Binary In HOL

$2^n a = a \frown 0 \cdots 0$ (n times)

$a \times b = \sum_{i=0}^{n-1} a_i b \frown 0 \cdots 0$ (i times)

$Bits = \{T, F\}$

$|b_{n-1} \cdots b_0| = n$

$(w_1 = w_2) \equiv (|w_1| = |w_2|) \wedge \forall i.i < |w_1| \Rightarrow w_1[i] = w_2[i]$

$b_{n-1} \cdots b_0[i] = (i \geq n) \rightarrow (0|b_i)$

$m \geq n \Rightarrow (w[m:n])[i] = ((i + n \leq m) \rightarrow (w[i+n]|0))$

$Bw(b)[0] = b$

$Bv(T) = 1$, $Bv(F) = 0$

$V(b_{n-1} \cdots b_0) = \sum_{i=0}^{n-1} 2^i b_i$

$V(W\, n\, m) = m \mod 2^n$

$w\{n \leftarrow b\}[i] = (i = n) \rightarrow (b|w[i])$

$w_1 \uplus w_2 = W\, (\max(|w_1|, |w_2|) + 1)\, (V(w_1) + V(w_2))$

$b_{m-1} \cdots b_0 \frown c_{n-1} \cdots c_0 = b_{m-1} \cdots b_0 c_{n-1} \cdots c_0$

$b.w = b \rightarrow (w|W\, |w|\, 0)$

Hardware In HOL

The behaviour of a device Dev is specified by a predicate that holds only when the signals it relates are allowable values on the corresponding lines of Dev. Circuits are specified by conjoining multiple device predicates and using $\exists$-quantification to hide internal lines.

To verify a circuit, show that $\vdash \forall a.Circuit\_Imp(a) \Rightarrow Circuit(a)$ or $\vdash \forall a.Circuit\_Imp(a) \iff Circuit(a)$.

Using the underspecified (partial specification) form allows design freedom, but may fail to stipulate some important aspect of device behaviour.

Inconsistent models trivially satisfying any specification, so you may want to prove a consistency theorem of the form $\exists v_1 \cdots v_n.M[v_1, \cdots, v_n]$ or $\forall i_1 \cdots i_n.\exists o_1 \cdots o_m.M[i_1, \cdots, i_n, o_1, \cdots, o_m]$.

Sequential devices are modelled my taking the values on lines to be sequences of values modelled as functions of time. Combinatorial devices can be modelled as sequential devices having no delay.

$Stable(t_1, t_2)(f) \equiv \forall t.t_1 \leq t \wedge t < t_2 \Rightarrow (f(t) = f(t_1))$ - f is constant from t_1 to just before t_2.

$Next(t_1, t_2)(f) \equiv t_1 < t_2 \wedge f(t_2) \wedge (\forall t.t_1 < t \wedge t < t_2 \Rightarrow \neg f(t))$ - t_2 is the first time strictly after t_1 that f holds.

$Rose(f)(t) \equiv \neg f(t-1) \wedge f(t)$

$Rise(f)(t) \equiv \neg f(t) \wedge f(t+1)$

Simple Switch Model

$Ntran(g, a, b) \equiv (g \Rightarrow (a = b))$

$Ptran(g, a, b) \equiv (\neg g \Rightarrow (a = b))$

This is somewhat inadequate as it suggests circuits can be used backwards (e.g. $Inv(i, o) \Rightarrow i = \neg o$). It also does not account for different conduction characteristics of transistors. N-type transistors transmit logic low well, but high poorly, and vice versa for P-types.

However, it is easy to use (combinatorial) and good for sanity checking.

Fourman's Switch Model

Modify the model to use three logic values: $\{Hi, Lo, X\}$.

$Ntran(g, a, b) \equiv (g = Hi) \Rightarrow ((a = Lo) = (b = Lo))$

$Ptran(g, a, b) \equiv (g = Lo) \Rightarrow ((a = Hi) = (b = Hi))$

$Strong(v) \equiv (v = Hi) \vee (v = Lo)$

$TBv(Hi) = 1$, $TBv(Lo) = 0$

Unidirectional Sequential Model

This allows us to reason about gates making use of dynamic logic. Use four logic values: $\{Hi, Lo, Fl, X\}$.

