Example of current research

```
TCAD Newsletter - March 2010 Issue
Placing you one click away from the best new CAD research!

Regular Papers
==============

Zheng, H.; "Compositional Reachability Analysis for Efficient Modular
Verification of Asynchronous Designs"

Abstract: Compositional verification is essential to address state
explosion in model checking. Traditionally, an over-approximate
context is needed for each individual component in a system for sound
verification. This may cause state explosion for the intermediate
results as well as inefficiency for abstraction refinement. This paper
presents an opposite approach, a compositional reachability method,
which constructs the state space of each component from an
under-approximate context gradually until a counter-example is found
or a fixpoint in state space is reached. This method has an additional
advantage in that counter-examples, if there are any, can be found
much earlier, thus leading to faster verification. Furthermore, this
modular verification framework does not require complex compositional
reasoning rules. The experimental results indicate that this method is
promising.

URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5419238&isnumber=5419222
```


Summary of CTL operators (primitive + defined)

- **CTL formulas:**

$\texttt{Atom}(p)$	(Atomic formula - p : *states*→*bool*)
$\neg P$	(Negation)
$P \wedge Q$	(Conjunction)
$P \vee Q$	(Disjunction)
$P \Rightarrow Q$	(Implication)
$\mathbf{AX}P$	(All successors)
$\mathbf{EX}P$	(Some successors)
$\mathbf{AF}P$	(Somewhere – along all paths)
$\mathbf{EF}P$	(Somewhere – along some path)
$\mathbf{AG}P$	(Everywhere – along all paths)
$\mathbf{EG}P$	(Everywhere – along some path)
$\mathbf{A}[P\ \mathbf{U}\ Q]$	(Until – along all paths)
$\mathbf{E}[P\ \mathbf{U}\ Q]$	(Until – along some path)
$\mathbf{A}[P\ \mathbf{W}\ Q]$	(Unless – along all paths)
$\mathbf{E}[P\ \mathbf{W}\ Q]$	(Unless – along some path)

- **Say 'P holds' if $P(\mathcal{R}, s)$ for all initial states s**

Example CTL formulas

- $\mathbf{EF}(\mathit{Started} \wedge \neg \mathit{Ready})$

  It is possible to get to a state where *Started* holds but *Ready* does not hold

- $\mathbf{AG}(\mathit{Req} \Rightarrow \mathbf{AF}\mathit{Ack})$

  If a request *Req* occurs, then it will eventually be acknowledged by *Ack*

- $\mathbf{AG}(\mathbf{AF}\mathit{DeviceEnabled})$

  ***DeviceEnabled* is always true somewhere along every path starting anywhere: i.e. *DeviceEnabled* holds infinitely often along every path**

- $\mathbf{AG}(\mathbf{EF}\mathit{Restart})$

  From any state it is possible to get to a state for which *Restart* holds

More CTL examples (1)

- $\mathbf{AG}(\mathit{Req} \Rightarrow \mathbf{A}[\mathit{Req}\ \mathbf{U}\ \mathit{Ack}])$

  If a request *Req* occurs, then it continues to hold, until it is eventually acknowledged

- $\mathbf{AG}(\mathit{Req} \Rightarrow \mathbf{AX}(\mathbf{A}[\neg \mathit{Req}\ \mathbf{U}\ \mathit{Ack}]))$

  Whenever *Req* is true either it must become false on the next cycle and remains false until *Ack*, or *Ack* must become true on the next cycle

  Exercise: is the AX necessary?

- $\mathbf{AG}(\mathit{Req} \Rightarrow (\neg \mathit{Ack} \Rightarrow \mathbf{AX}(\mathbf{A}[\mathit{Req}\ \mathbf{U}\ \mathit{Ack}])))$

  Whenever *Req* is true and *Ack* is false then *Ack* will eventually become true and until it does *Req* will remain true

  Exercise: is the AX necessary?

