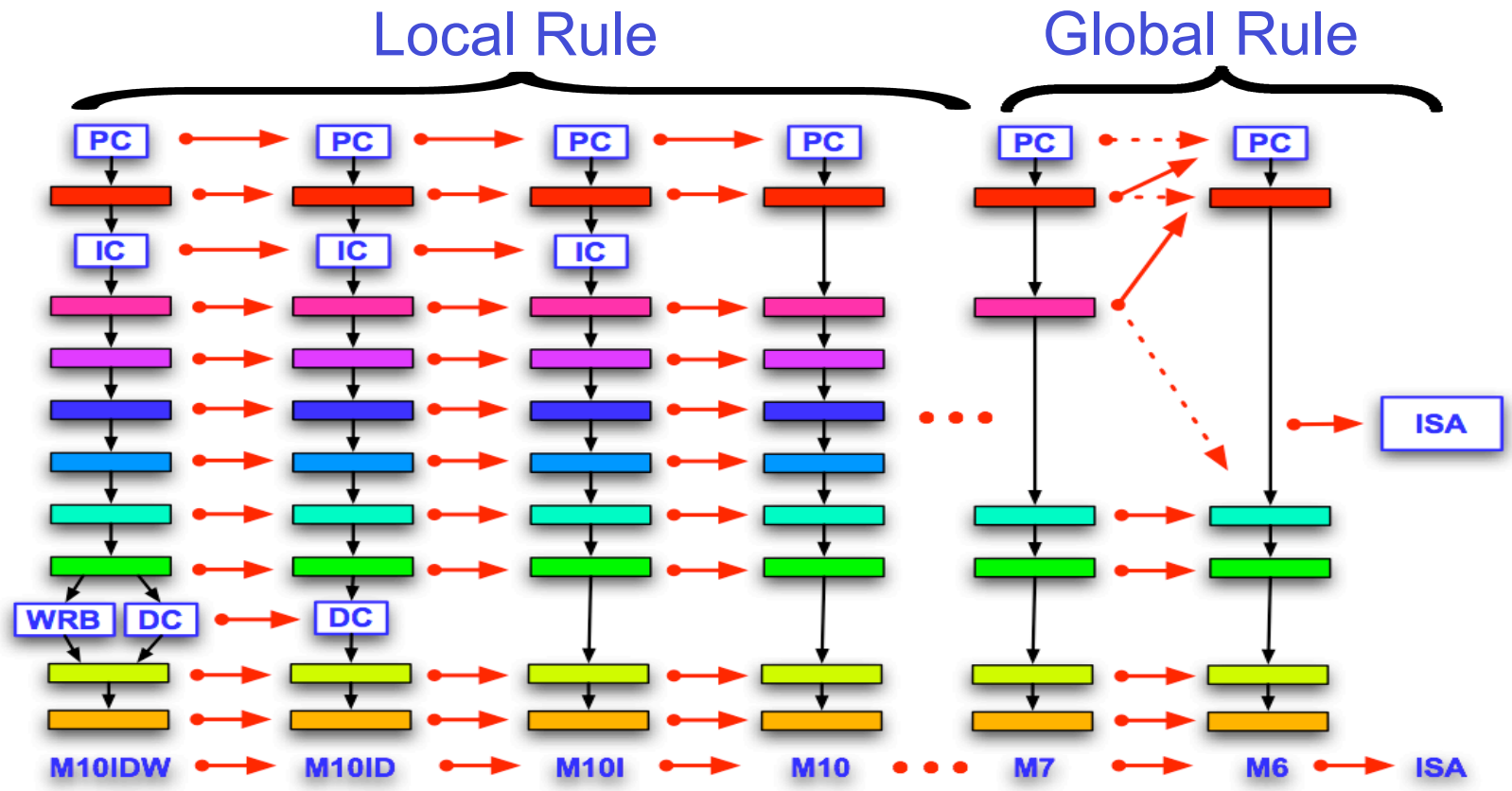
# IR Results


Effect of Intermediate Refinement Maps on Verification Times.
Verification Times (sec)
1000000
100000
10000
1000
IR0
IR1
IR2
IR3
IR4
IR5
IR6
IR7
IR8
IR9
Intermediate refinement maps
Actual
Extrapolated
Failed

# Compositional Verification

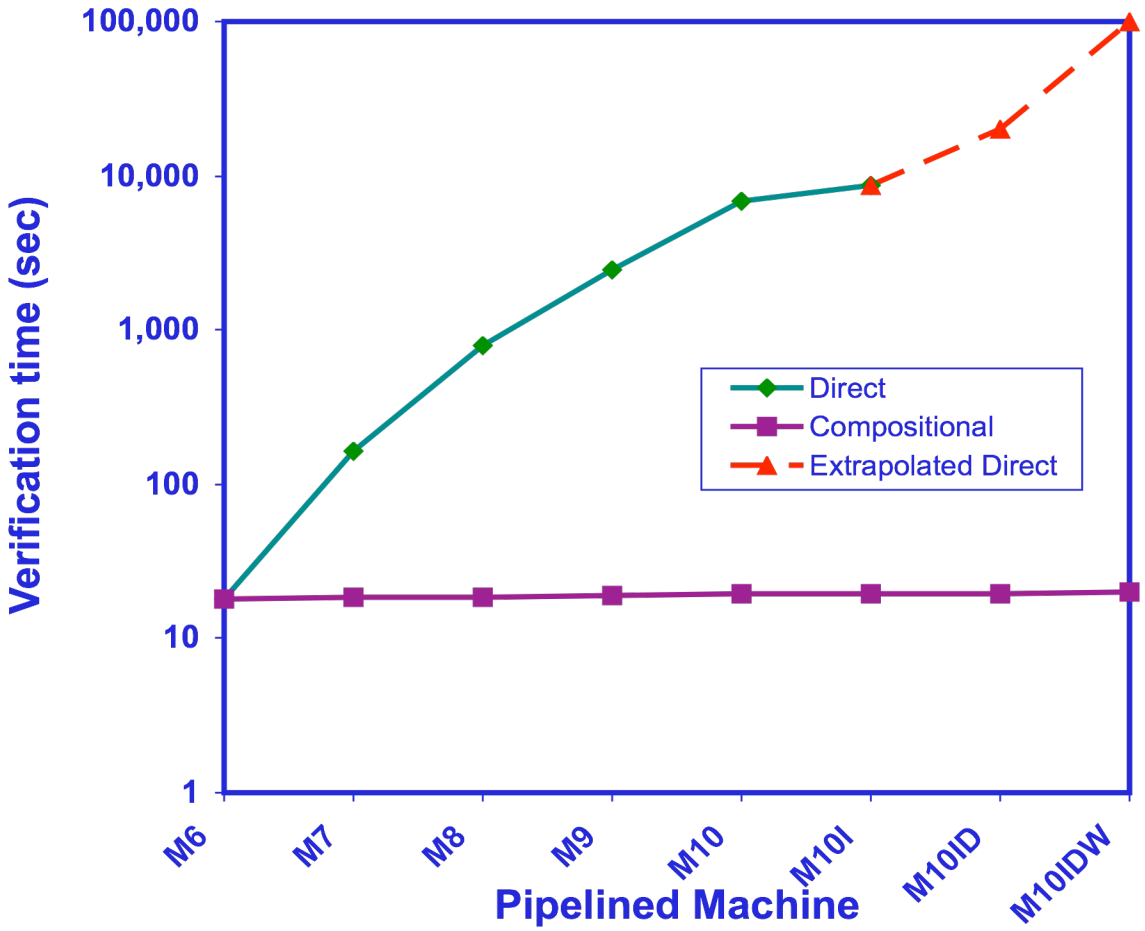
- Idea: verify the machines the way we defined them, one step at a time.
- Developed a complete, compositional framework for pipelined machine verification.
- Preserve safety & liveness.
- M10IDW now takes ~20 seconds to verify!
- Counterexamples tend to be much simpler: they can be isolated to a refinement step.
- Appeared in ICCAD 05.

# Compositional Proof




Refinement maps and ranks are easier to define piecewise.

# Compositional vs. Direct




Why such good results?

Complexity of proof depends on semantic gap between machines.

Our compositional framework makes this gap manageable.

# Counterexamples

- UCLID generates counterexamples when it fails.
  - Understanding counterexamples is hard!
  - Students spend a lot of time on this; prefer code inspection.
- Our framework leads to simpler counterexamples.
  - Occur at the refinement stage where the error appears.
  - Tend to be much smaller in size.
  - Tend to involve less simulation steps.
  - Can be generated more quickly.
- Aids debugging and design understanding.
- Example of a cache error.
  - Invariant: 1/2 second for both approaches.
  - Direct: 69 simulation steps, 6076 lines, 1026 sec.
  - Compositional: 2 simulation steps, 445 lines, <20 sec.

# Combining ACL2 & UCLID

- Idea: Use ACL2 to reduce correctness of bit-level, executable machines to term-level problems which UCLID can handle.
- ACL2 is used to manage the proof process.
- We can state the full refinement theorem.
- We can separate concerns.
  - Models and refinement maps are dealt with separately.
- ACL2 is used to reason at the bit level.
- UCLID is used to reason about the pipeline.
- Result: We can verify executable machines with bit-level interfaces without heroic effort.
- Enables us to reason about machine code running on the pipelined machine.
- See ICCAD 2005.

# Proof Outline

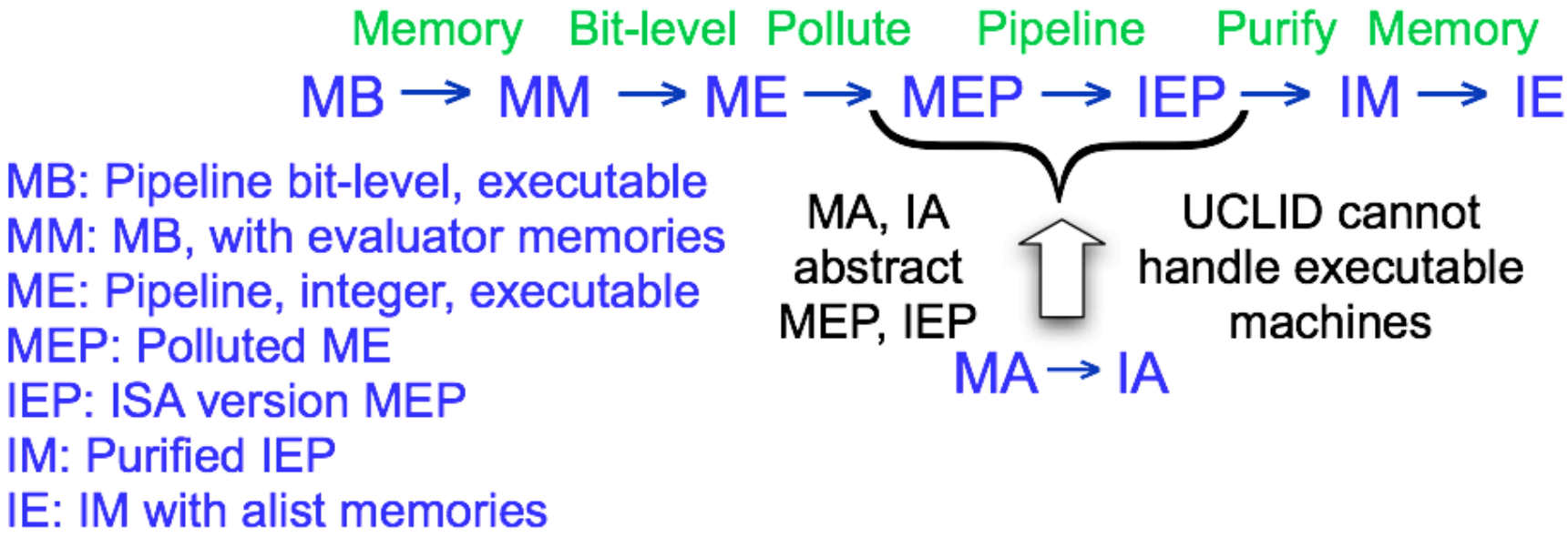
Memory Bit-level Pollute Pipeline Purify Memory

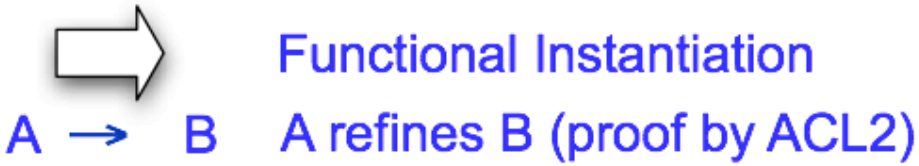
MB → MM → ME → MEP → IEP → IM → IE

MB: Pipeline bit-level, executable
MM: MB, with evaluator memories
ME: Pipeline, integer, executable
MEP: Polluted ME
IEP: ISA version MEP
IM: Purified IEP
IE: IM with alist memories

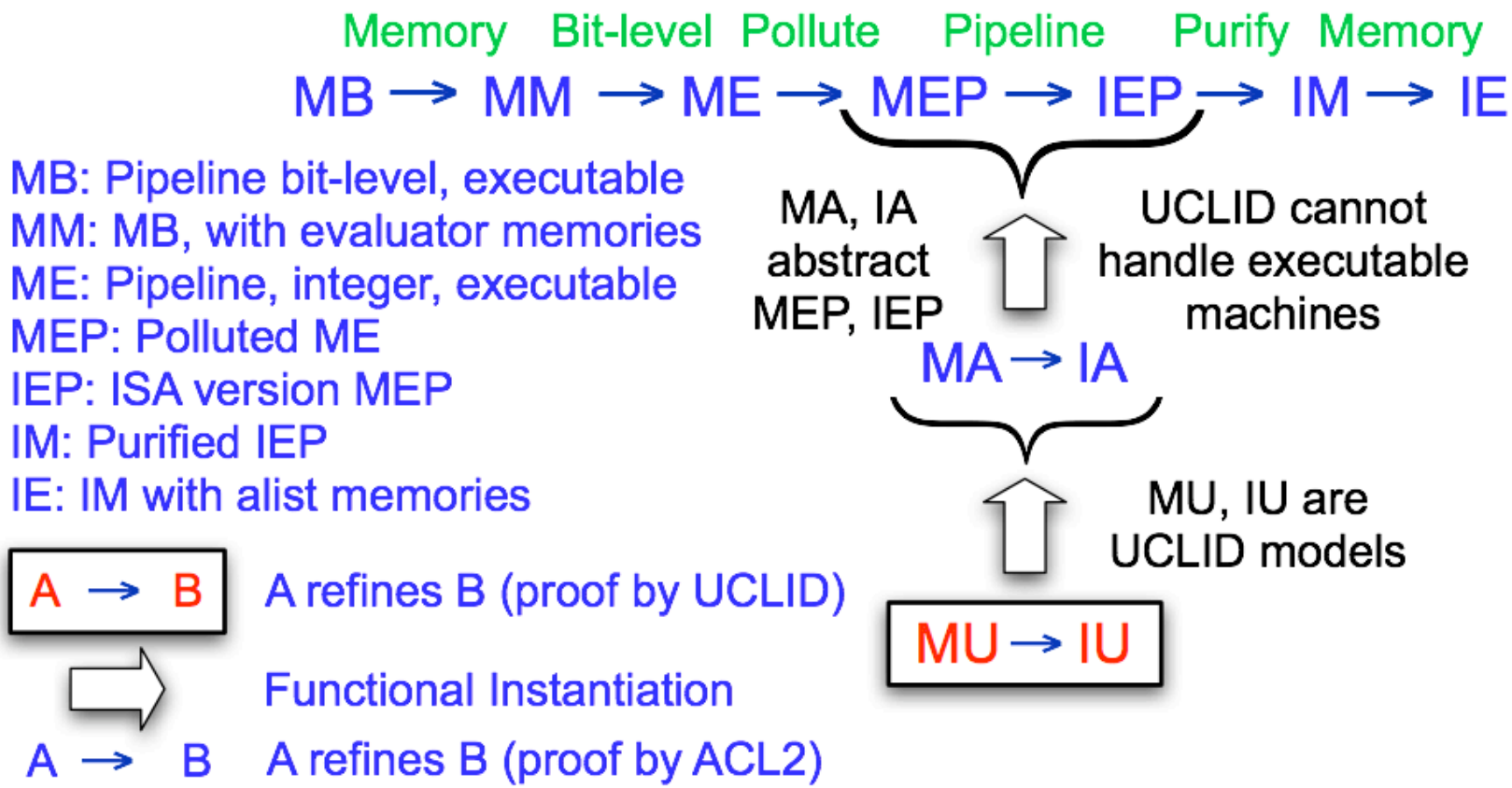
A → B A refines B (proof by ACL2)

# Proof Outline



# Proof Outline



# Verification Statistics

| Proof Step | Proof Time (sec) | User Effort (man-weeks) |
|---|---|---|
| MU → IU | 157 | 3 |
| MA → IA | 91 | 2 |
| MEP → IEP | 36 | 2 |
| IEP → IM | 4 | 1 |
| IM → IE | 601 | 1 |
| ME → MEP | 21 | 2 |
| MM → ME | 182 | 3 |
| MB → MM | 625 | 1 |

Times estimate the effort that would be required for an ACL2 & UCLID expert and do not include the integration effort.

# RTL Verification

- Refinement framework.
  - Formal connection from RTL to abstract models.
  - Parameterized wrt refinement maps .
- Composition plays a major role.
  - Decompose problem into separate tasks.
  - Can be integrated into the design cycle.
- Automation is a major challenge.
  - Even “automatic” methods require human effort.
  - The languages and tools used are important.
  - Understanding counterexamples is important.
- By combining ACL2 & UCLID we avoided heroic effort.
  - We verified the numerous abstractions used in term-level modeling.
  - We were able to relate term-level models with RTL-level designs.
  - Challenge is to reduce ~4x increase in effort to $1+\varepsilon$.

# Future Work

- Automate refinement steps.
  - Use intelligent search.
  - Many steps seem generic, e.g., caches, deep pipelines.
  - Develop a library of patterns.
- Counterexample guided abstraction-refinement.
  - Currently seems to work on simple systems/ shallow properties.
  - How to scale to complex systems & properties?
- Tools that operate directly on HDLs.
  - Current tools support very simple subsets.
  - What about decision procedures that operate directly on HDLs?
- We are currently exploring some of these directions.

# *A Functional HDL in ReFLect*

TOM MELHAM

Computing Laboratory
Oxford University
Wolfson Building
Parks Road
Oxford, OX2 3QD, England

JOHN O'LEARY

Strategic CAD Labs
Intel Corporation
Mail Stop JF4-211
2111 NE 25th Avenue
Hillsboro, OR 97124-5961, USA

*ReFLect* [4] is a functional programming language designed and implemented at Intel's Strategic CAD Labs under the direction of Jim Grundy. The language is strongly typed and similar to ML, but provides certain *reflection* features intended for applications in industrial hardware design and verification. Like LISP, *reFLect* has quotation and antiquotation constructs that may be used to construct and decompose expressions in the language itself. Unlike LISP, these mechanisms are typed. The language also provides a primitive mechanism for pattern-matching, and in particular for defining functions over code by pattern-matching on the structure of *reFLect* expressions. The design of *reFLect* draws on the experience of applying an earlier reflective language called *FL* [1] to large-scale formal verification problems within Intel's Forte framework [8].

One of the intended roles of *reFLect* is to be the host language for a functional HDL. As with other work based on Haskell [2, 7] or LISP [5, 6], a key requirement is the ability to simulate hardware models by program execution. Circuit descriptions are just functional programs, which we can simply run to simulate the circuits on test case inputs. But in addition to this simulation capability, we also wish to execute various operations on the abstract *syntax* of circuit descriptions written in the language. We want to be able to write programs that 'see' the code of a circuit description. This allows us, for example, to program circuit design transformations [10] as functions that traverse code—or simply to generate netlists for other design tools.

This talk at DCC 2006 will illustrate how the reflection features of *reFLect* can provide both simulation and a handle on circuit structure (intensional analysis) within a single, unified language framework. We will present a small HDL embedded within *reFLect*. In the spirit of the approach pioneered by Sheeran in $\mu$FP [9], circuit descriptions are built up in this HDL from primitives using higher-order functions that implement various ways of composing sub-circuits together. The reflection features of *reFLect* will then be employed to allow a single-source circuit description in this language both to be executed for simulation and to be executed to generate circuit netlists.

Lava [2] is an HDL based on Haskell that also supports simulation by execution and circuit netlist generation with a single functional source. Lava achieves this using non-standard interpretation, laziness, and functional data structures with 'observable sharing' [3]. Our presentation at DCC will show how the reflection features of *reFLect* can be employed to achieve single-source simulation and netlist generation in another, perhaps more direct, way. We will also offer some speculations on capabilities available with our approach that seem beyond what can be achieved in Lava—for example the syntactic analysis of circuit descriptions before flattening into netlists.

## References


[1] Mark D. Aagaard, Robert B. Jones, and Carl-Johan H. Seger. *Lifted-FL*: A pragmatic implementation of combined model checking and theorem proving. In Yves Bertot, Gilles Dowek, André Hirschowitz, Christine Paulin, and Laurent Théry, editors, *Theorem Proving in Higher Order Logics: 12th International Conference, TPHOLs 1999*, volume 1690 of *LNCS*, pages 323–340. Springer, 1999.

[2] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: Hardware design in Haskell. In *Functional Programming: International Conference, ICFP 1998*, pages 174–184. ACM, 1998.

[3] Koen Claessen and David Sands. Observable sharing for functional circuit description. In *Advances in Computing Science: 5th Asian Computing Science Conference, ASIAN 1999*, pages 62–73. Springer, 1999.

[4] Jim Grundy, Tom Melham, and John O'Leary. A reflective functional language for hardware design and theorem proving. *Journal of Functional Programming*, 16(2):157–196, March 2006.

[5] Steven D. Johnson. *Synthesis of Digital Designs from Recursion Equations*. MIT, 1984.

[6] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore, editors. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer, 2000.

[7] John Matthews, Byron Cook, and John Launchbury. Microprocessor specification in Hawk. In *Computer Languages: International Conference*, pages 90–101. IEEE Computer Society, 1998.

[8] Carl-Johan H. Seger, Robert B. Jones, John W. O'Leary, Tom Melham, Mark D. Aagaard, Clark Barrett, and Don Syme. An industrially effective environment for formal hardware verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(9):1381–1405, September 2005.

[9] Mary Sheeran. *μFP: An Algebraic VLSI Design Language.* PhD thesis, University of Oxford, 1983.

[10] Greg Spirakis. Leading-edge and future design challenges: Is the classical EDA ready? In *Design Automation: 40th ACM/IEEE Conference, DAC 2003*, page 416. ACM, 2003.

# Towards Automatically Compiling Efficient FPGA Hardware

Jean Baptiste Note, Jean Vuillemin
Ecole Normale Supérieure- Paris

We detail some aspects of our current research on compiling efficient FPGA designs from the source code of data flow applications. The output from our compiler is a FPGA hardware design for the Pamette [1] re-configurable co-processor. Three requirements are met by construction:

1. the source code software specifies the transform applied to the host memory content at each system cycle;
2. the compiled FPGA design bit-wise computes the very same cycle transform- yet at much higher speed thanks to [1];
3. the hardware design area is automatically minimized to meet a throughput requirement, which is specified a-priori within the source code.

Accordingly, the compiler carries its analysis/synthesis in three stages:

1. Unfold to SSA list, perform range analysis, bit-size and translate to RTL design:
    - The input source code is presented here in a C like syntax. In our experimental implementation, the Jazz language [2] is used- such source could be expressed just as well in other synchronous language: Lustre [3], Esterel [4], ...
    - Transform the source code to an equivalent SSA list operating on integers.
    - Analyse the range of each integer variable, and code each by a finite vector of bits; accordingly represent integer operations by Boolean functions.
    - The result is a RTL design equivalent to the bit-sized SSA description.

2. Map RTL to SPF form, re-time and FPGA compile:
    - Map RTL design to Serial/Parallel/Feedback SPF form.
    - Minimize re-timing registers to achieve optimal latency- based on reliable delay models for FPGA integer operators, and less reliable routing models.
    - Produce FPGA design from vendors tools (PamDC [5] and Xilinx [6]) and system software support [5].

3. FPGA design size/power meets set bandwidth requirement:
    - For RTL throughput below requirement, trade area for bandwidth according to [7].
    - For RTL throughput way above requirement, we first generate the Bit-Serial Design Realization: BSDR minimizes logic and throughput among designs.
        i. For BSDR throughput below requirement, unfold space as above.
        ii. For BSDR throughput much above requirement, we fold space at the expense of bandwidth through hyper-serial designs [7].
        iii. For low required throughput, a pure software implementation on the host processor is chosen, without using the co-processor.
        iv. Between these extremes, valuable hardware/software co-design tradeoffs are automatically met.

An experimental validation of the proposed techniques has been obtained.

- Including over half a dozen real-life designs in ***current multi-media***.
- This specific presentation highlights ***dithering in present digital printers***.
- Excluding so far ***automatic bandwidth adaptation***.

We automatically compile *efficient* FPGA designs for applications in the above list:

- *Efficient* means that, the compiled design compared to any hand-crafted for the specific case is no worse, by a factor of two in size/bandwidth/power.
- To permit efficiency, we let each stage automatically performed by the compiler be guided by annotations/pragmas- manually put in the source code.
- A fair measure of our system is thus the number of annotations added to the source code in order to compile an efficient hardware design, as defined above. Our experimental evidence, from all test cases, is: ***very few***!

This talk presents and explains part of the ***theoretical*** and ***experimental*** evidence, in the context of two half-toning algorithms: ***random*** and ***Floyd Steinberg*** digital dithering.

The conclusion from this study is quite **optimistic**: once a *small number* of specific *compiler directives* are added by the *learned designer*, the source code for many current multi-medias applications can be **automatically compiled** into **efficient FPGA** based hardware co-processors. We expect the compiler methodology to extend to other hardware/software technology targets as well- including SIMD machines and multi-core processors.

# Interconnect and Geometric Layout in Hydra (Abstract)

John T. O'Donnell*

Hydra is a functional computer hardware description language that allows circuit designs to be specified either with or without information about the geometric layout. Portions of a design may be specified at different levels of abstraction, and some may have geometric layouts while others do not. This presentation describes how geometric information is incorporated in Hydra circuit specifications, and discusses its interaction with equational reasoning about the behaviour and structure of circuits.

Hydra models every circuit as a function from inputs to outputs. A circuit may be defined directly as a function from signals to signals, or indirectly through the use of functions that generate parameterized circuits or combinators that define families of related circuits. There are also several domain-specific sublanguages, such as a language for designing control algorithms, from which control circuits can be synthesized.

A number of alternative circuit semantics are provided. Some of them are concerned only with behaviour, so that the application of a circuit to suitable inputs will perform a simulation. Others are concerned with structure, so that execution of the same circuit function will generate a netlist. The netlist semantics is based on an algebraic data type for representing the abstract structure of the circuit. The data type contains explicit descriptions of all the components in the circuit, the wires that connect them, and the hierarchical organisation of the circuit specification. It may also contain additional information about the geometric locations of components and wires in a layout, although the geometric information is optional and may be omitted.

A non-geometric circuit specification can use an arbitrary organisation of the input and output ports; they are simply function parameters. A geometric specification, however, treats a circuit as a rectilinear box with a sequence of ports on each of the four edges. Each port is identified as either input or output (there is also an experimental bidirectional port type). The

*Computing Science Department, University of Glasgow. jtod@dcs.gla.ac.uk

circuit function uses only input ports as parameters, and produces results only for the output ports.

Hydra is a functional CHDL, not a relational one, so adjacent circuits must in general be connected, not simply composed. This is performed by a combinator that takes two circuits with geometric layout, and defines a new circuit with the connections completed on their common edge.

Two mechanisms are provided to help the designer specify geometric layouts. Regular layouts can be generated using a set of geometric combinators. Alternatively, the positions of components and wires may be provided explicitly, either by giving their coordinates or by drawing them with a graphical user interface.

The algebraic data type that records the structure of the circuit is implemented via a program transformation. In earlier versions of Hydra the program transformation was performed manually. A new experimental prototype implementation uses Template Haskell to perform the program transformations automatically, making both the geometric combinators and the generation of netlists far more usable.

Hydra is similar in many ways to Lava + Wired. There are also some significant differences. Wired is relational, while Hydra is functional, and the mechanisms used for making connections along the edges are somewhat different. However, the motivations behind Hydra and Lava+Wired appear to be identical.

# The design of a floating point unit using the Integrated Design and Verification (IDV) system.

**Dr. Carl Seger, Strategic CAD Labs, Intel Corp.**

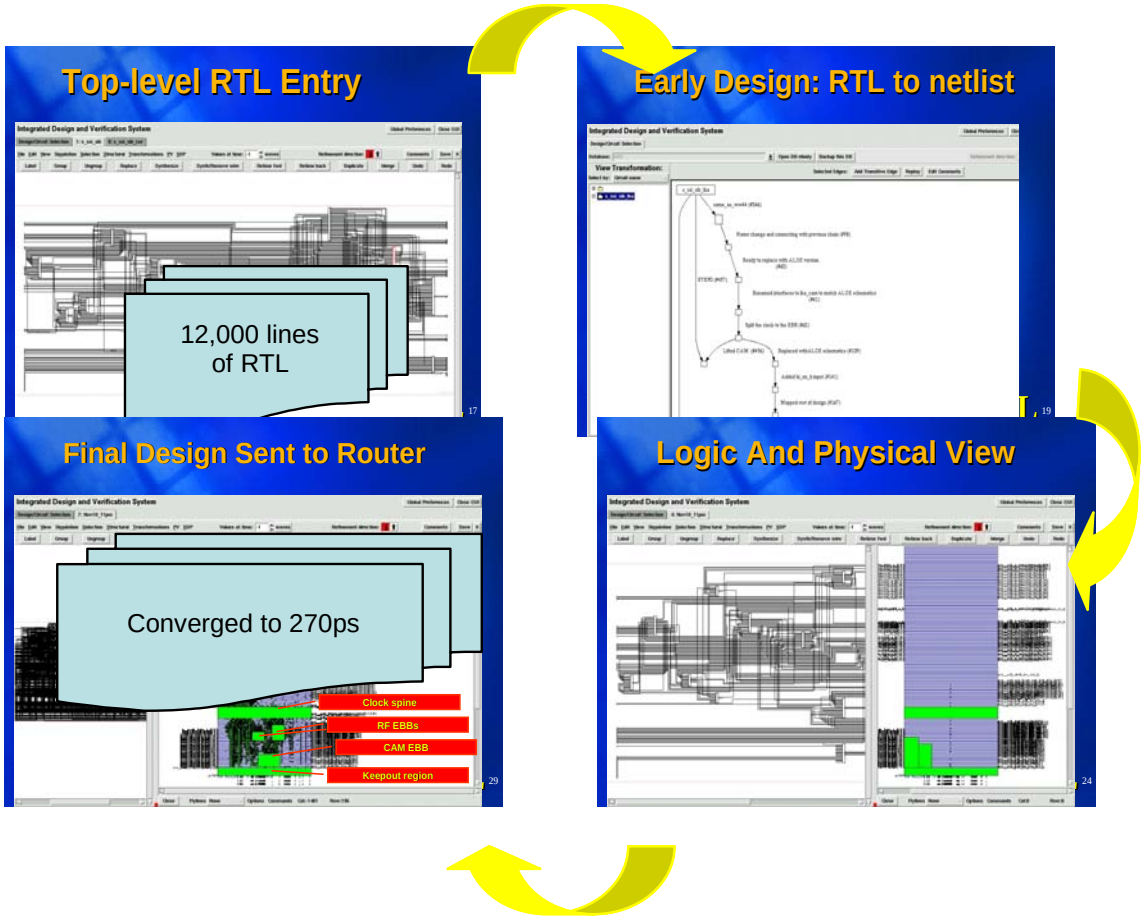
For many VLSI designs, validation has started to dominates the total design effort. In addition, historical trends are indicating that this problem will continue to grow. For example, data from Intel's lead microprocessor design efforts shows that the number of pre-silicon bugs has increased by a factor of four for every lead project for the last 25 years. If this trend is not broken, Intel's next lead design is likely to have to go through the "find the bug, evaluate it, root cause it, fix it, and validate the fix" process tens of thousands of times; potentially overwhelming the validation and design team. Thus one of the most critical goals for improving the design process is to break this bug trend.

In this presentation, we will introduce the Integrated Design and Verification (IDV) system that has been developed at Intel for the last 5 years. IDV combines the design and validation efforts so that the task of design validation (i.e., "Did we capture what we actually wanted?") is significantly simplified by means of a much smaller and much more stable high-level model. Furthermore, when the design is completed, so also is the implementation validation (i.e., "Did we implement what we intended?"). The latter is accomplished by linking the design process very tightly with the validation process. Although this idea is not new, the combination of correct-by-construction and correct-by-verification and the tight integration of a database of verified results is new and has led to a design environment that allows rapid design in which the validation problem has been significantly reduced.

To make the presentation more realistic, we will use the design, from a high-level model to layout, of a floating point execution unit as a driving example. We will discuss the early design

phase in which the high-level model is refined using algorithmic transformations to a viable micro architecture; continue with the middle level design in which the actual logic implementation is derived and conclude with the final placement and layout stage. Although the design process conceptually is performed sequentially, we will illustrate the tight loop that IDV enables between physical design and logical/micro-architectural design. The latter is a critical component in enabling design convergence. In fact, in today's process technology, integration of physical and logical design is not optional but rather mandatory.

Time and facilities permitting, some of the design steps in IDV will be demoed live.



# FAQ for Proof Producing Synthesis in HOL

Konrad Slind, Scott Owens, Juliano Iyoda, Mike Gordon
[Project web page: `http://www.cl.cam.ac.uk/~mjcg/dev/`]

## 1 What is proof producing synthesis?

Proof producing synthesis compiles a source specification (see 2) to an implementation and generates a theorem certifying that the implementation is correct. The specification is expressed in higher order logic.

## 2 What is the synthesisable subset of HOL?

The compiler automatically translates functions $f : \sigma_1 \times \cdots \times \sigma_m \rightarrow \tau_1 \times \cdots \times \tau_n$, where the argument ($\sigma_i$) and result ($\tau_j$) types are words. It can translate any tail recursive definition of such a function as long as the sub-functions used in the definition are in the library of primitive or previously defined functions. Formal refinement into this subset is by proof in the HOL4 system (13, 14, 31, 30, 34 have more discussion and examples).

## 3 Why not verify synthesis functions?

Synthesis functions would need to be coded inside higher order logic if they were to be proved correct. This would be impractical as the compiler uses many HOL4 system tools to automatically infer circuits – it would not be feasible to represent these tools (a substantial chunk of the HOL4 theorem proving infrastructure) in higher order logic.

## 4 Is proof producing synthesis really theorem-proving?

The compiler that synthesises circuits is a derived proof rule in the HOL4 system which is implemented by rewriting and a variety of custom proof strategies. It is a special purpose automatic theorem prover for proving correctness certifying theorems (see 12).

