Rule-based language from modular design to modular verification

June 11, 2024

1 Notes - session-1

Rule-based language from modular design to modular verification(mindset: Nothing is finite.)

Why hardware design language:

- AMD and Intel.
- specialized hardware.

Challenge:

- Specification for hardware.
- Deal with concurrency.

1.1 Modules

In order to model the rule-based languages describing these systems, we turn to "modules". Modules can be represented as tuples with 4 elements:

- S, the set of states of the system
- $R \subseteq S \times S$, the set of rules describing spontaneous transitions between states
- $\mathcal{A} = \{\alpha.m \mid \alpha \subseteq S \times \mathbb{N} \times S\}$, the set of "action" methods which are relations describing parameterized transitions between states
- $\mathcal{V} = \{\beta.m \mid \beta \subseteq S \times \mathbb{N} \times \mathbb{N}\}$, the set of "value" methods which are relations describing possible observations of the system

For simplicity, methods here take a single natural number as an argument, but in general, methods make take many arguments or no arguments at all from arbitrary sets.

Definition(modular): (S, type, {R}, {action}, {value})

$R \in S \times S$ $a.m \in S \times \mathcal{N} \times S$ $v.m \in S \times \mathcal{N} \times \mathcal{N}$

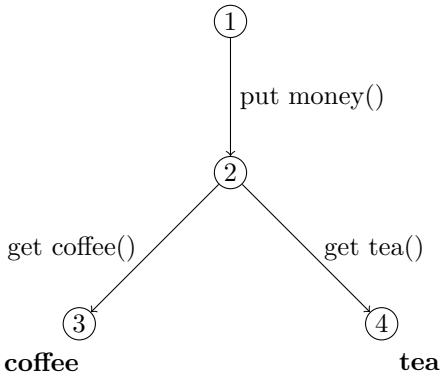
Example(coffee tea):

For hardware, design a language to compose hardware together.

Example 2(register): $\{\mathcal{N}, \{\}, \{write = \{(x, y, z) | \forall x, y \in \mathcal{N}^2\}, \{read = \{(x, 3, x) | \forall x, 3 \in \mathcal{N}^2\}\}\}\}$

Example: $\{\mathcal{N} \to \mathcal{N}, \{\}, \{write =\}, \{read = \{(\ell, arg, \ell(arg)\}\}\}$

Queue: $\{(list\mathcal{N}, \{\}, \{enq, deq\}, \{list\}\}$



1.2 Notions of Equivalence

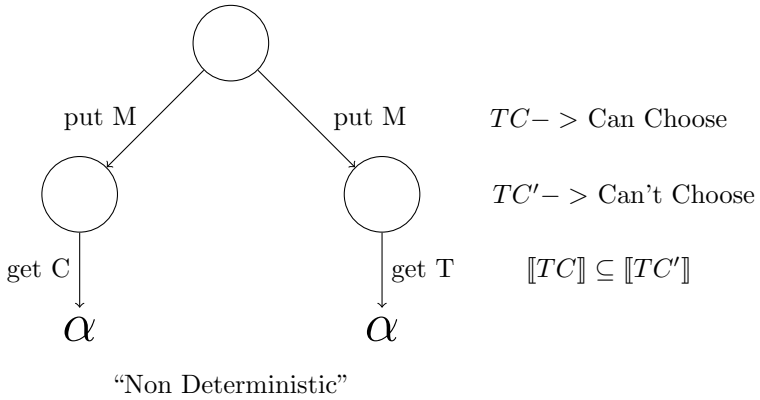
Modules specify an abstract interface with action methods for interacting with them and value methods for observing them, so naturally we might ask when one module may be replaced with another, that is we want some form of "equivalence" on modules. Strict equivalence of two modules is often too strong of a condition to be useful, so we instead define a related condition called "refinement" such that if two modules M and M' are refinements of each other, we recover equivalence. There are two natural definitions for what it means for one module to refine another.