More CTL examples (2)

- $\mathbf{AG}[\mathit{Enabled} \Rightarrow \mathbf{AG}[\mathit{Start} \Rightarrow \mathbf{A}[\neg \mathit{Waiting}\ \mathbf{U}\ \mathit{Ack}]]]$

  If $\mathit{Enabled}$ is ever true then if Start is true in any subsequent state then Ack will eventually become true, and until it does $\mathit{Waiting}$ will be false

- $\mathbf{AG}[\neg \mathit{Req}_1 \wedge \neg \mathit{Req}_2$
  $\Rightarrow$
  $\mathbf{A}[\neg \mathit{Req}_1 \wedge \neg \mathit{Req}_2\ \mathbf{U}\ (\mathit{Start} \wedge \neg \mathit{Req}_2)]]$

  Whenever Req_1 and Req_2 are false, they remain false until Start becomes true with Req_2 still false

- $\mathbf{AG}[\mathit{Req} \Rightarrow \mathbf{AX}(\mathit{Ack} \Rightarrow \mathbf{AF}\ \neg \mathit{Req})]$

  If Req is true and Ack becomes true one cycle later, then eventually Req will become false

Some abbreviations

- $\mathbf{AX}_i\ P\ \equiv\ \underbrace{\mathbf{AX}(\mathbf{AX}(\cdots(\mathbf{AX}\ P)\cdots))}_{i \text{ instances of } \mathbf{AX}}$

  P is true on all paths i units of time later

- $\mathbf{ABF}_{i..j}\ P\ \equiv$
  $\mathbf{AX}_i\,\underbrace{(P \vee \mathbf{AX}(P \vee\ \cdots\ \mathbf{AX}(P \vee \mathbf{AX}\ P)\cdots))}_{j-i \text{ instances of } \mathbf{AX}}$

  P is true on all paths sometime between i units of time later and j units of time later

- $\mathbf{AG}[\mathit{Req} \Rightarrow \mathbf{AX}[\mathit{Ack}_1 \wedge \mathbf{ABF}_{1..6}(\mathit{Ack}_2 \wedge \mathbf{A}[\mathit{Wait}\ \mathbf{U}\ \mathit{Reply}])]]$

  One cycle after Req, Ack_1 should become true, and then Ack_2 becomes true 1 to 6 cycles later and then eventually Reply becomes true, but until it does Wait holds from the time of Ack_2

- **More abbreviations in the 'Industry Standard' language PSL**

CTL model checking algorithm

- **A model is a relation $\mathcal{R}$**
- **A property is a CTL formula P**
- **Model checking:** **given CTL formula P compute $\{s \mid P(\mathcal{R}, s)\}$**
- **$P(\mathcal{R}, s_0)$ true if and only if $s_0 \in \{s \mid P(\mathcal{R}, s)\}$**
- **Assume set of states to be finite**
  (infinite state model checking possible for some models)
- **Already seen how to model check reachability**
  $\mathbf{AG}(\texttt{Atom}\ p)(\mathcal{R}, s)\ \equiv\ \forall s'.\ \texttt{Reach}\ \mathcal{R}\ (\texttt{Eq}\ s)\ s'\ \Rightarrow\ p(s')$
  so can model check AG of atomic properties – compute:
  $\{s' \mid\ \texttt{Reach}\ \mathcal{R}\ (\texttt{Eq}\ s)\ s'\ \Rightarrow\ p(s')\}$,
  e.g. via BDD of
  $\texttt{Reach}\ \mathcal{R}\ (\texttt{Eq}\ s)\ s'\ \Rightarrow\ p(s')$

Checking EF $\texttt{Atom}(p)$

$\mathbf{EF}(\texttt{Atom}\ p)(\mathcal{R}, s)$ if p holds along some path starting at s

- **Mark all the states satisfying p**
- **Repeatedly mark all the states which have at least one marked successor until no change**
- **$\{s \mid \mathbf{EF}(\texttt{Atom}\ p)(\mathcal{R}, s)\}$ computed by generating:**

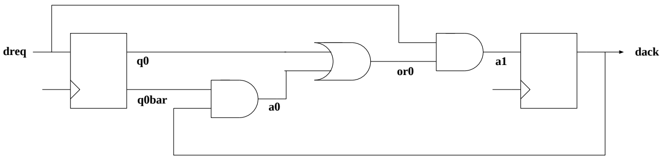
$$\begin{aligned}\mathcal{S}_0 &= \{s \mid (\texttt{Atom}\ p)(\mathcal{R}, s)\}\\ &= \{s \mid p(s)\}\end{aligned}$$

$$\mathcal{S}_{i+1} = \mathcal{S}_i\ \cup\ \{s \mid \exists s'.\ \mathcal{R}(s, s')\ \wedge\ s' \in \mathcal{S}_i\}$$