$Strong(v) \equiv (v = Hi) \vee (v = Lo)$

$Float(v) \equiv v = Fl$

$v1\, U\, v2 \equiv (Strong(v1) \wedge Float(v2)) \rightarrow (v1|(Float(v1) \wedge Strong(v2)) \rightarrow (v2|(Float(v1) \wedge Float(v2)) \rightarrow (Fl|X)))$

$Join(i1, i2, o) \equiv \forall t.out(t) = i1(t)\, U\, i2(t)$

$Cap(i, o) \equiv \forall t.o(t) = Strong(i(t)) \rightarrow (i(t)|(t = 0) \rightarrow (X|(Float(i(t)) \wedge Strong(i(t-1))) \rightarrow (i(t-1)|Fl)))$

$Ntran(g, a, b) \equiv \forall t.b(t) = (g(t) = Hi) \rightarrow (a(t)|(g(t) = Lo \vee a(t) = Fl) \rightarrow (Fl|X)))$

$Ptran(g, a, b) \equiv \forall t.b(t) = (g(t) = Lo) \rightarrow (a(t)|(g(t) = Hi \vee a(t) = Fl) \rightarrow (Fl|X)))$

$Pu(i, o) \equiv \forall t.o(t) = Float(i(t)) \rightarrow (Hi|i(t))$

This model is of dubious electrical validity, but can sanity check functional correctness of sometimes quite subtle dynamic logic circuits.

All of these models may break down when shorts are present in the circuit.

Temporal Abstraction

A time mapping is a function f such that $\forall t_1 t_2.(t_1 < t_2) \Rightarrow (f(t_1) \leq f(t_2))$, mapping abstract to concrete time.

To verify a model using temporal abstraction, check that $\vdash M[v_1, \cdots, v_n] \Rightarrow S[v_1 \circ f, \cdots, v_n \circ f]_{\mathcal{A}}$ since S is couched in terms of concrete time.

$IsTimeof\, 0\, s\, t = s(t) \wedge \forall t'.t' < t \Rightarrow \neg s(t')$

$IsTimeof\, (n+1)\, s\, t = \exists t'.IsTimeof\, n\, s\, t' \wedge Next\, t'\, t\, s$

$Timeof\, s\, n = \epsilon t.IsTimeof\, n\, s\, t$ - the time such that s is true for the nth time, if such a time exists.

$Inf\, p = \forall t.\exists t'.t' > t \wedge p\, t'$

$Inf\, p \Rightarrow \forall n.\exists! t.IsTimeof\, n\, p\, t$, so $Inf\, p \Rightarrow IsTimeof\, n\, p\, (Timeof\, p\, n)$.

$s\, when\, P = s \circ (Timeof\, P)$

$f@ck = f\, when\, (Rise\, ck)$

Temporal Logic Motivation

Hoare logic for data reasoning, temporal logic for time (control) reasoning.

There are some properties me might want to hold that cannot be expressed as Hoare style correctness specifications, such as the fact that a variable does not change during the computation.

Model Checking

Express models as state transition systems.

$\mathcal{B} :: states \rightarrow bool$, predicate tests whether state is an initial state.

$R :: states \times states \rightarrow bool$, predicate tests whether a state transitions to another.

Can have $\delta :: states \times inputs \rightarrow states$, so $\mathcal{R}(s, s') = \exists i.s' = \delta(s, i)$: this gives a deterministic machine with "input nondeterminism".

If you have n state variables v_i and transition functions δ_i:

$\mathcal{R}(\vec{v}, \vec{v'}) = \exists n. \bigvee_{i=1}^{n} \left[\bigwedge_{j=1}^{n} v'_j = (i = j) \rightarrow (\delta_j(\vec{v}, n) | v_j) \right]$

Can compute set of reachable states by $S_0 = \{s | \mathcal{B}(s)\}$, $S_{n+1} = S_i \cup \{s | \exists u.u \in S_n \wedge \mathcal{R}(u, s)\}$. If the state space is finite this will reach a fixed point.

In symbolic model checking, sets of states are represented via predicates rather than explicit representation. In general, the transition relation on states with n variables is represented as a formula with $2n$ variables.

