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

Example CTL formulas

- EF(Started ∧ ¬Ready)
  It is possible to get to a state where Started holds but Ready does not hold

- AG(Req ⇒ AFAck)
  If a request Req occurs, then it will eventually be acknowledged by Ack

- AG(AF(DeviceEnabled))
  DeviceEnabled is always true somewhere along every path starting anywhere: i.e. DeviceEnabled holds infinitely often along every path

- AG(EF(Restart))
  From any state it is possible to get to a state for which Restart holds

Summary of CTL operators (primitive + defined)

- CTL formulas:
  - Atom(p) (Atomic formula - p: states → bool)
  - ¬P (Negation)
  - P ∧ Q (Conjunction)
  - P ∨ Q (Disjunction)
  - P ⇒ Q (Implication)
  - AXP (All successors)
  - EXP (Some successors)
  - AFP (Somewhere - along all paths)
  - EFp (Somewhere - along some path)
  - AGP (Everywhere - along all paths)
  - EGP (Everywhere - along some path)
  - A[p U q] (Until - along all paths)
  - E[p U q] (Until - along some path)
  - A[p W q] (Unless - along all paths)
  - E[p W q] (Unless - along some path)

- Say ‘P holds’ if P(R, s) for all initial states s

More CTL examples (1)

- AG(Req ⇒ A[Req U Ack])
  If a request Req occurs, then it continues to hold, until it is eventually acknowledged

- AG(Req ⇒ AX(A[¬Req U 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?

- AG(Req ⇒ (¬Ack ⇒ AX(A[Req U 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)

- AG[Enabled \implies AG[Start \implies A[\neg Waiting \cup Ack]]]

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

- AG[\neg Req \land \neg Req, \Rightarrow A[\neg Req \land \neg Req \cup (Start \land \neg Req)]]

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

- AG[Req \implies AX(Ack \implies AF \neg Req)]

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

Some abbreviations

- AX, P \equiv AX(AX(\cdots(AX(P)\cdots))) \quad \text{\(i\) instances of AX}

  P is true on all paths \(i\) units of time later

- ABF_{i,j} P \equiv

  AX(\exists P \lor AX(\exists \cdots AX(\exists P \lor AX(P)\cdots))) \quad \text{\(j-i\) instances of AX}

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

- AG[Req \implies AX[Ack \land ABF_{1,6}((Ack \land A[Wait \cup Reply])]]]

  One cycle after Req, Ack; should become true, and then Ack becomes true \(i\) to \(6\) cycles later and then eventually Reply becomes true, but until it does Wait holds from the time of Ack

- More abbreviations in the ‘Industry Standard’ language PSL

CTL model checking algorithm

- A model is a relation R

- A property is a CTL formula P

- Model checking: given CTL formula P compute \(\{s \mid P(R, s)\}\)

- P(R, s) true if and only if \(s_0 \in \{s \mid P(R, s)\}\)

- Assume set of states to be finite

  (infinite state model checking possible for some models)

- Already seen how to model check reachability

  AG(Atom p)(R, s) = \forall s'. Reach R (Eq s) s' \Rightarrow p(s')

  so can model check AG of atomic properties – compute:

  \(\{s' \mid Reach R (Eq s) s' \Rightarrow p(s')\}\),

  e.g. via BDD of

  Reach R (Eq s) s' \Rightarrow p(s')

Checking EF Atom(p)

- EF(Atom p)(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

  - \(\{s \mid EF(Atom p)(R, s)\}\) computed by generating:

    \(S_0 = \{s \mid (Atom p)(R, s)\} = \{\{s \mid p(s)\}\}

    S_{i+1} = S_i \cup \{s \mid \exists s'. R(s, s') \land s' \in S_i\}

  - EF(Atom p) is true in marked states and false in unmarked states

  - Algorithm similar for AF(Atom p):

    repeatedly mark all the states which have all successors marked

  - To check AF EF(Atom p):

    - apply EF algorithm

    - starting with resulting marking apply AF algorithm
Recall handshake example

- Part of a handshake circuit

- Transition relation:
  \[(q_0' = dreq) \land (dack' = dreq \land (q_0 \lor dack))\]