- **$\mathbf{EF}(\texttt{Atom}\ p)$ is true in marked states and false in unmarked states**
- **Algorithm similar for $\mathbf{AF}(\texttt{Atom}\ p)$:**
  repeatedly mark all the states which have all successors marked
- **To check AF EF $(\texttt{Atom}\ p)$:**
  - **apply EF algorithm**
  - **starting with resulting marking apply AF algorithm**

Recall handshake example

- **Part of a handshake circuit**



- **Transition relation:**

  `(q0' = dreq) ∧ (dack' = dreq ∧ (q0 ∨ dack))`

- **Define $\mathcal{R}_{\texttt{RECEIVER}}$ by:**

  $\mathcal{R}_{\texttt{RECEIVER}}$`((dreq,q0,dack),(dreq',q0',dack')) =`
  `(q0' ⇔ dreq) ∧ (dack' ⇔ dreq ∧ (q0 ∨ dack))`

- **Primed variables (`dreq'`,`q0'`,`dack'`) represent 'next state'**

- **`dreq'` unconstrained, hence non-determinism**

Model checking `RECEIVER`

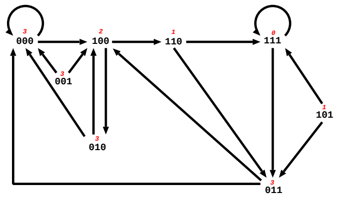
- **Possible states for `RECEIVER`:**

  $\{000, 001, 010, 011, 100, 101, 110, 111\}$

  where $b_2b_1b_0$ denotes state

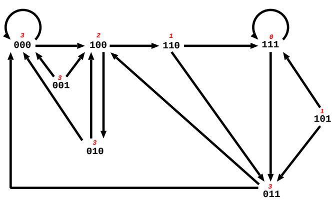
  $\texttt{dreq} = b_2 \ \wedge\ \texttt{q0} = b_1 \ \wedge\ \texttt{dack} = b_0$

- **Graph of the transition relation:**



- i **above a state indicates membership of $\mathcal{S}_i$ (defined below)**

Example: **EF**($\texttt{dreq} \wedge \texttt{q0} \wedge \texttt{dack}$)



- **Define:**

  $P \ = \ \texttt{Atom}(\lambda b_2b_1b_0.\ b_2 \ \wedge\ b_1 \ \wedge\ b_0)$

  $P(\mathcal{R}_{\texttt{RECEIVER}}, b_2b_1b_0) \ = \ b_2 \ \wedge\ b_1 \ \wedge\ b_0$

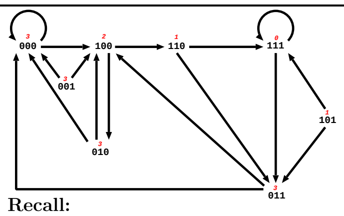
- **Define:**

  $\mathcal{S}_0 \quad = \{b_2b_1b_0 \mid P(\mathcal{R}_{\texttt{RECEIVER}}, b_2b_1b_0)\}$

  $\mathcal{S}_{i+1} = \mathcal{S}_i \ \cup\ \{s \mid \exists s'.\ \mathcal{R}(s,s') \ \wedge\ s' \in \mathcal{S}_i\}$

  $\quad\quad = \mathcal{S}_i \ \cup\ \{b_2b_1b_0 \mid \exists b_2'b_1'b_0'.\ (b_1' = b_2) \ \wedge\ (b_0' = b_2 \wedge (b_1 \vee b_0)) \ \wedge\ b_2'b_1'b_0' \in \mathcal{S}_i\}$

Checking **EF**($\texttt{dreq} \wedge \texttt{q0} \wedge \texttt{dack}$)



- **Recall:**

  $\mathcal{S}_0 \quad = \{b_2b_1b_0 \mid P(\mathcal{R}_{\texttt{RECEIVER}}, b_2b_1b_0)\}$

  $\mathcal{S}_{i+1} = \mathcal{S}_i \ \cup\ \{b_2b_1b_0 \mid \exists b_2'b_1'b_0'.\ (b_1' = b_2) \ \wedge\ (b_0' = b_2 \wedge (b_1 \vee b_0)) \ \wedge\ b_2'b_1'b_0' \in \mathcal{S}_i\}$