## 5 Is proof producing synthesis the same as formal synthesis?

Proof producing synthesis is a kind of formal synthesis [14] in which the synthesised circuit is not only formally inferred from the specification, but, in addition, a certifying theorem is produced (see 38 also).

## 6 Are there benefits of formal synthesis besides assurance?

Formal synthesis by theorem proving ensures that circuits are correct by construction. Users can safely tinker with the proof scripts used by the compiler, confident that they cannot produce incorrect implementations. Users familiar with the underlying HOL4 theorem proving infrastructure can easily experiment with application-specific extensions or optimisations. An example of an optimisation is combinational inlining (see 26). An example of an extension is `let`-expressions (see 30). Safe extensibility is thus a benefit.

## 7 Why use proof producing synthesis for crypto hardware?

Implementations of cryptographic algorithms are evaluated to a high standard of assurance such as Common Criteria Evaluation Assurance Level 7 (EAL7) [4, 5.9]. Formal methods are an established technique in this area. Proof producing synthesis provides a new way of certifying that cryptographic hardware implements high level specifications.

## 8 Formal synthesis is an old idea, so why is it still interesting?

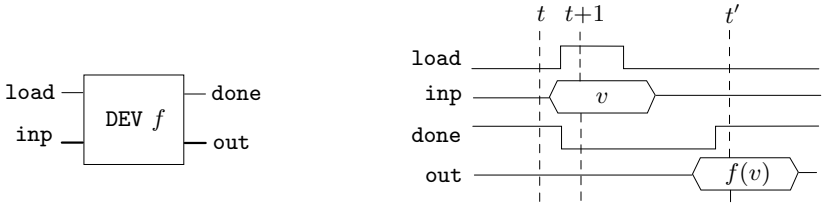
In the past it has been hard to justify the high cost of formal synthesis – the additional confidence of correctness it produces has not been considered worth the expense. However, we think there are niche applications (see 7) where the approach could be cost effective, because it may make it easier to achieve required levels of assurance. Also, we think that safe extensibility (see 6) is a feature that is worth exploring more.

## 9 Is proof producing synthesis automatic?

The synthesis of clocked synchronous circuits from tail recursive definitions of functions mapping words to words is fully automatic. Currently users must manually refine specifications that use general recursion schemes to tail recursive form (there is a tool in development to automatically compile linear recursion to tail recursion). Data refinement from functions operating on types other than words must also be done manually.

## 10 What is the hardware realisation of a HOL function?

A function $f$ defined in higher order logic is realised by a device `DEV` $f$ that computes $f$ via a four-phase handshake circuit on signals `load`, `inp`, `done` and `out`.




At the start of a transaction (say at time $t$) the device must be outputting T on `done` (to indicate it is ready) and the environment must be asserting F on `load`, i.e. in a state such that a positive edge on `load` can be generated. A transaction is initiated by asserting (at time $t+1$) the value T on `load`, i.e. `load` has a positive edge at time $t+1$. This causes the device to read the value, $v$ say, input on `inp` (at time $t+1$) and to set done to F. The device then becomes insensitive to inputs until T is next asserted on done, when the computed value $f(v)$ will be output on `out`. See 33 for an example.

## 11 What is the formal specification of a handshake device?

The specification of the four-phase handshake protocol is represented by the definition of the predicate `DEV`, which uses auxiliary predicates Posedge and HoldF. A positive

edge of a signal is defined as the transition of its value from low to high, i.e. from F to T. The formula HoldF $(t_1, t_2)$ $s$ says that a signal $s$ holds a low value F during a half-open interval starting at $t_1$ to just before $t_2$. The formal definitions are:

$\vdash$ Posedge $s\ t$ $\quad = \texttt{if}\ t{=}0\ \texttt{then F else}\ (\neg\, s(t{-}1)\ \wedge\ s\ t\ )$
$\vdash$ HoldF $(t_1, t_2)\ s$ $\quad = \forall t.\ t_1 \leq t < t_2\ \Rightarrow\ \neg(s\ t)$

The behaviour of the handshaking device computing a function $f$ is described by the term DEV $f$ $(load, inp, done, out)$ where:

$\vdash$ DEV $f$ $(load, inp, done, out) =$
$\quad (\forall t.\ done\ t\ \wedge$ Posedge $load$ $(t{+}1)$
$\qquad \Rightarrow$
$\qquad \exists t'.\ t' > t{+}1\ \wedge$ HoldF $(t{+}1, t')$ $done\ \wedge$
$\qquad\quad done\ t'\ \wedge\ (out\ t' = f(inp\ (t{+}1))))\ \wedge$
$\quad (\forall t.\ done\ t\ \wedge\ \neg(\text{Posedge}\ load\ (t{+}1))\ \Rightarrow\ done\ (t{+}1))\ \wedge$
$\quad (\forall t.\ \neg(done\ t)\ \Rightarrow\ \exists t'.\ t' > t\ \wedge done\ t')$

The first conjunct in the right-hand side specifies that if the device is available and a positive edge occurs on $load$, there exists a time $t'$ in future when $done$ signals its termination and the output is produced. The value of the output at time $t'$ is the result of applying $f$ to the value of the input at time $t{+}1$. The signal $done$ holds the value F during the computation. The second conjunct specifies the situation where no call is made on $load$ and the device simply remains idle. Finally, the last conjunct states that if the device is busy, it will eventually finish its computation and become idle.

# 12 What is the form of a correctness certifying theorem?

Synthesising a circuit implementing $f : \sigma_1 \times \cdots \times \sigma_m \rightarrow \tau_1 \times \cdots \times \tau_n$ (see 2) proves:

```
|- InfRise clk
    ==> CIR_f
    ==> DEV f (load at clk, inputs at clk, done at clk, outputs at clk)
```

$\texttt{CIR}_f$ is a formula representing a circuit containing variables `clk`, `load`, `inp1`, ..., `inpm` representing inputs and variables `done`, `out1`, ..., `outn` representing outputs. The type of `inp`$i$ matches $\sigma_i$ and the type of `out`$j$ matches $\tau_j$. `InfRise clk` asserts that clock `clk` has infinitely many rising edges. See 31 and 34 for examples. The term *inputs* stands for `inp1<>`$\cdots$`<>inpm` which is the concatenation of the variables `inp1`, ... , `inpm` using the word concatenation operator <> and *outputs* is `out1<>`$\cdots$`<>outn` representing the concatenation of the output variables. A term $s$ `at clk` is the temporal projection of signal $s$ at rising edges of `clk` (see 15, 21).

# 13 Are specifications using high level datatypes synthesisable?

During synthesis the compiler generates circuits that use polymorphic registers and combinational components (i.e. components having inputs and outputs of arbitrary types). However, the lower level phases instantiate all types to words (currently represented as lists of bits). Thus we can generate circuits with wires carrying abstract values (e.g. numbers), but these cannot be refined to a form that can be input to FPGA tools. Our intention is that users will derive Boolean level specifications inside higher order logic using data-refinement methods. Automating this is a possible future direction.

# 14 Are there tools to translate into the synthesisable subset?

There is an experimental proof producing tool called `linRec` that translates linear recursions to tail recursions. For example it translates:

```
FACT n = if n = 0 then 1 else Mult(n, FACT(n-1))
```

to:

```
FactIter(n,acc) = if n = 0 then (n,acc) else FactIter(n-1,Mult(n,acc)))
Fact n = SND(FactIter (n,1))
```

# 15 What hardware components are used in circuits?

The compiler generates circuits built from a user-specifiable library of combinational components, e.g. `AND`, `OR`, `NOT`, `MUX`, `ADD` (the default library is chosen for use with the Quartus II FPGA software). Synthesised circuits may also contain constants (`CONSTANT`), edge-triggered D-type registers with unspecified initial state (`Dtype`) and Dtypes that power up into an initial state storing the value T (`DtypeT`). Constants and the registers are specified in higher order logic by:

$$\begin{array}{ll} \mathsf{CONSTANT}\ v\ out & = \forall t.\ out(t) = v \\ \mathsf{Dtype}\ \ (clk, d, q) & = \forall t.\ q(t{+}1) = \texttt{if}\ \mathsf{Rise}\ clk\ t\ \texttt{then}\ d\ t\ \texttt{else}\ q\ t \\ \mathsf{DtypeT}(clk, d, q) & = (q\ 0 = \mathsf{T})\ \wedge\ \mathsf{Dtype}(clk, d, q) \end{array}$$

where Rise $s\ t$ means signal $s$ has a rising edge starting at time $t$:

$$\mathsf{Rise}\ s\ t\ =\ \neg s(t)\ \wedge\ s(t{+}1)$$

Both `Dtype` and `DtypeT` are implemented in Verilog by instantiating a single generic register module `dtype` that is parametised on its size and initial stored value (see 32).

# 16 How much do you rely on untrustworthy FPGA tools?

The 'sign-off' from logic to EDA tools occurs at the clocked synchronous RTL level (see 15). A circuit $\texttt{CIR}_f$ (see 12) is translated to Verilog using a pretty-printer written in ML, and this Verilog is then fed to FPGA tools (e.g. Quartus II). FPGA implementations thus rely on the Verilog pretty-printer and the subsequent industrial tools. Higher assurance could be gained by taking the proof producing synthesis to a lower level (e.g. to an FPGA netlist language).

# 17 Can users control how specifications are synthesised?

The architecture of synthesised circuits reflects the input specification, so can be tuned by adjusting the higher order logic source. For example, using `let`-expressions (see 30) prevents logic blocks from being duplicated. There are also user-settable parameters: for example, modules can be inserted directly as combinational logic (i.e. without an enclosing handshake interface) if they are declared "combinational" (see 26, 28).

# 18 How efficient is proof producing synthesis?

Because synthesisers invokes a theorem prover it is relatively slow. Simple one-line examples take a few seconds on a standard workstation, bigger examples take several minutes.

# 19 How fast are synthesised circuits?

Some simple experiments comparing synthesised and hand coded circuits suggest that performance is not too bad, but we do not have solid evidence. However, the user has some control over the amount of computation per clock cycle via a facility to declare functions to be inlined as combinational logic, rather that via a handshake (see 28). We think the approach will support the creation of optimised implementations (necessary for crypto applications), but so far the emphasis has been on proof of concept.

# 20 How large are synthesised circuits?

Some of the bigger examples we have synthesised did not at first fit onto the FPGAs we are using. Whole program compaction (see 28) and use of `let` (see 30) solved the problem for these examples, but we still worry that the circuits are too big.

# 21 How does the hardware compiler work?

The operation of the compiler can be decomposed into four phases.

1. Translate $\forall x_1 \ldots x_n.\ f(x_1, \ldots, x_n) = e$ to an equivalent equation $f = \mathcal{E}$, where the expression, $\mathcal{E}$ is built from combinators Seq (compute in sequence), Par (compute in parallel), Ite (if-then-else) and Rec (recursion).
2. Replace the combinators Seq, Par, Ite and Rec with corresponding circuit constructors SEQ, PAR, ITE and REC to create a circuit term and a theorem that this implements DEV $f$.
3. Replace circuit constructors with cycle-level implementations and prove a theorem that the resulting design implements DEV $f$.
4. Introduce a clock and perform temporal projection [10] from cycle level to a clocked RTL circuit and prove a theorem that the resulting RTL circuit implements DEV $f$.

# 22 What are the combinators Seq, Par, Ite and Rec?

Seq, Par, Ite and Rec are used to build the combinatory expression $\mathcal{E}$ that is generated when translating $\forall x_1 \ldots x_n.\ f(x_1, \ldots, x_n) = e$ to $f = \mathcal{E}$. They are defined by:

$$\begin{array}{ll}
\mathsf{Seq}\ f_1\ f_2 & = \lambda x.\ f_2(f_1\ x) \\
\mathsf{Par}\ f_1\ f_2 & = \lambda x.\ (f_1\ x,\ f_2\ x) \\
\mathsf{Ite}\ f_1\ f_2\ f_3 & = \lambda x.\ \texttt{if}\ f_1\ x\ \texttt{then}\ f_2\ x\ \texttt{else}\ f_3\ x \\
\mathsf{Rec}\ f_1\ f_2\ f_3 & = \lambda x.\ \texttt{if}\ f_1\ x\ \texttt{then}\ f_2\ x\ \texttt{else}\ \mathsf{Rec}\ f_1\ f_2\ f_3\ (f_3\ x)
\end{array}$$

For example:

$$\begin{array}{l}
\vdash\ \mathsf{FactIter}(n, acc)\ = \\
\quad \texttt{if}\ n = 0\ \texttt{then}\ (n, acc)\ \texttt{else}\ \mathsf{FactIter}(n - 1, n \times acc)
\end{array}$$

is translated to:

$$\begin{array}{l}
\vdash\ \mathsf{FactIter}\ = \\
\quad \mathsf{Rec}\ (\mathsf{Seq}\ (\mathsf{Par}\ (\lambda(n, acc).\ n)\ (\lambda(n, acc).\ 0))\ (=)) \\
\qquad (\mathsf{Par}\ (\lambda(n, acc).\ n)\ (\lambda(n, acc).\ acc)) \\
\qquad (\mathsf{Par}\ (\mathsf{Seq}\ (\mathsf{Par}\ (\lambda(n, acc).\ n)\ (\lambda(n, acc).\ 1))\ (-)) \\
\qquad\quad (\mathsf{Seq}\ (\mathsf{Par}\ (\lambda(n, acc).\ n)\ (\lambda(n, acc).\ acc))\ (\times)))
\end{array}$$

# 23 What components do circuit constructors use?

The circuit constructors are built using the following components, which are represented at an unclocked cycle level of abstraction.

$\vdash$ AND $(in_1, in_2, out) = \forall t.\ out\ t = (in_1\ t\ \wedge\ in_2\ t)$
$\vdash$ OR $(in_1, in_2, out) = \forall t.\ out\ t = (in_1\ t\ \vee\ in_2\ t)$
$\vdash$ NOT $(inp, out) = \forall t.\ out\ t = \neg(inp\ t)$
$\vdash$ MUX$(sw, in_1, in_2, out) = \forall t.\ out\ t = \texttt{if}\ sw\ t\ \texttt{then}\ in_1\ t\ \texttt{else}\ in_2\ t$
$\vdash$ COMB $f\ (inp, out) = \forall t.\ out\ t = f(inp\ t)$
$\vdash$ DEL $(inp, out) = \forall t.\ out(t+1) = inp\ t$
$\vdash$ DELT $(inp, out) = (out\ 0 = \mathsf{T})\ \wedge\ \forall t.\ out(t+1) = inp\ t$
$\vdash$ DFF$(d, sel, q) = \forall t.\ q(t+1) = \texttt{if}$ Posedge $sel\ (t+1)\ \texttt{then}\ d(t+1)\ \texttt{else}\ q\ t$
$\vdash$ POSEDGE$(inp, out) = \exists c_0\ c_1.$ DELT$(inp, c_0) \wedge$ NOT$(c_0, c_1) \wedge$ AND$(c_1, inp, out)$

# 24 What are the circuit constructors SEQ, PAR, ITE and REC?

SEQ, PAR, ITE and REC are circuit constructors that implement Seq, Par, Ite and Rec, respectively (see 22). They construct circuits that combine delay elements with combinational logic. The delay elements are refined to clocked synchronous registers.

The circuit constructors are defined in higher order logic below. The components they use are defined in 15 and schematic diagrams of the implementations are in 29.

Sequential composition of handshaking devices.

$\vdash$ SEQ $f\ g\ (load, inp, done, out) =$
$\exists c_0\ c_1\ c_2\ c_3\ data.$
NOT$(c_2, c_3)\ \wedge$ OR$(c_3, load, c_0)\ \wedge\ f(c_0, inp, c_1, data) \wedge$
$g(c_1, data, c_2, out)\ \wedge$ AND$(c_1, c_2, done)$

Parallel composition of handshaking devices.

$\vdash$ PAR $f\ g\ (load, inp, done, out) =$
$\exists c_0\ c_1\ start\ done_1\ done_2\ data_1\ data_2\ out_1\ out_2.$
POSEDGE$(load, c_0)\ \wedge$ DEL$(done, c_1)\ \wedge$ AND$(c_0, c_1, start) \wedge$
$f(start, inp, done_1, data_1)\ \wedge\ g(start, inp, done_2, data_2) \wedge$
DFF$(data_1, done_1, out_1)\ \wedge$ DFF$(data_2, done_2, out_2) \wedge$
AND$(done_1, done_2, done)\ \wedge\ (out = \lambda t.\ (out_1\ t, out_2\ t))$

Conditional composition of handshaking devices.

$\vdash$ ITE $e\ f\ g\ (load, inp, done, out) =$
$\exists c_0\ c_1\ c_2\ start\ start'\ done\_e\ data\_e\ q\ not\_e\ data\_f\ data\_g\ sel$
$done\_f\ done\_g\ start\_f\ start\_g.$
POSEDGE$(load, c_0)\ \wedge$ DEL$(done, c_1)\ \wedge$ AND$(c_0, c_1, start) \wedge$
$e(start, inp, done\_e, data\_e)\ \wedge$ POSEDGE$(done_e, start') \wedge$
DFF$(data_e, done_e, sel)\ \wedge$ DFF$(inp, start, q) \wedge$
AND$(start', data_e, start_f)\ \wedge$ NOT$(data_e, not_e) \wedge$
AND$(start', not_e, start_g)\ \wedge\ f(start_f, q, done_f, data_f) \wedge$
$g(start\_g, q, done\_g, data\_g)\ \wedge$ MUX$(sel, data_f, data_g, out) \wedge$
AND$(done_e, done_f, c_2)\ \wedge$ AND$(c_2, done_g, done)$

Tail recursion constructor.

$\vdash$ REC $e\ f\ g\ (load, inp, done, out) =$
$\exists done\_g\ data\_g\ start\_e\ q\ done\_e\ data\_e\ start\_f\ start\_g\ inp\_e\ done\_f$
$c_0\ c_1\ c_2\ c_3\ c_4\ start\ sel\ start'\ not\_e.$
POSEDGE$(load, c_0)\ \wedge$ DEL$(done, c_1)\ \wedge$ AND$(c_0, c_1, start)\ \wedge$
OR$(start, sel, start_e)\ \wedge$ POSEDGE$(done_g, sel)\ \wedge$
MUX$(sel, data_g, inp, inp_e)\ \wedge$ DFF$(inp_e, start_e, q)\ \wedge$
$e(start\_e, inp\_e, done\_e, data\_e)\ \wedge$ POSEDGE$(done_e, start')\ \wedge$
AND$(start', data_e, start_f)\ \wedge$ NOT$(data_e, not_e)\ \wedge$
AND$(not_e, start', start_g)\ \wedge\ f(start_f, q, done_f, out)\ \wedge$
$g(start\_g, q, done\_g, data\_g)\ \wedge$ DEL$(done_g, c_3)\ \wedge$
AND$(done_g, c_3, c_4)\ \wedge$ AND$(done_f, done_e, c_2)\ \wedge$ AND$(c_2, c_4, done)$

# 25 How are combinational circuits represented?

A function $f$ can be packaged as a handshaking device with constructor ATM:

$\vdash$ ATM $f\ (load, inp, done, out) =$
$\exists c_0\ c_1.$ POSEDGE$(load, c_0) \wedge$ NOT$(c_0, done) \wedge$
COMB $f\ (inp, c_1) \wedge$ DEL$(c_1, out)$

This creates a simple handshake interface that computes $f$ and satisfies the refinement theorem: $\vdash\ \forall f.$ ATM $f\ \Longrightarrow$ DEV $f$.

Formulas of the form COMB $g\ (inp, out)$ are compiled into circuits built only using components in user-supplied library of predefined circuits. The default library currently includes Boolean functions (e.g. $\wedge$, $\vee$ and $\neg$), multiplexers and simple operations on $n$-bit words (e.g. versions of $+$, $-$ and $<$, various shifts etc.). A special purpose proof rule uses a recursive algorithm to synthesise combinational circuits. For example:

$\vdash$ COMB $(\lambda(m, n).\ (m < n,\ \ m + 1))\ (inp_1 \text{<>} inp_2, out_1 \text{<>} out_2)\ =$
$\exists v_0.$ COMB $(<)\ (inp_1 \text{<>} inp_2, out_1)\ \wedge$ CONSTANT $1\ v_0\ \wedge$
COMB $(+)\ (inp_1 \text{<>} v_0, out_2)$

where <> is bus concatenation, CONSTANT $1\ v_0$ drives $v_0$ high continuously, and COMB $<$ and COMB $+$ are assumed given components (if they were not given, then they could be implemented explicitly, but one has to stop somewhere).

# 26 How is an explosion of internal handshakes avoided?

When processing Seq $f_1\ f_2$ (see 27), the compiler checks to see whether $f_1$ or $f_2$ are compositions of combinational functions and if so introduces PRECEDE or FOLLOW instead of SEQ, using the theorems:

$\vdash\ (P\ \Longrightarrow$ DEV $f_2)\ \Rightarrow$ (PRECEDE $f_1\ P\ \Longrightarrow$ DEV (Seq $f_1\ f_2$))
$\vdash\ (P\ \Longrightarrow$ DEV $f_1)\ \Rightarrow$ (FOLLOW $P\ f_2\ \Longrightarrow$ DEV (Seq $f_1\ f_2$))

where PRECEDE $f\ d$ processes inputs with $f$ before sending them to $d$ and FOLLOW $d\ f$ processes outputs of $d$ with $f$. The definitions are:

PRECEDE $f\ d\ (load, inp, done, out) = \exists v.$ COMB $f\ (inp, v)\ \wedge\ d(load, v, done, out)$
FOLLOW $d\ f\ (load, inp, done, out)\ = \exists v.\ d(load, inp, done, v)\ \wedge$ COMB $f\ (v, out)$

SEQ $d_1\ d_2$ introduces a handshake between the executions of $d_1$ and $d_2$, but PRECEDE $f\ d$ and FOLLOW $d\ f$ just 'wire' $f$ before or after $d$ without introducing a handshake.

# 27 How are the circuit constructors introduced?

The following theorems enable the compiler to compositionally deduce theorems of the form $\vdash Imp \implies \mathsf{DEV}\ f$, where $Imp$ is a formula constructed using the circuit constructors. The long arrow symbol $\implies$ denotes implication lifted to functions:
$f \implies g \ = \ \forall \mathit{load\ inp\ done\ out}.\ f(\mathit{load}, \mathit{inp}, \mathit{done}, \mathit{out}) \Rightarrow g(\mathit{load}, \mathit{inp}, \mathit{done}, \mathit{out})$.

$\vdash\ (P_1 \implies \mathsf{DEV}\ f_1) \wedge (P_2 \implies \mathsf{DEV}\ f_2)$
$\quad \Rightarrow\ (\mathsf{SEQ}\ P_1\ P_2 \implies \mathsf{DEV}\ (\mathsf{Seq}\ f_1\ f_2))$

$\vdash\ (P_1 \implies \mathsf{DEV}\ f_1) \wedge (P_2 \implies \mathsf{DEV}\ f_2)$
$\quad \Rightarrow\ (\mathsf{PAR}\ P_1\ P_2 \implies \mathsf{DEV}\ (\mathsf{Par}\ f_1\ f_2))$

$\vdash\ (P_1 \implies \mathsf{DEV}\ f_1) \wedge (P_2 \implies \mathsf{DEV}\ f_2) \wedge (P_3 \implies \mathsf{DEV}\ f_3)$
$\quad \Rightarrow\ (\mathsf{ITE}\ P_1\ P_2\ P_3 \implies \mathsf{DEV}\ (\mathsf{Ite}\ f_1\ f_2\ f_3))$

$\vdash\ \mathsf{Total}(f_1, f_2, f_3)$
$\quad \Rightarrow\ (P_1 \implies \mathsf{DEV}\ f_1) \wedge (P_2 \implies \mathsf{DEV}\ f_2) \wedge (P_3 \implies \mathsf{DEV}\ f_3)$
$\quad \Rightarrow\ (\mathsf{REC}\ P_1\ P_2\ P_3 \implies \mathsf{DEV}\ (\mathsf{Rec}\ f_1\ f_2\ f_3))$

The predicate Total is defined so that $\mathsf{Total}(f_1, f_2, f_3)$ ensures termination.

If $\mathcal{E}$ is an expression built using Seq, Par, Ite and Rec, then by instantiating the predicate variables $P_1$, $P_2$ and $P_3$, these theorems enable a logic formula $\mathcal{F}$ to be built from circuit constructors SEQ, PAR, ITE and REC such that $\vdash \mathcal{F} \implies \mathsf{DEV}\ \mathcal{E}$. We have $\vdash f = \mathcal{E}$ (see 21, phase 1), hence $\vdash \mathcal{F} \implies \mathsf{DEV}\ f$. This is the basic idea, but see also 26, 28.

# 28 What optimisation does the compiler perform?

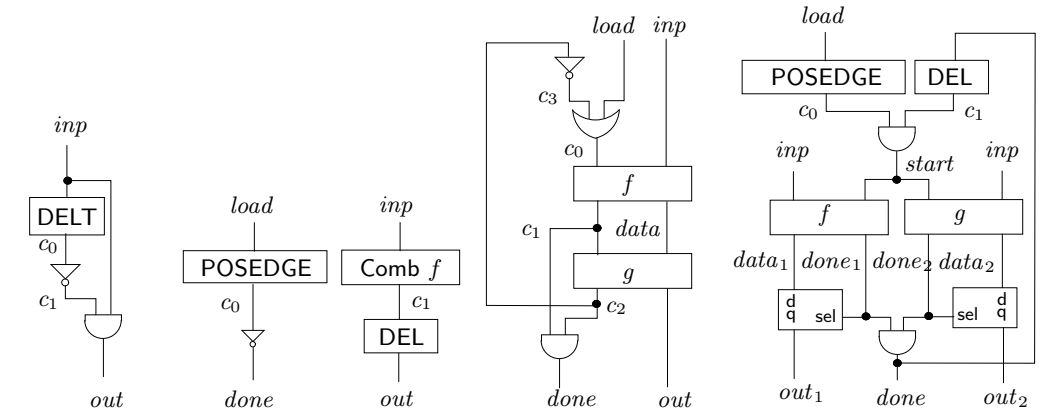
There are currently two main optimisations used in synthesis: reducing handshakes between implementations of functions (see 26) and whole program compaction.

The normal synthesis of Seq $f_1\ f_2$ (see 22) is to SEQ $d_1\ d_2$ (see 24), where the circuit combinator SEQ puts a handshake between the device $d_1$ implementing $f_1$ and the device $d_2$ implementing $f_2$. Users may declare $f_1$ to be 'combinational' and then a combinational logic block $c_1$ implementing $f_1$ is synthesised and put in series before $d_2$. Similarly $f_2$ can be declared combinational and then combinational logic $c_2$ will be put in series after $d_1$. Only components that can be realised using known combinational logic blocks can be declared combinational. Using this mechanism, all the computation from the arguments of a function to its recursive call can be synthesised as combinational logic, so there is just a single handshake to manage the iteration.

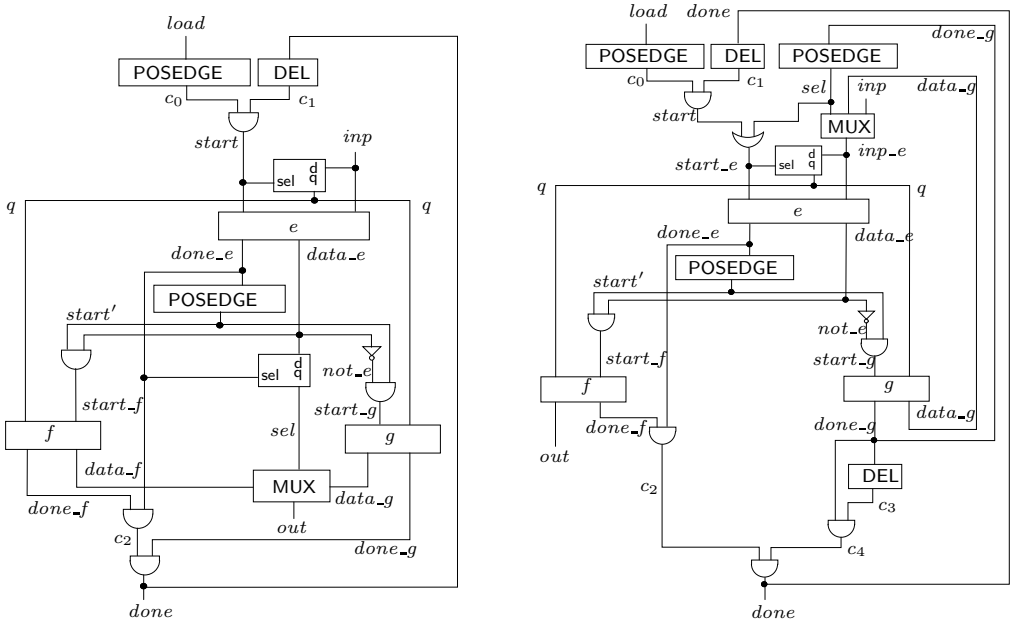
This optimisation is restricted to the scope of a single function. However, before applying this method, we can inline the function calls to produce a system defined by a single function (the function 'main'). The whole program compaction eliminates unnecessary handshake circuits that could have been generated to implement the function calls.

In practise these techniques can generate long logic paths (i.e. slow clocks), so some pipelining via internal handshakes can be appropriate.

# 29 What do the circuit constructor implementations look like?




(a) POSEDGE (b) ATM $f$ (c) SEQ $f$ $g$ (d) PAR $f$ $g$




(e) IF $e$ $f$ $g$ (f) REC $e$ $f$ $g$

# 30 How do `let`-expressions work?

A `let`-expression has the form `let` $v$ `=` $e_1$ `in` $e_2$ where $v$ is a "varstruct" (variable structure) which is either a single variable or, recursively, a non-empty tuple of varstructs (e.g. `(x,(m,n),y)`).

If $e_1$ is combinational, then a `let`-expression is synthesised into a circuit consisting of $e_1$ driving wires corresponding to $v$ that are inputs to the circuit corresponding to $e_2$.

If $e_1$ is not combinational, the `let`-expression is compiled using:

$\vdash\ \forall f_1\ f_2.\ (\lambda x.\ \texttt{let}\ v\ =\ f_1\ x\ \texttt{in}\ f_2\ (x, v))\ =\ \mathsf{Seq}\ (\mathsf{Par}\ (\lambda x.\ x)\ f_1)\ f_2$

For example, suppose H and J are defined by:

```
H x = x+1w
J x = let y = H x in y + y + y
```

where 1w is the 32-bit word denoting 1 and + is 32-bit addition.

If H is not declared combinational, then J compiles to:

```
|- InfRise clk ==>
   (∃v0 v1 v2 v3 v4 v5 v6 v7 v8 v9 v10 v11 v12 v13 v14 v15 v16 v17 v18 v19 v20 v21
      v22 v23 v24 v25 v26 v27.
      DtypeT (clk,load,v10) ∧ NOT (v10,v9) ∧ AND (v9,load,v8) ∧ Dtype (clk,done,v7) ∧
      AND (v8,v7,v6) ∧ DtypeT (clk,v6,v13) ∧ NOT (v13,v12) ∧ AND (v12,v6,v11) ∧
      NOT (v11,v5) ∧ Dtype (clk,inp,v3) ∧ DtypeT (clk,v6,v17) ∧ NOT (v17,v16) ∧
      AND (v16,v6,v15) ∧ NOT (v15,v4) ∧ CONSTANT 1w v18 ∧ ADD32 (inp,v18,v14) ∧
      Dtype (clk,v14,v2) ∧ DtypeT (clk,v5,v21) ∧ NOT (v21,v20) ∧ AND (v20,v5,v19) ∧
      MUX (v19,v3,v22,v1) ∧ Dtype (clk,v1,v22) ∧ DtypeT (clk,v4,v25) ∧ NOT (v25,v24) ∧
      AND (v24,v4,v23) ∧ MUX (v23,v2,v26,v0) ∧ Dtype (clk,v0,v26) ∧ AND (v5,v4,done) ∧
      ADD32 (v0,v0,v27) ∧ ADD32 (v27,v0,out)) ==>
   DEV J (load at clk,inp at clk,done at clk,out at clk)
```

but if H is declared to be combinational, then J compiles to:

```
|- InfRise clk ==>
   (∃v0 v1 v2 v3 v4 v5 v6.
      DtypeT (clk,load,v3) ∧ NOT (v3,v2) ∧ AND (v2,load,v1) ∧ NOT (v1,done) ∧
      CONSTANT 1w v5 ∧ ADD32 (inp,v5,v4) ∧ ADD32 (v4,v4,v6) ∧ ADD32 (v6,v4,v0) ∧
      Dtype (clk,v0,out)) ==>
   DEV J (load at clk,inp at clk,done at clk,out at clk)
```

# 31 What is a simple example?

A simple example is iterative accumulator-style multiplication on 32-bit words:

```
Mult32Iter(m,n,acc) =
 if m = 0w then (0w, n, acc) else Mult32Iter(m-1w, n, n+acc)
```

where 0w, 1w are 32-bit numbers and +, − are 32-bit operations.

This specification compiles to:

```
|- InfRise clk ==>
  (∃ v0 v1 v2 v3 v4 v5 v6 v7 v8 v9 v10 v11 v12 v13 v14 v15 v16 v17 v18 v19 v20 v21 v22 v23
     v24 v25 v26 v27 v28 v29 v30 v31 v32 v33 v34 v35 v36 v37 v38 v39 v40 v41 v42 v43 v44
     v45 v46 v47 v48 v49 v50 v51 v52 v53 v54 v55 v56 v57.
      DtypeT (clk,load,v21) ∧ NOT (v21,v20) ∧ AND (v20,load,v19) ∧ Dtype (clk,done,v18) ∧
      AND (v19,v18,v17) ∧ OR (v17,v16,v11) ∧ DtypeT (clk,v15,v23) ∧ NOT (v23,v22) ∧
      AND (v22,v15,v16) ∧ MUX (v16,v14,inp1,v3) ∧ MUX (v16,v13,inp2,v2) ∧
      MUX (v16,v12,inp3,v1) ∧ DtypeT (clk,v11,v26) ∧ NOT (v26,v25) ∧ AND (v25,v11,v24) ∧
      MUX (v24,v3,v27,v10) ∧ Dtype (clk,v10,v27) ∧ DtypeT (clk,v11,v30) ∧ NOT (v30,v29) ∧
      AND (v29,v11,v28) ∧ MUX (v28,v2,v31,v9) ∧ Dtype (clk,v9,v31) ∧ DtypeT (clk,v11,v34) ∧
      NOT (v34,v33) ∧ AND (v33,v11,v32) ∧ MUX (v32,v1,v35,v8) ∧ Dtype (clk,v8,v35) ∧
      DtypeT (clk,v11,v39) ∧ NOT (v39,v38) ∧ AND (v38,v11,v37) ∧ NOT (v37,v7) ∧
      CONSTANT 0w v40 ∧ EQ32 (v3,v40,v36) ∧ Dtype (clk,v36,v6) ∧ DtypeT (clk,v7,v44) ∧
      NOT (v44,v43) ∧ AND (v43,v7,v42) ∧ AND (v42,v6,v5) ∧ NOT (v6,v41) ∧ AND (v41,v42,v4) ∧
      DtypeT (clk,v5,v48) ∧ NOT (v48,v47) ∧ AND (v47,v5,v46) ∧ NOT (v46,v0) ∧
      CONSTANT 0w v45 ∧ Dtype (clk,v45,out1) ∧ Dtype (clk,v9,out2) ∧ Dtype (clk,v8,out3) ∧
      DtypeT (clk,v4,v53) ∧ NOT (v53,v52) ∧ AND (v52,v4,v51) ∧ NOT (v51,v15) ∧
      CONSTANT 1w v54 ∧ SUB32 (v10,v54,v50) ∧ ADD32 (v9,v8,v49) ∧ Dtype (clk,v50,v14) ∧
      Dtype (clk,v9,v13) ∧ Dtype (clk,v49,v12) ∧ Dtype (clk,v15,v56) ∧ AND (v15,v56,v55) ∧
      AND (v0,v7,v57) ∧ AND (v57,v55,done)) ==>
  DEV Mult32Iter (load at clk, (inp1<>inp2<>inp3) at clk, done at clk, (out1<>out2<>out3) at clk)
```

See 12 and 15 for explanations and 32 for the Verilog that is extracted from this circuit.