$ReachBy\, 0\, \mathcal{R}\, \mathcal{B}\, s = \mathcal{B}(s)$

$ReachBy\, (n+1)\, \mathcal{R}\, \mathcal{B}\, s = ReachBy\, n\, \mathcal{R}\, \mathcal{B}\, s \vee \exists u.ReachBy\, n\, \mathcal{R}\, \mathcal{B}\, u \wedge \mathcal{R}(u, s)$

$Reach\, \mathcal{R}\, \mathcal{B}\, s \equiv \exists n.ReachBy\, n\, \mathcal{R}\, \mathcal{B}\, s$

$(ReachBy\, n\, \mathcal{R}\, \mathcal{B} = ReachBy\, (n+1)\, \mathcal{R}\, \mathcal{B}) \Rightarrow (Reach\, \mathcal{R}\, \mathcal{B} = ReachBy\, n\, \mathcal{R}\, \mathcal{B})$

Reduced ordered binary decision diagrams (ROBDDs) are used for representing Boolean formulas. They are canonical (given a variable ordering) and efficient to manipulate, both in construction and finding satisfying assignments. Fast to detect fixed points due to hash-consing. However, efficiency of the representation depends heavily upon variable ordering (the difference between exponential and linear space consumption).

Early quantification / disjunctive partitioning can be used to reduce the size of BDDs we need to manipulate. In particular, you may never have to compute the transition relation BDD. For the n state variable case above:

$ReachBy\, (n+1)\, \mathcal{R}\, \mathcal{B}\, \vec{v} = ReachBy\, n\, \mathcal{R}\, \mathcal{B}\, \vec{v} \vee \bigvee_{i=1}^{n} \exists v'_i.ReachBy\, n\, \mathcal{R}\, \mathcal{B}\, (\vec{v}\{i \leftarrow v'_i\}) \wedge v_i = \delta_i(\vec{v}\{i \leftarrow v'_i\})$

To check if a property $\mathcal{Q}$ is true is all reachable states, compute the BDD of $Reach\, \mathcal{R}\, \mathcal{B}\, s \Rightarrow \mathcal{Q}\, s$ and check if it is the single node T. Otherwise extract counterexample satisfying $Reach\, \mathcal{R}\, \mathcal{B}\, s \wedge \neg(\mathcal{Q}\, s)$ by finding the first n at which $Reach\, n\, \mathcal{R}\, \mathcal{B}\, s \wedge \neg(\mathcal{Q}\, s)$ is satisfiable and hence a state s_n satisfying the formula. Then find a state s_{i-1} satisfying $ReachBy\, (i-1)\, \mathcal{R}\, \mathcal{B}\, s \wedge \mathcal{R}(s, s_i)$ iteratively until you have a trace of states demonstrating a counterexample.

CTL

CTL is a branching time logic: may talk about all or some paths.

A model is a pair $(\mathcal{R}, s)$ of a transition relation and an initial state.

Formula	Semantics
$Atom(p)$	$\lambda(\mathcal{R}, s).p(s)$
$\neg P$	$\lambda(\mathcal{R}, s).\neg P(\mathcal{R}, s)$
$P \wedge Q$	$\lambda(\mathcal{R}, s).P(\mathcal{R}, s) \wedge Q(\mathcal{R}, s)$
$P \vee Q$	$\lambda(\mathcal{R}, s).P(\mathcal{R}, s) \vee Q(\mathcal{R}, s)$
$P \Rightarrow Q$	$\lambda(\mathcal{R}, s).P(\mathcal{R}, s) \Rightarrow Q(\mathcal{R}, s)$
$AX\, P$	$\lambda(\mathcal{R}, s).\forall s'.\mathcal{R}(s, s') \Rightarrow P(\mathcal{R}, s')$
$EX\, P$	$\lambda(\mathcal{R}, s).\exists s'.\mathcal{R}(s, s') \wedge P(\mathcal{R}, s')$
$A[P\, U\, Q]$	$\lambda(\mathcal{R}, s).\forall \sigma.Path(\mathcal{R}, s)\, \sigma \Rightarrow HoldsUntil\, P\, Q\, \mathcal{R}\, \sigma$
$E[P\, U\, Q]$	$\lambda(\mathcal{R}, s).\exists \sigma.Path(\mathcal{R}, s)\, \sigma \wedge HoldsUntil\, P\, Q\, \mathcal{R}\, \sigma$
$AF\, P$	$A[T\, U\, P]$ - P holds somewhere along every path.
$EF\, P$	$E[T\, U\, P]$ - P holds somewhere along some path.
$AG\, P$	$\neg(EF\, (\neg P))$ - P holds everywhere along every path.
$EG\, P$	$\neg(AF\, (\neg P))$ - P holds everywhere along some path.
$A[P\, W\, Q]$	$\neg E[(P \wedge \neg Q) U (\neg P \wedge \neg Q)]$, partial $A[P\, U\, Q]$