- **Compute:**

  $\mathcal{S}_0 = \{111\}$

  $\mathcal{S}_1 = \{111\} \ \cup\ \{101, 110\}$
  $\quad = \{111, 101, 110\}$

  $\mathcal{S}_2 = \{111, 101, 110\} \ \cup\ \{100\}$
  $\quad = \{111, 101, 110, 100\}$

  $\mathcal{S}_3 = \{111, 101, 110, 100\} \ \cup\ \{000, 001, 010, 011\}$
  $\quad = \{111, 101, 110, 100, 000, 001, 010, 011\}$

  $\mathcal{S}_i = \mathcal{S}_3 \quad (i > 3)$

- **Hence** $\forall s.$ **EF**$(\texttt{Atom}(\lambda(\texttt{dreq}, \texttt{q0}, \texttt{dack}).\ \texttt{dreq} \wedge \texttt{q0} \wedge \texttt{dack}))(\mathcal{R}_{\texttt{RECEIVER}}, s)$

Symbolic model checking

- Represent sets of states with BDDs
- Represent Transition relation with a BDD
- If BDDs of $P(\mathcal{R}, s)$, $Q(\mathcal{R}, s)$ are known, then BDDs of
  $\neg P(\mathcal{R}, s)$
  $P(\mathcal{R}, s) \wedge Q(\mathcal{R}, s)$
  $P(\mathcal{R}, s) \vee Q(\mathcal{R}, s)$
  $P(\mathcal{R}, s) \Rightarrow Q(\mathcal{R}, s)$
  can be computed using standard BDD algorithms
- If BDDs of $P(\mathcal{R}, s)$, $Q(\mathcal{R}, s)$ are known, then BDDs of
  $\mathbf{AX}P(\mathcal{R}, s)$, $\mathbf{EX}P(\mathcal{R}, s)$, $\mathbf{A}[P \ \mathbf{U} \ Q](\mathcal{R}, s)$, $\mathbf{E}[P \ \mathbf{U} \ Q](\mathcal{R}, s)$
  computed using fairly straightforward algorithms (see textbooks)
- Model checking CTL generalises iteration for reachable states (`AG`)

History of Model checking

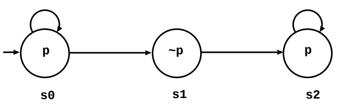
- CTL model checking invented by Emerson, Clarke and Sifakis
- Use of BDDs to represent and compute sets of states is called *symbolic model checking*
- Independently discovered by several people:
  Clarke & McMillan
  Coudert, Berthet & Madre
  Pixley
- SMV (McMillan) is a popular symbolic model checker

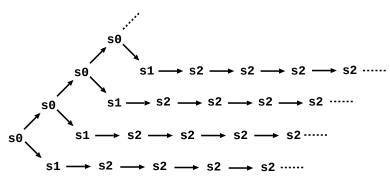
```
http://www.cs.cmu.edu/~modelcheck/smv.html   (original)
http://www.kenmcmil.com/smv.html             (Cadence extension by McMillan)
http://nusmv.irst.itc.it/                    (new implementation)
```

- Other temporal logics
  - Linear temporal logic (LTL): easier to use, more complicated to check
  - CTL*: combines CTL and LTL (also harder to check)
  - Industrial languages PSL and SVA designed to be 'engineer friendly'

Expressibility of CTL

- Consider the property
  "on every path there is a point after which **p** *is always true on that path"*
- Consider





- Property true, but cannot be expressed in CTL
  - would need something like AF P
  - where P is something like "property **p** *true from now on*"
  - but in CTL P must start with a path quantifier A or E
  - so cannot talk about current path, only about all or some paths
  - AF AG (`Atom` p) is false (consider path `s0s0s0`$\cdots$)

Linear Temporal Logic (LTL)

- CTL property is a predicate on a state in a tree: $P(\mathcal{R}, s)$
- LTL property is a predicate on a path: $P(\sigma)$
- Syntax of LTL well-formed formulae:

wff ::=	`Atom`(p)	(Atomic formula)
|	$\neg$*wff*	(Negation)
|	$wff_1 \vee wff_2$	(Disjunction)
|	**X***wff*	(successor)
|	**F***wff*	(sometimes)
|	**G***wff*	(always)
|	$[wff_1 \ \mathbf{U} \ wff_2]$	(Until)