# 32 What is the generated Verilog like?

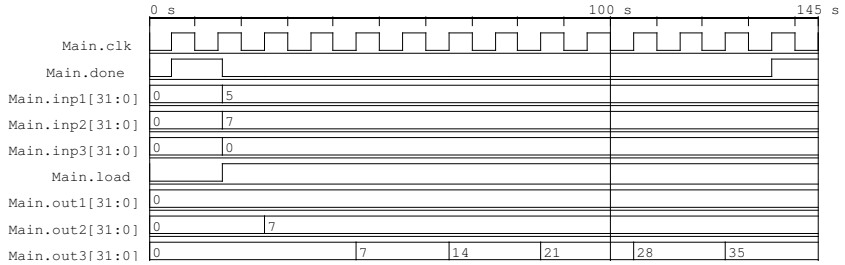
The iterative accumulator-style multiplication device (see 31) generates the following Verilog.

```
module dtype (clk,d,q);
 parameter size = 31; parameter value = 1;
 input clk; input [size:0] d; output [size:0] q; reg [size:0] q = value;

 always @(posedge clk) q <= d;

endmodule

module Mult32Iter (clk,load,inp1,inp2,inp3,done,out1,out2,out3);
 input clk,load; input [31:0] inp1; input [31:0] inp2; input [31:0] inp3;
 output done; output [31:0] out1; output [31:0] out2; output [31:0] out3;
 wire clk,done; wire [0:0] v0; wire [31:0] v1; wire [31:0] v2; wire [31:0] v3; wire [0:0] v4;
 wire [0:0] v5; wire [0:0] v6; wire [0:0] v7; wire [31:0] v8; wire [31:0] v9; wire [31:0] v10;
 wire [0:0] v11; wire [31:0] v12; wire [31:0] v13; wire [31:0] v14; wire [0:0] v15;
 wire [0:0] v16; wire [0:0] v17; wire [0:0] v18; wire [0:0] v19; wire [0:0] v20; wire [0:0] v21;
 wire [0:0] v22; wire [0:0] v23; wire [0:0] v24; wire [0:0] v25; wire [0:0] v26; wire [31:0] v27;
 wire [0:0] v28; wire [0:0] v29; wire [0:0] v30; wire [31:0] v31; wire [0:0] v32; wire [0:0] v33;
 wire [0:0] v34; wire [31:0] v35; wire [0:0] v36; wire [0:0] v37; wire [0:0] v38; wire [0:0] v39;
 wire [31:0] v40; wire [0:0] v41; wire [0:0] v42; wire [0:0] v43; wire [0:0] v44; wire [31:0] v45;
 wire [0:0] v46; wire [0:0] v47; wire [0:0] v48; wire [31:0] v49; wire [31:0] v50; wire [0:0] v51;
 wire [0:0] v52; wire [0:0] v53; wire [31:0] v54; wire [0:0] v55; wire [0:0] v56; wire [0:0] v57;

 dtype   dtype_0 (clk,load,v21);   defparam dtype_0.size = 0;
 assign v20 = ~ v21;
 assign v19 = v20 && load;
 dtype   dtype_1 (clk,done,v18);   defparam dtype_1.size = 0;
 assign v17 = v19 && v18;
 assign v11 = v17 || v16;
 dtype   dtype_2 (clk,v15,v23);    defparam dtype_2.size = 0;
 assign v22 = ~ v23;
 assign v16 = v22 && v15;
 assign v3 = v16 ? v14 : inp1;
 assign v2 = v16 ? v13 : inp2;
 assign v1 = v16 ? v12 : inp3;
 dtype   dtype_3 (clk,v11,v26);    defparam dtype_3.size = 0;
 assign v25 = ~ v26;
 assign v24 = v25 && v11;
 assign v10 = v24 ? v3 : v27;
 dtype   dtype_4 (clk,v10,v27);    defparam dtype_4.size = 31;
 dtype   dtype_5 (clk,v11,v30);    defparam dtype_5.size = 0;
 assign v29 = ~ v30;
 assign v28 = v29 && v11;
 assign v9 = v28 ? v2 : v31;
 dtype   dtype_6 (clk,v9,v31);     defparam dtype_6.size = 31;
 dtype   dtype_7 (clk,v11,v34);    defparam dtype_7.size = 0;
 assign v33 = ~ v34;
 assign v32 = v33 && v11;
 assign v8 = v32 ? v1 : v35;
 dtype   dtype_8 (clk,v8,v35);     defparam dtype_8.size = 31;
 dtype   dtype_9 (clk,v11,v39);    defparam dtype_9.size = 0;
 assign v38 = ~ v39;
 assign v37 = v38 && v11;
 assign v7 = ~ v37;
 assign v40 = 0;
 assign v36 = v3 == v40;
 dtype   dtype_10 (clk,v36,v6);    defparam dtype_10.size = 0;
 dtype   dtype_11 (clk,v7,v44);    defparam dtype_11.size = 0;
 assign v43 = ~ v44;
 assign v42 = v43 && v7;
 assign v5 = v42 && v6;
 assign v41 = ~ v6;
 assign v4 = v41 && v42;
 dtype   dtype_12 (clk,v5,v48);    defparam dtype_12.size = 0;
 assign v47 = ~ v48;
 assign v46 = v47 && v5;
 assign v0 = ~ v46;
 assign v45 = 0;
 dtype   dtype_13 (clk,v45,out1);  defparam dtype_13.size = 31;
 dtype   dtype_14 (clk,v9,out2);   defparam dtype_14.size = 31;
 dtype   dtype_15 (clk,v8,out3);   defparam dtype_15.size = 31;
 dtype   dtype_16 (clk,v4,v53);    defparam dtype_16.size = 0;
 assign v52 = ~ v53;
 assign v51 = v52 && v4;
 assign v15 = ~ v51;
 assign v54 = 1;
 assign v50 = v10 - v54;
 assign v49 = v9 + v8;
 dtype   dtype_17 (clk,v50,v14);   defparam dtype_17.size = 31;
 dtype   dtype_18 (clk,v9,v13);    defparam dtype_18.size = 31;
 dtype   dtype_19 (clk,v49,v12);   defparam dtype_19.size = 31;
 dtype   dtype_20 (clk,v15,v56);   defparam dtype_20.size = 0;
 assign v55 = v15 && v56;
 assign v57 = v0 && v7;
 assign done = v57 && v55;

endmodule
```

To conserve space, all comments and many line breaks have been removed from the preceding Verilog. Each Verilog statement is printed with a comment showing the HOL source to aid visual checking (e.g. for Common Criteria EAL7 certification evaluators [4, 5.9]). We are still tinkering with the Verilog: for example, experiments show that Quartus II configures Altera FPGAs to initialise faster with registers as instances of a separate module (`dtype` above) than with inlined behavioral statements `always @(posedge clk) q <= d`, where `q` is initialised with declaration `reg q = 1`.

# 33 What simulation tools have you used?

We currently use Icarus Verilog (`http://www.icarus.com`) for simulation and then view waveforms with GTKWave (`http://home.nc.rr.com/gtkwave`). These tools are both public domain. If we simulate the `Mult32Iter` example (see 31) with inputs $(5, 7, 0)$, then the resulting waveform is:




`load` is asserted at time 15 and `done` is T then, but `done` immediately drops to F in response to `load` being asserted. At the same time as `load` is asserted the values $5$, $7$ and $0$ are put on lines `inp1`, `inp2` and `inp3`, respectively. At time 135 `done` rises to T again, and by then the values on `out1`, `out2` and `out3` are $0$, $7$ and $35$, respectively, thus `Mult32Iter(5,7,0) = (0,7,35)`, which is correct.

# 34 What is a bigger example?

An example drawn from cryptography is the TEA block cipher [17]. The encryption algorithm is described by the following HOL definitions (all variables are 32-bit words):

```
TEAEncrypt (keys,txt) = FST(Rounds (32,(txt,keys,0)))

Rounds (n,s) = if n=0 then s else Rounds(n-1, Round s)

Round ((y,z),(k0,k1,k2,k3),s) =
  let s' = s + DELTA in
  let t = y + ShiftXor(z,s',k0,k1)
  in
     ((t, z + ShiftXor(t,s',k2,k3)), (k0,k1,k2,k3), s')

ShiftXor (x,s,k0,k1) = ((x << 4) + k0) # (x + s) # ((x >> 5) + k1)

DELTA = 0x9e3779b9w
```

There is a corresponding `TEADecrypt` function (omitted). In HOL-4 we can prove functional correctness:

```
|- ∀plaintext keys. TEADecrypt(keys,TEAEncrypt(keys,plaintext)) = plaintext
```

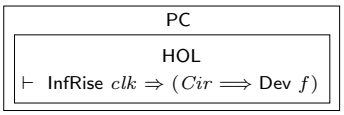
The compiler generates the following netlist for `TEAEncrypt` and also one for `TEADecrypt`:

```
|- InfRise clk ==>
    (∃v0 v1 v2 v3 v4 v5 v6 v7 v8 v9 v10 v11 v12 v13 v14 v15 v16 v17 v18
     v19 v20 v21 v22 v23 v24 v25 v26 v27 v28 v29 v30 v31 v32 v33 v34 v35
     v36 v37 v38 v39 v40 v41 v42 v43 v44 v45 v46 v47 v48 v49 v50 v51 v52
     v53 v54 v55 v56 v57 v58 v59 v60 v61 v62 v63 v64 v65 v66 v67 v68 v69
     v70 v71 v72 v73 v74 v75 v76 v77 v78 v79 v80 v81 v82 v83 v84 v85 v86
     v87 v88 v89 v90 v91 v92 v93 v94 v95 v96 v97 v98 v99 v100 v101 v102
     v103 v104 v105 v106 v107 v108 v109 v110 v111 v112 v113 v114 v115 v116
     v117 v118 v119 v120 v121 v122 v123 v124 v125 v126 v127 v128 v129 v130
     v131 v132 v133 v134 v135 v136 v137 v138 v139 v140 v141 v142 v143 v144
     v145 v146 v147 v148 v149 v150 v151 v152 v153 v154 v155 v156 v157 v158
     v159 v160 v161 v162 v163 v164 v165 v166 v167 v168 v169 v170 v171 v172
     v173 v174 v175 v176 v177 v178 v179 v180 v181 v182 v183 v184 v185 v186
     v187 v188 v189 v190 v191 v192 v193 v194 v195 v196 v197 v198 v199 v200
     v201 v202 v203 v204 v205 v206 v207 v208 v209 v210 v211 v212 v213 v214
     v215 v216 v217 v218 v219 v220 v221 v222 v223 v224 v225 v226 v227 v228
     v229 v230 v231 v232 v233 v234 v235 v236 v237 v238 v239 v240 v241 v242
     v243 v244 v245 v246 v247 v248 v249 v250 v251 v252 v253 v254 v255 v256
     v257 v258 v259 v260 v261 v262 v263 v264 v265 v266 v267 v268 v269 v270
     v271 v272 v273 v274 v275 v276 v277 v278 v279 v280 v281 v282 v283 v284
     v285 v286 v287 v288 v289 v290 v291 v292 v293 v294 v295 v296 v297 v298
     v299 v300 v301 v302 v303 v304 v305 v306 v307 v308 v309 v310 v311 v312
     v313 v314 v315 v316 v317 v318 v319 v320 v321 v322 v323 v324 v325 v326
     v327 v328 v329 v330 v331 v332 v333 v334 v335 v336 v337 v338 v339 v340
     v341 v342 v343 v344 v345 v346 v347 v348 v349 v350 v351 v352 v353 v354.
       CONSTANT 32w v6 ∧ CONSTANT 0w v5 ∧ DtypeT (clk,load,v43) ∧
       NOT (v43,v42) ∧ AND (v42,load,v41) ∧ Dtype (clk,done,v40) ∧
       AND (v41,v40,v39) ∧ OR (v39,v38,v28) ∧ DtypeT (clk,v37,v45) ∧
       NOT (v45,v44) ∧ AND (v44,v37,v38) ∧ MUX (v38,v36,v6,v15) ∧
       MUX (v38,v35,inp2,v14) ∧ MUX (v38,v34,inp3,v13) ∧ MUX (v38,v33,inp11,v12) ∧
       MUX (v38,v32,inp12,v11) ∧ MUX (v38,v31,inp13,v10) ∧ MUX (v38,v30,inp14,v9) ∧
       MUX (v38,v29,v5,v8) ∧ DtypeT (clk,v28,v48) ∧ NOT (v48,v47) ∧
       AND (v47,v28,v46) ∧ MUX (v46,v15,v49,v27) ∧ Dtype (clk,v27,v49) ∧
       DtypeT (clk,v28,v52) ∧ NOT (v52,v51) ∧ AND (v51,v28,v50) ∧
       MUX (v50,v14,v53,v26) ∧ Dtype (clk,v26,v53) ∧ DtypeT (clk,v28,v56) ∧
       NOT (v56,v55) ∧ AND (v55,v28,v54) ∧ MUX (v54,v13,v57,v25) ∧
       Dtype (clk,v25,v57) ∧ DtypeT (clk,v28,v60) ∧ NOT (v60,v59) ∧
       AND (v59,v28,v58) ∧ MUX (v58,v12,v61,v24) ∧ Dtype (clk,v24,v61) ∧
       DtypeT (clk,v28,v64) ∧ NOT (v64,v63) ∧ AND (v63,v28,v62) ∧
       MUX (v62,v11,v65,v23) ∧ Dtype (clk,v23,v65) ∧ DtypeT (clk,v28,v68) ∧
       NOT (v68,v67) ∧ AND (v67,v28,v66) ∧ MUX (v66,v10,v69,v22) ∧
       Dtype (clk,v22,v69) ∧ DtypeT (clk,v28,v72) ∧ NOT (v72,v71) ∧
       AND (v71,v28,v70) ∧ MUX (v70,v9,v73,v21) ∧ Dtype (clk,v21,v73) ∧
       DtypeT (clk,v28,v76) ∧ NOT (v76,v75) ∧ AND (v75,v28,v74) ∧
       MUX (v74,v8,v77,v20) ∧ Dtype (clk,v20,v77) ∧ DtypeT (clk,v28,v81) ∧
       NOT (v81,v80) ∧ AND (v80,v28,v79) ∧ NOT (v79,v19) ∧ CONSTANT 0w v82 ∧
       EQ32 (v15,v82,v78) ∧ Dtype (clk,v78,v18) ∧ DtypeT (clk,v19,v86) ∧
       NOT (v86,v85) ∧ AND (v85,v19,v84) ∧ AND (v84,v18,v17) ∧ NOT (v18,v83) ∧
       AND (v83,v84,v16) ∧ DtypeT (clk,v17,v89) ∧ NOT (v89,v88) ∧ AND (v88,v17,v87) ∧
       NOT (v87,v7) ∧ Dtype (clk,v26,out1) ∧ Dtype (clk,v25,out2) ∧ Dtype (clk,v24,v4) ∧
       Dtype (clk,v23,v3) ∧ Dtype (clk,v22,v2) ∧ Dtype (clk,v21,v1) ∧
       Dtype (clk,v20,v0) ∧ DtypeT (clk,v16,v104) ∧ NOT (v104,v103) ∧
       AND (v103,v16,v102) ∧  Dtype (clk,v37,v101) ∧ AND (v102,v101,v100) ∧
       DtypeT (clk,v100,v108) ∧ NOT (v108,v107) ∧ AND (v107,v100,v106) ∧
       NOT (v106,v99) ∧ CONSTANT 1w v109 ∧ SUB32 (v27,v109,v105) ∧
       Dtype (clk,v105,v97) ∧ DtypeT (clk,v100,v123) ∧ NOT (v123,v122) ∧
       AND (v122,v100,v121) ∧ Dtype (clk,v98,v120) ∧ AND (v121,v120,v119) ∧
       DtypeT (clk,v119,v132) ∧ NOT (v132,v131) ∧ AND (v131,v119,v130) ∧
       Dtype (clk,v118,v129) ∧ AND (v130,v129,v128) ∧ DtypeT (clk,v128,v143) ∧
       NOT (v143,v142) ∧ AND (v142,v128,v141) ∧ Dtype (clk,v127,v140) ∧
       AND (v141,v140,v139) ∧ DtypeT (clk,v139,v146) ∧ NOT (v146,v145) ∧
       AND (v145,v139,v144) ∧ NOT (v144,v138) ∧ Dtype (clk,v26,v136) ∧
       CONSTANT 2654435769w v148 ∧ ADD32 (v20,v148,v147) ∧ DtypeT (clk,v139,v152) ∧
       NOT (v152,v151) ∧ AND (v151,v139,v150) ∧ NOT (v150,v137) ∧
       CONSTANT 4 v158 ∧ LSL32 (v25,v158,v157) ∧ ADD32 (v157,v24,v156) ∧
       ADD32 (v25,v147,v155) ∧ XOR32 (v156,v155,v154) ∧ CONSTANT 5 v160 ∧
       ASR32 (v25,v160,v159) ∧ ADD32 (v159,v23,v153) ∧ XOR32 (v154,v153,v149) ∧
       Dtype (clk,v149,v135) ∧ DtypeT (clk,v138,v163) ∧ NOT (v163,v162) ∧
       AND (v162,v138,v161) ∧ MUX (v161,v136,v164,v134) ∧ Dtype (clk,v134,v164) ∧
       DtypeT (clk,v137,v167) ∧ NOT (v167,v166) ∧ AND (v166,v137,v165) ∧
       MUX (v165,v135,v168,v133) ∧ Dtype (clk,v133,v168) ∧ AND (v138,v137,v127) ∧
       ADD32 (v134,v133,v125) ∧ DtypeT (clk,v128,v179) ∧ NOT (v179,v178) ∧
       AND (v178,v128,v177) ∧ Dtype (clk,v126,v176) ∧ AND (v177,v176,v175) ∧
       DtypeT (clk,v175,v182) ∧ NOT (v182,v181) ∧ AND (v181,v175,v180) ∧
       NOT (v180,v174) ∧ Dtype (clk,v25,v172) ∧ NOT (v187,v190) ∧
       OR (v190,v175,v189) ∧ DtypeT (clk,v189,v201) ∧ NOT (v201,v200) ∧
```

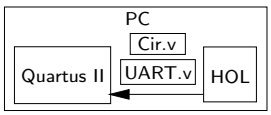
```
    AND (v200,v189,v199) ∧ Dtype (clk,v188,v198) ∧ AND (v199,v198,v197) ∧
    DtypeT (clk,v197,v212) ∧ NOT (v212,v211) ∧ AND (v211,v197,v210) ∧
    Dtype (clk,v196,v209) ∧ AND (v210,v209,v208) ∧ DtypeT (clk,v208,v215) ∧
    NOT (v215,v214) ∧ AND (v214,v208,v213) ∧ NOT (v213,v207) ∧
    Dtype (clk,v26,v205) ∧ CONSTANT 2654435769w v217 ∧ ADD32 (v20,v217,v216) ∧
    DtypeT (clk,v208,v221) ∧ NOT (v221,v220) ∧ AND (v220,v208,v219) ∧
    NOT (v219,v206) ∧ CONSTANT 4 v227 ∧ LSL32 (v25,v227,v226) ∧
    ADD32 (v226,v24,v225) ∧ ADD32 (v25,v216,v224) ∧ XOR32 (v225,v224,v223) ∧
    CONSTANT 5 v229 ∧ ASR32 (v25,v229,v228) ∧ ADD32 (v228,v23,v222) ∧
    XOR32 (v223,v222,v218) ∧ Dtype (clk,v218,v204) ∧ DtypeT (clk,v207,v232) ∧
    NOT (v232,v231) ∧ AND (v231,v207,v230) ∧ MUX (v230,v205,v233,v203) ∧
    Dtype (clk,v203,v233) ∧ DtypeT (clk,v206,v236) ∧ NOT (v236,v235) ∧
    AND (v235,v206,v234) ∧ MUX (v234,v204,v237,v202) ∧ Dtype (clk,v202,v237) ∧
    AND (v207,v206,v196) ∧ ADD32 (v203,v202,v194) ∧ DtypeT (clk,v197,v241) ∧
    NOT (v241,v240) ∧ AND (v240,v197,v239) ∧ NOT (v239,v195) ∧
    CONSTANT 2654435769w v242 ∧ ADD32 (v20,v242,v238) ∧ Dtype (clk,v238,v193) ∧
    Dtype (clk,v22,v192) ∧ Dtype (clk,v21,v191) ∧ DtypeT (clk,v196,v245) ∧
    NOT (v245,v244) ∧ AND (v244,v196,v243) ∧ MUX (v243,v194,v246,v186) ∧
    Dtype (clk,v186,v246) ∧ DtypeT (clk,v195,v249) ∧ NOT (v249,v248) ∧
    AND (v248,v195,v247) ∧ MUX (v247,v193,v250,v185) ∧ Dtype (clk,v185,v250) ∧
    DtypeT (clk,v195,v253) ∧ NOT (v253,v252) ∧ AND (v252,v195,v251) ∧
    MUX (v251,v192,v254,v184) ∧ Dtype (clk,v184,v254) ∧ DtypeT (clk,v195,v257) ∧
    NOT (v257,v256) ∧ AND (v256,v195,v255) ∧ MUX (v255,v191,v258,v183) ∧
    Dtype (clk,v183,v258) ∧ AND (v196,v195,v188) ∧ DtypeT (clk,v188,v262) ∧
    NOT (v262,v261) ∧ AND (v261,v188,v260) ∧ NOT (v260,v187) ∧
    CONSTANT 4 v268 ∧ LSL32 (v186,v268,v267) ∧ ADD32 (v267,v184,v266) ∧
    ADD32 (v186,v185,v265) ∧ XOR32 (v266,v265,v264) ∧ CONSTANT 5 v270 ∧
    ASR32 (v186,v270,v269) ∧ ADD32 (v269,v183,v263) ∧ XOR32 (v264,v263,v259) ∧
    Dtype (clk,v259,v171) ∧ AND (v188,v187,v173) ∧ DtypeT (clk,v174,v273) ∧
    NOT (v273,v272) ∧ AND (v272,v174,v271) ∧ MUX (v271,v172,v274,v170) ∧
    Dtype (clk,v170,v274) ∧ DtypeT (clk,v173,v277) ∧ NOT (v277,v276) ∧
    AND (v276,v173,v275) ∧ MUX (v275,v171,v278,v169) ∧ Dtype (clk,v169,v278) ∧
    AND (v174,v173,v126) ∧ ADD32 (v170,v169,v124) ∧ DtypeT (clk,v127,v281) ∧
    NOT (v281,v280) ∧ AND (v280,v127,v279) ∧ MUX (v279,v125,v282,v116) ∧
    Dtype (clk,v116,v282) ∧ DtypeT (clk,v126,v285) ∧ NOT (v285,v284) ∧
    AND (v284,v126,v283) ∧ MUX (v283,v124,v286,v115) ∧ Dtype (clk,v115,v286) ∧
    AND (v127,v126,v118) ∧ DtypeT (clk,v119,v290) ∧ NOT (v290,v289) ∧
    AND (v289,v119,v288) ∧ NOT (v288,v117) ∧ CONSTANT 2654435769w v291 ∧
    ADD32 (v20,v291,v287) ∧ Dtype (clk,v24,v114) ∧ Dtype (clk,v23,v113) ∧
    Dtype (clk,v22,v112) ∧ Dtype (clk,v21,v111) ∧ Dtype (clk,v287,v110) ∧
    DtypeT (clk,v118,v294) ∧ NOT (v294,v293) ∧ AND (v293,v118,v292) ∧
    MUX (v292,v116,v295,v96) ∧ Dtype (clk,v96,v295) ∧ DtypeT (clk,v118,v298) ∧
    NOT (v298,v297) ∧ AND (v297,v118,v296) ∧ MUX (v296,v115,v299,v95) ∧
    Dtype (clk,v95,v299) ∧ DtypeT (clk,v117,v302) ∧ NOT (v302,v301) ∧
    AND (v301,v117,v300) ∧ MUX (v300,v114,v303,v94) ∧ Dtype (clk,v94,v303) ∧
    DtypeT (clk,v117,v306) ∧ NOT (v306,v305) ∧ AND (v305,v117,v304) ∧
    MUX (v304,v113,v307,v93) ∧ Dtype (clk,v93,v307) ∧ DtypeT (clk,v117,v310) ∧
    NOT (v310,v309) ∧ AND (v309,v117,v308) ∧ MUX (v308,v112,v311,v92) ∧
    Dtype (clk,v92,v311) ∧ DtypeT (clk,v117,v314) ∧ NOT (v314,v313) ∧
    AND (v313,v117,v312) ∧ MUX (v312,v111,v315,v91) ∧ Dtype (clk,v91,v315) ∧
    DtypeT (clk,v117,v318) ∧ NOT (v318,v317) ∧ AND (v317,v117,v316) ∧
    MUX (v316,v110,v319,v90) ∧ Dtype (clk,v90,v319) ∧ AND (v118,v117,v98) ∧
    DtypeT (clk,v99,v322) ∧ NOT (v322,v321) ∧ AND (v321,v99,v320) ∧
    MUX (v320,v97,v323,v36) ∧ Dtype (clk,v36,v323) ∧ DtypeT (clk,v98,v326) ∧
    NOT (v326,v325) ∧ AND (v325,v98,v324) ∧ MUX (v324,v96,v327,v35) ∧
    Dtype (clk,v35,v327) ∧ DtypeT (clk,v98,v330) ∧ NOT (v330,v329) ∧
    AND (v329,v98,v328) ∧ MUX (v328,v95,v331,v34) ∧ Dtype (clk,v34,v331) ∧
    DtypeT (clk,v98,v334) ∧ NOT (v334,v333) ∧ AND (v333,v98,v332) ∧
    MUX (v332,v94,v335,v33) ∧ Dtype (clk,v33,v335) ∧ DtypeT (clk,v98,v338) ∧
    NOT (v338,v337) ∧ AND (v337,v98,v336) ∧ MUX (v336,v93,v339,v32) ∧
    Dtype (clk,v32,v339) ∧ DtypeT (clk,v98,v342) ∧ NOT (v342,v341) ∧
    AND (v341,v98,v340) ∧ MUX (v340,v92,v343,v31) ∧ Dtype (clk,v31,v343) ∧
    DtypeT (clk,v98,v346) ∧ NOT (v346,v345) ∧ AND (v345,v98,v344) ∧
    MUX (v344,v91,v347,v30) ∧ Dtype (clk,v30,v347) ∧ DtypeT (clk,v98,v350) ∧
    NOT (v350,v349) ∧ AND (v349,v98,v348) ∧ MUX (v348,v90,v351,v29) ∧
    Dtype (clk,v29,v351) ∧ AND (v99,v98,v37) ∧ Dtype (clk,v37,v353) ∧
    AND (v37,v353,v352) ∧ AND (v7,v19,v354) ∧ AND (v354,v352,done))
==>
 DEV TEAEncrypt
   (load at clk,
    ((inp11 <> inp12 <> inp13 <> inp14) <> inp2 <> inp3) at clk,
    done at clk,(out1 <> out2) at clk) : thm
```

# 35 How are HOL designs downloaded to an FPGA?

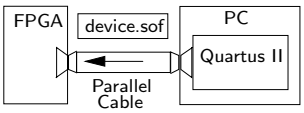
There are four steps to download our circuits to an FPGA.



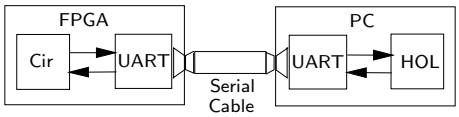

(g) Proof producing synthesis.




(h) Verilog compilation.




(i) Download the object file.




(j) Run the circuit.

**Proof Producing Synthesis.** The initial step is concerned with the production of the theorem: ⊢ InfRise $clk \Rightarrow (Cir \Longrightarrow$ Dev $f)$.

**Verilog compilation.** A pretty-printer translates the circuit $Cir$ into the Verilog file `Cir.v`. No formal verification is applied to this translation as Verilog has no formal semantics. The primitive components — operators like AND, OR, MUX — are mapped to Verilog modules. The file `UART.v` contains a Verilog implementation of an interface that connects a serial cable to the circuit (this interface has not being formally verified). Both files are sent to Quartus II for compilation.

**Download the object file.** Quartus II translates the Verilog files into the object file `device.sof`, which is downloaded to the FPGA via the parallel cable.

**Run the circuit.** HOL is connected to the serial cable by a UART program previously coded in C. The circuit is triggered interactively via an automatically generated function defined in HOL which communicates with the UART program.

All the four steps can be carried out from the HOL system provided that the pretty-printer is able to map every primitive combinational operator to Verilog.

# 36 What are the plans for the future?

We hope to use the compiler to generate various kinds of cryptographic hardware. We expect more aggressive compaction may be needed to fit bigger examples (e.g. AES) onto the FPGAs we are using.

Eventually, it is hoped to provide wrapper circuitry to enable synthesised HOL functions to be invoked from ARM code as hardware co-processors. The FPGA board we are using (Altera Excalibur) has an ARM processor on it.

# 37 Is the compiler freely available?

The compiler is distributed with the HOL-4 system (`http://hol.sourceforge.net/`) in the directory `examples/dev`.

## 38 What related work is there?

There is considerable previous work on using functional programming to specify and design hardware. Examples include $\mu$FP (Sheeran [15]), Ruby (Jones & Sheeran [9, 8]), Hydra (O'Donnell [12]), Lava (Bjesse et al [1]), DDD (Johnson & Bose [7]), LAMBDA (Finn et al [5]), Gropius (Blumenröhr [2]), DUAL-EVAL (Brock & Hunt [3]) and SAFL (Mycroft & Sharp [11]).

Our work was initially inspired by SAFL, a hardware compiler for a language based on a first-order subset of ML, though we use a subset of higher order logic rather than a separate special-purpose design specification language. The general way we employ serial/parallel combinators for compositional translation has similarities to compilers for Handel (Page [13]) and SAFL. However, many details differ in our approach: in particular, our compiler is proof-producing.

The paper "Formal Synthesis in Circuit Design – A Classification and Survey" [14] provides an excellent overview. In terms of the classification in that paper, our approach is *formal synthesis* by *transformational derivation* in a *general purpose calculus*.

The way we realise HOL functions by handshaking devices is reminiscent of some self-timed design methods [6, 16], though we produce clocked synchronous circuits.

## Acknowledgements

David Greaves gave us advice on the hardware implementation of handshake protocols and also helped us understand the results of simulating circuits produced by our compiler. Simon Moore and Robert Mullins lent us an Excalibur FPGA board on which we are running compiled hardware at Cambridge, and they helped us with the Quartus II design software that we are using to drive the board. Ken Larsen used his dynlib library to write an ML version of our original C interface to the serial port (this is used to communicate with the Excalibur board, see 35).

## References

[1] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: Hardware design in Haskell. *ACM SIGPLAN Notices*, 34(1):174–184, January 1999.

[2] Christian Blumenröhr. A formal approach to specify and synthesize at the system level. In *GI Workshop Modellierung und Verifikation von Systemen*, pages 11–20, Braunschweig, Germany, 1999. Shaker-Verlag.

[3] Bishop Brock and Warren A. Hunt Jr. The DUAL-EVAL hardware description language and its use in the formal specification and verification of the fm9001 microprocessor. *Formal Methods in System Design*, 11(1):71–104, 1997.

[4] Common Criteria for Information Security Evaluation, 2004. Part 3: Security Assurance Requirements, `http://niap.nist.gov/cc-scheme/cc_docs/cc_v22_part3.pdf`.

[5] Simon Finn, Michael P. Fourman, Michael Francis, and Robert Harris. Formal system design—interactive synthesis based on computer-assisted formal reasoning. In Luc Claesen, editor, *IMEC-IFIP International Workshop on Applied Formal Methods for Correct VLSI Design, Volume 1*, pages 97–110, Houthalen, Belgium, November 1989. Elsevier Science Publishers, B.V. North-Holland, Amsterdam.

[6] Scott Hauck. Asynchronous design methodologies: An overview. *Proceedings of the IEEE*, 83(1):69–93, 1995.

[7] Steven D. Johnson and Bhaskar Bose. DDD – A System for Mechanized Digital Design Derivation. Technical Report TR323, Indiana University, IU Computer Science Department, 1990.

[8] G. Jones and M. Sheeran. Circuit design in Ruby. In J. Staunstrup, editor, *Formal Methods for VLSI Design*, pages 13–70. Elsevier Science Publications, North-Holland, 1990.

[9] G. Jones and M. Sheeran. Relations and refinement in circuit design. In C. Morgan, editor, *BCS FACS Workshop on Refinement*. Springer-Verlag, 1991.

[10] Thomas F. Melham. *Higher Order Logic and Hardware Verification*. Cambridge University Press, Cambridge, England, 1993. Cambridge Tracts in Theoretical Computer Science 31.

[11] Alan Mycroft and Richard Sharp. Hardware synthesis using SAFL and application to processor design. In *Proceedings of the 11th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME'01)*, Livingston, Scotland, September 2001. Springer Verlag. Invited Talk. LNCS Vol. 2144.

[12] John O'Donnell. Overview of Hydra: A concurrent language for synchronous digital circuit design. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium*. IEEE Computer Society Press, 2002.

[13] I. Page. Constructing hardware-software systems from a single description. *Journal of VLSI Signal Processing*, 12(1):87–107, 1996. `citeseer.ist.psu.edu/page96constructing.html`.

[14] R. Kumar, C. Blumenroehr, D. Eisenbiegler, and D. Schmid. Formal synthesis in circuit design-A classification and survey. In M. Srivas and A. Camilleri, editors, *First international conference on formal methods in computer-aided design*, volume 1166, pages 294–299, Palo Alto, CA, USA, 1996. Springer Verlag.

[15] Mary Sheeran. $\mu$FP, A Language for VLSI Design. In *Proceedings of the ACM Symposium on LISP and Functional Programming*, pages 104–112. ACM Press, Austin, Texas, 1984.

