Choice of Temporal Logic Specifications

Narayanan Sundaram
EE219C Lecture

CTL Vs LTL

The Final Showdown

Why should we choose one over the other?

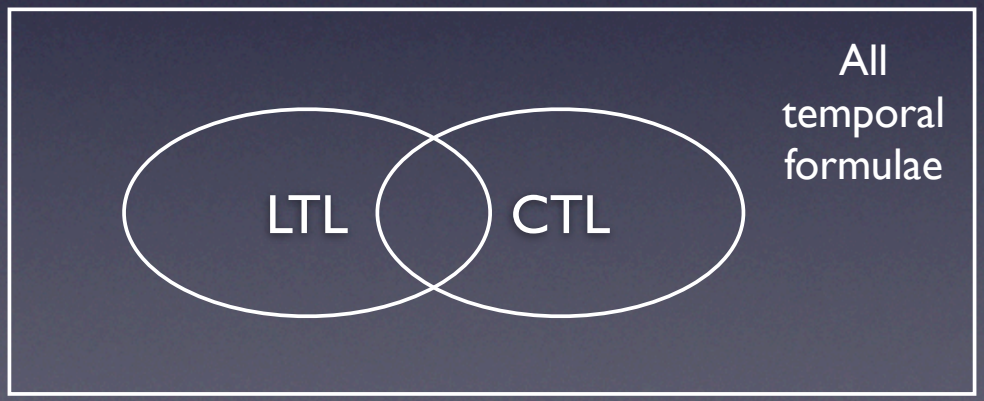
- Expressiveness
- Clarity/Intuitiveness
- Algorithmic Complexity for Verification
- Ease of analyzing error reports
- Compositionality

Expressiveness - CTL

- CTL can express formulae that LTL cannot
  - Try expressing $AG(p \rightarrow ((AX\ q) \vee (AX\ \neg q))$ in LTL (This formula is used in the context of database transactions)
  - How about AF AX p or AF AG p ?

Expressiveness - LTL

- LTL can express temporal formulae that CTL cannot !
  - Try expressing F G p in CTL (AF AG p is stronger and AF EG p is weaker)



Expressiveness

- CTL characterizes bisimulation i.e. two states in a transition system are bisimilar iff they satisfy the same CTL properties
- Bisimulation is a structural relation
- We need a way to specify behavioural properties

Verdict

Property	CTL	LTL	Tie/No Answer
Expressiveness			√
Clarity/ Intuitiveness			
Complexity			
Debugging			
Composinality			

Clarity/Intuitiveness

- Which is more intuitive - CTL or LTL ?
- Claims made for clarity on both sides
- Tightly linked with expressiveness
- Does more expressive mean more or less clear/intuitive?

Clarity/Intuitiveness

- Most properties are very simple like AG p
- Linear time is more intuitive than branching time for most people
- F X p and X F p mean the same thing
- AF AX p and AX AF p do not
- Do we need expressiveness or clarity ?

Clarity/Intuitiveness

- LTL uses language containment (Buchi automaton approach)
- CTL uses reachability analysis
- With LTL, both system and properties are FSMs
  - Does this mean that LTL is more intuitive ?

Verdict

Property	CTL	LTL	Tie/No Answer
Expressiveness			√
Clarity/ Intuitiveness		√	
Complexity			
Debugging			
Composinality			

Complexity Classes

Complexity

- For CTL, model checking algorithms run in O(nm) time (n is the size of transition system and m is the size of temporal formula)
- For LTL, model checking algorithms run in $n.2^{O(m)}$ time
- Is CTL better?
- Remember : m << n

Complexity Closed/Open systems

- CTL complexity bound is better than LTL only in closed systems
- For open systems, we get totally different results
  - For LTL, it is PSPACE Complete
  - For CTL, it is EXPTIME Complete
  - For CTL*, it is 2EXPTIME Complete

Complexity

- Are these comparisons valid?
- Should we only compare properties that are expressible in both CTL and LTL?
- The $2^{O(m)}$ in the LTL complexity comes from creating the Buchi automaton
- For LTL formulae that are expressible as $\forall$CTL, there is a Buchi automaton whose size is linear in the size of the LTL formula

Complexity

- Hierarchical systems
  - Both LTL and CTL model checking are PSPACE Complete
  - LTL : Polynomial in the size of the system
  - CTL : Exponential in the size of the system
- Size of system >> Size of formula
- Similar results for pushdown systems

Verdict

Property	CTL	LTL	Tie/No Answer
Expressiveness			√
Clarity/ Intuitiveness		√	
Complexity			√
Debugging			
Composinality			

Debugging from error traces

- Error trace analysis is needed for
  - Debugging the design
  - Semi-formal verification
- Don't CTL and LTL give similar error traces?