- **Note:** no path quantifiers A or E

Semantics of LTL (shallow embedding)

- **Define** Tail m σ $=$ $\lambda n.\ \sigma(n{+}m)$
- **Define:**

$$
\begin{array}{ll}
\texttt{Atom}(p) & = \lambda\sigma.\ p(\sigma(0)) \\
\neg P & = \lambda\sigma.\ \neg(P\ \sigma) \\
P \vee Q & = \lambda\sigma.\ P\ \sigma\ \vee\ Q\ \sigma \\
\mathbf{X}P & = \lambda\sigma.\ P(\text{Tail } 1\ \sigma) \\
\mathbf{F}P & = \lambda\sigma.\ \exists m.\ P(\text{Tail } m\ \sigma) \\
\mathbf{G}P & = \lambda\sigma.\ \forall m.\ P(\text{Tail } m\ \sigma) \\
[P\ \mathbf{U}\ Q] & = \lambda\sigma.\ \exists i.\ Q(\text{Tail } i\ \sigma)\ \wedge\ \forall j.\ j < i\ \Rightarrow\ P(\text{Tail } j\ \sigma)
\end{array}
$$

- **Example:**

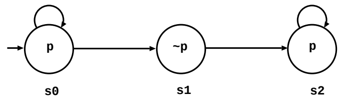
$$\mathbf{X}(\texttt{Atom}(p))(\sigma)\ =\ \texttt{Atom}(p)(\text{Tail } 1\ \sigma)\ =\ p(\text{Tail } 1\ \sigma\ 0)\ =\ p(\sigma(0{+}1)) = p(\sigma(1))$$

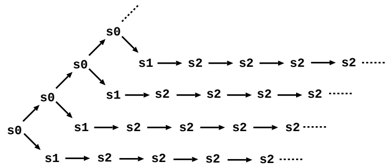
FG

- **FGP is true if there is a point after which P is always true**

$$
\begin{array}{l}
\mathbf{FG}P(\sigma) \\
\quad = \mathbf{F}(\mathbf{G}(P))(\sigma) \\
\quad = \exists m_1.\ (\mathbf{G}(P))(\text{Tail } m_1\ \sigma) \\
\quad = \exists m_1.\ \forall m_2.\ P(\text{Tail } m_2\ (\text{Tail } m_1\ \sigma)) \\
\quad = \exists m_1.\ \forall m_2.\ P(\text{Tail } (m_1{+}m_2)\ \sigma)
\end{array}
$$

- **Recall:**





- **LTL can express things that CTL can't express**

CTL can express things that LTL can't express

- **AG(EF P) says:**

  "from every state it is possible to get to a state for which P holds"

- **Can't say this in LTL (proof omitted)**
- **Consider disjunction:**

  "along every path there is a state from which P will hold forever
  or
  from every state it is possible to get to a state for which P holds"

- **Can't say this in either CTL or LTL! (proof omitted)**
- **CTL* combines CTL and LTL and can express this property**

CTL*

- **Two kinds of formulas: state formulas** (*swff*) **& path formulas** (*pwff*)
  - **state formulas are true of a state s in a tree $\mathcal{R}$** .... $\lambda(\mathcal{R}, s)$ **like CTL**
  - **path formulas are true of a path σ through a tree $\mathcal{R}$** .... $\lambda(\mathcal{R}, \sigma)$ **like LTL**
- **Defined mutually recursively**

$$
\begin{array}{lll}
\mathit{swff} ::= & \texttt{Atom}(p) & \text{(Atomic formula)} \\
\quad\mid & \neg\mathit{swff} & \text{(Negation)} \\
\quad\mid & \mathit{swff}_1 \vee \mathit{swff}_2 & \text{(Disjunction)} \\
\quad\mid & \mathbf{A}\mathit{pwff} & \text{(All paths)} \\
\quad\mid & \mathbf{E}\mathit{pwff} & \text{(Some paths)}
\end{array}
$$