[16] Kees van Berkel. *Handshake circuits: an asynchronous architecture for VLSI programming*. Cambridge University Press, New York, NY, USA, 1993.

[17] David Wheeler and Roger Needham. TEA, a tiny encryption algorithm. In *Fast Software Encryption: Second International Workshop*, volume 1008 of *LNCS*, pages 363–366. Springer Verlag, 1999.

# Towards the Correct Design of Multiple Clock Domain Circuits

Ed Czeck, Ravi Nanavati and Joe Stoy
Bluespec Inc.
Waltham MA 02451, USA

## Abstract

*We present a set of guiding principles for the management of multiple clocks domains in the design of a high-level hardware description language. Our motivation of the requirements is based on typical design problems; the solutions are based on common engineering practices and appropriate language abstractions. We include examples, and conclude with some comments based on a design experience.*

## 1. Introduction

Hardware designs these days typically make use of several clocks. This is partly to save power (by gating the clock to a part of the circuit temporarily not in use, and by ensuring that parts of the design are not run unnecessarily fast, both of which reduce the design's "dynamic" power consumption), and also to allow the design to communicate with parts of the external environment running asynchronously. Moreover, designs are increasingly "systems on a chip" ("SoC"s): these bring together blocks (IPs) which come from various sources, and of which each is likely to have its own clocking requirements. These different requirements may arise simply because each block was designed independently; but it might be because different blocks are constrained by different standards (for example for external buses, or audio or video I/O), each with its own clocking regime.

Where signals cross clock domain boundaries, the normal design conventions of digital logic break down. Special precautions must be taken to ensure that the signals get across correctly; and care must be taken that there are no accidental crossings which neglect such precautions. A good hardware description language, particularly one which claims to operate at a high level of abstraction, should have features which make it natural for the designer to construct circuits which correctly observe these constraints.

## 2 Principles

A hardware description language, notation or system which supports multiple-clock-domain designs should ideally have the following characteristics.

1. The aim should be to make the simplest situations trivial, other simple situations easy, and all situations expressible in a sensible way.
2. Thus in a design, or part of a design, with just one clock domain, clock handling should be completely implicit. Each instantiated module needs to be connected to "the" clock, and having to say so explicitly adds unnecessary clutter.
3. *Clock* should be a datatype of the language (values of which will include the oscillator, as well as an optional gating signal); the type system should ensure that clock oscillators are never confused with level-sampled signals.
4. The system should keep track of which signals are clocked by which clocks, and ensure that no signal crosses a clock-domain boundary without the use of appropriate synchronizing logic.
5. For efficiency's sake, the system should be able to recognize when two clocks are driven by the same oscillator (that is, they differ only in gating); this should be exploited to simplify domain-crossing logic between them.
6. Many groups of designers have their own preferred designs for domain-crossing logic; the system should allow the automatic deployment of such designs when appropriate (though also providing default designs for use when these are not available).

In the remainder of this paper we address, by way of example, how these principles are addressed in Bluespec SystemVerilog.

## 3. A Brief Introduction to BSV

Bluespec SystemVerilog is a strongly typed, high-level behavioral hardware description language. As with Verilog, designs are structured into modules. The internal behavior of a module is specified by *rules* (instead of always-blocks), which have a guard and an action part. Modules communicate with their environment by the *methods* of their interfaces, which may be invoked by rules in other modules. All

methods have "ready" conditions, which become "implicit" conditions" of rules which invoke them—only if all the conditions, explicit and implicit, of a rule are satisfied may the rule be executed. Taken together, the rules of a design have the semantics of a state-transition system (or term-rewriting system), and execute atomically. The Bluespec compiler generates logic which schedules as many rules as possible in each clock cycle; but the overall effect of each cycle is exactly the same as if the rules executed one at a time in some order—this greatly simplifies the analysis of the design's correctness. (It avoids many race conditions, replacing them with compile time warnings that two rules could not be scheduled simultaneously because of a resource conflict; it also means that proofs of correctness can analyse each rule separately, without worrying about their interaction.)

A slightly fuller description is provided in the Appendix; the reader is also referred to the Language Reference Guide [BLU06] for a complete account.

# 4. Clocks—The Simplest Case

The simplest case is one where a design uses just one clock. In line with the principles outlined above, in this case the clock is not mentioned in the BSV text at all. Each module in the RTL generated by the Bluespec compiler will have a clock port, called *CLK*; this is connected to any module instantiated within that module. At the lowest level of the generated module hierarchy, this clock is connected to the flip-flops, registers and other primitive state elements.

The advantage of an implicit clock is that designer is spared the tedium of mentioning the clock and its explicit use in the generation of the flops, registers, and other primitives. The use of standard primitives based on common engineering practice (in this case positive-edge-triggered flops) further reinforces that abstraction for the common case, while the complete Bluespec SystemVerilog language allows for a full range of flexibility.

# 5. The Type "Clock"

In designs with more complicated clocking arrangements, clocks are mentioned explicitly. A clock is a value of type *Clock*. It enjoys most of the general "first-class citizenship" properties of other BSV values: it may be passed as an argument, or returned as the result, of a function; it may be a field of an interface. A clock is not allowed, however, to pass through combinational logic generated by the Bluespec tool, because there are likely to be special requirements for the handling of clocks (affecting such things as phase and skew). All dynamic manipulation of clocks must be done using primitives written in Verilog: the Bluespec library provides a repertoire of general-purpose primitives, and designers are encouraged to write or request others, to deal with special situations and requirements.

Thus, for example, if *c1* and *c2* are clocks, the definition

```
Clock c = (b ? c1 : c2);
```

is valid, but only if *b* is a value known at "compile-time". The dynamic selection between two clocks must be done by a special multiplexing primitive such as *mkClockMux*.

A BSV clock value contains two signals: an oscillator and a gating signal. If the gating signal is high, the clock is assumed to be ungated, and clearly the oscillator should be running. The tool remains agnostic about whether the reverse is true. Stopping the oscillator when the clock is gated saves the power being dissipated by charging and discharging the capacitance in the clock-tree itself, but it may require complicated interaction with the clock-handling tools downstream in the synthesis flow. The tool will, however, ensure that when the gating signal is low, all state transitions in that clock's domain are inhibited.

New clock values arise in several ways. They are often passed in as arguments to the design, having been generated by external electronics. They may also be generated in IP cores (for which the BSV designer will provide a wrapper) to handle external interfaces (such as a SPI-4) which provide their own clocks. Bluespec provides facilities, described in more detail below, for adding a gating condition to an existing clock. Facilities are also available for generating clocks with periods specified numerically; but these are not suitable for synthesis, and are provided for simulation purposes only.

# 6. Gated Clocks

A simple way to save power is to switch off the clock for parts of the design when they are temporarily not in use. Differently gated versions of the same clock allow a simplified treatment, since when both are running they are exactly in phase. BSV provides special facilities to handle this case efficiently.

If a clock $B$ is a gated version of clock $A$, we say that $A$ is an *ancestor* of $B$. We therefore know that if a clock is running, then so are all of its ancestors. Clocks which are driven by the same oscillator, and differ merely in gating, are said to be in the *same family*. Both these relationships may be tested by functions available to the design: thus a library package can automatically arrange to exploit the extra simplicity of this situation without the user's having to be aware of it.

Each method of a primitive module is explicitly associated with a particular clock, often (but not necessarily) the default clock of the primitive module's instantiation. These methods are invoked by other methods (of other modules, written in BSV) or by rules: the tool rigorously insists that all methods invoked by any one method or rule are in the

same family, thereby avoiding the risk of paths accidentally crossing between different clock domains. The invoking method or rule will be clocked by a clock in that same family, which is running if and only if all the invoked methods' clocks are running—if necessary, a new clock in the same family will be produced which satisfies this condition. The guard of any method which effects a state transition (that is, in general, any method which has an *ENABLE* signal) will include the gating condition of the method's clock—so such a method will not be *READY* unless its clock is running, and neither will any rule which invokes that method. (A method which merely returns a value, without executing a state transition, remains *READY* when its clock is switched off, returning the value set by the latest transition, provided that it was already *READY* when the switch-off occurred.)

All this mechanism obviates the need for any special clock-domain-crossing logic between same-family domains: everything is handled by the normal implicit-condition mechanism of BSV methods.

# 7. Clock-domain Crossing Between Families

When the clocks concerned are from different families (which in general implies that they have different oscillators), domain crossing is more complicated and requires special logic. It is always handled by primitives written in Verilog. The tool will ensure that no domain crossing occurs without the use of such a primitive and that the logic used connects the domains appropriately. As with any design language, the tools can only ensure that a given module is used properly, and cannot verify that the module was correctly written or correctly selected in the first place.

Bluespec provides general-purpose primitives in its library, but users are encouraged to provide (or to request) others to cover special cases. For example, each clocking edge of the slower clock might coincide with a clocking edge of the faster one; or the clocks might have the same frequency but a different phase; or very nearly the same frequencies; or very different frequencies. The general-purpose primitives will handle all these cases, but perhaps not as efficiently as special-purpose ones.

The facilities provided by Bluespec fall into two groups, following two different approaches. The "hardware approach" provides modules with source and destination ports, which the designer instantiates and connects up explicitly; the low-level primitives are provided only in this form. The "linguistic approach" provides modules which transform an interface into one of the same type, but differently clocked: this allows a smoother treatment in the BSV notation. These two approaches are described and illustrated below.

## 7.1. The Hardware Approach

Synchronizers are provided to handle the following cases of moving data from a source clock domain to a destination clock domain. Like the single clock primitives, the provided synchronizers are based on common engineering practice, while the full language allows other designs which users may desire.

(1) bits: a bit change to the source will cause a bit change in the destination;

(2) pulses: a pulse on the source will cause a pulse in the destination.

If the crossing is from a fast clock domain to a slower one, there is a danger in these two cases that information may be lost: a bit change might not be noticed if it persists only for a short time, and a sequence of fast pulses might result in fewer pulses on the destination side. The next case guards against this:

(3) the same as (2), but a second source pulse is not accepted until a pulse has been delivered at the destination.

(4) words: a word delivered to the source eventually appears at the destination; a subsequent send cannot occur until the first has been delivered.

Note that this word synchronizer is not simply a parallel composition of a number of bit synchronizers: precautions must be taken to ensure that all the bits of a word appear at the destination at the same time.

In these last two cases, even though an event is guaranteed to have happened on the destination side, there is no guarantee that it has been noticed. The final case avoids this problem.

(5) A FIFO: data items enqueued on the source side will arrive at the destination side, and remain there until they are dequeued.

Examples of these synchronizers will be shown in the examples which follow.

## 7.2. The Linguistic Approach

Many BSV designs make extensive use of *Get* and *Put* interfaces, provided in the library. An interface of type *Put#*($a$) is the simplest interface into which one can put values of type $a$; similarly, an interface of type *Get#*($a$) is the simplest interface from which values of type $a$ can be retrieved just once (that is, they are not merely read but also removed). A *Get* interface and a corresponding *Put* interface may be connected by the module *mkConnection* (the name is overloaded, and may be used to connect other compatible interfaces too).

These are just two of the kinds of interface which may be converted in this second approach. If *ifc* is such an interface, clocked by any clock, the instantiation

```
mkConverter#(n) the_conv(ifc, new_ifc);
```

will produce an interface of the same type, but clocked by the clock of the current environment. There is no need to specify the clock of the original *ifc*, as the *mkConverter* module can determine that for itself. The parameter $n$ specifies the depth of the conversion FIFO to be used between the two domains.

Since the types of the original and the new interfaces are the same, and many of the details are implicit, this approach lends itself to generalisation: the *mkConverter* name is also overloaded, and can be used to implement clock-domain conversion on a whole class of interfaces.

# 8. Examples

Figure 1 shows the use of *mkSlowClock* and also the use of gated clocks, using the "hardware approach". The module *mkGenPair* produces a pair of *Get* interfaces (perhaps sources of pseudo-random numbers, produced by splitting the output from a random-number generator). The other two sub-modules are user modules, and they also each have a pair of interfaces. One of these is a *Put* interface, and is connected to one of the *Get* interfaces; the other is exported as part of the main module's interface.

The required domain crossing is achieved by the primitive *mkSyncFIFO*, which is supplied with the two clocks concerned (and the source-side reset signal). The designer provides two rules, *enqueue_ff* and *dequeue_ff*, which supply and retrieve data items: note that these two rules are in different clock domains, which of course the tool automatically verifies.

Figure 2 shows the last few lines of the same module, but using the "linguistic approach" instead. The interface *user1ifc* is of the same type as the original interface *user1.fst*, but it is clocked by the default clock of the surrounding module which, since it is in the same family as *c1*, is suitable for direct connection to *gens.snd*.

Our final example is in Figure 3, which shows *mkConverter* for a *Put* interface. This demonstrates how the linguistic approach is implemented using the hardware approach primitives. It is suitable for any type $a$ of data, with the proviso that it can be represented in bits (this is necessary because values of this type are to be stored in a FIFO).

The internal domain-crossing primitive is *mkSyncFIFOFromCC*, a variant of *mkSyncFIFO* which assumes that the source side is to be clocked by the current clock of the surrounding module. *mkConverter* implements the *put* method of the interface it is providing, which enqueues items on the synchronizing FIFO; the internal *dequeue* rule

```
(* synthesize *)
module mkRandTop(UInt#(4) ratio,
                 Bool g1, Bool g2, ExtIfc ifc);
   // Declare the gated clocks and their
   // associated resets:
   // c1 will be a slower clock:
   Clock c1 <- mkSlowClock(ratio, g1);
   Reset r1 <- mkSyncResetFromCC(3, c1);
   // c2 is a gated version of currentClock:
   Clock c2 <- mkGatedClock(g2);
   Reset r2 <- mkSyncResetFromCC(3, c2);
   // c0 is similar, on when either of the
   // consumers is on:
   Clock c0 <- mkGatedClock gate0(g1 || g2);
   Reset r0 <- mkSyncResetFromCC(3, c0);

   // Instantiate the sub-modules,
   // appropriately clocked:
   GenPair gens <-
        mkGenPair(clocked_by c0, reset_by r0);
   UserIfc user1 <-
        mkUser1(clocked_by c1, reset_by r1);
   UserIfc user2 <-
        mkUser2(clocked_by c2, reset_by r2);

   // Since c2 and c0 are in the same
   // family, there is no need for explicit
   // conversion:
   mkConnection(gens.fst, user2.fst);

   // c1 is unrelated to c0, however, so
   // explicit conversion is necessary.
   // This version uses the "hardware approach".

   SyncFIFOIfc#(Bit#(6)) ff <-
        mkSyncFIFO(4, c0,r0, c1);

   // We provide two rules to enqueue values
   // from the generator onto ff, and to
   // dequeue them to send to user1:
   rule enqueue_ff;
      let x <- gens.snd.get;
      ff.enq(x);
   endrule
   rule dequeue_ff;
      user1.fst.put(ff.first);
      ff.deq;
   endrule

   // The external interfaces:
   interface ifcA = user1.snd;
   interface ifcB = user2.snd;
   // Also export the clock for ifcA:
   interface cA = c1;
endmodule
```

Figure 1: Use of various clocks

```
   // This one uses the "linguistic approach".

   // There's no need to specify an explicit
   // clock for the converter, since the current
   // clock is in the same family as c1.
   let user1ifc <- mkConverter(4, user1.fst);

   mkConnection(gens.snd, user1ifc);

   interface ifcA = user1.snd;
   interface ifcB = user2.snd;
   // Export the clock for ifcA:
   interface cA = c1;
endmodule
```

Figure 2: The linguistic approach (last few lines)

```
module mkConverter#(Integer d)
     (Put#(a) used_put, Put#(a) provided_put)
   provisos (Bits#(a,sa));

   SyncFIFOIfc#(a) ff <-
     mkSyncFIFOFromCC(d, clockOf(used_put));

   rule dequeue;
      used_put.put(ff.first);
      ff.deq;
   endrule

   method Action put(x);
      ff.enq(x);
   endmethod
endmodule
```

Figure 3: Implementation of *mkConverter*

dequeues items and supplies them to the *put* method of the "used" interface argument.

# 9 Conclusion

The handling of multiple clocks domains in a design can be a rich source of error. The original way of preventing such errors was visual inspection of the source code, coupled with testing; it was not very reliable, particularly since many of the runtime errors would only manifest themselves rarely, as they depended on particular phase-relationships of the participating clocks. Various companies have provided "lint-like" tools [ATR05, CAD04, SYN06], which pro-actively analyse the source code for likely errors and other infelicities. A better solution is for the source code language to include a set of features which allows designers "to get it right first time". As far as we know, apart from ourselves only the recently-announced version of Esterel [EST06] attempts to do this.

The features described here have been used in a substantial design (a UTMI block [UTM01] for USB2.0). This handled USB transmission both at 480MHz and 12MHz; most of the logic was clocked at 120MHz (dealing with 4-bit nibbles in parallel). Thus many clocks were involved, and some careful design was required, some of it iterative, to perform the required domain crossing within the latency constraints of the specification (as tested by the Verisity Specman test suite [VER04]). It was found that the support provided by the language avoided all accidental mistakes, allowing iterative improvements to be implemented and tested quickly. It is estimated that the design was completed (and passed the test suite) in a time at least twice as fast as if it had been written in standard Verilog.

# 10 Acknowledgments

The authors thank their colleagues at Bluespec Inc., and members of the Computation Structures Group at MIT's Computer Science and Artificial Intelligence Laboratory, for advice and assistance with this project.

# References

[ATR05] Atrenta, Inc. *1team:Verify*, 2005.

[BLU06] Bluespec, Inc. *Bluespec Language Reference Guide*, 2006. Please consult http://www.bluespec.com.

[CAD04] Cadence Design Systems, Inc. *Incisive HDL Analysis (HAL)*, 2004.

[EST06] Esterel Technologies, Inc. *Esterel Studio Version 5.3*, 2006.

[SYN06] Synopsys, Inc. *Leda Programmable RTL Checker*, 2006.

[SYS05] IEEE Standard for SystemVerilog—Unified Hardware Design, Specification , and Verification Language, November 2005. IEEE Std 1800-2005, http://standards.ieee.org.

[UTM01] Intel Corporation. *USB 2.0 Transceiver Macrocell Interface (UTMI) Specification*, 2001.

[VER04] Verisity Design, Inc. *USB eVC*, 2004.

# A. Bluespec SystemVerilog

Previous attempts at behavioral synthesis have tried to optimize along three axes simultaneously: choice of micro-architecture, allocation of resources, and scheduling of concurrent operations. Doing all this together, however, is computationally "hard". Besides, designers are good at evaluating micro-architectures, and like to be in control of resource allocation; handling concurrency, however, often becomes excessively complex. The Bluespec tool takes over this task, while leaving the designer in control of the other two. The result is a flexible tool, with which it is easy to do architectural experiments, while producing RTL of comparable quality to hand-crafted Verilog.

The HDL for the Bluespec tool is BluespecSystemVerilog (BSV). This is a variant of SystemVerilog [SYS05] in which behavior is specified, not by the usual "always-blocks", but by "design assertions" also known as "rules". A rule consists of a condition and an action part. Only if the condition is satisfied may the state transition specified by

the action part be executed. Each action executes in a single clock cycle. The tool generates scheduling logic which executes as many rules as possible in each cycle, but with the restriction that the overall effect must be the same as if they had each executed one at a time in some order. This restriction avoids many race conditions, replacing them with a compile-time warning that two rules could not be simultaneously scheduled because of a resource conflict, which is much easier to deal with.

As in standard SystemVerilog, a BSV design is partitioned into modules. The designer can control which modules are synthesized by the Bluespec tool into separate RTL modules (output in low-level Verilog2001) and which are inlined. The action part of a rule effects a state transition by invoking a *method* of some other module's interface. Even individual registers are actually instantiations of primitive modules (written in standard Verilog)—references to them are "desugared" into invocations of their *_read* and *_write* methods.

The interfaces of these other modules might be available in the module's environment; or the modules might be instantiated within the module being defined; or they might be supplied as arguments to that module. These appear in the list corresponding to the port list of a standard Verilog module; the input/output distinction is inappropriate for these arguments (as each interface, and indeed each of its methods, contain both input and output signals), so instead we distinguish between the interfaces "used" by a module, and the interface (by convention the last one in the list) it "provides" by defining each of its methods.

For modules which are separately synthesized, the methods of a module's interface become collections of ports in the RTL version. As well as the data ports (input or output), each method in general has an output *READY* signal, asserting that the method may validly be invoked; methods which effect state transitions also have an input *ENABLE* signal, asserting that the transition is to be executed. The tool enforces the protocol that the *ENABLE* signal may not be asserted unless the corresponding *READY* signal is also asserted. At the language level, the method's validity conditions are implicitly added to the conditions of any rule (or other method) which invokes it; similarly, its action becomes part of the rule's (or other method's) one-cycle action.

## A.1. Example—Factorial

Figure 4 shows a simple complete BSV design, containing a module *mkFact* for computing the factorial function, and a testbench module *mkTestFact* for exercising it.

The interface provided by *mkFact* is of type *NumFn*; it consists of a method *start* to initiate a calculation, and a method *result* to retrieve the result. The actual computation is performed by the rule *calc*, which can run only when $n \neq$

```
package Factorial;

typedef UInt#(32) Nat;

interface NumFn;
   method Action start(Nat x);
   method Nat result();
endinterface

(* synthesize *)
module mkFact(NumFn ifc);
   Reg#(Nat) n <- mkReg(0);
   Reg#(Nat) a <-  mkRegU;

   rule calc (n!=0);
      a <= a * n;
      n <= n - 1;
   endrule

   method Action start(x) if (n==0);
      a <= 1;
      n <= x;
   endmethod

   method result if (n==0);
      return a;
   endmethod
endmodule

(* synthesize *)
module mkTestFact(Empty);
   NumFn fact <- mkFact;
   Reg#(UInt#(2)) state <- mkReg(0);
   rule start_test (state==0);
      state <= 1;
      fact.start(7);
   endrule

   rule show_result (state==1);
      state <= 2;
      $display("%d", fact.result());
   endrule

   rule end_test (state==2);
      $finish(0);
   endrule
endmodule

endpackage
```

Figure 4: A simple BSV package

```
module mkFact(CLK,
      RST_N,
      start_x,
      EN_start,
      RDY_start,
      result,
      RDY_result);
  input  CLK;
  input  RST_N;

  // action method start
  input  [31 : 0] start_x;
  input  EN_start;
  output RDY_start;

  // value method result
  output [31 : 0] result;
  output RDY_result;
  ...
```

Figure 5: Part of the RTL for the simple example

0. The two methods, on the other hand, are valid only if $n = 0$: the result cannot be read while a computation is still in progress, nor can a new computation be started.

The first few lines of the RTL synthesized from *mkFact* are shown in Figure 5. As well as the methods' signals, already described, clock and reset ports will be noticed.

In the testbench, *mkTestFact*, the *mkFact* module is instantiated, giving an interface called *fact*. Its *start* method is invoked by the *start_test* rule of *mkTestFact*. Thus the complete condition of *start_test* is `state==0 & n==0`: the first test comes from the rule's condition and the second from the method's. Similarly, the action part is

```
state <= 1;
a <= 1;
n <= x;
```

amalgamating the actions of the rule and the method; all the assignments are executed simultaneously.

Notice that in this very simple design, all the rules are mutually exclusive—at most one is enabled at any one time. There is therefore no possibility of conflict, and the scheduling is trivial: each rule may fire whenever it is enabled. In general, however, many non-conflicting rules may fire during any one cycle.

# Two-level Languages and Circuit Design and Synthesis

Walid Taha*

Rice University, Houston, TX, USA
taha@cs.rice.edu

The next two decades are anticipated to move digital circuit design from the million transistor level to the billion and trillion transistor levels. In addition to challenges that this goal poses at the physical level, fundamental computational complexity barriers suggest that common design and verification tasks can also become a bottleneck. Examples include placement and routing, as well as a host of design rule checking (DRC) techniques. Increase in circuit size will increase both the time needed for DRC (from days to weeks or months) as well as the overall effort needed to produce a design likely to pass DRC. At the same time, increasing variability in implementation technologies and their characteristics will fuel the need for better methods to manage families of related circuits.

New programming language techniques recently developed to improve software design can provide a powerful tool for managing and checking families of related circuits. Program generation techniques in general, and two-level languages in particular, have been proposed and found to be useful for managing families of related software products. Static type checking in general, and dependent type systems in particular, have been proposed and found to be useful for early checking of a wide range of properties that would otherwise be expensive to check in generated programs. Our goal is to show that adapting these techniques to the specific needs of circuit design can lead to fundamental changes in the design process. In particular, it would allow the capture of significant design experience in the form of executable *and statically checkable* specifications for families of related circuits. Such specifications would be highly parameterized with respect to the specifics of the manufacturing technology, as well as the specifics of the problem being solved and the rest of the design. Comprehensive, manifest interfaces would allow fast, compositional checking of compatibility with the rest of the design.

Over the last two years we have made concrete advances toward this ambitious, long-term goal. Our first study showed that a standard type system for two-level languages can be systematically integrated with a type system for a resource-bounded language [5]. The result of such an integration, called a resource-aware programming (RAP) language [4], provides an expressive (non-resource bounded) language for writing generators of resource bounded computations. At the same time, a static type system is provided that checks that a generator can only generate well-formed, resource-bounded computations. Depending on the specific resources considered, such resource-bounded programs can be embedded software systems or hardware circuits.

A case study focusing on FFT showed that annotated versions of the basic Cooley-Tuckey recurrence can be executed as generators that produce high-quality circuits [2,

* Joint work with Stephan Ellner, Jennifer Gillenwater, Oleg Kiselyov, and Gregory Malecha. Funded by the National Science Foundation, the Texas Advanced Technology Program, National Instruments, and a grant from Rice University.

3]. In addition to confirming that this family of circuits can be specified by a generator closely resembling the textbook form of a standard recurrence, the experiment lead directly to two intriguing insights about FFT: First, unlike what the work on FFTW suggests, only a small number of domain-specific optimizations is needed to generate FFT circuits with the same arithmetic operation count as Split-radix or FFTW. Second, producing circuits that have counts identical to either Split-radix or FFTW requires *only* changing the definition of complex multiplication.

The RAP approach uses a purely functional language to describe hardware circuits, and so should be viewed as a direct descendant of Sheeran's family of hardware description languages. Focusing on two-level languages amounts to pursuing the insight that circuits are a strict subset of the generation language. Focusing on statically typed two-level languages reflects emphasis on performing the checking at the level of a *family* of circuits rather than on individual circuits. It is useful to note that this approach is complementary to model checking, which can perform more extensive, albeit more computationally intensive, checking on individual circuits.

Our emphasis on static checking discourages the transformation of circuits after they are generated. This contrasts with the transformational approach promoted by other systems (such as reFLect). Instead of first generating and then transforming, our approach focuses on incorporating domain-specific optimizations directly in the generator. This can have a two benefits. First, the designer can follow the methodology of abstract interpretation, widely used for program analysis, as a method for building optimizing generators that are correct by construction. The approach preserves the extensional nature of generated objects, and the soundness of equational reasoning principles. Second, avoiding the generation of numerous intermediate circuits can greatly improve the efficiency of the generation process.

Our recent and ongoing work focuses on the formal treatment of the connection between circuits and programs to allow the incorporation of various non-textual concepts into standard formal accounts of two-level languages [1]. Over the last year, we worked on building a prototype implementation to facilitate further work in this particular research direction. The prototype, called Uccello (previously PreVIEW), implements the translations between the graphical and textual representations used in our formal studies, in addition to implementing basic circuit layout algorithms.

## References


1. Stephan Ellner. PreVIEW: An untyped graphical calculus for resource-aware programming. Masters thesis, Rice University, 2004.
2. Oleg Kiselyov, Kedar Swadi, and Walid Taha. A methodology for generating verified combinatorial circuits. *EMSOFT '04*, Pisa, Italy, 2004.
3. Oleg Kiselyov and Walid Taha. Relating FFTW and split radix. *ICESS '04*, Hangzhou, China, 2004.
4. Walid Taha. Resource-aware programming. *ICESS '04*, Hangzhou, China, 2004. Invited Paper.
5. Walid Taha, Stephan Ellner, and Hongwei Xi. Generating Imperative, Heap-Bounded Programs in a Functional Setting. *EMSOFT'03*, Philadelphia, PA, October 2003.

# The Semantics of Graphical Languages*

Stephan Ellner and Walid Taha

Rice University, Houston, TX, USA
{besan,taha}@cs.rice.edu

**Abstract.** Graphical notations are pervasive in circuit design, control systems, and increasingly in mainstream programming environments. Yet many of the foundational advances in programming language theory are taking place in the context of textual notations. In order to map such advances to the graphical world, and to take the concerns of the graphical world into account when working with textual formalisms, there is a need for rigorous connections between textual and graphical expressions of computation.
To this end, this paper presents a graphical calculus called Uccello. Our key insight is that Ariola and Blom's work on sharing in the cyclic lambda calculi provides an excellent foundation for formalizing the semantics of graphical languages. As an example of what can be done with this foundation, we use it to extend a graphical language with staging constructs.

## 1 Introduction

Visual programming languages are finding increasing popularity in a variety of domains, and are often the preferred programming medium for experts in these domains. Examples of such domains include circuit design and control system design, and examples of mainstream tools include a wide range of hardware CAD design environments, data-flow languages like LabVIEW [10, 14], Simulink [20], and Ptolemy [12], spreadsheet-based languages such as Microsoft Excel, or data modeling languages such as UML. Compared to modern text-based languages, many visual languages are limited in expressivity. For example, while they are often purely functional, they generally do not support first-class functions. More broadly, the wealth of abstraction mechanisms, reasoning principles, and type systems developed over the last thirty years is currently available mainly for textual languages. Yet there is real need for migrating many ideas and results developed in the textual setting to the graphical setting.

Recognizing this need, we sought existing accounts of the semantics of graph-based representations of programs, or of formal connections between graph-based representations and visual-representations. The visual programming research literature focuses largely on languages that are accessible to novice programmers and domain-experts, rather than general-purpose calculi. Examples include form-based [4] and

* Supported by NSF ITR-0113569 "Putting Multi-Stage Annotations to Work", Texas ATP 003604-0032-2003 "Advanced Languages Techniques for Device Drivers", and NSF SOD-0439017 "Synthesizing Device Drivers".

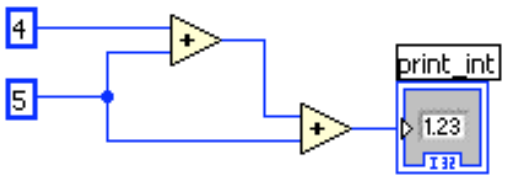
spreadsheet-based [2, 9, 11] languages. Citrin et al. give a purely graphical description of an object-oriented language called VIPR [5] and a functional language called VEX [6], but the mapping to and from textual representations is only treated informally. Erwig [8] presents a denotational semantics for VEX using inductive definitions of graph representations to support pattern matching on graphs, but this style of semantics does not preserve information about the syntax of graphs, as it maps syntax to "meaning".

Our key observation is that Ariola and Blom's work on sharing the cyclic lambda calculus [3] provide an excellent starting point. They establish a formal connection between textual and graph-based representation of programs. The two representations are not one-to-one because of a subtle mismatch between textual and graphical representations in how they express sharing of values. Ariola and Blom overcome this problem by defining a notion of equivalence for terms that represent the same graph, and establish an isomorphism between graphs and equivalence classes of textual terms. In the graph representation, sharing is modeled by having an edge from the output of one component to the inputs of multiple different components. Especially if we are using visual languages to describe circuits, this model of sharing is natural. For example, without sharing, the butterfly circuit for computing the FFT would be exponentially larger [7, Figure 32.5].

We do not know of a notion in textual syntax that corresponds exactly to the notion of sharing provided by graphs. Local variable declarations almost work, but not quite: they only correspond to local declarations that are used *more than once*. For example, the following two C code fragments

```
int x = 4;
int y = 5;                int y = 5;
print_int(x+y+y);         print_int(4+y+y);
```

both correspond to the following LabVIEW graph:

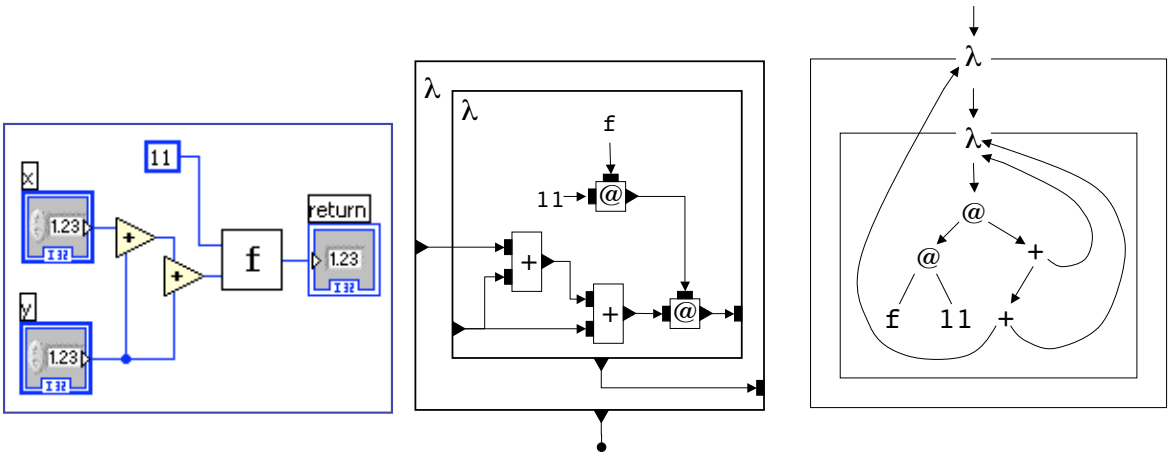



While the first code fragment assigns a local variable name to the constant 4, the second snippet uses the constant 4 directly. But there is no corresponding distinction in a graph. Disallowing variable declarations that are used only once, or requiring all subterms to be explicitly named are not options, because they would be unnatural restrictions for the programmer. They are also problematic from the technical point of view. For example, they are not preserved by standard reasoning principles such as substitution.

We postulate that Ariola and Blom's treatment of the issue of sharing in both representations is a necessary complication in any connection between a textual representation of a programming language with the richness of the lambda calculus and a graphical representation of the same language.

### 1.1 Contributions

To illustrate how the Ariola/Blom connection can be used to map new concepts in programming languages to a graph-based setting, we extend their original calculus with staging constructs typical in textual multi-stage languages [19]. The resulting calculus is based on a one-to-one correspondence between visual programs and a variation of the text-based lambda-calculus. We then use this formal connection to lift the semantics of multi-stage languages to the graphical setting. We show that graph reductions have corresponding reductions at the term level, and similarly, term reductions have corresponding reductions at the graph level.