- Define RECEIVER by:
  \[\text{RECEIVER}((dreq,q_0,dack),(dreq',q_0',\text{dack}')) = (q_0' = dreq) \land (dack' = dreq \land (q_0 \lor dack))\]

- Primed variables \(\langle dreq',q_0',\text{dack}' \rangle\) represent 'next state'

- \(dreq\) unconstrained, hence non-determinism

Example: \(\text{EF}(dreq \land q_0 \land \text{dack})\)

- Define:
  \[P = \text{atom}(\lambda \text{b'}| \text{b} \land b_1 \land b_0)\]
  \[P(\text{RECEIVER}, b_0 b_1 b_2) = b_2 \land b_1 \land b_0\]

- Define:
  \[S_0 = \{b_0 b_1 b_2 | P(\text{RECEIVER}, b_0 b_1 b_2)\}\]
  \[S_{i+1} = S_i \cup \{s | \exists s', R(s,s') \land s' \in S_i\}\]

  \[S_2 = \{s | \exists s', R(s,s') \land s' \in S_2\}\]

  \[\text{EF}(dreq \land q_0 \land \text{dack})/\text{RECEIVER}, s\]

Checking \(\text{EF}(dreq \land q_0 \land \text{dack})\)

- Recall:
  \[S_0 = \{b_0 b_1 b_2 | P(\text{RECEIVER}, b_0 b_1 b_2)\}\]
  \[S_{i+1} = S_i \cup \{b_0 b_1 b_2 | \exists b'_0 b'_1 b'_2 (b'_1 = b_0) \land (b'_2 = b_2 \land (b_0 \lor b_1)) \land b'_2 b'_0 b'_1 \subseteq S_i\}\]

- Compute:
  \[S_1 = \{111\}\]
  \[S_1 = \{111\} \cup \{101,110\}\]
  \[= \{111,101,110\}\]
  \[S_2 = \{111,101,110\} \cup \{100\}\]
  \[= \{111,101,110,100\}\]
  \[S_3 = \{111,101,110,100\} \cup \{000,001,010,011\}\]
  \[= \{111,101,110,100,000,001,010,011\}\]
  \[S_i = S_1 (i > 3)\]

- Hence \(\forall s. \text{EF}(\text{atom}(\lambda (dreq,q_0,\text{dack}), dreq \land q_0 \land \text{dack})/\text{RECEIVER}, s)\)
Symbolic model checking

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

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://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 SYA 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

```
  p
  s0 → s1 → s2 → s2 → ... → s2 → s2 ...
```

- Property true, but cannot be expressed in CTL
  - would need something like $\text{AF} \neg p$
  - where $\neg p$ is something like "property $p$ true from now on"
  - but in CTL $\neg p$ must start with a path quantifier $A$ or $E$
  - so cannot talk about current path, only about all or some paths
  - $\text{AF AG} (\text{Atom} p)$ is false (consider path $\alpha_0 \alpha_0 \alpha_0 ...$

Linear Temporal Logic (LTL)

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

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

- Notice no path quantifiers $A$ or $E$
Semantics of LTL (shallow embedding)

- Define \( \text{Tail}_m \sigma = \lambda n. \sigma(n + m) \)
- Define:
  - \( \text{Atom}(p) = \lambda \sigma. p(\sigma(0)) \)
  - \( \neg P = \lambda \sigma. \neg P(\text{Tail}_1 \sigma) \)
  - \( P \lor Q = \lambda \sigma. P \lor Q \)
  - \( X P = \lambda \sigma. P(\text{Tail}_1 \sigma) \)
  - \( F \neg P = \lambda \sigma. \exists m. \neg P(\text{Tail}_m \sigma) \)
  - \( G P = \lambda \sigma. \forall m. P(\text{Tail}_m \sigma) \)
  - \( [P \lor Q] = \lambda \sigma. \exists i. Q(\text{Tail}_i \sigma) \land \forall j. j < i \Rightarrow P(\text{Tail}_j \sigma) \)