$$
\begin{array}{lll}
\mathit{pwff} ::= & \texttt{PathForm}(\mathit{swff}) & \text{(Every state formula is a path formula)} \\
\quad\mid & \neg\mathit{pwff} & \text{(Negation)} \\
\quad\mid & \mathit{pwff}_1 \vee \mathit{pwff}_2 & \text{(Disjunction)} \\
\quad\mid & \mathbf{X}\mathit{pwff} & \text{(Successor)} \\
\quad\mid & \mathbf{F}\mathit{pwff} & \text{(Sometimes)} \\
\quad\mid & \mathbf{G}\mathit{pwff} & \text{(Always)} \\
\quad\mid & [\mathit{pwff}_1\ \mathbf{U}\ \mathit{pwff}_2] & \text{(Until)}
\end{array}
$$

- **CTL is CTL* restricted with X, F, G, [–U–] preceded by A or E**
- **LTL consists of CTL* formulas of form A*pwff*,**
  where the only state formulas in *pwff* are atomic
- **Selection of primitives above arbitrary: $\vee$, $\neg$, X, U, E enough**

CTL* semantics

- **Combining state semantics of CTL with path semantics of LTL:**

$$
\begin{array}{ll}
\texttt{Atom}(p) & = \lambda(\mathcal{R}, s).\ p(s) \\
\neg S & = \lambda(\mathcal{R}, s).\ \neg(S(\mathcal{R}, s)) \\
S_1 \vee S_2 & = \lambda(\mathcal{R}, s).\ S_1(\mathcal{R}, s)\ \vee\ S_2(\mathcal{R}, s) \\
\mathbf{A}P & = \lambda(\mathcal{R}, s).\ \forall \sigma.\ \texttt{Path}(\mathcal{R}, s)\sigma\ \Rightarrow\ P(\mathcal{R}, \sigma) \\
\mathbf{E}P & = \lambda(\mathcal{R}, s).\ \exists \sigma.\ \texttt{Path}(\mathcal{R}, s)\sigma\ \wedge\ P(\mathcal{R}, \sigma)
\end{array}
$$

$$
\begin{array}{ll}
\texttt{PathForm}(S) & = \lambda(\mathcal{R}, \sigma).\ S(\mathcal{R}, \sigma(0)) \\
\neg P & = \lambda(\mathcal{R}, \sigma).\ \neg(P(\mathcal{R}, \sigma)) \\
P_1 \vee P_2 & = \lambda(\mathcal{R}, \sigma).\ P_1(\mathcal{R}, \sigma)\ \vee\ P_2(\mathcal{R}, \sigma) \\
\mathbf{X}P & = \lambda(\mathcal{R}, \sigma).\ P(\mathcal{R}, \mathrm{Tail}\ 1\ \sigma) \\
\mathbf{F}P & = \lambda(\mathcal{R}, \sigma).\ \exists m.\ P(\mathcal{R}, \mathrm{Tail}\ m\ \sigma) \\
\mathbf{G}P & = \lambda(\mathcal{R}, \sigma).\ \forall m.\ P(\mathcal{R}, \mathrm{Tail}\ m\ \sigma) \\
[P_1\ \mathbf{U}\ P_2] & = \lambda(\mathcal{R}, \sigma).\ \exists i.\ P_2(\mathcal{R}, \mathrm{Tail}\ i\ \sigma)\ \wedge\ \forall j.\ j < i\ \Rightarrow\ P_1(\mathcal{R}, \mathrm{Tail}\ j\ \sigma)
\end{array}
$$

- **Note semantics of state and path formulas have different types**
  - $\lambda(\mathcal{R}, s)$ versus $\lambda(\mathcal{R}, \sigma)$
- **Semantics looks simpler if we assume $\mathcal{R}$ fixed**

Simplified CTL* semantics (textbook semantics)