(a) LabVIEW [14] (b) Uccello (this paper) (c) Ariola and Blom [3]

**Fig. 1.** The syntax of Uccello as middle-ground between that of LabVIEW and lambda-graphs

### 1.2 Organization of this Paper

The rest of the paper is organized as follows. Section 2 explains how the syntax for visual languages such as LabVIEW and Simulink can be modeled using a variation of Ariola and Blom's cyclic lambda-graphs. Section 3 introduces the syntax for a graphical calculus called Uccello. Section 4 defines textual representations for Uccello and shows that graphs and terms in a specific normal form are one-to-one. Section 5 describes a reduction semantics for both terms and graphs, and Section 6 concludes. Proofs for the results presented in this paper are available online [1]

## 2 LabVIEW and Lambda-Graphs

The practical motivation for the calculus studied in the rest of this paper is to extend popular languages such as LabVIEW or Simulink with higher-order functional and staging features. The main abstraction mechanism in LabVIEW is to declare functions;

Figure 1 (a) displays the syntax for defining a function with two formal parameters in LabVIEW. Uccello abstracts away from many of the details of LabVIEW and similar languages. We reduce the complexity of the calculus by supporting only functions with one argument and by making functions first-class values. We can then use nested lambda abstractions to model functions with multiple parameters, as illustrated in Figure 1 (b).

Graph (c) illustrates Ariola and Blom's lambda-graph syntax [3] for the same computation. In this representation, lambda abstractions are drawn as boxes describing the scope of the parameter bound by the abstraction. Edges represent subterm relationships in the syntax tree, and parameter references are drawn as back-edges to a lambda abstraction. While the lambda-graph (c) may appear less closely related to (a) than the Uccello graph (b), note that the graphs (b) and (c) are in fact dual graphs. That is, by flipping the direction of edges in the lambda-graph (c) to represent data-flow instead of subterm relationships, and by making connection points in the graph explicit in the form of ports, we get the Uccello program (b). Based on this observation, we take Ariola and Blom's lambda-graphs as the starting point for our formal development.

# 3 Syntax of Uccello

The core language features of Uccello are function abstraction and function application as known from the $\lambda$-calculus, and the staging constructs Bracket "$\langle\rangle$", Escape "$\sim$", and Run "!". Brackets are a quotation mechanism delaying the evaluation of an expression, while the Escape construct escapes the delaying effect of a Bracket (and so must occur inside a Bracket). Run executes such a delayed computation. The semantics and type theory for these constructs has been studied extensively in recent years [19]. Before defining the syntax of Uccello formally, we give an informal description of its visual syntax. Note that this paper focuses on abstract syntax for both terms and graphs, while issues such as an intuitive concrete syntax and parsing are part of future work (see Section 6).

## 3.1 Visual Syntax

A Uccello program is a graph built from the following components:

1. **Nodes** represent function abstraction, function application, the staging constructs Brackets, Escape, and Run, and "black holes". Black holes are a concept borrowed from Ariola and Blom [3] and represent unresolvable cyclic dependencies that can arise in textual languages with recursion. [1] As shown in Figure 2, nodes are drawn as boxes labeled $\lambda$,@,$\langle\rangle$,$\sim$, !, and $\bullet$ respectively. Each lambda node contains a subgraph inside its box which represents the body of the function, and the node's box visually defines the scope of the parameter bound by the lambda abstraction. Bracket and Escape boxes, drawn using dotted lines, also contain subgraphs. The

[1] In functional languages, recursion is typically expressed using a letrec-construct. The textual program letrec x=x in x introduces a cyclic dependency that cannot be simplified any further. Ariola and Blom visualize such terms as black holes.

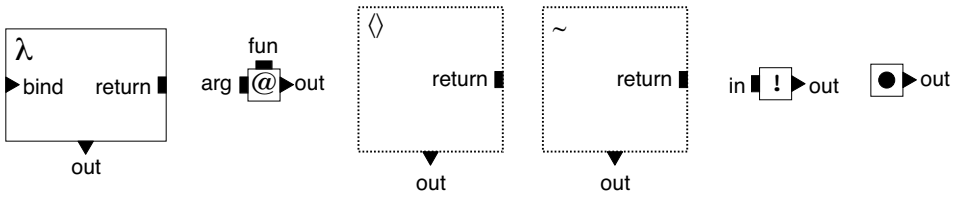



**Fig. 2.** Uccello nodes

subgraph of a Bracket node represents code being generated for a future-stage computation, while the subgraph of an Escape node represents a computation resulting in a piece of code that will be integrated into a larger program at runtime.

2. **Free variables**, displayed as variable names, represent name references that are not bound inside a given Uccello graph.
3. **Ports** mark the points in the graph which edges can connect. We distinguish between *source ports* (drawn as triangles) and *target ports* (drawn as rectangles). As shown in Figure 2, a lambda node provides two source ports: out carries the value of the lambda itself, since functions are first-class values in Uccello. When the function is applied to an argument, then bind carries the function's parameter, and the return port receives the result of evaluating the function body, represented by the lambda node's subgraph. Intuitively, the fun and arg ports of an application node receive the function to be applied and its argument respectively, while out carries the value resulting from the application. The out port of a Bracket node carries the delayed computation represented by the node's subgraph, and return receives the value of that computation when it is executed in a later stage. Conversely, the out port of an Escape node carries a computation that escapes the surrounding Brackets delaying effect, and return receives the value of that computation.
4. **Edges** connect nodes and are drawn as arrows:

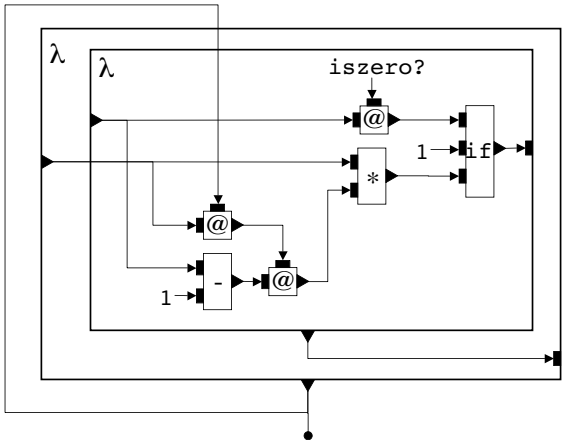

The source of any edge is either the source port of a node or a free variable `x`. The target of any edge is the target port of some node. The only exception to this is the **root** of the graph. Similar to the root of an abstract syntax tree, it marks the entry-point for evaluating the graph. It is drawn as a dangling edge without a target port, instead marked with a dot.

For convenience, the examples in this paper assume that Uccello is extended with integers, booleans, binary integer operators, and conditionals.

*Example 1 (Functional Constructs).* Consider the following recursive definition of the power function in OCaml. The function computes the number $x^n$ for two inputs $x$ and $n$:

```
let rec power = fun x -> fun n ->
    if iszero? n then 1
    else x * (power x (n-1))
in power
```

In Uccello, this program is expressed as follows:




Closely following the textual definition, we visualize the power function as two nested lambda nodes. Consequently, two cascaded application nodes are necessary for the function call `power x (n-1)`. Note that the recursive nature of the definition is represented visually by an edge from the out-port of the outer lambda node back into the lambda box.

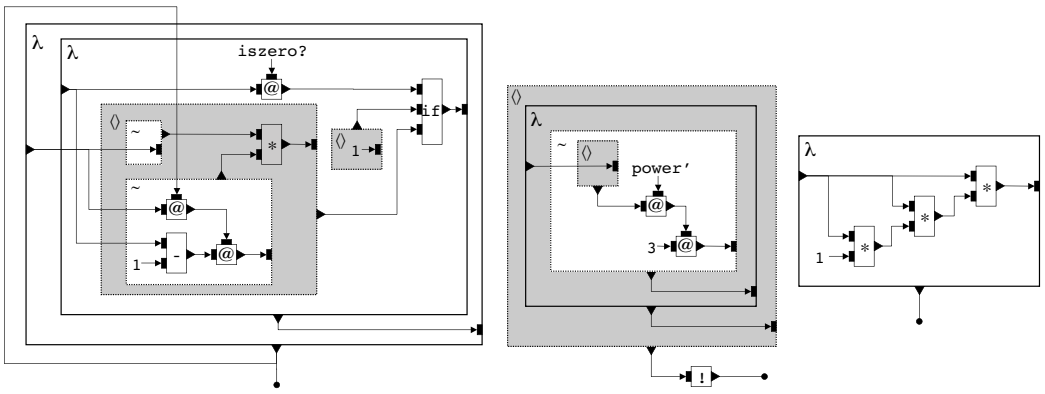



**Fig. 3.** Generating power functions in Uccello

*Example 2 (Multi-stage Constructs).* The power function can be staged by annotating it as follows in MetaOCaml [13]:[2]

```
let rec power' = fun x -> fun n ->
    if iszero? n then .<1>.
    else .<.~x * .~(power' x (n-1))>.
in power'
```

[2] MetaOCaml adds staging constructs to OCaml. Dots are used to disambiguate the concrete syntax: Brackets around an expression *e* are written as `.<`*e*`>.`, an Escaped expression *e* is written as `.~` *e*, and ! *e* is written as `.!`*e*.

The same program is represented in Uccello as shown to the left of Figure 3. As in the text-based program, in Uccello we only need to add a few staging "annotations" (in the form of Bracket and Escape boxes) to the unstaged version of the power function.

*Example 3 (Generating Graphs).* In MetaOCaml, the staged power function can be used to generate efficient specialized power functions by applying the staged version only to its second input (the exponent). For instance, evaluating the term $M_1$:

```
.! .<fun x -> .~(power' .<x>. 3)>.
```

yields the non-recursive function `fun x -> x*x*x*1`. Similarly, evaluating the Uccello graph in the middle of Figure 3 yields the specialized graph on the right side; the graph in the middle triggers the specialization by providing the staged power function with its second input parameter. Note the simplicity of the generated graph. When applying this paradigm to circuit generation, controlling the complexity of resulting circuits can be essential, and staging constructs were specifically designed to give the programmer more control over the structure of generated programs.

### 3.2 Formal Syntax

The following syntactic sets are used for defining Uccello graphs:

$$
\begin{array}{rcl}
\textit{Nodes} & u, v, w & \in \mathbb{V} \\
\textit{Free variables} & x, y & \in \mathbb{X} \\
\textit{Source port types} & o & \in \mathbb{O} ::= \mathsf{bind} \mid \mathsf{out} \\
\textit{Target port types} & i & \in \mathbb{I} ::= \mathsf{return} \mid \mathsf{fun} \mid \mathsf{arg} \mid \mathsf{in} \\
\textit{Source ports} & r, s & \in \mathbb{S} ::= v.o \mid x \\
\textit{Target ports} & t & \in \mathbb{T} ::= v.i \\
\textit{Edges} & e & \in \mathbb{E} ::= (s, t)
\end{array}
$$

As a convention, we use regular capital letters to denote concrete sets. For example, $E \subseteq \mathbb{E}$ stands for a concrete set of edges $e$. We write $\mathcal{P}(V)$ to denote the power set of $V$.

A Uccello **graph** is then defined as a tuple $g = (V, L, E, S, r)$ where $V$ is a finite set of **nodes**, $L : V \to \{\lambda, @, \langle\rangle, \sim, !\ , \bullet\}$ is a **labeling function** that associates each node with a label, $E$ is a finite set of **edges**, $S : \{v \in V \mid L(v) \in \{\lambda, \langle\rangle, \sim\}\} \to \mathcal{P}(V)$ is a **scoping function** that associates each lambda, Bracket, and Escape node with a subgraph, and $r$ is the **root** of the graph. When it is clear from the context, we refer to the components $V$, $L$, $E$, $S$, and $r$ of a graph $g$ without making the binding $g = (V, L, E, S, r)$ explicit.

### 3.3 Auxiliary Definitions

For any Uccello graph $g = (V, L, E, S, r)$ we define the following auxiliary notions. The set of **incoming edges** of a node $v \in V$ is defined as $pred(v) = \{(s, v.i) \in E\}$ for any edge targets $i$. Given a set $U \subseteq V$, the set of **top-level nodes** in $U$ that are not in the scope of any other node in $U$ is defined as $toplevel(U) = \{u \in U \mid \forall v \in U : u \in S(v) \Rightarrow v = u\}$. If $v \in V$ has a scope, then the **contents** of $v$ are defined as $contents(v) = S(v) \backslash \{v\}$. For a given node $v \in V$, if there exists a node $u \in V$ with $v \in toplevel(contents(u))$,

then $u$ is a **surrounding scope** of $v$. Well-formedness conditions described in the next section will ensure that such a surrounding scope is unique when it exists. A **path** $v \rightsquigarrow w$ in $g$ is an acyclic path from $v \in V$ to $w \in V$ that only consists of edges in $\{(s,t) \in E \mid \forall u : s \neq u.\mathsf{bind}\}$. The negative condition excludes edges starting at a bind port.

### 3.4 Well-Formed Graphs

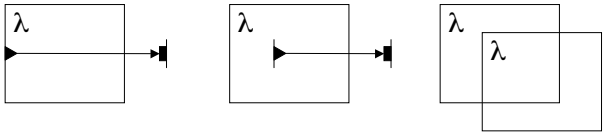
Whereas context-free grammars are generally sufficient to describe well-formed terms in textual programming languages, characterizing well-formed graphs (in particular with respect to scoping) is more subtle. The well-formedness conditions for the functional features of Uccello are taken directly from Ariola and Blom. Since Bracket and Escape nodes also have scopes, these conditions extend naturally to the multi-stage features of Uccello. Note however that the restrictions associated with Bracket and Escape are simpler since unlike lambdas these are not binding constructs.

The set $\mathbb{G}$ of **well-formed graphs** is the set of graphs that satisfy the following conditions:

1. **Connectivity** - Edges may connect ports belonging only to nodes in $V$ with the correct port types. Valid *inports* and *outports* for each node type are defined as follows:

| $L(v)$ | *inports*(v) | *outports*(v) |
|---|---|---|
| $\lambda$ | {return} | {bind, out} |
| @ | {fun, arg} | {out} |
| $\langle\rangle, \sim$ | {return} | {out} |
| ! | {in} | {out} |
| $\bullet$ | $\emptyset$ | {out} |

   We require that an edge $(v.o, w.i)$ connecting nodes $v$ and $w$ is in $E$ only if $v, w \in V$ and $o \in \mathit{outports}(v)$ and $i \in \mathit{inports}(w)$. Similarly, an edge $(x, w.i)$ originating from a free variable $x$ is in $E$ only if $w \in V$ and $i \in \mathit{inports}(w)$.
   We also restrict the in-degree of nodes: each target port (drawn as a rectangle) in the graph must be the target of exactly one edge, while a source port (drawn as a triangle) can be unused, used by one or shared by multiple edges. Thus we require for any node $v$ in the graph that $\mathit{pred}(v) = \{(s, v.i) \mid i \in \mathit{inports}(v)\}$.
2. **Scoping** - Intuitively, source ports in Uccello correspond to bound names in textual languages, and scopes are drawn as boxes. Let $w, w_1, w_2 \in V$ and $v, v_1, v_2 \in \mathit{dom}(S)$ be distinct nodes. By convention, all nodes that have a scope must be in their own scope ($v \in S(v)$). The following three graph fragments illustrate three kinds of scoping errors that can arise:



A name used outside the scope where it is bound corresponds to an edge from a bind or an out port that *leaves* a scope. We prohibit the first case by requiring that $(v.\mathsf{bind}, t) \in \mathit{pred}(w)$ only if $w \in S(v)$. For the second case, we require that if $w_1 \notin S(v)$ and $w_2 \in S(v)$ and $(w_2.\mathsf{out}, t) \in \mathit{pred}(w_1)$ then $w_2 = v$. Partially overlapping scopes correspond to overlapping lambda, Bracket, or Escape boxes. We disallow this by requiring that $S(v_1) \cap S(v_2) = \emptyset$ or $S(v_1) \subseteq S(v_2) \setminus \{v_2\}$ or $S(v_2) \subseteq S(v_1) \setminus \{v_1\}$.

3. **Root Condition** - The root $r$ cannot be the port of a node nested in the scope of another node. Therefore, the root must either be a free variable ($r \in \mathbb{X}$) or the out port of a node $w$ that is visible at the "top-level" of the graph ($r = w.\mathsf{out}$ and $w \in \mathit{toplevel}(V)$).

# 4 Graph-Term Connection

To develop the connection between Uccello graphs and their textual representations, this section begins by defining a term language and a translation from graphs to terms. Not all terms can be generated using this translation, but rather only terms in a specific normal form. A backward-translation from terms to graphs is then defined, and it is shown that a term in normal form represents all terms that map to the same graph. Finally, sets of graphs and normal forms are shown to be in one-to-one correspondence.

## 4.1 From Graphs to Terms

Building on Ariola and Blom's notion of cyclic lambda terms, we use *staged* cyclic lambda terms to represent Uccello programs textually, and define them as follows:

$$\begin{aligned} \textit{Terms } M \in \mathbb{M} ::= {} & x \mid \lambda x.M \mid M\ M \mid \mathsf{letrec}\ d^*\ \mathsf{in}\ M \\ & \mid \sim M \mid \langle M \rangle \mid\ !\ M \\ \textit{Declarations } \ d \in \mathbb{D} \ ::= {} & x = M \end{aligned}$$

**Conventions:** By assumption, all recursion variables $x$ in letrec declarations are distinct, and the sets of bound and free variables are disjoint. We write $d^*$ for a (possibly empty) sequence of letrec declarations $d$. Different permutations of the same sequence of declarations $d^*$ are identified. Therefore, we often use the set notation $D$ instead of $d^*$. Given two sequences of declarations $D_1$ and $D_2$, we write $D_1, D_2$ for the concatenation of the two sequences. We write $M_1[x := M_2]$ for the result of substituting $M_2$ for all free occurrences of the variable $x$ in $M_1$, without capturing any free variables in $M_2$. We use $\equiv_\alpha$ to denote syntactic equality up to $\alpha$-renaming of both lambda-bound variables and recursion variables.

To translate a graph into a term, we define the **term construction** $\tau : \mathbb{G} \to \mathbb{M}$. Intuitively, this translation associates all nodes in the graph with a unique variable name in the term language. These variables are used to explicitly name each subterm of the resulting term. Lambda nodes are associated with an additional variable name, which is used to name the formal parameter of the represented lambda abstraction.

**Definition 1 (Term construction).** *Let $g = (V, L, E, S, r)$ be a well-formed graph in $\mathbb{G}$.*

1. *For every node $v \in V$, we define a unique name $x_v$, and a second distinct name $y_v$ if $L(v) = \lambda$. We then associate a name with each edge source $s$ in the graph as follows:*

$$name(s) = \begin{cases} x_v & \text{if } s = v.\mathsf{out} \\ y_v & \text{if } s = v.\mathsf{bind} \\ x & \text{if } s = x \end{cases}$$

2. *To avoid the construction of empty letrec terms* (letrec _ in $M$) *in the translation, we use the following function:*

$$mkrec(D, M) = \begin{cases} M & \text{if } D = \emptyset \\ \mathsf{letrec}\ D\ \mathsf{in}\ M & \text{otherwise} \end{cases}$$

3. *We construct a term corresponding to each node $v \in V$:*

$$\frac{L(v) = \bullet \quad pred(v) = \emptyset}{term(v) = x_v}$$

$$\frac{L(v) = \lambda \quad pred(v) = \{(s, v.\mathsf{return})\}}{term(v) = \lambda y_v.mkrec(decl(contents(v)), name(s))}$$

$$\frac{L(v) = @ \quad pred(v) = \{(s_1, v.\mathsf{fun}), (s_2, v.\mathsf{arg})\}}{term(v) = name(s_1)\ name(s_2)}$$

$$\frac{L(v) = \langle\rangle \quad pred(v) = \{(s, v.\mathsf{return})\}}{term(v) = \langle mkrec(decl(contents(v)), name(s))\rangle}$$

$$\frac{L(v) = \sim \quad pred(v) = \{(s, v.\mathsf{return})\}}{term(v) = \sim mkrec(decl(contents(v)), name(s))}$$

$$\frac{L(v) = ! \quad pred(v) = \{(s, v.\mathsf{in})\}}{term(v) = !\ name(s)}$$

4. *We construct letrec declarations for any set of nodes $W \subseteq V$:*

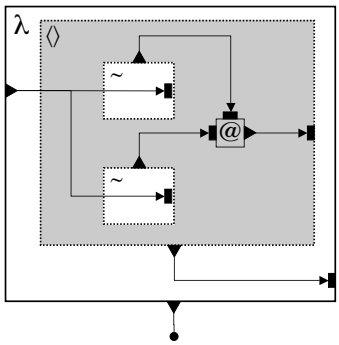
$$decl(W) = \{x_v = term(v) \mid v \in toplevel(W)\}$$

5. *The term construction $\tau$ is then defined as:*

$$\tau(g) = mkrec(decl(V), name(r))$$

The translation $\tau$ starts by computing the set of top-level nodes in $V$ (see Section 3.3), and creates a letrec declaration for each of these nodes. For a node $v$ with no subgraph, the letrec declaration binds the variable $x_v$ to a term that combines the variables associated with the incoming edges to $v$. If $v$ contains a subgraph, then $\tau$ is applied recursively to the subgraph, and $x_v$ is bound to the term that represents the subgraph. The constraint $v \in toplevel(W)$ in the definition of *decl* ensures that exactly one equation is generated for each node: if $v \notin toplevel(W)$, then $v$ is in the scope of a different node $w \in W$, and an equation for $w$ is instead included in $term(w)$.

*Example 4 (Term Construction).* The function $\tau$ translates the graph




as follows: Let $v_1$ be the lambda node, $v_2$ the Bracket node, $v_3$ and $v_4$ the top and bottom Escape nodes, and $v_5$ the application node in the graph $g$. We associate a variable name $x_j$ with each node $v_j$. In addition, the name $y_1$ is associated with the parameter of the lambda node $v_1$. The result is:

$$\begin{array}{l}\mathsf{letrec}\ x_1 = \lambda y_1.(\mathsf{letrec}\ x_2 = \langle \mathsf{letrec}\ x_3 = \sim y_1, x_4 = \sim y_1, x_5 = x_3 x_4 \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{in}\ x_5\rangle \\ \qquad\qquad\qquad\qquad \mathsf{in}\ x_2) \\ \mathsf{in}\ x_1\end{array}$$

All nodes are in the scope of $v_1$ so it is the only "top-level" node in $g$. We create a letrec declaration for $v_1$, binding $x_1$ to a term $\lambda y_1.N$ where $N$ is the result of recursively translating the subgraph inside $v_1$. When translating the subgraph of the Bracket node $v_2$, note that this subgraph contains three top-level nodes ($v_3$, $v_4$, $v_5$). Therefore, the term for $v_2$ contains three variable declarations ($x_3$, $x_4$, $x_5$).

### 4.2 Terms in Normal Form

The term construction function $\tau$ only constructs terms in a very specific form. For example, while the graph in the previous example represents the computation $\lambda y_1.\langle\sim y_1 \sim y_1\rangle$, the example shows that $\tau$ constructs a different term. Compared to $\lambda y_1.\langle\sim y_1 \sim y_1\rangle$, every subterm in the constructed term is explicitly named using letrec. This explicit naming of subterms expresses the notion of value sharing in Uccello graphs, where the output port of *any* node can be the source of multiple edges. Such **normal forms** are essentially the same as A-normal form [17], and can be defined as follows:

$$\begin{array}{rl} \textit{Terms}\ N \in \mathbb{M}_{norm} & ::= x \mid \mathsf{letrec}\ q^+\ \mathsf{in}\ x \\ \textit{Declarations}\ \ q \in \mathbb{D}_{norm} & ::= x = x \mid x = y\ z \mid x = \lambda y.N \\ & \quad \mid x = \langle N\rangle \mid x = \sim N \mid x = !\ y \end{array}$$

where $q^+$ is a non-empty sequence of declarations $q$. In normal forms, nested terms are only allowed in function bodies and inside Brackets or Escapes, i.e. only for language constructs that correspond to nodes with subgraphs. All other expressions are explicitly named using letrec declarations, and pure "indirection" declarations of the form $x = y$ with $x \not\equiv y$ are not allowed.

**Lemma 1 (Normal forms are terms).** $\mathbb{M}_{norm} \subseteq \mathbb{M}$.

**Lemma 2 ($\tau$ maps graphs to normal forms).** *If* $g \in \mathbb{G}$ *then* $\tau(g) \in \mathbb{M}_{norm}$.

As we will show, $\tau$ is an injection, i.e. not every term corresponds to a distinct graph. However, we will show that every term has a normal form associated with it, and that these normal forms are one-to-one with graphs. To this end, we define the **normalization function** $\nu : \mathbb{M} \to \mathbb{M}_{norm}$ in two steps: general terms are first mapped to **intermediate forms**, which are then converted into normal forms in a second pass. We define the set $\mathbb{M}_{pre}$ of intermediate forms as follows:

$$\begin{aligned} \textit{Terms } N' \in \mathbb{M}_{pre} &::= x \mid \mathsf{letrec}\ q'^{*}\ \mathsf{in}\ x \\ \textit{Declarations } q' \in \mathbb{D}_{pre} &::= x = y \mid x = y\ z \mid x = \lambda y.N' \\ &\quad \mid x = \langle N' \rangle \mid x = \sim N' \mid x = !\ y \end{aligned}$$

Note that this set consists of normal forms with fewer restrictions: empty letrec terms and indirections of the form $x = y$ are allowed.

$$\frac{}{[\![x]\!]_{pre} = \mathsf{letrec}\ \_\ \mathsf{in}\ x} \qquad \frac{[\![M]\!]_{pre} = N' \quad x_1 \text{ fresh}}{[\![\lambda x.M]\!]_{pre} = (\mathsf{letrec}\ x_1 = \lambda x.N'\ \mathsf{in}\ x_1)}$$

$$\frac{[\![M_1]\!]_{pre} = \mathsf{letrec}\ Q_1\ \mathsf{in}\ x_1 \quad [\![M_2]\!]_{pre} = \mathsf{letrec}\ Q_2\ \mathsf{in}\ x_2 \quad x_3 \text{ fresh}}{[\![M_1 M_2]\!]_{pre} = (\mathsf{letrec}\ Q_1, Q_2, x_3 = x_1 x_2\ \mathsf{in}\ x_3)}$$

$$\frac{[\![M]\!]_{pre} = \mathsf{letrec}\ Q\ \mathsf{in}\ y \quad \overrightarrow{[\![M_j]\!]_{pre} = \mathsf{letrec}\ Q_j\ \mathsf{in}\ y_j}}{[\![\mathsf{letrec}\ \overrightarrow{x_j = M_j}\ \mathsf{in}\ M]\!]_{pre} = (\mathsf{letrec}\ Q, \overrightarrow{Q_j, x_j = y_j}\ \mathsf{in}\ y)}$$

$$\frac{[\![M]\!]_{pre} = N' \quad x_1 \text{ fresh}}{[\![\langle M \rangle]\!]_{pre} = (\mathsf{letrec}\ x_1 = \langle N' \rangle\ \mathsf{in}\ x_1)} \qquad \frac{[\![M]\!]_{pre} = N' \quad x_1 \text{ fresh}}{[\![\sim M]\!]_{pre} = (\mathsf{letrec}\ x_1 = \sim N'\ \mathsf{in}\ x_1)}$$

$$\frac{[\![M]\!]_{pre} = \mathsf{letrec}\ Q\ \mathsf{in}\ y \quad x_1 \text{ fresh}}{[\![!\ M]\!]_{pre} = (\mathsf{letrec}\ Q, x_1 = !\ y\ \mathsf{in}\ x_1)}$$

$$\frac{}{[\![N]\!]_{norm} = N} \qquad \frac{}{[\![\mathsf{letrec}\ \_\ \mathsf{in}\ x]\!]_{norm} = x}$$

$$\frac{N' \notin \mathbb{M}_{norm} \quad [\![N']\!]_{norm} = N_1 \quad [\![\mathsf{letrec}\ y = \lambda z.N_1, Q\ \mathsf{in}\ x]\!]_{norm} = N_2}{[\![\mathsf{letrec}\ y = \lambda z.N', Q\ \mathsf{in}\ x]\!]_{norm} = N_2}$$

$$\frac{N' \notin \mathbb{M}_{norm} \quad [\![N']\!]_{norm} = N_1 \quad [\![\mathsf{letrec}\ y = \langle N_1 \rangle, Q\ \mathsf{in}\ x]\!]_{norm} = N_2}{[\![\mathsf{letrec}\ y = \langle N' \rangle, Q\ \mathsf{in}\ x]\!]_{norm} = N_2}$$

$$\frac{N' \notin \mathbb{M}_{norm} \quad [\![N']\!]_{norm} = N_1 \quad [\![\mathsf{letrec}\ y = \sim N_1, Q\ \mathsf{in}\ x]\!]_{norm} = N_2}{[\![\mathsf{letrec}\ y = \sim N', Q\ \mathsf{in}\ x]\!]_{norm} = N_2}$$

$$\frac{[\![(\mathsf{letrec}\ Q\ \mathsf{in}\ x)[y := z]]\!]_{norm} = N \quad y \not\equiv z}{[\![\mathsf{letrec}\ y = z, Q\ \mathsf{in}\ x]\!]_{norm} = N}$$

**Fig. 4.** The translation functions $[\![\_]\!]_{pre} : \mathbb{M} \to \mathbb{M}_{pre}$ and $[\![\_]\!]_{norm} : \mathbb{M}_{pre} \to \mathbb{M}_{norm}$

**Definition 2 (Term Normalization).** *Given the definitions of the translations* $[\![\_]\!]_{pre} : \mathbb{M} \to \mathbb{M}_{pre}$ *and* $[\![\_]\!]_{norm} : \mathbb{M}_{pre} \to \mathbb{M}_{norm}$ *in Figure 4, we define the normalization function* $\nu : \mathbb{M} \to \mathbb{M}_{norm}$ *by composition:* $\nu = [\![\_]\!]_{norm} \circ [\![\_]\!]_{pre}$.

The translation $[\![\_]\!]_{pre}$ maps any term $M$ to a letrec term, assigning a fresh letrec variable to each subterm of $M$. We preserve the nesting of lambda abstractions, Bracket and Escapes by applying $[\![\_]\!]_{pre}$ to subterms recursively. [3] Once every subterm has a letrec variable associated with it, and all lambda, Bracket, and Escape subterms are normalized recursively, the function $[\![\_]\!]_{norm}$ eliminates empty letrec terms and letrec indirections of the form $x = y$ (where $x \not\equiv y$) using substitution. The clause $N' \notin \mathbb{M}_{norm}$ in the definition of $[\![\_]\!]_{norm}$ ensures that normalization terminates: without this restriction we could apply $[\![\_]\!]_{norm}$ to a fully normalized term without making any progress.

*Example 5 (Term Normalization).* Given the following terms:

$$
\begin{aligned}
M_1 &\equiv \lambda x.\langle\sim x \ \ \sim x\rangle \\
M_2 &\equiv \mathsf{letrec}\ y = \lambda x.\langle\sim x \ \ \sim x\rangle\ \mathsf{in}\ y \\
M_3 &\equiv \lambda x.\mathsf{letrec}\ y = \langle\sim x \ \ \sim x\rangle\ \mathsf{in}\ y
\end{aligned}
$$

Then $\nu(M_1)$, $\nu(M_2)$, and $\nu(M_3)$ all yield a term alpha-equivalent to:

$$
\begin{aligned}
&\mathsf{letrec}\ y_1 = \lambda x.(\mathsf{letrec}\ y_2 = \langle \mathsf{letrec}\ y_3 = \sim x, y_4 = \sim x, y_5 = y_3\ y_4 \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{in}\ y_5\rangle \\
&\qquad\qquad\qquad\qquad \mathsf{in}\ y_2) \\
&\mathsf{in}\ y_1
\end{aligned}
$$

Note that the basic structure of the original terms (lambda term with Bracket body and application of two escaped parameter references inside) is preserved by normalization, but every subterm is now named explicitly.

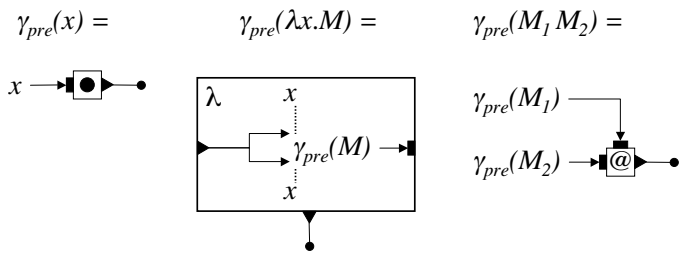
**Lemma 3 ($\nu$ maps terms to normal forms).** *If* $M \in \mathbb{M}$ *then* $\nu(M) \in \mathbb{M}_{norm}$.

### 4.3 From Terms to Graphs

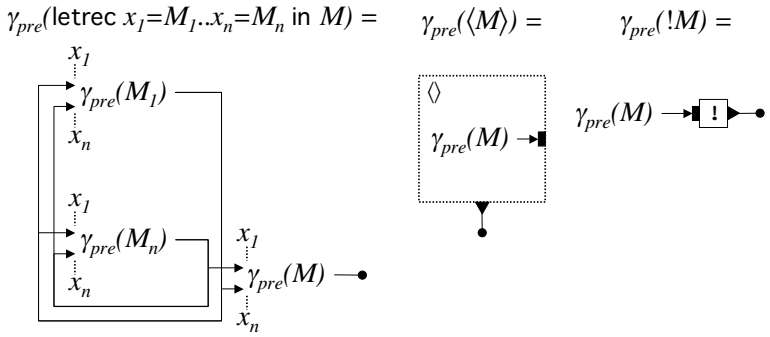
To simplify the definition of a translation from terms to graphs, we introduce a notion analogous to Ariola and Blom's scoped pre-graphs. The set $\mathbb{G}_{pre}$ of **intermediate graphs** consists of all graphs for which a well-formedness condition is relaxed: nodes with label $\bullet$ may have 0 or 1 incoming edge. Formally, whenever $L(v) = \bullet$ then $pred(v) = \emptyset$ or $pred(v) = \{(s, v.\mathsf{in})\}$. If such a node has 1 predecessor, we call it an **indirection node**. Since free variables are not represented as nodes in Uccello, the idea is to associate an indirection node with each variable occurrence in the translated lambda-term. This simplifies connecting subgraphs constructed during the translation, as it provides "hooks" for connecting bound variable occurrences in the graph to their binders. We will also use indirection nodes to model intermediate states in the graph reductions presented in Section 5.2.

[3] This is similar to the translation $\tau$ from graphs to terms presented above, where lambda, Bracket and Escape *nodes* are translated to terms recursively.

We translate terms to Uccello graphs in two steps: A function $\gamma_{pre}$ maps terms to intermediate graphs, and a simplification function $\sigma$ maps intermediate graphs to proper Uccello graphs. Before defining these translations formally, we give visual descriptions of $\gamma_{pre}$ and $\sigma$.