**Example:**
\[
X(\text{Atom}(p))(\sigma) = \text{Atom}(p)(\text{Tail}_1 \sigma) = p(\sigma(0 + 1)) = p(\sigma(1))
\]

FG

- \( \text{FG}P \) is true if there is a point after which \( P \) is always true
  \[
  \text{FG}P(\sigma) = F(G(P))(\sigma) = \exists m_1. (G(P))(\text{Tail}_{m_1} \sigma)
  \]

**Recall:**

- 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 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 (\( \text{swff} \)) & path formulas (\( \text{pwff} \))
  - state formulas are true of a state \( s \) in a tree \( R \ldots \ldots \lambda(R, s) \) like CTL
  - path formulas are true of a path \( \sigma \) through a tree \( R \ldots \ldots \lambda(R, \sigma) \) like LTL

- Defined mutually recursively
  - \( \text{swff} := \text{Atom}(p) \) (Atomic formula)
    \[
    | \neg \text{swff} \quad \text{Negation}
    | \text{swff}_1 \lor \text{swff}_2 \quad \text{Disjunction}
    | \text{Apwff} \quad \text{(All paths)}
    | \text{Epwff} \quad \text{(Some paths)}
    \]
  - \( \text{pwff} := \text{PathForm}(\text{swff}) \) (Every state formula is a path formula)
    \[
    | \neg \text{pwff} \quad \text{(Negation)}
    | \text{pwff}_1 \lor \text{pwff}_2 \quad \text{Disjunction}
    | \text{Xpwff} \quad \text{(Successor)}
    | \text{Fpwff} \quad \text{(Sometimes)}
    | \text{Gpwff} \quad \text{(Always)}
    | [\text{pwff}_1 \lor \text{pwff}_2] \quad \text{(Until)}
    \]

- CTL is CTL* restricted with X, F, G, \([-U-]\) preceded by A or E

- LTL consists of CTL* formulas of form \( \text{Apwff} \), where the only state formulas in \( \text{swff} \) are atomic

- Selection of primitives above arbitrary: \( \lor, \neg, X, U, E \) enough
CTL* semantics

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

\[ \text{Atom}(p) = \lambda(R, s). p(s) \]
\[ \neg S = \lambda(R, s). \neg S(R, s) \]
\[ S_1 \lor S_2 = \lambda(R, s). S_1(R, s) \lor S_2(R, s) \]
\[ AP = \lambda(R, s). \forall \sigma. \text{Path}(R, s) \sigma \Rightarrow P(R, \sigma) \]
\[ EP = \lambda(R, s). \exists \sigma. \text{Path}(R, s) \sigma \land P(R, \sigma) \]

**PathForm**

\[ \text{PathForm}(S) = \lambda(R, \sigma). S(R, \sigma(0)) \]
\[ \neg P = \lambda(R, \sigma). \neg P(R, \sigma) \]
\[ P_1 \lor P_2 = \lambda(R, \sigma). P_1(R, \sigma) \lor P_2(R, \sigma) \]
\[ XP = \lambda(R, \sigma). P(R, \text{Tail} i \sigma) \]
\[ GP = \lambda(R, \sigma). \exists m. P(R, \text{Tail} m \sigma) \]
\[ [P_1 \cup P_2] = \lambda(R, \sigma). \exists i. P_1(R, \text{Tail} i \sigma) \land \forall j. j < i \Rightarrow P_1(R, \text{Tail} j \sigma) \]

**Note semantics of state and path formulas have different types**

- \( \lambda(R, s) \) versus \( \lambda(R, \sigma) \)
- Semantics looks simpler if we assume \( R \) fixed

Simplified CTL* semantics (textbook semantics)

**Let** Path \( s \sigma \) abbreviate Path\((R, s)\sigma\), then:

\[ \text{Atom}(p) = \lambda s. p(s) \]
\[ \neg S = \lambda s. \neg S(s) \]
\[ S_1 \lor S_2 = \lambda s. S_1 s \lor S_2 s \]
\[ AP = \lambda s. \forall \sigma. \text{Path} s \sigma \Rightarrow P s \sigma \]
\[ EP = \lambda s. \exists \sigma. \text{Path} s \sigma \land P s \sigma \]