- **Let $\texttt{Path}\ s\ \sigma$ abbreviate $\texttt{Path}(\mathcal{R}, s)\sigma$, then:**

$$
\begin{array}{ll}
\texttt{Atom}(p) & = \lambda s.\ p(s) \\
\neg S & = \lambda s.\ \neg(S\ s) \\
S_1 \vee S_2 & = \lambda s.\ S_1\ s\ \vee\ S_2\ s \\
\mathbf{A}P & = \lambda s.\ \forall \sigma.\ \texttt{Path}\ s\ \sigma\ \Rightarrow\ P\ \sigma \\
\mathbf{E}P & = \lambda s.\ \exists \sigma.\ \texttt{Path}\ s\ \sigma\ \wedge\ P\ \sigma
\end{array}
$$

$$
\begin{array}{ll}
\texttt{PathForm}(S) & = \lambda \sigma.\ S(p(0)) \\
\neg P & = \lambda \sigma.\ \neg(P\sigma) \\
P_1 \vee P_2 & = \lambda \sigma.\ P_1\ \sigma\ \vee\ P_2\ \sigma \\
\mathbf{X}P & = \lambda \sigma.\ P(\mathrm{Tail}\ 1\ \sigma) \\
\mathbf{F}P & = \lambda \sigma.\ \exists m.\ P(\mathrm{Tail}\ m\ \sigma) \\
\mathbf{G}P & = \lambda \sigma.\ \forall m.\ P(\mathrm{Tail}\ m\ \sigma) \\
[P_1\ \mathbf{U}\ P_2] & = \lambda \sigma.\ \exists i.\ P_2(\mathrm{Tail}\ i\ \sigma)\ \wedge\ \forall j.\ j < i\ \Rightarrow\ P_1(\mathrm{Tail}\ j\ \sigma)
\end{array}
$$

Fairness

- **May want to assume a component or the environment is 'fair'**
- **Example 1: fair arbiter**
  the arbiter doesn't ignore one of its requests forever
  - not every request need be granted
  - want to exclude infinite number of requests and no grant
- **Example 2: reliable channel**
  no message continuously transmitted but never received
  - not every message need be received
  - want to exclude an infinite number of sends and no receive
- **Want if P holds infinitely often along a path then so does Q**
- **In LTL is expressible as $\mathbf{G}(\mathbf{F}\ P)\ \Rightarrow\ \mathbf{G}(\mathbf{F}\ Q)$**
- **Can't say this in CTL**
  - why not – what's wrong with $\mathbf{AG}(\mathbf{AF}\ P)\ \Rightarrow\ \mathbf{AG}(\mathbf{AF}\ Q)$?
  - in CTL* expressible as $\mathbf{A}(\mathbf{G}(\mathbf{F}\ P)\ \Rightarrow\ \mathbf{G}(\mathbf{F}\ Q))$
  - fair CTL model checking is implemented in the model checking algorithm
  - fair LTL just needs a fairness assumption like $\mathbf{G}(\mathbf{F}\ P)\ \Rightarrow\ \cdots$
- **Fairness is a tricky and subtle subject**
  - several notions or fairness: 'weak fairness', 'strong fairness' etc
  - exist whole books on fairness

Propositional modal μ-calculus

- **Modal μ-calculus is an even more powerful property language**
- **Has fixed-point operators**
  - both maximal and minimal fixed points
  - model checking consists of calculating fixed points
  - many logics (e.g. CTL*) can be translated into μ-calculus
- **Strictly stronger than CTL***
  - expressibility in μ-calculus strictly increases as allowed nesting increases
  - need fixed point operators nested 2 deep for CTL*
- **The μ-calculus is very non-intuitive to use!**
  - intermediate code rather than a practical property language
  - nice meta-theory and algorithms, but terrible usability!

Interval Temporal Logic (ITL)

- ITL specifies properties of intervals
- An interval is a sequence of states with a beginning and an end
- Useful for talking about 'transactions'
- ITL specifies properties of finite intervals not infinte traces
- Has an executable subset called *Tempura* suitable for simulation
- Developed by Ben Moszkowski at Stanford then here at Cambridge
- Moszkowski is now at De Montford University

ITL (simplified and with expressions omitted)

- Syntax of ITL well-formed formulae:

$$
\begin{array}{lll}
\mathit{wff} ::= & \mathtt{Atom}(p) & \text{(Atomic formula)} \\
\mid & \mathbf{true} & \text{(Truth)} \\
\mid & \neg \mathit{wff} & \text{(Negation)} \\
\mid & \mathit{wff}_1 \lor \mathit{wff}_2 & \text{(Disjunction)} \\
\mid & \mathbf{skip} & \text{(interval with exactly two states)} \\
\mid & \mathit{wff}_1 \ ; \ \mathit{wff}_2 & \text{(Chop)} \\
\mid & \mathit{wff}* & \text{(Repeat)}
\end{array}
$$