$HoldsUntil\, P\, Q\, \mathcal{R}\, \sigma \equiv \exists i.Q(\mathcal{R}, \sigma(i)) \wedge \forall j.j < i \Rightarrow P(\mathcal{R}, \sigma(j))$

$Path(\mathcal{R}, s)\, \sigma = (\sigma(0) = s) \wedge \forall t.\mathcal{R}(\sigma(t), \sigma(t+1))$

It is straightforward to apply model checking to CTL. For example, to check $EF(Atom\,p)(\mathcal{R}, s)$ mark states satisfying p and repeatedly mark the states with *at least one* marked successor until a fixed point is reached: $\mathcal{S}_0 = \{s|p(s)\}$, $S_{i+1} = S_i \cup \{s|\exists s'.\mathcal{R}(s,s') \wedge s' \in \mathcal{S}_i\}$. The formula is true in exactly those states that are marked. Similar algorithm for $AF(Atom\,p)$ (mark if all successors marked). Can recursively decompose problem.

Cannot express fairness properties: must be implemented in the model checking algorithm somehow.

Cannot express property "on every path there is a point after which p is always true *on that path*".

LTL

LTL is a linear time logic: may talk about all paths and particular paths.

Formula	Semantics
$Atom(p)$	$\lambda\sigma.p(\sigma(0))$
$\neg P$	$\lambda\sigma.\neg(P(\sigma))$
$P \vee Q$	$\lambda\sigma.P(\sigma) \vee Q(\sigma)$
$X\,P$	$\lambda\sigma.P(Tail\,1\,\sigma)$
$F\,P$	$\lambda\sigma.\exists m.P(Tail\,m\,\sigma)$
$G\,P$	$\lambda\sigma.\forall m.P(Tail\,m\,\sigma)$
$[P\,U\,Q]$	$\lambda\sigma.\exists i.Q(Tail\,i\,\sigma) \wedge \forall j.j < i \Rightarrow P(Tail\,j\,\sigma)$

$Tail\,m\,\sigma = \lambda n.\sigma(n+m)$

Can express fairness properties such as "if p holds infinitely often on path then so does q", using $A(G(F\,p) \Rightarrow G(F\,q))$.

Can express "on every path there is a point after which p is always true on that path", using $F\,G\,p$.

Cannot express property "from every state it is possible to get to a state for which p holds".

CTL*

Allows the expression of both state and path properties.

CTL is CTL* restricted with X, F, G and $[-U-]$ preceded by A or E.

LTL consists of CTL* formulas of the form $A-$ where the only state formulas are atomic.

Semantics and form fairly obvious. State semantics are of form $\lambda(\mathcal{R}, s)$, path semantics of form $(\mathcal{R}, \sigma)$.

ITL

Specifies properties of finite intervals (sequences of states with a beginning and an end).

Useful for talking about transactions.

Formula	Semantics
$Atom(p)$	$\lambda\langle s_0, \cdots, s_n\rangle.p(s_0)$
$true$	$\lambda\langle s_0, \cdots, s_n\rangle.T$
$\neg P$	$\lambda\langle s_0, \cdots, s_n\rangle.\neg P\langle s_0, \cdots, s_n\rangle$
$P \vee Q$	$\lambda\langle s_0, \cdots, s_n\rangle.P\langle s_0, \cdots, s_n\rangle \vee Q\langle s_0, \cdots, s_n\rangle$
$skip$	$\lambda\langle s_0, \cdots, s_n\rangle.n = 1$
$P; Q$	$\lambda\langle s_0, \cdots, s_n\rangle.\exists k.k \leq n \wedge$ $P\langle s_0, \cdots, s_k\rangle \wedge Q\langle s_k, \cdots, s_n\rangle$
$P*$	$\lambda\langle s_0, \cdots, s_n\rangle.\exists w_1 \cdots w_l.$ $\langle s_0, \cdots, s_n\rangle = w_1 \cdots w_l \wedge \bigwedge_{i=1}^{l} P(w_i)$

PSL

SEREs: finite state decidable matchers, similar to but weaker than ITL.