**PathForm**

\[ \text{PathForm}(S) = \lambda s. S(p(0)) \]
\[ \neg P = \lambda s. \neg P(s) \]
\[ P_1 \lor P_2 = \lambda s. P_1 s \lor P_2 s \]
\[ XP = \lambda s. P(Tail i s) \]
\[ GP = \lambda s. \exists m. P(Tail m s) \]
\[ [P_1 \cup P_2] = \lambda s. \exists i. P_1(Tail i s) \land \forall j. j < i \Rightarrow P_1(Tail j s) \]

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 \( G(F P) = G(F Q) \)
- Can’t say this in CTL
  - why not – what’s wrong with AG(AF P) \( \Rightarrow \) AG(AF Q)?
- in CTL* expressible as \( A(GF P) \Rightarrow GF Q) \)
- fair CTL model checking is implemented in the model checking algorithm
- fair LTL just needs a fairness assumption like \( GF P) \Rightarrow GF Q \)

**Propositional modal \( \mu \)-calculus**

- Modal \( \mu \)-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 \( \mu \)-calculus
- Strictly stronger than CTL*
  - expressibility in \( \mu \)-calculus strictly increases as allowed nesting increases
  - need fixed point operators nested 2 deep for CTL*
- The \( \mu \)-calculus is very non-intuitive to use!
  - intermediate code rather than a practical property language
  - nice meta-theory and algorithms, but terrible usability!

- Fairness is a tricky and subtle subject
  - several notions or fairness: ‘weak fairness’, ‘strong fairness’ etc
  - exist whole books on fairness
**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 infinite 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

**Syntax of ITL well-formed formulae:**

\[
\text{wff ::= } \text{Atom}(p) \quad (\text{Atomic formula}) \\
\quad \mid \text{true} \quad (\text{Truth}) \\
\quad \mid \neg \text{wff} \quad (\text{Negation}) \\
\quad \mid \text{wff}_1 \lor \text{wff}_2 \quad (\text{Disjunction}) \\
\quad \mid \text{skip} \quad (\text{interval with exactly two states}) \\
\quad \mid \text{wff}_1 ; \text{wff}_2 \quad (\text{Chop}) \\
\quad \mid \text{wff}^* \quad (\text{Repeat})
\]

**Semantics (properties are predicates on intervals):**

\[
\text{Atom}(p) = \lambda \langle s_0 \cdots s_n \rangle. p(s_0) \land n = 0
\]

\[
\text{true} = \lambda \langle s_0 \cdots s_n \rangle. T
\]

\[
\neg P = \lambda \langle s_0 \cdots s_n \rangle. \neg (P(s_0 \cdots s_n))
\]

\[
P \lor Q = \lambda \langle s_0 \cdots s_n \rangle. P(s_0 \cdots s_n) \lor Q(s_0 \cdots s_n)
\]

\[
\text{skip} = \lambda \langle s_0 \cdots s_n \rangle. n = 1
\]

\[
P ; Q = \lambda \langle s_0 \cdots s_n \rangle. \exists k : k \leq n \land P(s_0 \cdots s_k) \land Q(s_k \cdots s_n)
\]

\[
P^* = \lambda \langle s_0 \cdots s_n \rangle. \\
\exists w_1 \cdots w_l . \langle s_0 \cdots s_k \rangle = w_1 \cdots w_l \land P w_1 \land \cdots \land P w_l
\]

**Examples of ITL**

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1 ; P_2$</td>
<td>$P_1$ holds then $P_2$ holds (overlapping state)</td>
</tr>
<tr>
<td>$P_1 ; \text{skip} ; P_2$</td>
<td>$P_1$ holds then $P_2$ holds (no overlapping state)</td>
</tr>
<tr>
<td>skip ; $P$</td>
<td>$P$ true on the next state</td>
</tr>
<tr>
<td>true ; $P$</td>
<td>$P$ sometimes true</td>
</tr>
<tr>
<td>$\neg \text{true} ; \neg P$</td>
<td>$P$ always true</td>
</tr>
</tbody>
</table>

**Too many logics: CTL, ITL, 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 Committee
  - 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 checking 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)