1.2.1 Trace Refinement

Def(trace of a module/behavior): $S_o \rightarrow S_\tau \rightarrow action(1)$.
$S_o \rightarrow (\epsilon)s\phi \rightarrow S_c - S$

[arg(1), first(1)$\rightarrow$ 1]
Starting from initial state S_0, a trace of a module is a list of all invocations of action and value methods during the execution of the module. Note that spontaneous transitions via rules do not appear in traces. Concretely, we say that a module M refines a module M' iff $[\![M]\!] \subseteq [\![M']\!]$, where $[\![M]\!]$ is the set of all traces of M. The issue with this definition of refinement is that it does not allow us to distinguish between modules with similar interfaces, yet completely different behavior.

For example, consider a variation of the tea-and-coffee machine in which the machine decides internally whether to dispense tea or coffee:



"Non Deterministic"

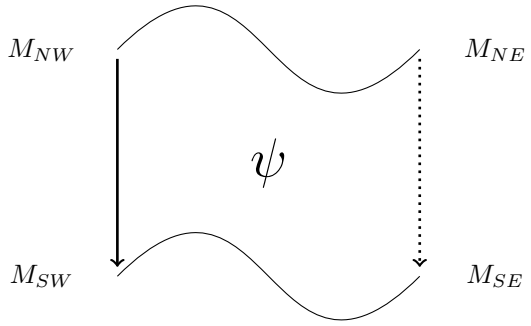
We have that $[\![TC]\!] \subseteq [\![TC']\!]$, yet the choice between coffee and tea is determined for us in the case of TC'.

1.2.2 Weak Simulation

We say that a module A weakly simulates a module B, $(A \sqsubseteq_\varphi B)$, witnessed by a relation $\varphi : S_A \times S_B$ if the following hold:

1. $\varphi(a_0, b_0)$, that is the initial states of A and B are related by φ.
2. $\forall v \in \mathcal{V}, \forall a_m \in \mathcal{A}, \forall arg, ret \in \mathbb{N}, (a_0, arg, ret) \in \mathcal{V} \implies (b_0, arg, ret) \in \mathcal{V}$
3. $\forall a \in S_A, b \in S_B, \varphi(a, b) \implies \forall \alpha \in \mathcal{A}, \forall arg \in \mathbb{N}, (a, arg, a') \in \alpha, \exists b', b'' \in S_B, (b, arg, b')$ and $\phi(m_2, m_2')\exists m_2'' m_2' \rightarrow (R)m_2''$

Furthermore, it is possible to prove that for modules M and M', $M \sqsubseteq_\varphi M' \implies [\![M]\!] \subset [\![M']\!]$.



$$\frac{(m, \ell, m') \in enq_M}{m.enqE(arg) \to m'}\ enqE()$$

$$\frac{(m, *, m') \in deq_M}{m.cleaq(arg) \to m'}\ \text{cleqE()}$$

$$\frac{(m, *, ret) \in deq_M \qquad g(ret') = ret}{m.put(e) \to ret}\ \text{protE()}$$

$$\frac{m_1 deq \to m_1', m_1 pop() \to ret, \qquad m_2 enq(g(ret)) \to m_2'}{(m_1, m_2) \to (m_1', m_2')}\ \text{internal}$$

2 Notes - session-2

2.1 Plan for the day

- **Language Definition**
- **Compilation to a circuit**
- **Examples : Weak Simulation + Refinement Theorems**

2.2 Language definition

e ::=$f(e)$	**s** ::=$s; s$
$m.vm(e)$	(if (e) then s else s)
	$m.am(e)$
	let var = e
	$abort$

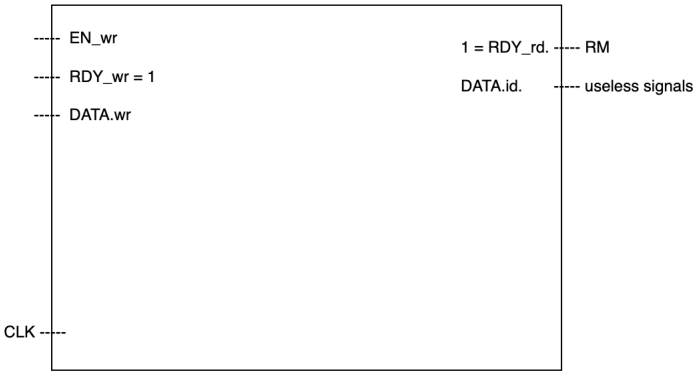
2.3 Weak Simulation

$M \sqsubseteq_\beta M'$ is defined by:
- $\psi M_o M'_o$

$\oplus$

$$\forall XY, \psi \longrightarrow X < y"X < Y" := \forall vm, arg, ret, X.vm(org) = ret \Longrightarrow Y.vm(arg) = ret$$