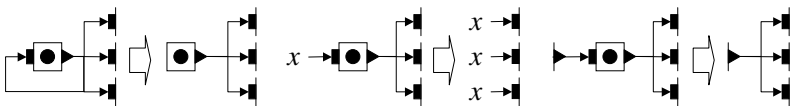



A free variable $x$ is mapped by $\gamma_{pre}$ to an indirection node with $x$ connected to its in port. A lambda term $\lambda x.M$ maps to a lambda node $v$, where the pre-graph for $M$ becomes the subgraph of $v$ and all free variables $x$ in the subgraph are replaced by edges originating at the lambda node's bind port. An application $M_1 M_2$ translates to an application node $v$ where the roots of the pre-graphs for $M_1$ and $M_2$ are connected to the fun and arg ports of $v$.




Given a letrec term (letrec $x_1 = M_1, .., x_n = M_n$ in $M$), $\gamma_{pre}$ translate the terms $M_1$ through $M_n$ and $M$ individually. The root of the resulting pre-graph is the root of $\gamma_{pre}(M)$. Any edge that starts with one of the free variable $x_j$ is replaced by an edge from the root of the corresponding graph $\gamma_{pre}(M_j)$. The cases for $\langle M\rangle$ and $\sim M$ are treated similarly to the case for $\lambda x.M$, and the case for $!M$ is treated similarly to the case for application.

Simplification eliminates indirection nodes from the pre-graph using the following local graph transformations:




Any indirection node with a self-loop (i.e. there is an edge from its out port to its in port) is replaced by a black hole. If there is an edge from a free variable $x$ or from a different node's port $s$ to an indirection node $v$, then the indirection node is "skipped" by replacing all edges originating at $v$ to edges originating at $x$ or $s$. Note that the second and third cases are different since free variables cannot be shared in Uccello.

To define these translations formally, we use the following notation: $E[s_1 := s_2]$ denotes the result of substituting any edge in $E$ that originates from $s_1$ with an edge that starts at $s_2$:

$$E[s_1 := s_2] = \{(s,t) \in E \mid s \neq s_1\} \cup \{(s_2,t) \mid (s_1,t) \in E\}$$

$S\backslash u$ stands for the result of removing node $u$ from any scope in the graph: $(S\backslash u)(v) = S(v)\backslash\{u\}$. The substitution $r[s_1 := s_2]$ results in $s_2$ if $r = s_1$ and in $r$ otherwise.

$$\frac{v \text{ fresh}}{\gamma_{pre}(x) = (\{v\}, \{v \mapsto \bullet\}, \{(x, v.\mathsf{in})\}, \emptyset, v.\mathsf{out})}$$

$$\frac{\gamma_{pre}(M) = (V, L, E, S, r) \quad v \text{ fresh}}{\gamma_{pre}(\lambda x.M) = (V \uplus \{v\}, L \uplus \{v \mapsto \lambda\}, E[x := v.\mathsf{bind}] \uplus \{(r, v.\mathsf{return})\}, S \uplus \{v \mapsto V \uplus \{v\}\}, v.\mathsf{out})}$$

$$\frac{\gamma_{pre}(M_1) = (V_1, L_1, E_1, S_1, r_1) \quad \gamma_{pre}(M_2) = (V_2, L_2, E_2, S_2, r_2) \quad v \text{ fresh}}{\gamma_{pre}(M_1 M_2) = (V_1 \uplus V_2 \uplus \{v\}, L_1 \uplus L_2 \uplus \{v \mapsto @\}, E_1 \uplus E_2 \uplus \{(r_1, v.\mathsf{fun}), (r_2, v.\mathsf{arg})\}, S_1 \uplus S_2, v.\mathsf{out})}$$

$$\frac{\overrightarrow{\gamma_{pre}(M_j) = (V_j, L_j, E_j, S_j, r_j)} \quad \gamma_{pre}(M) = (V, L, E, S, r) \quad v \text{ fresh}}{\gamma_{pre}(\mathsf{letrec}\ \overrightarrow{x_j = M_j}\ \mathsf{in}\ M) = (V \uplus \overrightarrow{V_j}, L \uplus \overrightarrow{L_j}, (E \uplus \overrightarrow{E_j})[\overrightarrow{x_j := r_j}], S \uplus \overrightarrow{S_j}, r)}$$

$$\frac{\gamma_{pre}(M) = (V, L, E, S, r) \quad v \text{ fresh}}{\gamma_{pre}(\langle M \rangle) = (V \uplus \{v\}, L \uplus \{v \mapsto \langle\rangle\}, E \uplus \{(r, v.\mathsf{return})\}, S \uplus \{v \mapsto V \uplus \{v\}\}, v.\mathsf{out})}$$

$$\frac{\gamma_{pre}(M) = (V, L, E, S, r) \quad v \text{ fresh}}{\gamma_{pre}(\sim M) = (V \uplus \{v\}, L \uplus \{v \mapsto \sim\}, E \uplus \{(r, v.\mathsf{return})\}, S \uplus \{v \mapsto V \uplus \{v\}\}, v.\mathsf{out})}$$

$$\frac{\gamma_{pre}(M) = (V, L, E, S, r) \quad v \text{ fresh}}{\gamma_{pre}(!\ M) = (V \uplus \{v\}, L \uplus \{v \mapsto !\}, E \uplus \{(r, v.\mathsf{in})\}, S, v.\mathsf{out})}$$

$$\frac{\forall v \in V : L(v) = \bullet \Rightarrow \mathrm{pred}(v) = \emptyset}{\sigma(V, L, E, S, r) = (V, L, E, S, r)}$$

$$\frac{\sigma(V \uplus \{v\}, L \uplus \{v \mapsto \bullet\}, E, S, r) = g}{\sigma(V \uplus \{v\}, L \uplus \{v \mapsto \bullet\}, E \uplus \{(v.\mathsf{out}, v.\mathsf{in})\}, S, r) = g}$$

$$\frac{s \neq v.\mathsf{out} \quad (v.\mathsf{out}, t) \notin E \quad \sigma(V, L, E \uplus \{\overrightarrow{(s, t_j)}\}, S\backslash v, r[v.\mathsf{out} := s]) = g}{\sigma(V \uplus \{v\}, L \uplus \{v \mapsto \bullet\}, E \uplus \{(s, v.\mathsf{in})\} \uplus \{\overrightarrow{(v.\mathsf{out}, t_j)}\}, S, r) = g}$$

**Fig. 5.** The translation functions $\gamma_{pre} : \mathbb{M} \to \mathbb{G}_{pre}$ and $\sigma : \mathbb{G}_{pre} \to \mathbb{G}$

**Definition 3 (Graph construction).** *Given the definitions of the translations $\gamma_{pre} : \mathbb{M} \to \mathbb{G}_{pre}$ and $\sigma : \mathbb{G}_{pre} \to \mathbb{G}$ in Figure 5, we define the graph construction $\gamma : \mathbb{M} \to \mathbb{G}$ by composition: $\gamma = \sigma \circ \gamma_{pre}$.*

**Lemma 4 ($\gamma$ maps terms to well-formed graphs).** *For any $M \in \mathbb{M}$, $\gamma(M)$ is defined and is a unique, well-formed graph.*

Using the mappings $\nu$, $\gamma$, and $\tau$, we can now give a precise definition of the connections between terms, graphs, and normal forms. Two terms map to the same graph if and only if they have the same normal form. Thus, normal forms represent equivalence classes of terms that map to the same graph by $\gamma$. The function $\nu$ gives an algorithm for computing such representative terms. Given two well-formed graphs $g, h \in \mathbb{G}$, we write $g = h$ if $g$ and $h$ are isomorphic graphs with identical node labels.

**Lemma 5 (Soundness of Normalization).** *If $M \in \mathbb{M}$. then $\gamma(M) = \gamma(\nu(M))$.*

**Lemma 6 (Recovery of normal forms).** *If $N \in \mathbb{M}_{norm}$ then $N \equiv_\alpha \tau(\gamma(N))$.*

**Lemma 7 (Completeness of Normalization).** *Let $M_1, M_2 \in \mathbb{M}$. If $\gamma(M_1) = \gamma(M_2)$ then $\nu(M_1) \equiv_{alpha} \nu(M_2)$.*

*Example 6.* In Example 5 we showed that the three terms $M_1$, $M_2$, and $M_3$ have the same normal form. By Lemma 5, they translate to the same graph. This graph is shown in Example 4. By Lemma 7, the terms $M_1$, $M_2$, and $M_3$ must have the same normal form since they map to the same graph by $\gamma$.

**Theorem 1 (Correctness of Graphical Syntax).** *Well-formed graphs and normal forms are one-to-one:*

1. *If $M \in \mathbb{M}$ then $\nu(M) \equiv_{alpha} \tau(\gamma(M))$.*
2. *If $g \in \mathbb{G}$ then $g = \gamma(\tau(g))$.*

# 5 Semantics For Uccello

This section presents a reduction semantics for staged cyclic lambda terms and graphs, and establishes the connection between the two.

## 5.1 Staged Terms

Ariola and Blom study a call-by-need reduction semantics for the lambda-calculus extended with a letrec construct. In order to extend this semantics to support staging constructs, we use the notion of *expression families* proposed for the reduction semantics of call-by-name $\lambda$-U [18]. In the context of $\lambda$-U, expression families restrict beta-redeces to terms that are valid at level 0. Intuitively, given a staged term $M$, the **level** of a subterm of $M$ is the number of Brackets minus the number of Escapes surrounding the subterm. A term $M$ is **valid at level** $n$ if all Escapes inside $M$ occur at a level greater than $n$.

*Example 7.* Consider the lambda term $M \equiv \langle \lambda x.\ \sim (f\langle x \rangle) \rangle$. The variable $f$ occurs at level 0, while the use of $x$ occurs at level 1. Since the Escape occurs at level 1, $M$ is valid at level 0.

The calculus $\lambda$-U does not provide a letrec construct to directly express sharing in lambda terms. Therefore, we extend the notion of expression families to include the letrec construct as follows:

$$\begin{aligned} M^0 \in \mathbb{M}^0 \quad &::= x \mid \lambda x.M^0 \mid M^0 M^0 \mid \mathsf{letrec}\ D^0\ \mathsf{in}\ M^0 \\ &\quad \mid \langle M^1 \rangle \mid !\ M^0 \\ M^{n+} \in \mathbb{M}^{n+} \ &::= x \mid \lambda x.M^{n+} \mid M^{n+} M^{n+} \mid \mathsf{letrec}\ D^{n+}\ \mathsf{in}\ M^{n+} \\ &\quad \mid \langle M^{n++} \rangle \mid \sim M^n \mid !\ M^{n+} \\ D^n \in \mathbb{D}^n \quad &::= \overrightarrow{x_j = M_j^n} \end{aligned}$$

In order to combine Ariola and Blom's reduction semantics for cyclic lambda-terms with the reduction semantics for $\lambda$-U, we need to account for the difference in beta-reduction between the two formalisms: While $\lambda$-U is based on a standard notion of substitution, Ariola and Blom's beta-rule uses the letrec construct to express a binding from the applied function's parameter to the argument of the application, without immediately substituting the argument for the function's parameter. Instead, substitution is performed on demand by a separate reduction rule. Furthermore, substitution in $\lambda$-U is restricted (implicitly by the $\beta$-rule) to $M^0$-terms. We make this restriction explicit by defining which contexts are valid at different levels:

$$\begin{aligned} C \in \mathbb{C} \quad &::= \square \mid \lambda x.C \mid C\ M \mid M\ C \mid \mathsf{letrec}\ D\ \mathsf{in}\ C \\ &\quad \mid \mathsf{letrec}\ x = C, D\ \mathsf{in}\ M \mid \langle C \rangle \mid \sim C \mid !\ C \\ C^n \in \mathbb{C}^n \quad &= \{C \in \mathbb{C} \mid C[x] \in \mathbb{M}^n\} \end{aligned}$$

We write $C[M]$ for the result of replacing the hole $\square$ in $C$ with $M$, potentially capturing free variables in $M$ in the process. Furthermore, we adopt the notation $D \perp M$ from [3] to denote that the set of variables occurring as the left-hand side of a letrec declaration in $D$ does not intersect with the set of free variables in $M$.

Using these families of terms and contexts, we extend Ariola and Blom's reductions as shown in Figure 6. We write $\rightarrow$ for the compatible extension of the rules in $\mathcal{R}$, and we write $\rightarrow^*$ for the reflexive and transitive closure of $\rightarrow$. The idea behind the rules *sub* is to perform substitution on demand after a function application has been performed. In this sense, the reduction rules *sub* and the rule $\beta\circ$ together mimic the behavior of beta-reduction in $\lambda$-U.

## 5.2 Staged Graphs

To define a reduction semantics for Uccello, we define similar notions as used in the previous section: the **level** of a node is the number of surrounding Bracket nodes minus the surrounding Escape nodes, and a set of nodes $U$ is **valid at level** $n$ if all Escape nodes in $U$ occur at a level greater than $n$.

$$
\begin{array}{rcl}
\text{letrec } x = M^0, D^n \text{ in } C^0[x] & \to_{sub} & \text{letrec } x = M^0, D^n \text{ in } C^0[M^0] \\
\text{letrec } x = C^0[y], y = M^0, D^n \text{ in } M^n & \to_{sub} & \text{letrec } x = C^0[M^0], y = M^0, D^n \text{ in } M^n \\
\\
(\lambda x.M_1^0)\ M_2^0 & \to_{\beta\circ} & \text{letrec } x = M_2^0 \text{ in } M_1^0 \\
\sim \langle M^0 \rangle & \to_{esc} & M^0 \\
! \langle M^0 \rangle & \to_{run} & M^0 \\
\\
\text{letrec } D_1^n \text{ in } (\text{letrec } D_2^n \text{ in } M^n) & \to_{merge} & \text{letrec } D_1^n, D_2^n \text{ in } M^n \\
\text{letrec } x = (\text{letrec } D_1^n \text{ in } M_1^n), D_2^n \text{ in } M_2^n & \to_{merge} & \text{letrec } x = M_1^n, D_1^n, D_2^n \text{ in } M_2^n \\
\\
(\text{letrec } D^n \text{ in } M_1^n)\ M_2^n & \to_{lift} & \text{letrec } D^n \text{ in } (M_1^n\ M_2^n) \\
M_1^n\ (\text{letrec } D^n \text{ in } M_2^n) & \to_{lift} & \text{letrec } D^n \text{ in } (M_1^n\ M_2^n) \\
\text{letrec } D^n \text{ in } \langle M^n \rangle & \to_{lift} & \langle \text{letrec } D^n \text{ in } M^n \rangle \\
\\
\text{letrec } \_ \text{ in } M^n & \to_{gc} & M^n \\
\text{letrec } D_1^n, D_2^n \text{ in } M^n & \to_{gc} & \text{letrec } D_1^n \text{ in } M^n \\
 & & \text{if } D_2^n \neq \emptyset \wedge D_2^n \perp \text{letrec } D_1^n \text{ in } M^n
\end{array}
$$

**Fig. 6.** Term Reduction Rules

**Definition 4 (Node level).** *Given a graph $g = (V, L, E, S, r) \in \mathbb{G}$, a node $v \in V$ has level $n$ if there is a derivation for the judgment $level(v) = n$ defined as follows:*

$$\frac{v \in toplevel(V)}{level(v) = 0} \qquad \frac{surround(v) = u \quad L(u) = \lambda \quad level(u) = n}{level(v) = n}$$

$$\frac{surround(v) = u \quad L(u) = \langle\rangle \quad level(u) = n}{level(v) = n + 1}$$

$$\frac{surround(v) = u \quad L(u) = \sim \quad level(u) = n + 1}{level(v) = n}$$

*We write $level(v_1) < level(v_2)$ as a shorthand for $level(v_1) = n_1 \wedge level(v_2) = n_2 \wedge n_1 < n_2$. A set $U \subseteq V$ is valid at level $n$ if there is a derivation for the judgment $\vdash^n U$ defined as follows:*

$$\frac{\vdash^n v \quad \forall v \in toplevel(U)}{\vdash^n U} \qquad \frac{L(v) \in \{@, \bullet, !\}}{\vdash^n v}$$

$$\frac{L(v) = \lambda \quad \vdash^n contents(v)}{\vdash^n v} \qquad \frac{L(v) = \langle\rangle \quad \vdash^{n+1} contents(v)}{\vdash^n v}$$

$$\frac{L(v) = \sim \quad \vdash^n contents(v)}{\vdash^{n+1} v}$$

Context families and node levels are closely related. In the term reductions presented in the previous section, context families restrict the terms in which a variable may be substituted. In the graph reductions described in this section, determining whether two nodes constitute a redex will require comparing the levels of the two nodes. Furthermore, we can show that the notion of a set of nodes valid at a given level corresponds directly to the restriction imposed on terms by expression families.

**Lemma 8 (Properties of graph validity).**

1. *Whenever* $M^n \in \mathbb{M}^n$ *and* $g = \gamma(M^n)$*, then* $\vdash^n V$.
2. *Whenever* $g \in \mathbb{G}$ *with* $\vdash^n V$*, then* $\tau(g) \in \mathbb{M}^n$.

When evaluating a graph $g = (V, L, E, S, r)$, we require that $g$ be well-formed (see Section 3.4) and that $\vdash^0 V$. This ensures that $level(v)$ is defined for all $v \in V$.

**Lemma 9 (Node levels in well-formed graphs).** *For any graph* $g \in \mathbb{G}$ *with* $\vdash^0 V$ *and* $v \in V$*, we have* $level(v) = n$ *for some* $n$.

We now define three reduction rules that can be applied to Uccello graphs. Each of these rules is applied in two steps: 1) If necessary, we copy nodes to expose the redex in the graph. This step corresponds to using the term reduction rules *sub* or the rules *merge*, $lift$, and *gc* (see Figure 6) on the original term. 2) We contract the redex by removing nodes and by redirecting edges in the graph. This step corresponds to performing the actual $\beta\circ$-, *esc*-, or *run*-reduction on a term. In the following, we write $j \oplus V$ for the set $\{j \oplus v \mid v \in V\}$ where $j \in \{1, 2\}$. Furthermore, we write $U \oplus V$ for the set $(1 \oplus U) \cup (2 \oplus V)$.

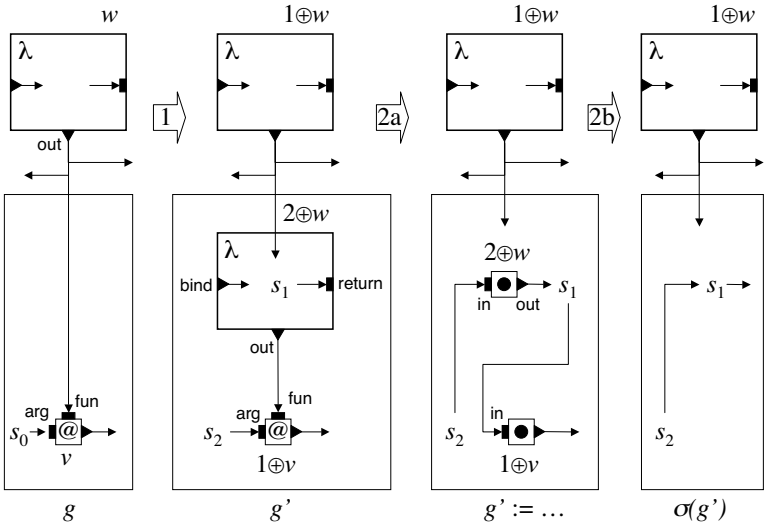



**Fig. 7.** Beta-reduction for Uccello graphs

**Beta** A $\beta\circ$-redex in a Uccello graph consists of an application node $v$ that has a lambda node $w$ as its first predecessor. The contraction of the redex is performed in two steps (see Figure 7):

1. Check that the edge $(w.\mathsf{out}, v.\mathsf{fun})$ is the only edge originating at $w.\mathsf{out}$, and that the application node $v$ is outside the scope of $w$. If any of these conditions do not hold, copy the lambda node in a way that ensures that the conditions hold for the copy of $w$. The copy of $w$ is called $2 \oplus w$, and the original of $v$ is called $1 \oplus v$. Place $2 \oplus w$ and its scope in the same scope as $1 \oplus v$.
2. Convert $1 \oplus v$ and $2 \oplus w$ into indirection nodes, which are then removed by the graph simplification function $\sigma$ (defined in Section 4.3). Redirect edges so that after

simplification, edges that originated at the applied function's parameter ($2 \oplus w$.bind) now start at the root $s_2$ of the function's argument, and edges that originated at the application node's output ($1 \oplus v$.out) now start at the root $s_1$ of the function's body.

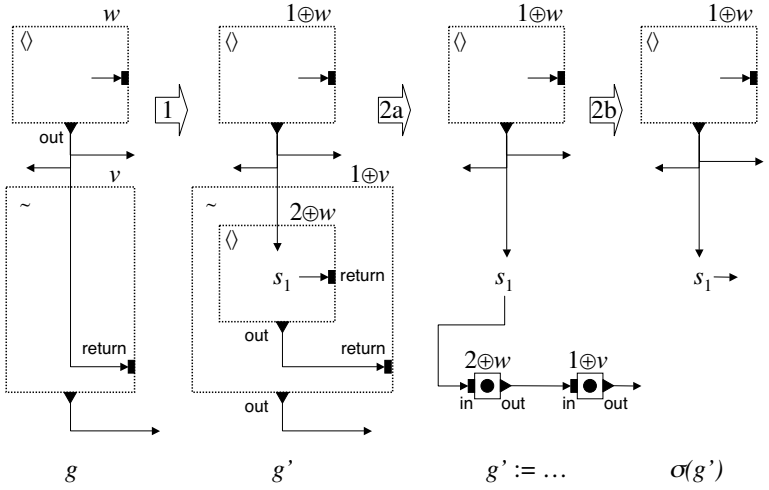



**Fig. 8.** Escape-reduction for Uccello graphs

**Definition 5 (Graph Beta).** *Given a graph $g \in \mathbb{G}$ with $\vdash^0 V$ and $v, w \in V$ such that $L(v) = @$, $L(w) = \lambda$, $(w.\mathsf{out}, v.\mathsf{fun}) \in E$, $\vdash^0 contents(w)$, $\vdash^0 \{u \mid u \in S(surround(v)) \wedge u \rightsquigarrow v\}$, and $level(w) \leq level(v)$ Then the contraction of the $\beta\circ$-redex v, written $g \rightarrow_{\beta\circ} h$, is defined as follows:*

1. *We define a transitional graph $g' = (V', L', E', S', r')$ using the functions $f_1$ and $f_2$ that map edge sources in E to edge sources in E′:*

$$
\begin{aligned}
f_1(x) &= x \\
f_1(u.o) &= 1 \oplus u.o \\
f_2(x) &= x \\
f_2(u.\mathsf{bind}) &= \begin{cases} 2 \oplus u.\mathsf{bind} & \textit{if } u \in S(w) \\ 1 \oplus u.\mathsf{bind} & \textit{otherwise} \end{cases} \\
f_2(u.\mathsf{out}) &= \begin{cases} 2 \oplus u.\mathsf{out} & \textit{if } u \in S(w) \backslash \{w\} \\ 1 \oplus u.\mathsf{out} & \textit{otherwise} \end{cases}
\end{aligned}
$$

*Let $s_0$ be the origin of the unique edge in $E$ with target $v.\mathsf{arg}$. The components of $g'$ are constructed as follows:*

$$V' = \begin{cases} (V \backslash S(w)) \oplus S(w) & \text{if } |\{(w.\mathsf{out}, t) \in E\}| = 1 \\ & \text{and } v \notin S(w) \\ V \oplus S(w) & \text{otherwise} \end{cases}$$

$$\begin{aligned} E' = {} & \{(f_1(s), 1 \oplus u.i) \mid 1 \oplus u \in V' \wedge (s, u.i) \in E \wedge u \neq v\} \\ & \cup \{(2 \oplus w.\mathsf{out}, 1 \oplus v.\mathsf{fun}), (f_1(s_0), 1 \oplus v.\mathsf{arg})\} \\ & \cup \{(f_2(s), 2 \oplus u.i) \mid 2 \oplus u \in V' \wedge (s, u.i) \in E\} \end{aligned}$$

$$L'(j \oplus u) = L(u) \qquad \text{for } j \in \{1, 2\}$$

$$\begin{aligned} S'(2 \oplus u) &= 2 \oplus S(u) \\ S'(1 \oplus u) &= 1 \oplus S(u) && \text{if } v \notin S(u) \\ S'(1 \oplus u) &= S(u) \oplus S(w) && \text{if } v \in S(u) \end{aligned}$$

$$r' = f_1(r)$$

2. *Let $s_1$ and $s_2$ be the origins of the unique edges in $E'$ with targets $2 \oplus w.\mathsf{return}$ and $1 \oplus v.\mathsf{arg}$ respectively. We modify $E'$, $L'$, and $S'$ as follows:*

$$\begin{aligned} (2 \oplus w.\mathsf{out}, 1 \oplus v.\mathsf{fun}) &:= (s_1, 1 \oplus v.\mathsf{in}) \\ (s_1, 2 \oplus w.\mathsf{return}) &:= (s_2, 2 \oplus w.\mathsf{in}) \\ (s_2, 1 \oplus v.\mathsf{arg}) &:= \textit{removed} \\ L'(1 \oplus v) &:= \bullet \\ L'(2 \oplus w) &:= \bullet \\ S'(2 \oplus w) &:= \textit{undefined} \end{aligned}$$

*Furthermore, any occurrence of port $2 \oplus w.\mathsf{bind}$ in $E'$ is replaced by $2 \oplus w.\mathsf{out}$. The resulting graph $h$ of the $\beta\circ$-reduction is then the simplification $\sigma(g')$.*

**Escape** An *esc*-redex consists of an Escape node $v$ that has a Bracket node $w$ as its predecessor. We contract the redex in two steps (see Figure 8):

1. Check that the edge $(w.\mathsf{out}, v.\mathsf{return})$ is the only edge originating at $w.\mathsf{out}$, and that the Escape node $v$ is outside the scope of $w$. If any of these conditions do not hold, copy the Bracket node in a way that ensures that the conditions hold for the copy of $w$. The copy of $w$ is called $2 \oplus w$, and the original of $v$ is called $1 \oplus v$. Place $2 \oplus w$ (and its scope) in the scope of $1 \oplus v$.
2. Convert $1 \oplus v$ and $2 \oplus w$ into indirection nodes, which are then removed by the function $\sigma$. Redirect edges so that after simplification, edges that originated at the Escape node's output port $(1 \oplus v.\mathsf{out})$ now start at the root $s_1$ of the Bracket node's body.

**Definition 6 (Graph Escape).** *Given a graph $g \in \mathbb{G}$ with $\vdash^0 V$ and $v, w \in V$ such that $L(v) = \sim$, $L(w) = \langle\rangle$, $(w.\mathsf{out}, v.\mathsf{return}) \in E$, $\vdash^0 contents(w)$, and $level(w) < level(v)$. Then the contraction of the esc-redex $v$, written $g \to_{esc} h$, is defined as follows:*

1. *We define a transitional graph* $g' = (V', L', E', S', r')$ *where* $V'$,$L'$,$S'$, *and* $r'$ *are constructed as in Definition 5.* [4] *The set of edges* $E'$ *is constructed as follows:*

$$\begin{aligned} E' = &\{(f_1(s), 1 \oplus u.i) \mid 1 \oplus u \in V' \wedge (s, u.i) \in E \wedge u \neq v\} \\ &\cup \{(2 \oplus w.\mathsf{out}, 1 \oplus v.\mathsf{return})\} \\ &\cup \{(f_2(s), 2 \oplus u.i) \mid 2 \oplus u \in V' \wedge (s, u.i) \in E\} \end{aligned}$$

2. *Let* $s_1$ *be the origin of the unique edge in* $E'$ *with target* $2 \oplus w.\mathsf{return}$. *We modify* $E'$, $L'$, *and* $S'$ *as follows:*

$$\begin{aligned} (2 \oplus w.\mathsf{out}, 1 \oplus v.\mathsf{return}) &:= (2 \oplus w.\mathsf{out}, 1 \oplus v.\mathsf{in}) \\ (s_1, 2 \oplus w.\mathsf{return}) &:= (s_1, 2 \oplus w.\mathsf{in}) \\ L'(1 \oplus v) &:= \bullet \\ L'(2 \oplus w) &:= \bullet \\ S'(1 \oplus v) &:= \textit{undefined} \\ S'(2 \oplus w) &:= \textit{undefined} \end{aligned}$$

*The resulting graph* $h$ *of the esc-reduction is* $\sigma(g')$.

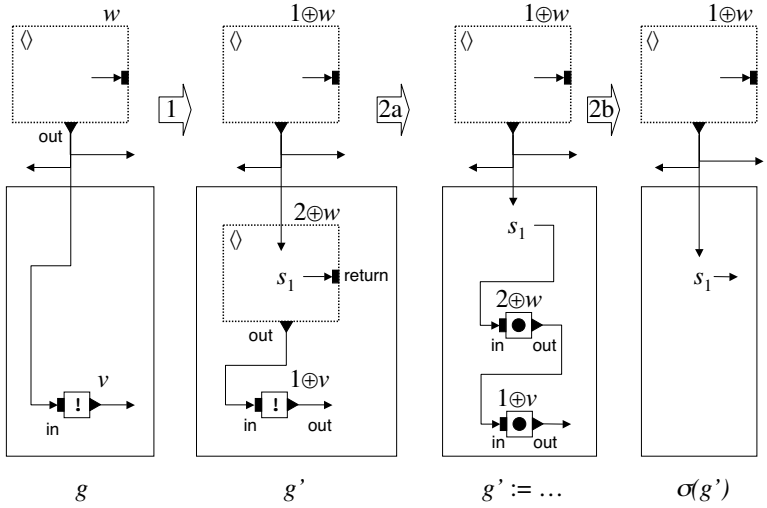



**Fig. 9.** Run-reduction for Uccello graphs

**Run** A *run*-redex consists of a Run node $v$ that has a Bracket node $w$ as its predecessor. The contraction of the redex is performed in two steps (see Figure 9):

1. Check that the edge $(w.\mathsf{out}, v.\mathsf{in})$ is the only edge originating at $w.\mathsf{out}$, and that the Run node $v$ is outside the scope of $w$. If any of these conditions do not hold, copy the Bracket node in a way that ensures that the conditions hold for the copy of $w$. The copy of $w$ is called $2 \oplus w$, and the original of $v$ is called $1 \oplus v$. Place $2 \oplus w$ (and its scope) in the same scope as $1 \oplus v$.

[4] In Definition 5, $v$ and $w$ refer to the application- and lambda nodes of a $\beta\circ$-redex. Here, $v$ stands for the Escape node, and $w$ stands for the Bracket node of the *esc*-redex.

2. Convert $1 \oplus v$ and $2 \oplus w$ into indirection nodes, which are then removed by $\sigma$. Redirect edges so that after simplification, edges that originated at the Run node's output port ($1 \oplus v.\mathsf{out}$) now start at the root $s_1$ of the Bracket node's body.

**Definition 7 (Graph Run).** *Given a graph $g \in \mathbb{G}$ with $\vdash^0 V$ and $v, w \in V$ such that $L(v) = !$, $L(w) = \langle\rangle$, $(w.\mathsf{out}, v.\mathsf{in}) \in E$, $\vdash^0 contents(w)$, and $level(w) \leq level(v)$. Then the contraction of the run-redex v, written $g \rightarrow_{run} h$, is defined as follows:*

1. *We define a transitional graph $g' = (V', L', E', S', r')$ where $V'$,$L'$,$S'$, and $r'$ are constructed as in Definition 5. The set of edges $E'$ is constructed as follows:*

$$\begin{aligned} E' = &\{(f_1(s), 1 \oplus u.i) \mid 1 \oplus u \in V' \wedge (s, u.i) \in E \wedge u \neq v\} \\ &\cup \{(2 \oplus w.\mathsf{out}, 1 \oplus v.\mathsf{in})\} \\ &\cup \{(f_2(s), 2 \oplus u.i) \mid 2 \oplus u \in V' \wedge (s, u.i) \in E\} \end{aligned}$$

2. *Let $s_1$ be the origin of the unique edge in $E'$ with target $2 \oplus w.\mathsf{return}$. We modify $E'$, $L'$, and $S'$ as follows:*

$$\begin{aligned} (s_1, 2 \oplus w.\mathsf{return}) &:= (s_1, 2 \oplus w.\mathsf{in}) \\ L'(1 \oplus v) &:= \bullet \\ L'(2 \oplus w) &:= \bullet \\ S'(2 \oplus w) &:= \textit{undefined} \end{aligned}$$

*The resulting graph h of the run-reduction is $\sigma(g')$.*

### 5.3 Results

Any reduction step on a graph $g = \gamma(M)$ corresponds to a sequence of reduction steps on the term $M$ to expose a redex, followed by a reduction step to contract the exposed redex. Conversely, the contraction of any redex in a term $M$ corresponds to the contraction of a redex in the graph $\gamma(M)$.

**Theorem 2 (Correctness of Graphical Reductions).** *Let $g \in \mathbb{G}$, $\delta \in \{\beta\circ, esc, run\}$, $M_1^0 \in \mathbb{M}^0$ and $g = \gamma(M_1^0)$.*

1. *Graph reductions preserve well-formedness:*

$$g \rightarrow_\delta h \textit{ implies } h \in \mathbb{G}$$

2. *Graph reductions are sound:*

$$\begin{aligned} &g \rightarrow_\delta h \textit{ implies } M_1^0 \rightarrow^* M_2^0 \rightarrow_\delta M_3^0 \\ &\textit{for some } M_2^0, M_3^0 \in \mathbb{M}^0 \textit{ such that } h = \gamma(M_3^0) \end{aligned}$$

3. *Graph reductions are complete:*

$$\begin{aligned} &M_1^0 \rightarrow_\delta M_2^0 \textit{ implies } g \rightarrow_\delta h \textit{ for some } h \in \mathbb{G} \\ &\textit{such that } h = \gamma(M_2^0) \end{aligned}$$

## 6 Conclusions and Future Work

With the goal of better understanding how to extend visual languages with programming constructs and techniques available for modern textual languages, this paper studies and extends a graph-text connection first developed by Ariola and Blom. While the motivation for Ariola and Blom's work was the graph-based compilation of functional languages, only minor changes to their representations and visual rendering were needed to make their results a suitable starting point for our work. We extended this formalism with staging constructs, thereby developing a formal model for generative programming in the visual setting.

In this paper we only presented an abstract syntax for Uccello. In the future, it will be important to develop a more user-friendly concrete syntax with features such as multi-parameter functions or color shading to better visualize stage distinctions. This step will raise issues related to parsing visual languages, where we expect to be able to build on detailed previous work on layered [16] and reserved graph grammars [21].

Another important step in developing the theory will be lifting both type checking and type inference algorithms defined on textual representations to the graphical setting. Given the interactive manner in which visual programs are developed, it will also be important to see whether type checking and the presented translations can be incrementalized so that errors can be detected locally and without the need for full-program analysis.

**Acknowledgments:** Kedar Swadi, Samah Abu Mahmeed, Roumen Kaiabachev and Edward Pizzi read and commented on early drafts of this paper, and we would like to thank them for their insightful suggestions. We also thank Keith Cooper, Moshe Vardi, Robert "Corky" Cartwright, and Peter Druschel for serving on the first author's thesis committee.