Formula	Semantics
$Atom(p)$	$\lambda w.p(head(w))$
$r_1 \vert r_2$	$\lambda w.r_1(w) \wedge r_2(w)$
$r_1; r_2$	$\lambda w.\exists w_1, w_2.w = w_1.w_2 \wedge r_1(w_1) \wedge r_2(w_2)$
$r_1 : r_2$	$\lambda w.\exists w_1, s, w_2.w = w_1.s.w_2 \wedge r_1(w_1.s) \wedge r_2(s.w_2)$
$r_1 \&\& r_2$	$\lambda w.r_1(w) \wedge r_2(w)$
$r[*]$	$\lambda w.w = \langle\rangle \vee \exists w_1 \cdots w_l.w = w_1 \cdots w_l \wedge \bigwedge_{i=1}^{l} r(w_i)$

Formula	Semantics
$Atom(p)$	$\lambda\sigma.p(\sigma(0))$
$\neg f$	$\lambda\sigma.\neg(f(\sigma))$
$f_1 \vee f_2$	$\lambda\sigma.f_1(\sigma) \vee f_2(\sigma)$
$next\,f$	$\lambda\sigma.f(Tail\,1\,\sigma)$
$\{r\}(f)$	$\lambda\sigma.\exists w, \sigma'.\sigma = w.\sigma' \wedge r(w) \wedge f(\sigma')$
$\{r_1\}\vert- > \{r_2\}$	$\lambda\sigma.\exists w_1, \sigma'.\sigma = w_1.\sigma' \wedge r_1(w_1) \Rightarrow$ $\exists w_2, \sigma''.\sigma' = w_2.\sigma'' \wedge r_2(w_2)$
$\{f_1\ until\ f_2\}$	$\lambda\sigma.\exists i.f_2(Tail\,i\,\sigma) \wedge \forall j.j < i \Rightarrow f_1(Tail\,j\,\sigma)$

PSL allows SEREs and formulae to be clocked. This can be translated away by pushing $@clk$ inwards: e.g. $b@clk = \{!clk[*]; clk\&b\}$.

PSL's optional branching extension (OBE) contains a complete copy of CTL.

PSL may be statically checked using finite automata and LTL/CTL methods. It may also be checked dynamically against simulation runs, but that requires the semantics to allow for the possibility of infinite paths.

Layered structure:

- Boolean layer: atomic predicates
- Temporal layer: LTL or CTL
- Verification layer: how to use predicates, constructs such as *assert* and *assume*

- Modelling layer: HDL constructs for specifying and verifying things that logics cannot check, typically involving event counts (temporal logic has no arithmetic)

Hardware Verification

Static verification:

- Does not execute the model
- Uses a model checker, type checker or equivalence checker

Dynamic verification:

- Actually executes the model on test data
- Uses a simulator

Relating Trace And Event Models

Register transfer level (RTL): behaviour is a state machine, registers with zero-delay combinatorial logic. May or may not have explicit clocks.

Trace level: clock edges are explicit, combinatorial logic can have delays.

Cycle semantics: similar to state machine view of RTL.

Behavioural level: machine cycles merge and only I/O events are significant.

HDLs use discrete event simulation: variable changes cause threads to be enabled, the threads are executed non-deterministically and potentially trigger more events. Event triggers are captured by the *always* construct of Verilog: positive/negative edges or combinatorial triggers.

State of a simulation consists of values of variables and a set of enabled threads.

Verilog simulation cycle:

1. If there is no enabled thread, advance simulation time and go to 1
2. Otherwise, choose one thread and execute it
3. Fire event controls activated by last thread to enable new threads

VHDL simulation cycle:

1. If there is no enabled thread, advance simulation time and go to 1
2. Otherwise, execute all enabled threads in parallel
3. Fire event controls activated by last threads to enable new threads

Steps at the same simulation time happen in δ-time.

We would like to unify the event semantics of the simulators used by designers with the trace semantics of logic. Ideally, we could prove that traces are sequences of quiescent simulation states. Impossible in general in Verilog due to simulation non-determinism, unless you use non-blocking assignment (equivalent to all simulated events in a cycle referring to old rather than current state values). VHDL simulation semantics is race-free. For consistency, both semantics require that no inputs change on a clock edge or we could get time violations that reflect metastability of physical implementations.