Register



Conflict matrix CM(M)

	wr	rd
wr	C(Conflict)	>
rd	<	CF(Conflict free)

Table 1: The CM

	r_1	r_2	r_3
r_1	$\supset$		
r_2		$\supset$	
r_3			$\supset$

Table 2: Caption

Compilation without if

- case 1: abort $\rightarrow$ do nothing
- case 2: double write $\rightarrow$ error
- case 3: (m.am1(e_1); m.am2(e_2);m.am3(e_3))
  1. RDY_r := ...(R ready if all modules are ready)
  2. EN_m_am1:=En_r; DATA_m_am1 := [[e]];
  3. ... for module 2

4. ... for module 3

- case 4:
  1. rule 1: let x = a.acl()?
     b.wr(x)
  2. rule 2: ket x == l.acl()?
     a.wr(x)

Lattice of conflicts

level 1: C
level 2: $<$ and $>$
level 3: CF

For a pair of rules, if multiple conflicts $c_1, \ldots, c_n$ occur between them, then the resulting conflict is given by the least upper bound of $\{c_1, \ldots, c_n\}$. For example, if $r_1 < r_2$, and $r_1 > r_2$ (as in the previous example, then they have a conflict (lub$\{<, >\} = C$.

Queue

M := 1-element queue (valid, data)
enq(e) :
  if (valid.read() == 0)
    valid.rd(1),
    data.rd(0),
  else
    abort

M' list based queue

$(\ell : ListN)$

$\ell - (enq(e)) \rightarrow e :: \ell$

$\ell H[e] - (deq()) \rightarrow \ell$

$\ell H[e] - (first()) \rightarrow e$

Definitions

$$\frac{\exists \ell, M \sqsubseteq_{\psi} M'.\psi(v,d)}{\psi(0,*)[\ ]}$$

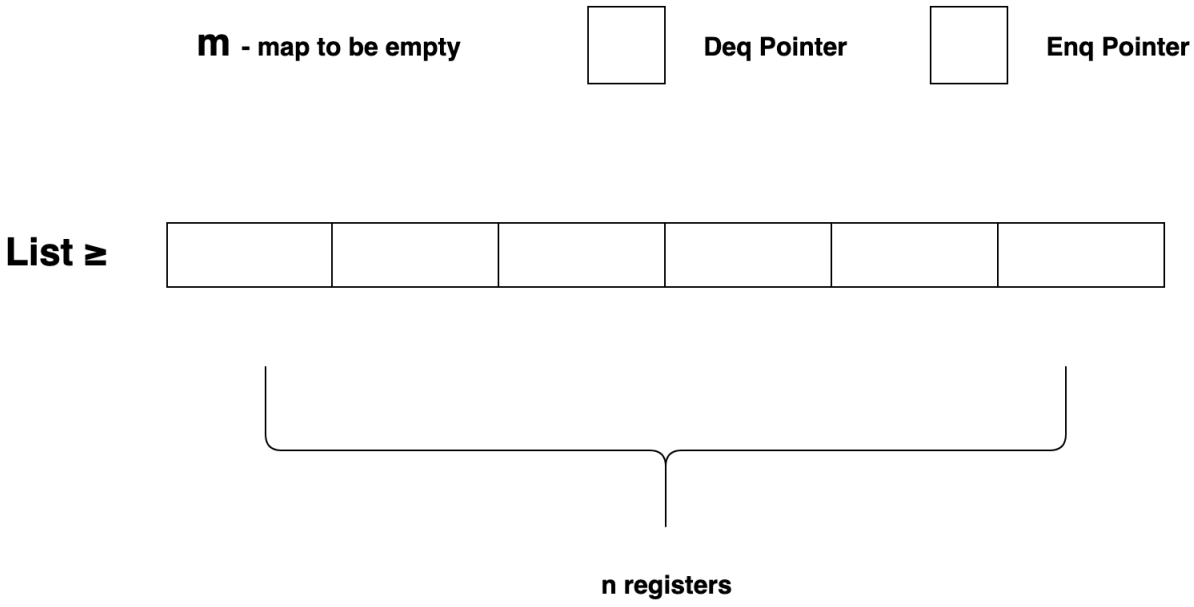
$$\frac{\exists \ell, M \sqsubseteq_{\psi} M'.\psi(v,d)}{\psi(1,e)[e]}$$