## References

1. `http://www.cs.rice.edu/~besan/proofs.pdf`.
2. Robin Abraham and Martin Erwig. Header and unit inference for spreadsheets through spatial analyses. In *IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 165–172, 2004.
3. Z. M. Ariola and S. Blom. Cyclic lambda calculi. *Lecture Notes in Computer Science*, 1281:77, 1997.
4. M. Burnett, J. Atwood, R. Walpole Djang, H. Gottfried, J. Reichwein, and S. Yang. Forms/3: A first-order visual language to explore the boundaries of the spreadsheet paradigm. *Journal of Functional Programming*, 11:155–206, March 2001.
5. W. Citrin, M. Doherty, and B. Zorn. Formal semantics of control in a completely visual programming language. In Allen L. Ambler and Takayuki Dan Kimura, editors, *Proceedings of the Symposium on Visual Languages*, pages 208–215, Los Alamitos, CA, USA, October 1994. IEEE Computer Society Press.
6. W. Citrin, R. Hall, and B. Zorn. Programming with visual expressions. In Volker Haarslev, editor, *Proc. 11th IEEE Int. Symp. Visual Languages*, pages 294–301. IEEE Computer Society Press, 5–9 September 1995.
7. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to algorithms*. MIT Press and McGraw-Hill Book Company, 14th edition, 1994.

8. M. Erwig. Abstract syntax and semantics of visual languages. *Jounral of Visual Languages and Computing*, 9:461–483, October 1998.
9. Martin Erwig and Margaret M. Burnett. Adding apples and oranges. In *4th International Symposium on Practical Aspects of Declarative Languages*, pages 173–191, 2002.
10. National Instruments. *LabVIEW Student Edition 6i*. Prentice Hall, 2001.
11. S. Peyton Jones, A. Blackwell, and M. Burnett. A user-centred approach to functions in Excel. *ICFP*, pages 165–176, 2003.
12. Edward A. Lee. What's ahead for embedded software? *IEEE Computer*, pages 18–26, September 2000.
13. MetaOCaml: A compiled, type-safe multi-stage programming language. Available online from `http://www.metaocaml.org/`, 2004.
14. National Instruments. LabVIEW. Online at `http://www.ni.com/labview`.
15. Oregon Graduate Institute Technical Reports. P.O. Box 91000, Portland, OR 97291-1000,USA. Available online from `ftp://cse.ogi.edu/pub/tech-reports/README.html`.
16. Jan Rekers and Andy Schuerr. Defining and parsing visual languages with layered graph grammars. *Journal of Visual Languages and Computing*, 8(1):27–55, 1997.
17. Amr Sabry. *The Formal Relationship between Direct and Continuation-Passing Style Optimizing Compilers: A Synthesis of Two Paradigms*. PhD thesis, Rice University, August 1994.
18. Walid Taha. *Multi-Stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, 1999. Available from [15].
19. Walid Taha and Michael Florentin Nielsen. Environment classifiers. In *The Symposium on Principles of Programming Languages (POPL '03)*, New Orleans, 2003.
20. The MathWorks. Simulink. Online at `http://www.mathworks.com/products/simulink`.
21. Da-Qian Zhang, Kang Zhang, and Jiannong Cao. A context-sensitive graph grammar formalism for the specification of visual languages. *The Computer Journal*, 44(3):186–200, 2001.

# Reconfigurable Manifolds

**Sarah Thompson** **Alan Mycroft**

{sarah.thompson,alan,mycroft}@cl.cam.ac.uk

Computer Laboratory, University of Cambridge, William Gates Building, 15 JJ Thomson Ave., Cambridge CB3 0FD, UK

## 1 Introduction

Spacecraft design is, without doubt, one of the most challenging areas of modern engineering. In order to be viable, spacecraft must mass relatively little, whilst being capable of surviving the considerable G-forces and vibration of launch. In space, they must withstand extreme temperatures, hard vacuum and high levels of radiation, for several years without maintenance.

Conventionally, spacecraft wiring harnesses are built with architectures that are fixed at the time of manufacture. They must therefore be designed to endure the lifetime of the mission with a very high probability, though the conventionally necessary redundant duplication of signals has significant implications for mass. Given that launch costs are typically in excess of $30,000 per kg, reducing the mass of a spacecraft's wiring harness, without compromising reliability, is highly desirable. As a motivating example, the network cabling in the International Space Station (ISS) is known to mass more than 10 metric tonnes.

Recent advances in MEMS-based switching [9] have made it possible to consider the construction of *reconfigurable manifolds* – essentially, wiring harnesses that behave like macroscopic FPGA routing networks. Redundant wiring can be shared between many signals, thereby significantly reducing the total amount of cable required. Reconfigurability has a significant further benefit, in that it also allows adaptation to mission requirements that change over time, whilst also significantly reducing design time.

In a recent initiative, the US Air Force has been moving toward a *responsive space* paradigm which aims to reduce the time from design concept to launch (currently several years) to less than one week [7]. Such a target is unlikely to be achievable with existing bespoke one-off design techniques; a parts bin driven, plug-and-play approach to satellite construction will become essential. It must be possible to choose a satellite chassis of a size appropriate to the task in terms of accommodating sufficient manoevering propellant as well as the necessary instrumentation payload, then bolt everything together and have the resulting satellite 'just work.'

We present an algorithm that allows such a reconfigurable manifold to be automatically self-configured, then dynamically tested in-situ, such that signals are automatically rerouted around non-functioning wires and switches as soon as faults are detected. Break-before-make switching is used in order to achieve transparency from the point of view of subsystems that are interconnected by the manifold, whilst also making it possible to achieve near-100% testability.

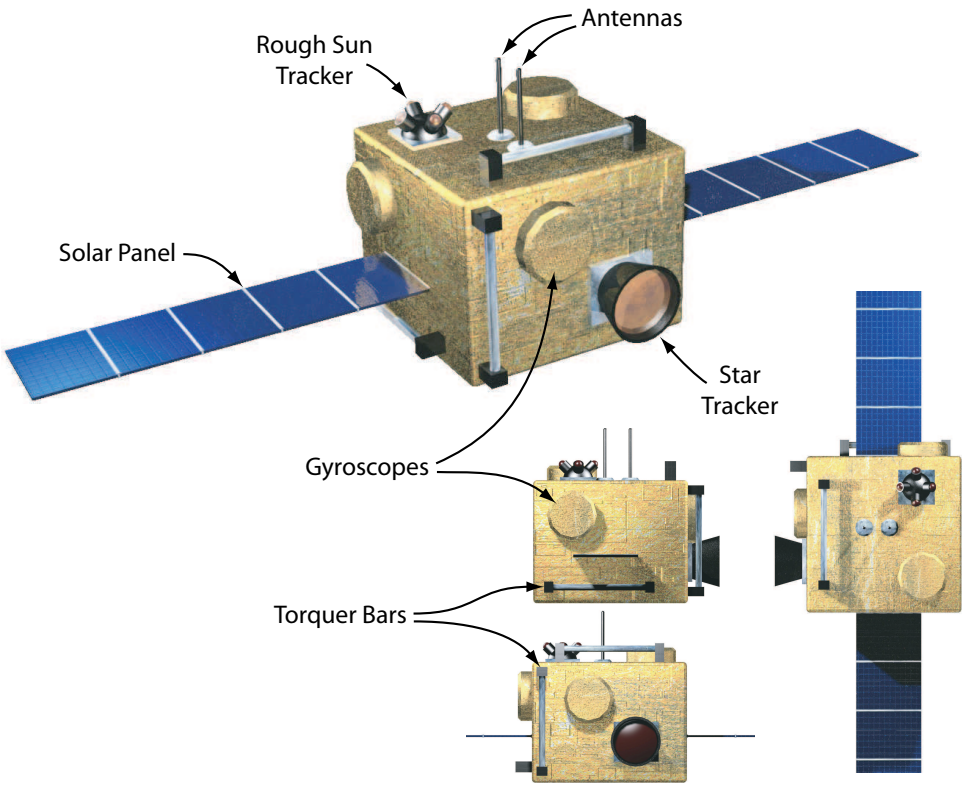



Figure 1: A typical near-earth small satellite configuration

## 1.1 Physical satellite wiring architectures

Conventionally, satellites are constructed with fixed wiring architectures. Reliability must therefore be engineered-in through multiple redundancy – duplication or triplication (or more) of signal paths is common, which carries with it an attendant mass penalty. Typically, one of two kinds of wiring architecture are common. Fig. 2 shows a typical passive backplane with multiple subsystems, each slotting in to a rack on separate cards[1]. Wiring harnesses, in the sense that they exist in cars and aircraft as bundles of physical cables, tend to be avoided where possible.

Another common approach is shown in Fig. 3, where a single motherboard has a number of daughter boards attached to it on standoffs. Normally (though not visible in the diagram) these daughter boards plug directly into connectors on the motherboard, again avoiding the need for cables.

Typically, card frames have passive backplanes, which do not normally contain active electronics beyond perhaps some simple power regulation or line termination. Motherboard approaches more commonly include active electronics on the main board itself, though this is not a prerequisite.

[1]Note that the image is representational – actual satellite hardware differs in detail

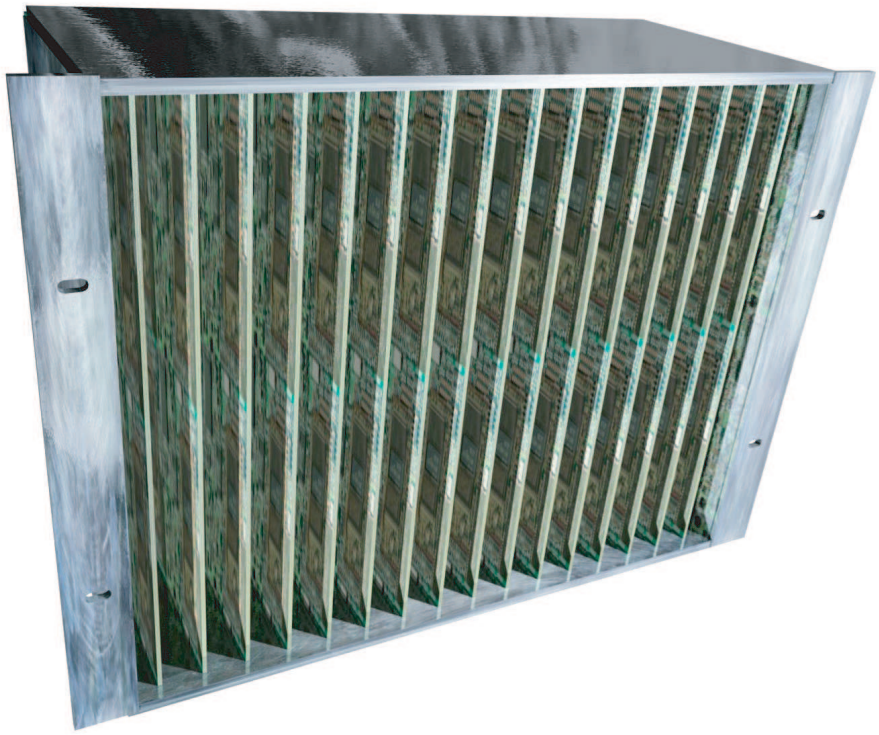


Figure 2: Card frame with backplane

## 1.2 Logical satellite wiring architectures

At a logical, block diagram level, fixed architecture satellite wiring harnesses typically follow the structure shown in Fig. 4. All of the main subsystems are attached to a motherboard or backplane that provides most of the necessary interconnection infrastructure, with external devices plugging directly into the relevant subsystems. All required redundancy must be in place from the outset. Typically, satellites are one-off designs, so any design changes before launch require physical modifications – of course, such changes *after* launch are typically impossible. As a further consequence of this approach, subsystem re-use is relatively uncommon, requiring considerable effort in terms of design, validation and verification, of the order of several years from concept to launch.

# 2 Reconfigurable manifolds

The responsive space paradigm [7] implies the requirement to move away from fixed architectures and their consequential design and validation costs toward an autonomous, self-organising approach. In essence, a *reconfigurable manifold* is a self-organising, self-testing, self-repairing replacement for a fixed architecture wiring harness. Ideally, at a system level, a spacecraft adopting this approach should have an architecture similar to that shown in Fig. 5.

Ideally, all wiring should be routed by the manifold rather than connected directly to subsystems. From a the point of view of rapid construction, this is ideal – a subsystem

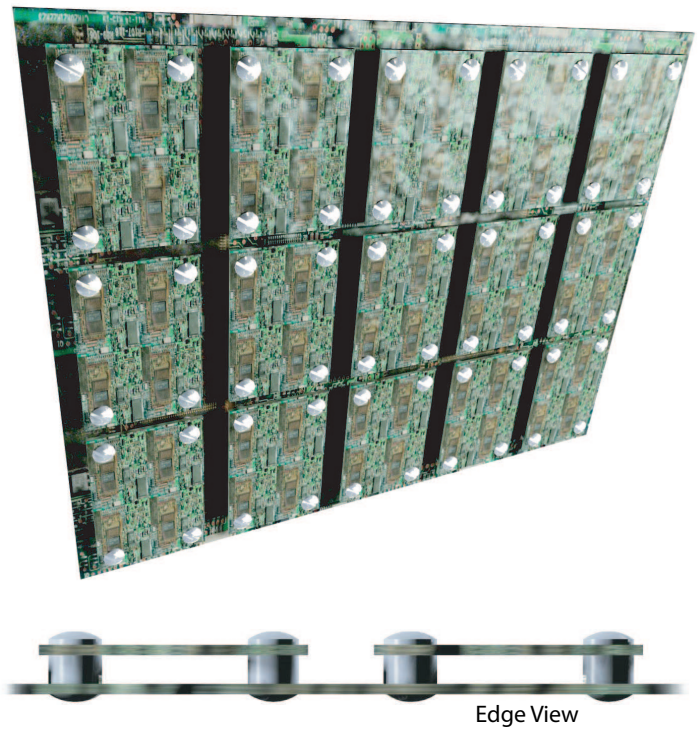



Figure 3: Motherboard with attached daughter boards

such as a gyroscope, star tracker, sun tracker or antenna could be bolted to the spacecraft chassis anywhere that is physically convenient, with all of the necessary wiring being 'discovered' and automatically routed after power-up.

## 2.1 Signal types

Spacecraft wiring harnesses (reconfigurable or otherwise) must be able to carry a wide variety of signals, varying in terms of power, voltage and bandwidth, with similarly variable electrical considerations in terms of impedance, end-to-end resistance, etc. Typical signal types found in satellites, along with example applications are listed as follows[2]:

**Power** Normally a single $+28$V DC unregulated supply rail powers the entire spacecraft, with local step-down regulators providing lower voltage high quality supply rails to each subsystem. Where higher voltages are necessary, e.g. to drive cryocoolers for low background noise imaging sensors, this is normally achieved with local step-up switching DC-DC converters.

**Heavy current analogue** High current feeds to torquer bars, motor drives, solenoid power, explosive bolts, etc.

**Low current, low speed analogue** Analogue sensor feeds, thermocouples, rough sun tracker photocells, etc.

[2]This list is not exhaustive

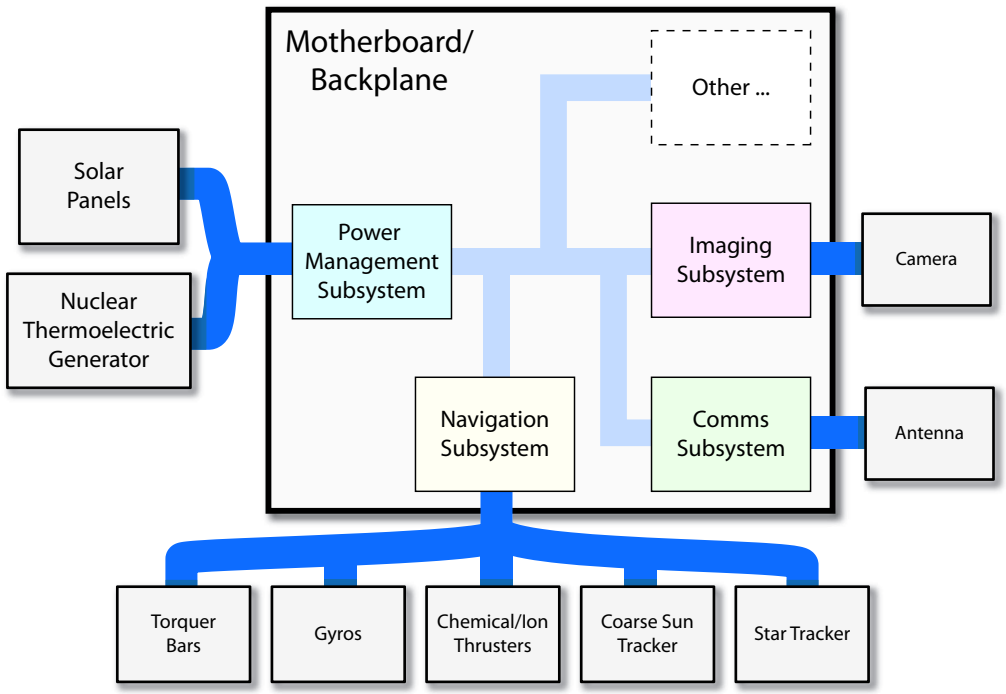



Figure 4: Conventional, fixed-architecture motherboard/backplane

**Low current, high speed analogue** Higher speed sensor wiring, video feeds from cameras and star trackers, etc.

**Low speed digital** Simple on/off telemetry sensors, e.g. mechanical limit switches.

**High speed digital** Digital communications between subsystems.

**Low power microwave** Radio receiver antenna feeds, low power radio transmitter antenna feeds.

**High power microwave** High power antenna feeds, ion thruster power cabling, etc.

**Optical** High speed network connectivity, lower speed sensor applications that require a significant degree of electrical isolation[3].

No single switching architecture, at the time of writing, can accommodate more than a few of the above signal types.

## 2.2 Constructing practical reconfigurable manifolds

A practical reconfigurable manifold must encompass most, if not all, signal types in order to be effective. Since no single switch fabric is suitable, it makes sense to split the

[3]Optical switching is beyond the scope of this work and will not be discussed further

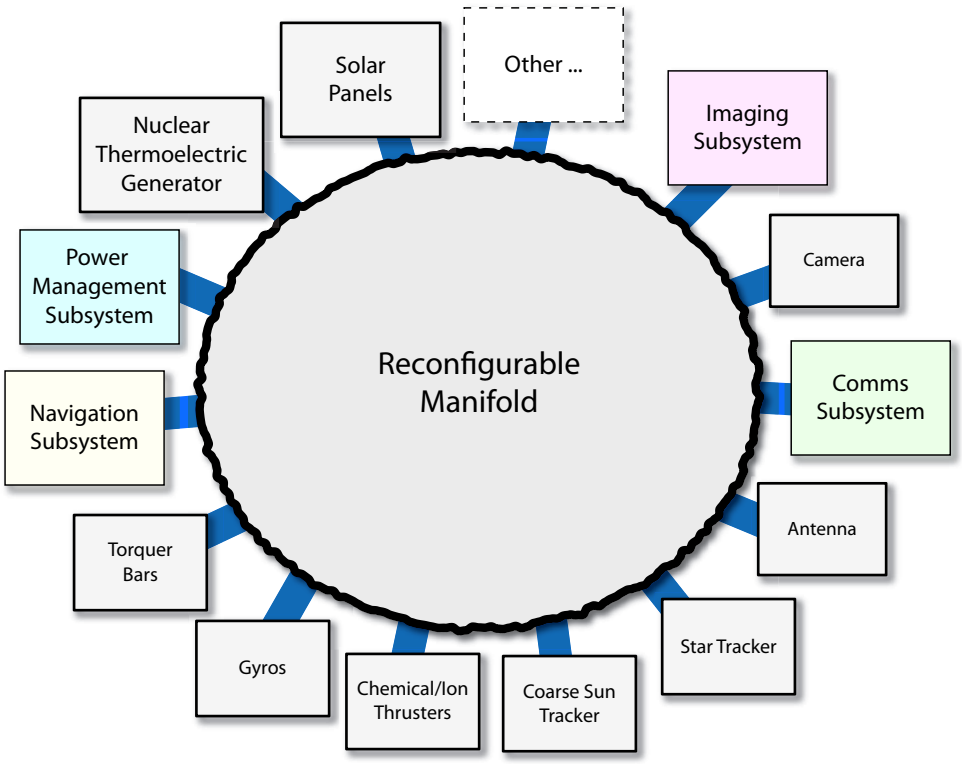



Figure 5: Reconfigurable manifold architecture

manifold into separate sub-manifolds, each of handling a different signal type, as shown in Fig. 6.

Some cross-connectivity between the sub-manifolds makes sense, since, for example, several MEMS relays could potentially be connected in parallel in order to to switch heavier current, or DC-biased analogue routing with sufficient bandwidth could, in an emergency, be used to carry digital data.

Fig. 7 shows a reconfigurable manifold implemented as a replacement for a passive backplane or passive motherboard. In contrast with Fig. 4, external systems connect to the manifold rather than direct to the subsystems themselves. Configuring such a satellite might be as simple as installing cards in a backplane or motherboard in any convenient order, then plugging external devices into the manifold. Spare slots could, given sufficient mass budget, be used to provide extra redundancy simply by plugging in extra duplicate cards; appropriate firmware could potentially handle this automatically.

An alternative architecture is shown in Fig. 8. Rather than a single manifold routing between devices connected to its periphery, the manifold is itself distributed between the subsystems. Interconnection between subsystems is passive, with the subsystems cooperating to establish longer distance, multi-hop routes.

The single manifold approach is perhaps best suited to small satellites, whereas the (more complex, though more flexible and scalable) distributed approach lends itself to larger spacecraft such as large satellites, manned spacecraft, space stations or indeed

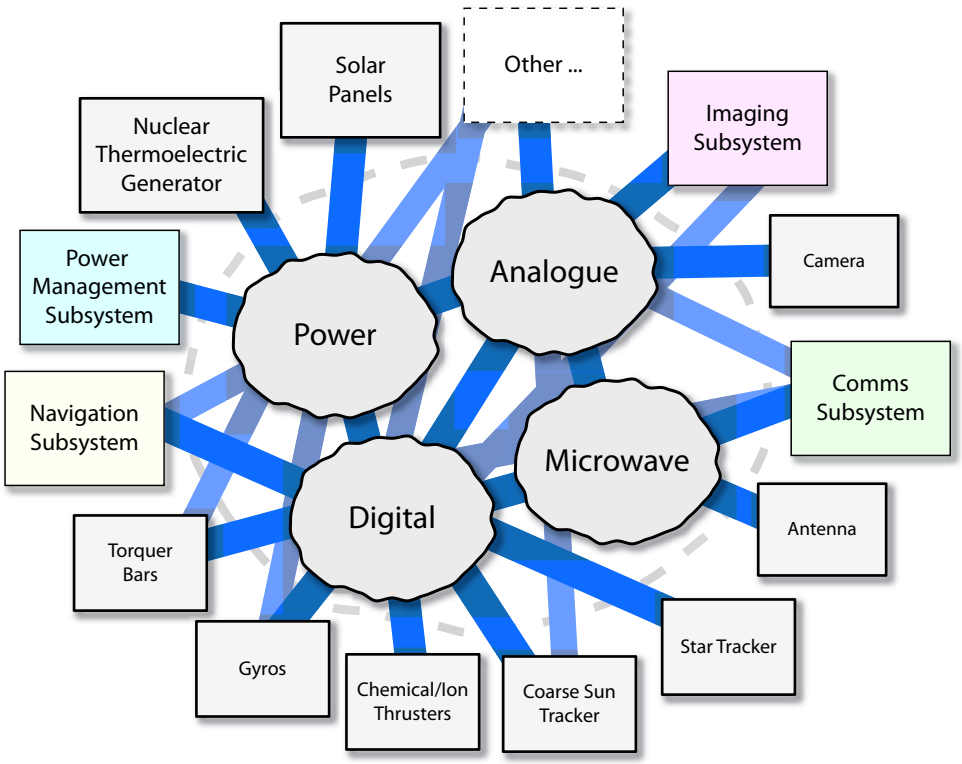



Figure 6: Separate routing networks for power, analogue, digital and microwave

also to terrestrial aircraft.

## 2.3 Switching technologies

Many switching technologies exist that differ considerably in capability:

**FPGAs** Field-programmable gate arrays can be used to route digital data, and are also comparatively cheap and readily available.

**FPTAs** Field-programmable transistor arrays [10] have some similarities to FPGAs, though they are aimed more closely at analogue applications. As with FPGAs, they are not intended from the outset as routing devices for use within a the switch fabric of a reconfigurable manifold, though it would seem feasible to apply them to the switching of low- to medium-speed analogue signals.

**Digital Crossbar Switch ASICs** A number of commercial, off-the-shelf (COTS) digital crossbar switch chips are available, though this application appears to be becoming dominated by FPGAs as a consequence of the larger FPGA manufacturers getting more directly involved by releasing support for using their devices in this way [2].

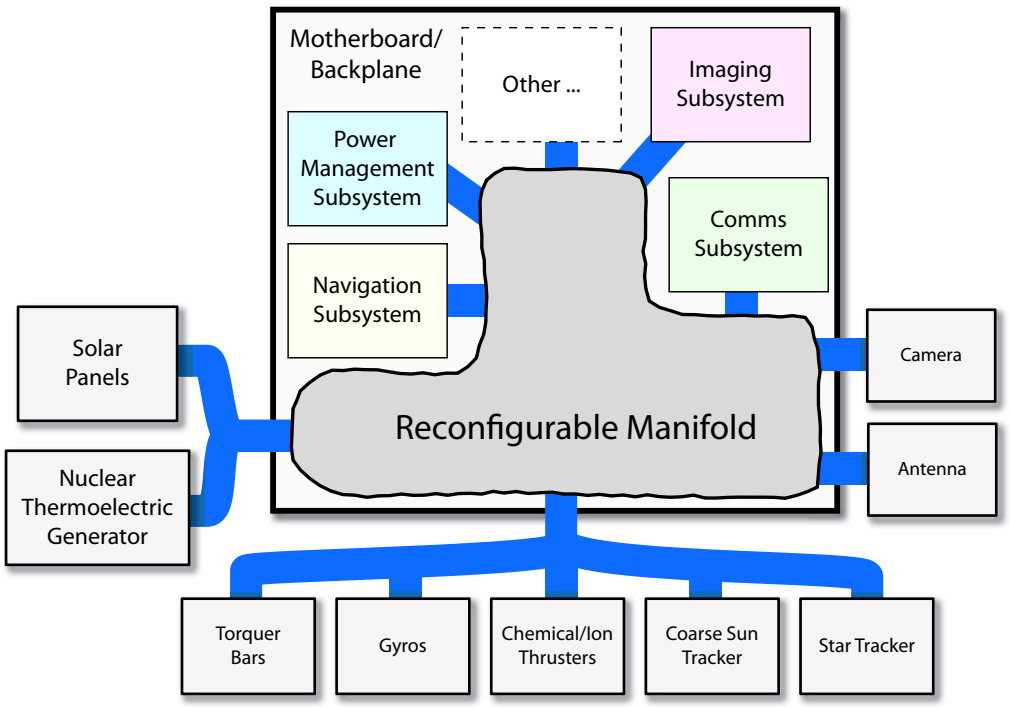



Figure 7: Reconfigurable manifold as a motherboard or backplane

**Analogue Crossbar Switch ASICs** Though not so widely supported as digital crossbar switch devices, analogue crossbar switches are available, mostly aimed at switching analogue video signals[1].

**MEMS switches** Micron-scale electromechanical switches have been demonstrated to be an effective candidate technology [9]. Though physically far larger than CMOS transistor-based switches, MEMS switches are nevertheless orders of magnitude smaller and lighter than full-size mechanical relays, and have excellent electrical characteristics that renders them capable of being applied to almost any low-current switching application, including microwave.

**Electromechanical Relays** Somewhat old-fashioned, relays are nevertheless capable of switching very heavy currents. They are sufficiently massive, however, that it is difficult to imagine them being used in large numbers in a spacecraft application.

**Discrete MOSFET/IGBT Switching** Large power transistors, both MOS and bipolar, are commonly used to switch heavy current and moderately high voltage (up to a few hundred volts and/or hundreds of amps) signals, particularly in motor drive applications. They exhibit high reliability and relatively good radiation hardness characteristics due to their very large (in comparison with ASICs) geometries, though their gate drive circuitry can be tricky to engineer. Though physically bulky, they nevertheless remain a useful possibility for constructing heavy current and/or power

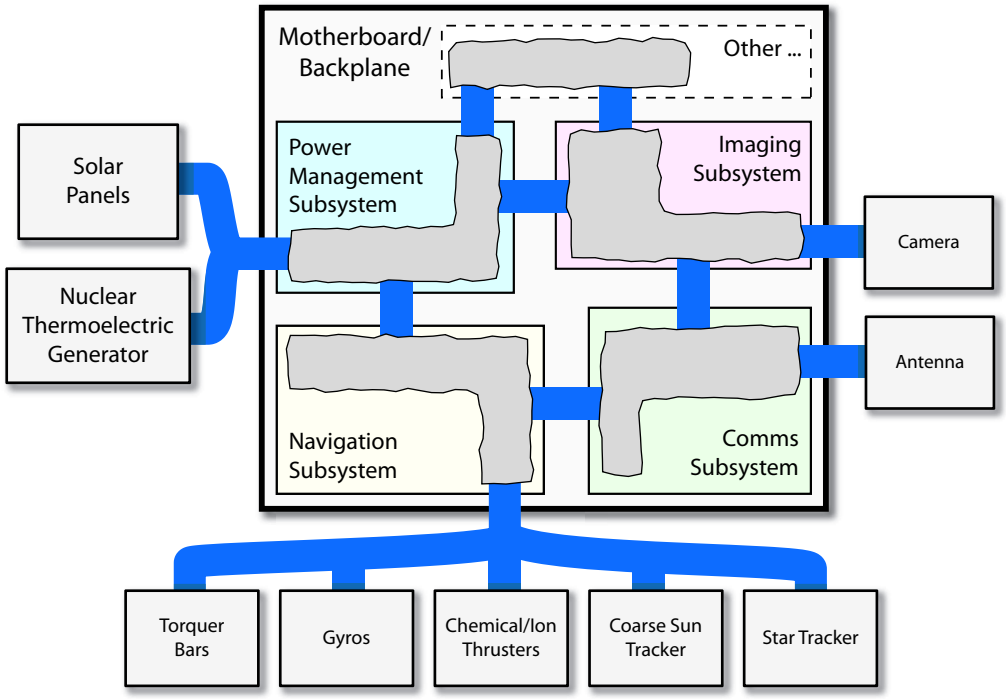



Figure 8: Reconfigurable manifold distributed across subsystems

switching networks.

Table 1 shows compatibility between switch technologies and signal types. The notation '?,' denoting 'possibly compatible,' indicates that, under normal circumstances, an automated routing algorithm would not normally attempt to make a connection of this type, though under certain circumstances, possibly only when authorised by a human, such connections might be made in the absence of more appropriate infrastructure. Normally, signals would be prioritised, so critical signals would almost always be routed, but less important connections may be degraded or even omitted. For example, a non-critical redundant temperature sensor might be disconnected in favour of keeping an instrument package running.

## 2.4 Routing architectures

The major alternative switching architectures that may be considered when designing a reconfigurable manifold are as follows:

**Crossbar Switch** An $M \times N$ grid of switches configured to provide a $M$-input, $N$-output routing network.

**Permutation Network** A *permutation network* performs an arbitrary permutation on $N$ inputs, such that any possible reordering of the inputs is supported.

| | FPGA | FPTA | Digital X-bar | Analogue X-bar | MEMS | Relays | MOSFET/IGBT |
|---|---|---|---|---|---|---|---|
| **Power** | × | × | × | × | ? | √ | √ |
| **Heavy current analogue** | × | × | × | × | ? | √ | √ |
| **Low current, low speed analogue** | × | √ | × | √ | √ | √ | ? |
| **Low current, high speed analogue** | × | √ | × | √ | √ | ? | ? |
| **Low speed digital** | √ | √ | √ | √ | √ | √ | √ |
| **High speed digital** | √ | ? | √ | ? | √ | ? | × |
| **Low power microwave** | × | × | × | × | √ | × | × |
| **High power microwave** | × | × | × | × | ? | × | × |

× – Not compatible ? – Possibly compatible √ – Compatible

Table 1: Compatibility between switch technologies and signal types

**Ad-Hoc and Hybrid Approaches** Practical considerations make it appropriate to consider the possibility of leveraging existing technologies, possibly in combination, to create reconfigurable manifolds. Though the result network topology and routing algorithms may be technically inferior to a purer design, economic considerations are nevertheless still important for practical designs.

**Embedding into Networks of Arbitrary Topology** Given a sufficiently large and complex graph, with nodes representing switches and edges representing wires, it is possible to compute a switch configuration that implements an arbitrary circuit.

Each approach is described in detail below.

### 2.4.1 Crossbar switches

Crossbar switches have a long history, having originally been introduced as a means of routing telephone calls through electromechanical telephone exchanges. Conceptually extremely simple, a crossbar switch is constructed from two sets of orthogonal wires (bus bars in telecommunications nomenclature), such that each crossing can be bridged by a switch. Fig. 9 depicts the circuit of a small $8 \times 8$ crossbar switch.

To route a particular input to a given output, all that is necessary is for the switch corresponding to that input and output to be closed. Crossbar switches are somewhat inefficient in terms of hardware requirements, and also in terms of providing more routing capability than is strictly necessary in many cases – it is possible, for example, to route a single input to any number of outputs, or to common inputs together. Achieving reliability is relatively straightforward, however – replacing each non redundant switch (Fig. 10) with a partially- or fully-redundant alternative (Fig. 11 or Fig. 12 respectively) allows single point failures to be recovered. A fully redundant switch configuration allows any of its four component switches to fail-open or fail-closed without affecting functionality. The partially redundant version only requires half as many switches, but is only safe against fail-closed faults – however, given one or more spare bus bars on each axis, fail-open faults can easily be patched around and are therefore still recoverable. In cost terms, building a fully-redundant $M \times N$ switch requires $4 \times M \times N$ switches, whereas the partially redundant approach requires $2 \times (M+1) \times (N+1)$ switches, though clearly the larger circuit is more fault-tolerant.