- Semantics (properties are predicates on intervals):

$$
\begin{array}{ll}
\mathtt{Atom}(p) & = \lambda\langle s_0 \cdots s_n\rangle.\ p(s_0)\ \land\ n=0 \\
\mathbf{true} & = \lambda\langle s_0 \cdots s_n\rangle.\ T \\
\neg P & = \lambda\langle s_0 \cdots s_n\rangle.\ \neg(P\langle s_0 \cdots s_n\rangle) \\
P \lor Q & = \lambda\langle s_0 \cdots s_n\rangle.\ P\langle s_0 \cdots s_n\rangle\ \lor\ Q\langle s_0 \cdots s_n\rangle \\
\mathbf{skip} & = \lambda\langle s_0 \cdots s_n\rangle.\ n=1 \\
P\ ;\ Q & = \lambda\langle s_0 \cdots s_n\rangle.\ \exists k.\ k \le n\ \land\ P\langle s_0 \cdots s_k\rangle\ \land\ Q\langle s_k \cdots s_n\rangle \\
P* & = \lambda\langle s_0 \cdots s_n\rangle. \\
 & \quad \exists w_1\ \cdots\ w_l.\ \langle s_0 \cdots s_n\rangle\ = w_1 \cdots w_l\ \land\ P\ w_1\ \land\ \cdots\ \land\ P\ w_l
\end{array}
$$

Examples of ITL

Abbreviation	Meaning
$P_1 ; P_2$	P_1 holds then P_2 holds (overlapping state)
$P_1 ;\mathbf{skip}; P_2$	P_1 holds then P_2 holds (no overlapping state)
$\mathbf{skip} ; P$	P true on the next state
$\mathbf{true} ; P$	P sometimes true
$\neg\mathbf{true} ; \neg P$	P always true

Too many logics: CTL, LTL, CTL*, ITL, ...

- Large variety of separate logics
- Can be viewed as idioms in higher order logic
- Can model complete hardware systems in higher order logic
- Can model programming languages and logics in higher order logic
- Why not dump ad hoc languages and just work in logic?
  - specialized logics support specialized specification and verification methods
  - compact assertions developed for specific applications

Assertion-based verification (ABV)

- **Claimed that assertion based verification:**
  "*is likely to be the next revolution in hardware design verification*"
- **Basic idea:**
  - document designs with formal properties
  - check properties using both simulation (dynamic) and model checking (static)
- **Accellera organisation and IEEE are specifying languages**
- **Frequently used acronyms**

```
PSL:  Property Specification Language
OVL:  Open Verification Library (Verilog modules)
OVA:  Open Vera Language
SVA:  System Verilog Assertions
SVL:  System Verilog assertion Library (SVA version of OVL)
```

- **Problem: too many languages**
  - PSL from Accellera Formal Verification Technical Committee
  - OVA/SVA from Accellera SystemVerilog Assertion Committee
  - OVL from Accellera Open Verification Library Technical Commitee
  - all Accellera committees + some new IEEE committees!
- **PSL and OVA/SVA have been 'aligned'**
- **OVL is a checker library for dynamic property verification**
  - currently VHDL, Verilog and PSL versions
  - eventually PSL version golden and others derived .... maybe

IBM's *Sugar* and Accellera's PSL

- *Sugar 1* **is the property language of IBM's RuleBase model checker**
- *Sugar 1* **is CTL plus** *Sugar Extended Regular Expressions* **(SEREs)**
- **SEREs are ITL-like constructs**
- *Accellera* **ran a competition to select a 'standard' property language**
- **Finalists were IBM's** *Sugar 2* **and Motorola's** *CBV*
  - Intel/Synopsys ForSpec eliminated earlier
    (apparently industry politics involved)
- *Sugar 2* **is based on LTL rather than CTL**
  - has CTL constructs called "Optional Branching Extension" (OBE)
  - has clocking constructs for temporal abstraction
- **Accellera purged "Sugar" from it property language**
  - the word "Sugar" was too associated with IBM
  - language renamed to PSL
  - SEREs now *Sequential Extended Regular Expressions*
- **People lobby to make PSL more like ForSpec (align with SVA)**