$$\frac{(0,d)-(enq(e))\rightarrow(v',d') \quad \ell=[]}{\exists ei[]\rightarrow(enq(e))\rightarrow\ell' \wedge \psi(1,e)\ell \quad instantiate\ \ell'=[e]}$$

$$\frac{\exists \psi, I \sqsubseteq S \wedge S \sqsubseteq_{\psi} I}{\psi(q_1,q_2,q_3)}$$

$map\ g \cdot f\ q_1 + +map\ gq_2 + +q_3 = q_5 = g(f(e)).q_5$
$(q_1 + +[e]) \Rightarrow (map\ g \cdot f\ q_1 + +map\ g(f(e) :: q_2) + +q_3)$

Refinement Is Compositional
$M(N)$
$N \sqsubseteq_{\phi} N'$
$M(N) \sqsubseteq_{\phi} M(N')$



3 Notes - session-3

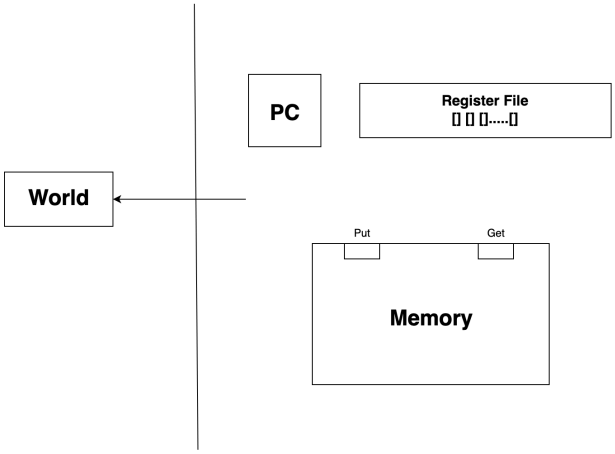


Figure 1: Processor Visualized

- Progam Counter :: next instruction to be executed
- Memory :: Instructions, Data of the program
- Registers :: Usually 32, holds values that should be brought back

Steps for Execution

- Fetch = get men[pc]
- Decode $r_1 \leftarrow ac$, r_2, r_3
- Execute ALU control flow memory
- write back