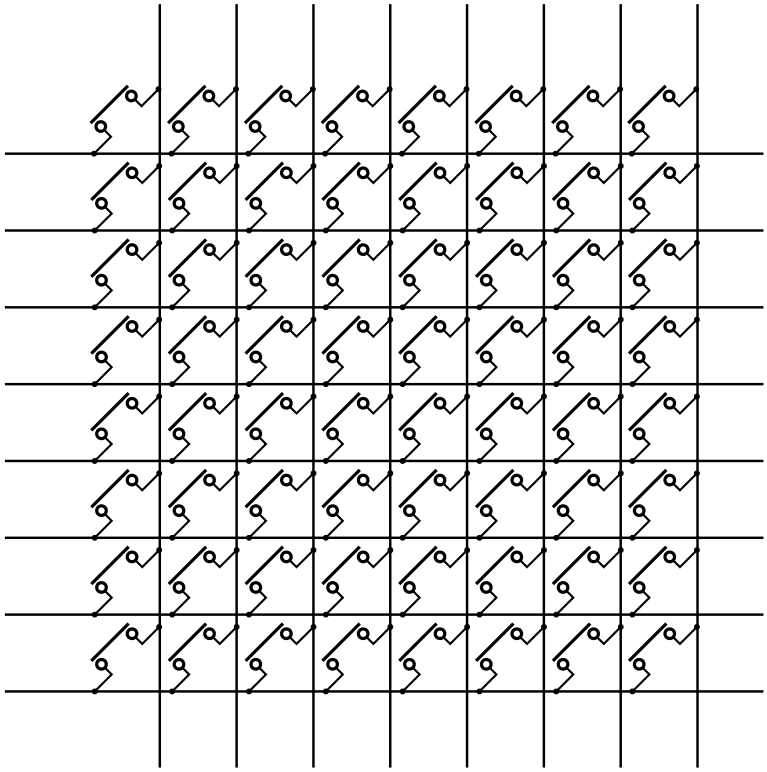


Figure 9: Crossbar Switch

Figure 10: Non-redundant switch

Figure 11: Partially redundant switch configuration

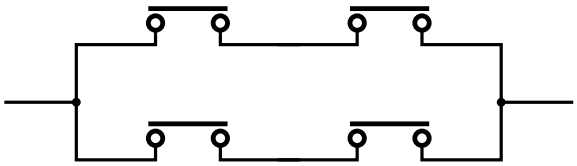


Figure 12: Fully redundant switch configuration

### 2.4.2 Permutation networks

Permutation networks are an alternative approach to routing that, in many cases, requires substantially fewer switches for a given number of inputs – rather than $O(N^2)$, they tend toward $O(N \log N)$, which can be a very significant advantage when the num-

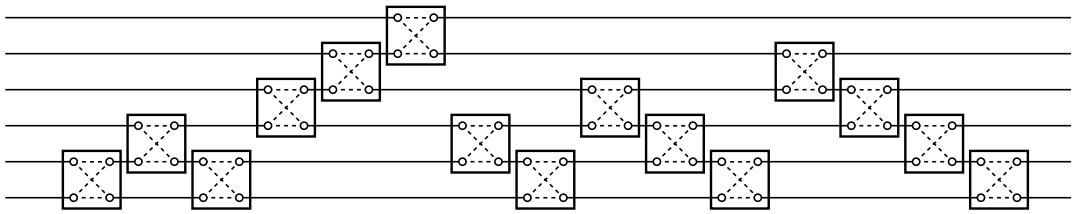


Figure 13: 6-way permutation network

ber of inputs is large. Fig. 13 illustrates the concept with a 6-way permutation network. Its 15 switches can each be in either of two states: pass the inputs left to right unchanged, or swap them. For 6 inputs, a crossbar switch is likely to be cheaper, in that it is likely to require only $36$ switches, in comparison with $60$ for the permutation network shown in Fig. 13. However, for $1000$ inputs, assuming $N \log_2 N$, approximately $40,000$ switches are required, whereas a $1000 \times 1000$ crossbar switch would require 1 million switches.

Designing a permutation network can be somewhat baroque, though a useful relationship with *sorting networks* can be exploited. A sorting network is a sort algorithm that can be modified (if necessary) to allow its architecture to be predetermined, regardless of the data that it is given. Typically, a network is constructed whereby each *swap node* has two inputs and two outputs, where the outputs are swapped (if necessary) in order to respect a given partial order. Though the popular Quicksort is unsuitable, many other well-known sort algorithms, e.g. merge sort, bubble sort, transposition sort, bitonic sort or shell sort, can be adapted. Since a sort may also be seen as just a particular kind of permutation, sort networks – by definition – must be capable of performing permutations. Furthermore, since the data to be sorted might initially be in any order, a sort network must be capable of supporting all possible permutations – therefore, if a sort algorithm can be adapted to create a sort network of arbitrary dimension, it follows that an equivalently structured permutation network would also be capable of any possible permutation. Usefully, the underlying sort algorithm can be leveraged to efficiently generate switch configurations, as follows:

1. Let $\langle W, \sqsubset \rangle$ be a totally ordered set such that $|W|$ is the number of wires in the switch network, and each $w \in W$ represents exactly one input and one output.

2. Let the total bijective map $P : \wp(W \times W)$ represent the desired permutation to be implemented by the switch network.

3. Sort $P$ with the underlying sort network, such that for each $(a, b) \in P$, $a$ represents the input, and $b$ represents the output. This can be achieved trivially by feeding tuples into the network ordered on $a$, then having the network sort these tuples ordered on $b$.

4. Note whether each swap node passed its data through unchanged, or whether it performed a swap. This gives the switch configuration for an isomorphic permutation network that performs an equivalent permutation.

Since suitable sort algorithms exist that have $O(N \log N)$ time complexity, computing a switch plan is therefore also an $O(N \log N)$ operation.

Permutation networks are nevertheless not guaranteed to be a better solution than crossbar switches, particularly when constructed as ASICS – their complex wiring reduces the effective advantage of their reduced switch count, particularly when considering that regular grids (crossbar switches being a particularly ideal example) are cheap and easy to lay out in comparison with the more spaghetti-like nature of large permutation networks. Limitations on chip packaging limit the number of wires that a single chip might be able to switch, and therefore also the number of switches that need sensibly be integrated in one die, reducing the impact of the $O(N^2)$ complexity problem with crossbar switches. However, when switches are large and/or expensive, as is the case with MEMS relays or any discrete component approach (e.g. full-size relays, MOSFETs, IGBTs), the reduction in component count could prove important.

### 2.4.3 Shuffle networks

Shuffle networks are essentially degenerate, incomplete permutation networks that do not support all possible permutations. They are perhaps best known in the parallel computing world, where they are commonly used as high speed inter-processor interconnect architectures. Omega networks, a commonly used shuffle network architecture, typically require some kind of blocking or queueing hardware at each swap node so that collisions can be arbitrated. Their incompleteness is probably not tolerable for our application, so they will not be considered further.

### 2.4.4 Ad-hoc COTS approaches

In some cases, COTS devices may be used to implement routing fabric. FPGAs, in particular, are ubiquitous, low cost and can be used (with appropriate considerations) in high radiation environments. There are a number of potential approaches:

1. Implement a general purpose crosspoint switch or permutation network as a HDL model, then synthesise it.

2. Generate HDL that routes the FPGA's inputs and outputs according to the desired switching plan, then synthesise the design.

The first option clearly limits the size of switch that can be implemented in a particular FPGA, though is inherently general purpose and can be reconfigured very rapidly. The second option is probably infeasible for embedded use at the time of writing due to the requirement for a complete tool chain in order to perform reconfiguration.

### 2.4.5 Embedding into networks of arbitrary topology

In this approach, a reconfigurable manifold is represented by a graph where its nodes represent switches and its edges represent wires. Embedding a desired circuit into such a network is essentially equivalent to computing a switch configuration. For the general case, this is a difficult computational problem that seems almost certainly to be in $NP$, with complexity rising exponentially with the number of switches in the network. Though this approach ultimately encompasses all others, in that both crossbar

switches and permutation networks may be seen as special cases, the difficulty of computing switching plans makes it unlikely that this approach could be feasible in practice.

## 2.5 Make-before-break switching

At the device level, make-before-break switching requires the capability to establish a new connection, in parallel, before an old connection is disconnected. Where a reconfigurable manifold is routing signals that should not be temporarily interrupted, make-before-break switching allows a connection to be moved to an alternative route transparently to the signal's endpoints.

Power, heavy current analogue, low-speed digital and low-speed analogue signals are all well suited to make-before-break switching, in that they are not particularly sensitive to minor changes in end-to-end resistance or discontinuities in impedance. However, high-speed digital, high-speed analogue, or (particularly) microwave signals need more careful consideration – in such cases, it may be necessary for the subsystems concerned to become involved in the routing process, at least from the point of view of being able to request that the manifold should not re-route particular signals during critical periods.

Crossbar switches support make-before-break switching by default: it is just necessary to turn on the switch for the new connection, waiting long enough (if necessary) for the switch to close fully and stop bouncing, then turn off the switch for the old connection. Implementing make-before-break switching in a permutation network is much more difficult, however, and will almost certainly require the network to be carefully designed (see Section 5.1).

In a reconfigurable manifold that does not alter its wiring plan after it has been initially configured, support for make-before-break switching is unnecessary – however, such a capability is essential in order to support continuous automated testing and fault recovery (see Section 4).

## 2.6 Grounding

Grounding of electronic systems within satellites is broadly similar to the grounding of Earth-based electronics; as-such, the same techniques and best practice applies in both cases. In satellites, grounding is particularly important because of the *charging effect,* whereby charged particles impacting the spacecraft impart a (potentially large) electric charge – careful grounding all conductive parts typically reduces or eliminates any consequential problems.

It is normal practice for a spacecraft to implement a ground network with a star topology – a single central grounding point is connected radially to the grounds on all subsystems. Cycles in the ground network are avoided, because they can form unwanted single-turn secondaries that may pick up hum or other unwanted noise from any heavy current subsystems in the vicinity.

Normally, grounds should not need to be switched by a reconfigurable manifold – a conventional, fixed, star ground topology should be sufficient for nearly all cases. Signals that are routed along shielded paths may require switchable ground lifts at one or both ends in order to avoid ground loops, though careful consideration of possible ground routing requirements may avoid this.

# 3 Self-organisation

In some circumstances, it is undesirable or even impossible to precalculate routing for a reconfigurable manifold. The responsive space paradigm requires that disparate subsystems should be able to be plugged together in any convenient manner, at which point they should self-organise and work together without human intervention. Achieving concept-to-launch times of the order of one week does not leave much time for anything other than physical assembly of the spacecraft, so the electronic subsystems must, of absolute necessity, not require a lengthy design process.

Self-organisation, at a fundamental level, requires subsystems to be able to discover each other, negotiate and configure any necessary wiring, and also to cooperate in maintaining the long-term reliability of the connectivity. These issues are discussed in detail in the remainder of this section.

## 3.1 'Space Velcro'

Some technologies absolutely require self-organisation in order to function at all. Fig. 14 is an electron micrograph of Joshi et. al.'s *Microcilia* concept [8, 11, 3]. MEMS technology is used to construct micron scale, articulated 'cilia' that are capable of manipulating small objects and of allowing the docking of small microsatellites. Assuming that electrical connections between the mated surfaces can be achieved, a self-organising, reconfigurable manifold based satellite could automatically configure any necessary connections during docking, then automatically recover the routing resources once the microsatellite has undocked.

Brei et. al. have investigated a passive interconnect architecture known as *Active Velcro* [6, 5, 4]. Fig. 15 illustrates the concept[4]. Mating, Velcro-like surfaces also contain a (possibly large) number of connectors, a proportion of which happen to make valid connections. Discovering these connections, then routing them via a reconfigurable manifold, potentially allows extremely straightforward ad-hoc construction. In manned spaceflight applications, an astronaut could connect or disconnect a piece of equipment simply by sticking or unsticking it to a Velcro-like pad[5]. In satellite applications, assuming that launch G force and vibration constraints are met, the same approach could allow extremely rapid construction and deployment.

## 3.2 Local routing

In a very small satellite, or within a single subsystem of a more complex satellite, routing may be exclusively *local*, i.e. switched only by a single level of switch networks. All connections in such a case would occur only to the edge of a single manifold, or cluster of sub-manifolds configured to act logically as a single manifold, with the consequence that the routing of all signals is equivalent only to routing across the manifold itself.

[4]Note that this is the author's rendering, and is intended to be representational of the connectivity approach rather than an accurate physical description

[5]The use of Velcro to avoid small object floating around the cabin of manned spacecraft has long been standard practice.

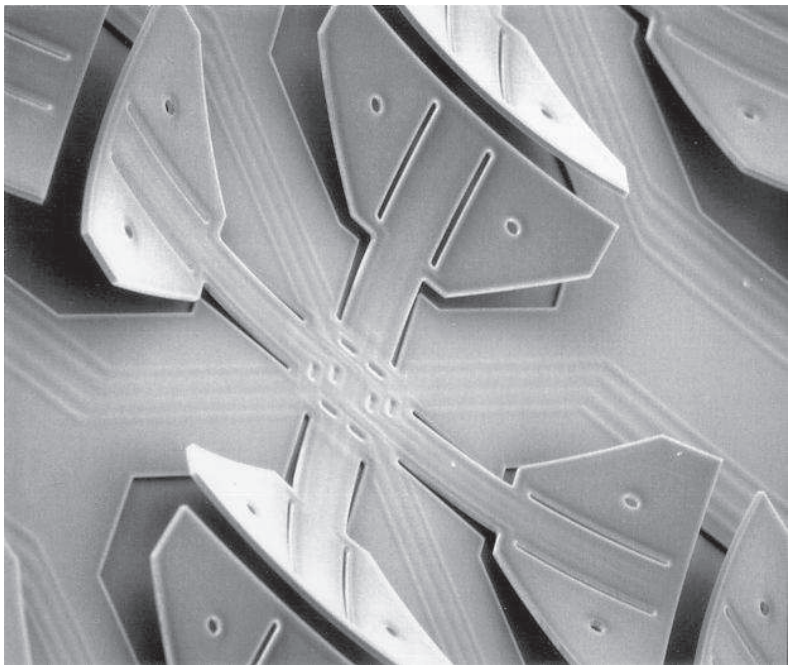


Photo: John Suh, University of Washington

Figure 14: Microcilia Cell

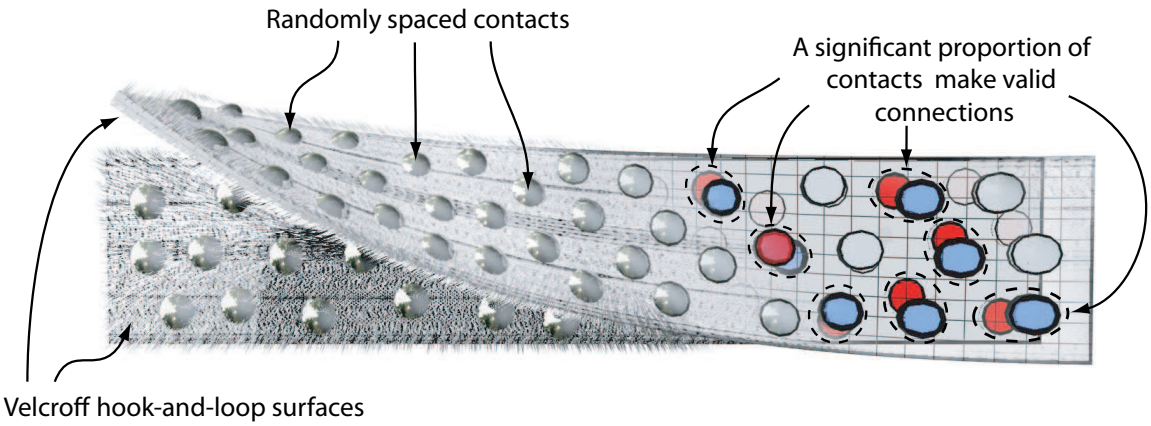



Figure 15: Active Velcro

Computationally, routing for such an architecture is relatively trivial, with complexity of the order of $O(N^2)$ for a crossbar architecture or $O(N \log N)$ for a permutation network.

## 3.3 System level routing

Purely local routing requires a strict star architecture, with the manifold at the hub. This physical geometry does not suit all applications – in many cases, particularly in larger spacecraft, it is likely to be more appropriate to distribute the switching around the

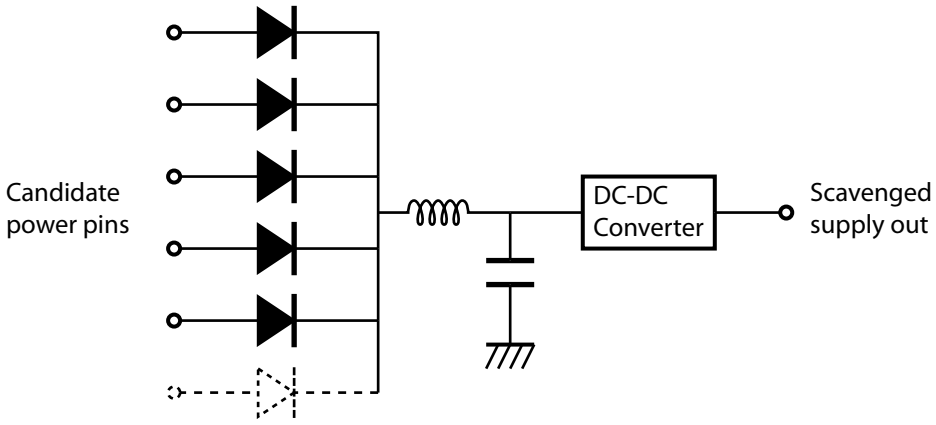



Figure 16: Power Scavenging Circuit

craft. Though it is theoretically possible to construct a large crossbar switch by ganging together smaller switches, this would be an expensive approach since the amount of inter-switch cabling would rise in proportion to the square of the number of switches. A more sensible and practical approach would be to construct a manifold-of-manifolds with an architecture resembling that of a circuit-switched telephone network – a number of manifolds handle primarily local connections internally, whilst handing off longer-distance connections via multicore trunk connections to other manifolds.

Computationally, the system level routing problem tends towards $NP$ in the worst case (e.g. a manifold-of-manifolds where each manifold consists of exactly one switch and connectivity between manifolds is arbitrary is essentially the same problem that is discussed in Section 2.4.5), though the relatively small number of manifolds and relatively large amount of connectivity within each manifold is likely to minimise the consequences of this.

## 3.4 Dynamic discovery

The *dynamic discovery* of connections is something that is becoming increasingly common in general-purpose computing. The USB standard, for example, allows devices to be discovered and configured automatically without significant human intervention. From the point of view of reconfigurable manifolds, the dynamic discovery problem is somewhat trickier, in that it is necessary to first power up any neighbouring subsystems, establish contact with them (potentially with zero prior knowledge of their wiring configuration), negotiate any required connections, then route the necessary signals. As a second requirement, it is then necessary to continuously re-test the existing connectivity in order that faults can be corrected and that subsystems coming on line or going off line can be connected and disconnected correctly.

In this section, the requirements for achieving reliable dynamic discovery, continuous testing and fault recovery are discussed.

### 3.4.1 The chicken-and-egg problem

It is a truism that any automatic discovery algorithm can only possibly run on hardware that is itself powered up. However, if a subsystem's power connections have not yet been discovered and configured, it will not (yet) be powered up – hence there is a chicken-and-egg problem. Though no longer in common use, a well-known solution already exists. For many years, the most commonly used PC peripheral interface standards, RS232 and Centronics, both suffered from a design oversight – no power supply pins – that proved maddening for any hardware engineer attempting to design small peripherals without separate mains power supply connections. Designers nevertheless succeeded in working around the limitation by including circuits that scavenged power from the I/O pins themselves. The technique is illustrated in Fig. 16 – a diode network, effectively a large-scale generalisation of a full-wave rectifier circuit, synthesises power rails effectively by implementing a minimum/maximum function on the voltages that are present. The clamping, smoothing and DC-DC converter circuitry takes the potentially rather unpredictable raw output from the diode network and turns it into clean power that can be safely used to power up discovery circuitry prior to permanent routes being put in place.

Given suitable power scavenging circuits, a feasible power-up procedure for a large, manifold-of-manifolds architecture might be follows:

1. Power is applied to the first manifold through any arbitrary power pin.
2. The power scavenger circuit synthesises a suitable voltage rail for the embedded processor and discovery hardware responsible for the manifold.
3. All switches within the manifold are initialised to open circuit
4. The power connection is detected, then connected via the manifold, thereby disabling the diode network. This step avoids the inherent voltage drop across the diode network, whilst also reducing power consumption and heat dissipation slightly.
5. The manifold starts to listen for connection requests from other subsystems (see Section 3.4.3)
6. Power is temporarily routed to arbitrary pins on neighbouring subsystems that currently do not appear to be active, giving them the chance to power up and begin their own discovery process. They may request that power is supplied through a different pin, if necessary, or request that the existing pin should remain connected indefinitely[6].

Eventually, all subsystems will be powered up, with the discovery process continuing to bring online all other necessary connections.

### 3.4.2 Watchdogs

It is standard practice for embedded processors in high reliability, mission critical and safety critical systems to be equipped with *watchdog circuits,* see Fig. 17.

[6]though it may be subject to change as part of the self-test algorithm

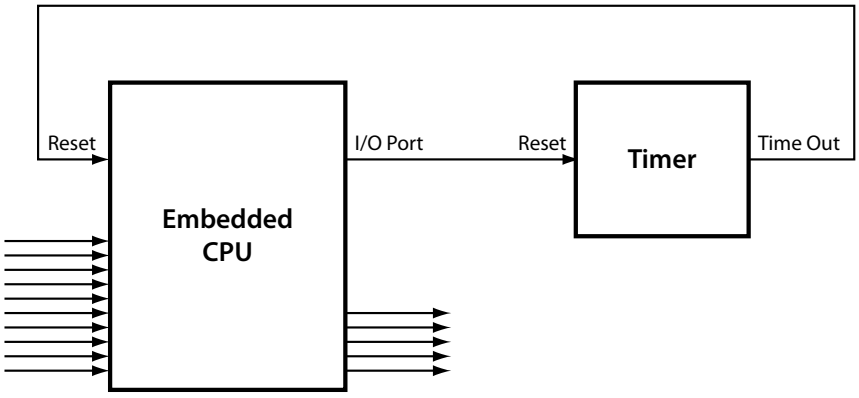



Figure 17: Typical watchdog circuit

A watchdog circuit is essentially a simple timer that is periodically reset by the host processor in such a way that, if the host processor happens to fail to reset it within a predetermined interval, the watchdog timer performs a hard reset on the host processor. Generally, this is integrated into a critical loop within the embedded software, such that if the program crashes this will cause the timer to fail to be reset, causing an automatic restart of the processor.

At a simplistic level, there is no reason why such a restart should cause problems for a manifold-of-manifolds architecture, though careful attention must be given to the following issues:

1. In the event of a watchdog reset, all external connections must be torn down, just in case the crash was itself caused by a faulty connection or, for example, by a SEE affecting the manifold itself.

2. Any negotiation protocol must be able to cope, e.g. by implementing timeouts, with connections going down without any corresponding explicit notification.

### 3.4.3 Discovery probe circuits

Connection discovery depends upon an ability to safely probe connections to find out what neighbouring subsystem they are connected to. The outline circuit shown in Fig. 18 shows how a suitable 'discovery probe' might be implemented. The circuit shows a UART (bidirectional serial interface) connected to a host processor, whose serial I/O ports (marked TxD and RxD) assume good quality, logic-level signals. On the transmit side, the signal is first buffered in order to protect the UART, then high pass filtered to achieve AC coupling and connected to the probe output via a resistor, whose value should be carefully selected in order to limit worst case current in the event of an accidental connection to a power or high current analogue signal to a level that can not cause damage. On the receive side, a similar current limiting resistor and high pass network protects the active components from direct connection to otherwise potentially damaging signals. A DC-coupled linear amplifier boosts the signal, then a Schmidt trigger (comparator with hysteresis) squares up the signal and raises it to logic levels suitable for the RxD input of

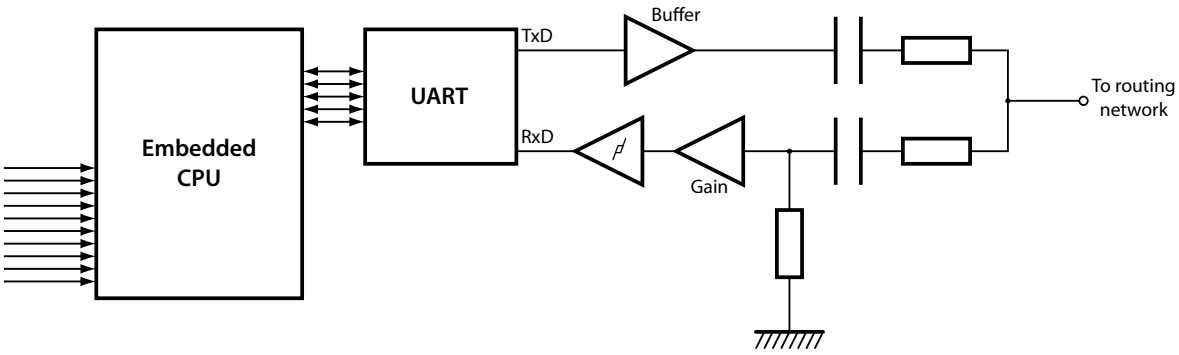



Figure 18: A possible discovery probe circuit

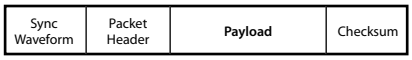



Figure 19: Typical packet format

the UART. Current limiting resistors should be chosen with values that are not too over-specified, since lower values are likely to result in better noise performance and higher achievable data rates.

In essence, the probe circuit is a simplified, extremely robust variation of a shared bus CSMA/CD network interface, in the style of 10Base2 Ethernet. AC coupling and a relatively high series resistance minimises the chance of damage due to accidental connection to higher voltage signals, whilst the ability to send and receive digital data without needing to switch between transmit and receive modes makes implementing higher level protocols relatively straightforward.

Sending serial data across AC coupled connections requires careful design of the low-level line protocol. Sending, for example, a long string of ones will cause the voltage to decay back to a centre value over a period of time that is determined by the time constant of the high pass filter. Similarly, a data packet that consists predominantly of ones (or zeros) will tend to shift away from the most common value, causing an unwanted DC bias and consequential reduction in noise margins. Typically this is addressed by arranging for the data encoding to implicitly retain an equal number of 0s and 1s – a trivial, though inefficient, approach is to spread an 8 bit byte across 16 bits, where each input bit corresponds to an inverted and a non-inverted copy in the output word. More efficient encodings exist that spread 2 bytes across 24 bits.

### 3.4.4 Line protocol

The main function of a suitable line protocol is to allow the discovery of of connections, then to allow routing negotiation for signals. Probe circuits will typically alternate between sending packets that announce the identity of the relevant wire and listening for incoming packets that identify the other side of the connection. A suitable packet format is likely to follow the pattern shown in Fig. 19. Initially, a synchronisation waveform begins the transmission, whose purpose is to overcome any DC bias, whilst allowing the

receiving UART time to lock on to the data. A packet header follows, identifying the kind of packet that is being sent, followed by the packet payload and finally a checksum.

### 3.4.5 Connection establishment

Connections are established as follows (assuming a single manifold):

1. Both endpoints announce their identity, and announce the identifier of the signal that they wish to connect to.
2. Manifold detects the announcements
3. Manifold replies to both end points to say that the connection is being established, then ceases to probe either connection
4. Manifold establishes the connection, within a predetermined maximum time interval
5. Both endpoints are now free to use the connection.

More complex manifold-of-manifolds architectures will require more complex negotiation and routing, though the necessary protocols are likely to remain similar.

### 3.4.6 Stale connection tear-down

In the event that a subsystem crashes, stale connections should be torn down after a known time-out interval. The discovery probe protocol should also allow a connection to be torn down more rapidly by announcing that a neighbouring connection is no longer in use. Assuming that a dynamic testing and fault recovery process will be continuously applied, there is no requirement for a 'keep alive' protocol to ensure that valid connections stay up (see also Section4).

# 4 Dynamic testing and fault recovery

The same probe architecture necessary for discovery is also well suited to end-to-end testing of connections – if a connection is faulty (e.g. open circuit, shorted to ground or shorted to power), it will not be used, since the discovery process will fail to recognise it. As a consequence of this, at least for a short time after the discovery process has completed, all discovered connections may be regarded as functioning correctly. Over time, there is an increasing probability that, for example, permanent latch-up damage to a digital crossbar switch, may cause one or more connections to fail. This limitation can be avoided by constantly re-testing connections, ideally such that no connection may be established for a period longer than the minimum necessary to achieve the desired level of reliability.

## 4.1 Fault recovery protocol

There is actually no specific requirement to implement a fault recovery protocol as-such; the ability to set up and tear down connections, with make-before-break capabilities, is sufficient. Each end-point manifold should implement the following procedure (discovery and initial establishment of connections is assumed to have happened already):

1. Choose a signal on a round-robin basis
2. Establish a second route to the same remote end-point through the discovery protocol, which has the side-effect of ensuring that end-to-end connectivity is currently valid.
3. Connect the signal to the newly established route, at both ends, whilst leaving the original connection in place
4. Tear down the original connection
5. Repeat.

Note that in larger systems, connections between manifolds must always provide sufficient spare connections to allow the discovery protocol to remain in operation at all times.

The stale connection timeout (see Section 3.4.6) should be longer than the worst-case time necessary to cycle through all connections.

When a connection fails, it will be repaired automatically the next time that the fault recovery procedure cycles through the relevant signal, because the failed route will no longer be detected, so it will naturally fall out of the pool of available connections.

## 4.2 Graceful degradation

In a situation where cumulative failures have exceeded the number of available connections, it is sensible to define a graceful degradation strategy in order to maximise the spacecraft's remaining functionality. A simple approach is to rank all signals in order of importance, with signals toward the end of the list simply being disconnected if insufficient connectivity is available, though more sophisticated approaches may allow greater levels of recovery:

**Routing signals on a less-ideal sub-manifold** Normally, for example, digital data would be routed through dedicated digital switch networks. In the event that insufficient digital switching capacity remains, it is potentially feasible to route signals through spare capacity in other switch networks, e.g. via MEMS switching that would normally be used for microwave signals or via high speed analogue routes.

**Multiplexing** Manifolds could potentially be equipped with multiplexing hardware, in order that multiple low speed signals could be routed through a single connection. Though this may degrade any signals carried in this way, it may still be preferable to disconnecting signals entirely.

**Emergency backup routing** As an extension to the multiplexing approach, in an emergency backup routes could be established by non-standard means, such as via low power local digital radio links.

# 5 Conclusions

At the time of writing, this project is at a relatively early stage; nevertheless, it is possible Se determine the following advantages of reconfigurable manifolds over conventional fixed-architecture spacecraft wiring harnesses:

**Cost Reduction** Since a reconfigurable manifold doesn't need to be designed from-scratch for each satellite, considerable cost reductions in terms of initial design, validation and verification are likely.

**Reduction in Time To Launch (Responsive Space)** Reduced design effort has a direct effect in terms of calendar time, potentially helping reduce a design process that is conventionally measured in years to just weeks or even days.

**Possibility for Re-purposing After Launch** If a spacecraft is no longer required for its initial purpose, given a modular design, it is quite likely that it could be re-purposed after launch at very low cost. For example, an imaging satellite with excess communications bandwidth could, assuming it has enough fuel, be shifted to another orbit to act as a communications relay.

**Disaster Recovery** Now legendary, the recovery of Apollo 13 after an explosion that deprived the command module of all three of its fuel cells and its entire oxygen reserve, with all crew alive and unhurt [12], was a direct consequence of heroic efforts to jury-rig the lunar lander's oxygen systems in order to keep the crew alive. A conventional satellite has no astronauts with a kit of spare parts available to make repairs – typically, failures tend to be terminal. A reconfigurable manifold offers great potential for jury-rigging the craft, either from Earth or possibly autonomously, so as to allow it to continue with some or all of its mission.

**Mass reduction** By sharing redundant wiring capacity across all subsystems, the total amount of copper necessary is reduced considerably in comparison with modular-redundant conventional wiring. At approximately $30,000 per Kg to low earth orbit, even small savings can have considerable consequences in terms of cost.

## 5.1 Future Work

We conjecture that, in general, make-before-break switching is not feasible for permutation network based switch fabrics; further theoretical work is necessary in order to confirm this assumption. Ideally, it is hoped that a (probably non-optimal) permutation network architecture might be possible that can cope with interruption-free reconfiguration, though it is not clear at the time of writing how this might be achieved.

Many, if not all of the prerequisites for the practical construction of satellites based upon reconfigurable manifold technology are well-established, so the problem is primarily one of systems integration rather than difficult original R&D. The next step we intend to take is to build a software simulation of a reconfigurable manifold in order to test the feasibility of the approach. Beyond that, given appropriate funding and the necessary political will, it just remains to design a practical implementation and, hopefully, to trial it in space.

## Acknowledgements

This work was supported by the US Air Force Office of Scientific Research Space Vehicles Directorate, through an EOARD grant. The first author wishes to thank AFOSR at Kirtland AFB, and Jim Lyke in particular, for their help and advice, without which this work would not have been possible.

## References

[1] *AD8116 - 200 MHz, 16 × 16 Buffered Video Crosspoint Switch*. Analog Devices, 2006. http://www.analog.com/en/prod/0,2877,768

[2] *High Performance Crossbar Switch for Virtex-II and Virtex-II Pro FPGAs*. Xilinx, 2006. www.xilinx.com/esp/xbarswitch.htm.

[3] BOHRINGER, K. F. A docking system for microsatellites based on microelectromechanical system actuator arrays. Tech. Rep. AFRL-VS-TR-2000-1099, US Air Force Research Laboratory, Space Vehicles Directorate, September 2000.

[4] BREI, D., AND CLEMENT, J. Proof-of-concept investigation of active velcro for smart attachment mechanisms. Tech. Rep. AFRL-VS-TR-2000-1097, US Air Force Research Laboratory, Space Vehicles Directorate, September 2000.

[5] BREI, D., AND CLEMENT, J. Velcro for smart attachment mechanisms. Tech. Rep. AFRL-VS-TR-2001-1104, US Air Force Research Laboratory, Space Vehicles Directorate, August 2001.

[6] CLEMENT, J. W., AND BREI, D. E. Proof-of-concept investigation of Active Velcro for smart attachment mechanisms. In *In Proc. 42nd AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference and Exhibit* (2001). AIAA Paper 2001-1503 (AIAA Accession number 25238).

[7] FOUST, J. Smallsats and standardization. *The Space Review* (2005).

[8] JOSHI, P. B. On-orbit asssembly of a universally interlocking modular spacecraft (7225-020). Tech. Rep. NASA SBIR 2003 Solicitation Proposal 03- II F5.03-8890, NASA, 2003.

[9] LYKE, J., WILSON, W., AND CONTINO, P. MEMS-based reconfigurable manifold. In *Proc. MAPLD* (2005).

[10] STOICA, A., ARSLAN, T., KEYMEULEN, D., DUONG, V., GUO, X., ZEBULUM, R., FERGUSON, I., AND DAUD, T. Evolutionary recovery of electronic circuits from radiation induced faults. In *Proc. IEEE Conference on Evolutionary Computation* (2004), CEC.

[11] SUH, J. W., DARLING, R. B., BOHRINGER, K. F., DONALD, B., BALTES, H., AND KOVACS, G. T. A. SMOS integrated ciliary actuator array as a general-purpose micromanipulation tool for small objects. 1999.

[12] TURNILL, R. *The Moonlandings: an eyewitness account*. Cambridge, 2003.