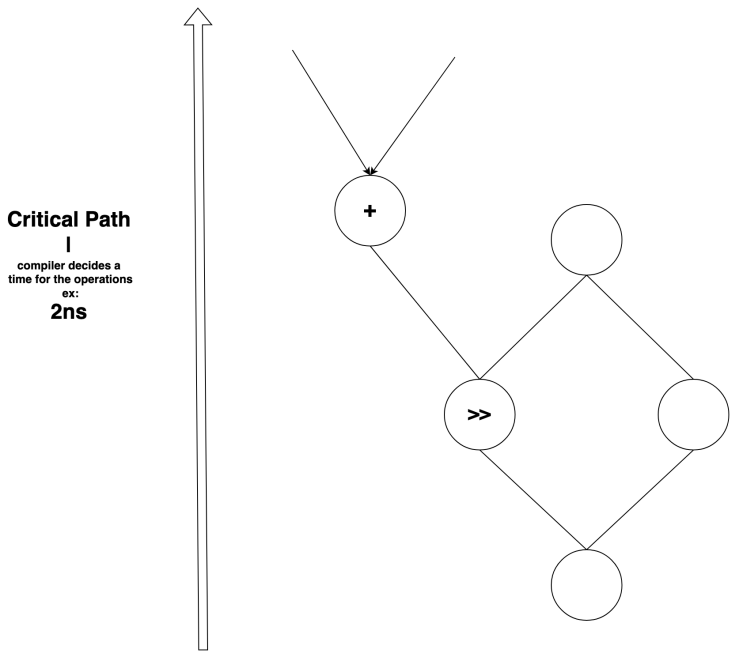


Figure 2: Critical Path Visualized

Code example:

```
typedef enum {Fetch, Decode, Execute, Writeback}
    StateProcessor deriving(Eq, Bits);

function Bool isMMIO(Bit#(32), addr);
    return (addr == 32'hf000fff0 || addr == 32'hf000fff8);
end function

module mkmulticycle(Empty);
BRAM1Port#(Bit#(16), Bit#(32)) mem -> mkMemory();

Reg#(Bit#(32)) pc -> mkReg(0);
Vector#(32, Reg#(Bit#(32) rf -> replicateM(mkReg(0));
Reg#(Stateprocessor) current_state -> mkReg(Fetch);

rule fetch if (current$_$state == Fetch);
    let req = BRAMRequest {
```

```
        write: false,
        address truncate(pc >> 2):,
        datain: ?,
        responseOnWrite: False
    }

    mem.portA.request.put(req);
    current_state <= Decode;
endrule

rule decode if (current_state == Decode);
    let instr -> mem.portA.response.get();
    let decodedinstr = decodeInst(instr);
    let rs1_idx = getInstFields(instr).rs1;
    let rs2_idx = getInstFields(instr).rs2;

    let rs1 = (rs1_idx == 0 ? 0 : rf[rs1_idx]);
    let rs1 = (rs2_idx == 0 ? 0 : rf[rs2_idx]);
    dInst <= decodedInstr;
    rd <= rs1;
    rs1 <= rs2;
    current_state <= Execute;
endrule

rule execute if (current_state == Excute);
    let imm =getImmediate(dInst);
    let data = execALU32(dInst.inst, rv1, rv2, imm , pc);
    let addr = rv1 + imm;
    if (ifMemoryInst(dInst)) begin
        data = rv2;
        let type_mem = (dInst.inst[5] == 1);
        let req = BRAMRequest {
            write: false,
            address truncate(pc >> 2):,
            datain: data,
```

```
            responseOnWrite: False
        };
        if (isMMIO(addr)) begin
            if (addr == 'hf000_fff0) $fwrite(stdout, "%c", data[7:0]);
            if (addr == 'hf000_fff8) begin
                $display("TERMINTATE");
                $finish;
            end
        end else begin
            mem.portA.request.put(req);
        end
    end
    else begin
        if (isControlInst(dInst)) begin
            data = pc + 4;
        end
    end
    let nextPc = execControl32(dInst.inst, rv1, rv2, imm, pc);
    pc <= nextPc;
    rd <= data;
    current_data <= Writeback;
endrule

rule writeback if( current_state == Writeback);
    let data = rd;
    if (isMemoryInst(dInst)) && !isMMIO(addr)) begin
        let resp <- memo.portA.response.get();
        data = resp;
    end
    // use data that corresponds to either coming from memory,
    // or coming from previous stage
    if (dInst.valid_rd) begin
        let rd_idx = getInstFields(dInst.inst).rd;
        if (rd_idx != 0) rf[rd_idx] <= data;
    end
```

4 Notes - session-4

4.1 Tricks of computer architects and inductive refinement maps

There is plenty of room at the top a bit more motivation for providing correctness of architecture.
a nice way to do refinement maps - Inductive refinement maps.
Tricks of Architects - Codex Preview

- Pure Pipelining
- Stateful Pipelining
- Duplicating (Parallel lanes)
- Banking

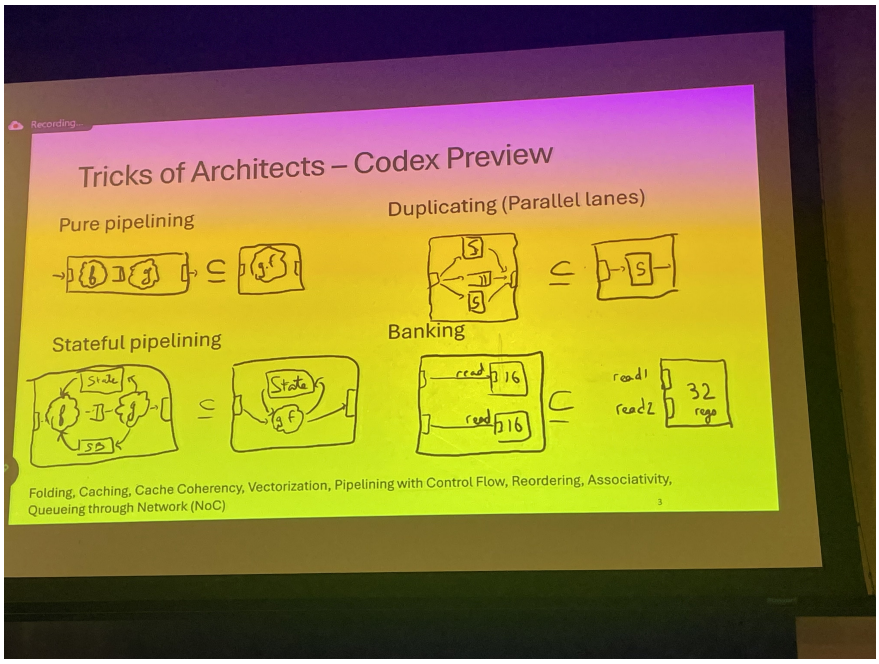


Figure 3: CodeX

Version	Implementation	Running time (s)	GFLOPS	Absolute speedup	Relative speedup	Fraction of peak (%)
1	Python	25,552.48	0.005	1	–	0.00
2	Java	2,372.68	0.058	11	10.8	0.01
3	C	542.67	0.253	47	4.4	0.03
4	Parallel loops	69.80	1.969	366	7.8	0.24
5	Parallel divide and conquer	3.80	36.180	6,727	18.4	4.33
6	plus vectorization	1.10	124.914	23,224	3.5	14.96
7	plus AVX intrinsics	0.41	337.812	62,806	2.7	40.45

Figure 4: Room for Perfomance Language Comparison

What we like/don't like

- Yay:
  - property of implementation flushes and specification agree post flushing is a very rich relation.
  - Architecturally meaningful.
  - Criteria amenable to automatic verification when pipeline has bounded depth.
- Abstain: prove that the ultimately lazy machine is Burch&Dill correct.
- Nay:
  - Requires to write the flushing steps as shadow logic.
  - Ambiguous – more than one way to "flush" -¿ are those notion of correctness equivalent.

Flushing our $(f \cdot g)$ pipeline: an inductive simulation relation

$$\begin{aligned}\text{Inductive } \phi : \text{ImplState} \to \text{SpecState} \to \text{Prop} := \\ &| \text{Flushed} : \forall l, \phi([],[],l)([],1) \\ &| \text{one\_more\_f} : \forall i\ i'\ s, \\ &\quad (i \sim (\text{do\_f}) \sim i') \to \\ &\quad \phi\ i'\ s \to \phi\ i\ s \\ &| \text{one\_more\_g} : \forall i\ i'\ s, \\ &\quad (i \sim (\text{do\_g}) \sim i') \to \\ &\quad \phi\ i'\ s \to \phi\ i\ s\end{aligned}$$

phi i s $->$ i $<$ s
by induction on phi:

- base case: phi ([], [], l) ([], l), masquerading all good
- inductive case (do_f easy, do_g not completely immediate)

Unshelving the issues

- how hard is it to write phi?
  - did you consider inductive flushing?
- how big is phi?
  - Linear in the size (number of transitions < 10 when doing hierarchical proofs) of the system
- how much does phi change when doing a little design update?
  - very little, but the proof might change
- what is the scam?
  - N^2 cases in the inductive case

Actual problems with refinements
Two systems

- Implementation processor
- Simple specification processor
- A Pipelined Processor is not $\not\sqsubseteq$ Simple specification processor

Generalizing the specification

4 sequential steps:
Fetch, Decode, Execute, Writeback always works on exactly one instruction, no specification/ prediction.

Two non-deterministic load machine:
Processor does not directly emit loads to memories. instead processor queries the load

buffers. Load buffers are refilled nondeterministically.

Why is the generalization valid?

Architecturally it is obvious: loads don't matter

'loads don't matter, from the perspective of the MMIO trace of the full system'

Is that intuition formalizable?

What prevents us to make a mistake? We just changed our specification with no discussion?

Generalized specification that emits random stores? Clearly wrong.

Modular Proof

implementation processor $\sqsubseteq$

Generalized specification processor $\sqsubseteq$

simple specification processor