Error traces

- CTL is inherently branching time based
- Consider AF AX p is not satisfied - There is no linear trace that can disprove the property
- In contrast, all LTL property failures can produce a single linear trace

Error traces

- Closely related to intuitiveness of the specification
- Semiformal verification involves combining formal verification and simulation
- Harder to do this with CTL than LTL
- Current approaches to semiformal verification limit themselves to invariants to get around the problem - Too restrictive for wide usage

Verdict

Property	CTL	LTL	Tie/No Answer
Expressivenss			√
Clarity/ Intuitiveness		√	
Complexity			√
Debugging		√	
Compositionality			

Compositionality

- Compositional or modular verification used to tackle the space-explosion problem inherent in any formal verification method
- Use Assume-Guarantee paradigm

$$\left.\begin{array}{l} M_1 \models \psi_1 \\ M_2 \models \psi_2 \\ C(\psi_1, \psi_2, \psi) \end{array}\right\} M_1 \| M_2 \models \psi$$

Compositionality

$$\left.\begin{array}{l}\langle\varphi_1\rangle M_1\langle\psi_1\rangle \\ \langle\mathbf{true}\rangle M_1\langle\varphi_1\rangle \\ \langle\varphi_2\rangle M_2\langle\psi_2\rangle \\ \langle\mathbf{true}\rangle M_2\langle\varphi_2\rangle\end{array}\right\} \langle\mathbf{true}\rangle M_1 \| M_2 \langle\psi_1 \wedge \psi_2\rangle$$

- <φ>M<ψ> specifies that whenever M is a part of a system satisfying the formula φ, the system satisfies the formula ψ too.
- This branching modular model-checking problem for ∀CTL is PSPACE complete

Compositionality

- What is generally done in CTL model checking?
- People generally use (1) instead of (2)
- $M_2 \leqslant A_2$ is based on "intuition", which may be wrong
- $\leqslant$ is the simulation refinement relation

$$\left.\begin{array}{l} M_2 \preceq A_2 \\ M_1 || A_2 \models \varphi \end{array}\right\} M_1 || M_2 \models \varphi$$

(1)

$$\left.\begin{array}{l} M_1 \preceq A_1 \\ A_1 || M_2 \preceq A_2 \\ M_1 || A_2 \models \varphi \end{array}\right\} M_1 || M_2 \models \varphi$$

(2)

Compositionality - LTL

- Compositionality works easily with LTL!
- To prove $\langle\varphi\rangle M\langle\psi\rangle$ with LTL, we only need to prove $M \models \varphi \rightarrow \psi$
- To prove the linear-time properties of the parallel composition $M||E_1||E_2||... ||E_k$, it suffices to consider the linear-time properties of components $M, E_1, E_2, ... E_k$
- Possible because if $L(M) \subseteq L(P)$ and $L(E_i) \subseteq L(P)$, then

  $L(M) \cap L(E_i) \subseteq L(P)$

Verdict

Property	CTL	LTL	Tie/No Answer
Expressiveness			√
Clarity/ Intuitiveness		√	
Complexity			√
Debugging		√	
Compositionality		√	

Final Verdict

Property	CTL	LTL	Tie/No Answer
Expressiveness			√
Clarity/ Intuitiveness		√	
Complexity			√
Debugging		√	
Compositionality		√	

LTL declared as winner

LTL - Other advantages

- Abstraction can be mapped to language containment which LTL can handle
- To verify if design P_1 is a refinement of P_2, we have to just check $L(P_1) \subseteq L(P_2)$
- BMC fits naturally within a linear time framework as we only search for a counter-example trace of bounded length

Is LTL sufficient ?

- It is proven that LTL cannot express certain ω-regular expressions
- LTL is inadequate to express all assumptions about the environment in modular verification
- What is the "ultimate" temporal property specification language?
- ETL is an extension of LTL with temporal connectives that correspond to ω-automata

More Proposals

- Use past connectives - not necessary but can be convenient when referring to program locations where some modifications were made rather than just the external behaviour
- "In order to perform compositional specification and verification, it is *convenient* to use the past operators but *necessary* to have the full power of ETL" - Pnueli

Some Libraries & Tools in use

- Cadence SMV is CTL based (It has a linear time model checker built on top of a CTL model checker)
- FTL is a linear temporal logic with limited form of past connectives and with the full expressive power of ω-regular expressions
  - Used in ForSpec, Intel's formal verification language

Some more Libraries & Tools in use

- Open Verification Library (OVL)
- Process Specification Language (PSL)
- System Verilog Assertions (SVA)

Integrating Verification

- Designers use VHDL/Verilog for hardware designs
- Programmers use C/C++/Java etc
- Verification engines use FSMs with temporal property specifications
- How to make them talk to each other?

OVL

- The OVL library of assertion checkers is intended to be used by design, integration, and verification engineers to check for good/bad behavior in simulation, emulation and formal verification
- OVL is a Verification methodology, which can find bugs (even in mature designs)
- OVL is a Library of predefined assertions, currently available in Verilog, SVA and PSL

Types of OVL Assertions

Combinatorial

Combinatorial Assertions

- assert_proposition, assert_never_unknown_async

Single-Cycle

Single-cycle Assertions

- assert_always, assert_implication, assert_range, …

2-Cycles

Sequential over 2 cycles

- assert_always_on_edge, assert_decrement, …

n-Cycles

Sequential over num_cks cycles

- assert_change, assert_cycle_sequence, assert_next, …

Event-bound

Sequential between two events

- assert_win_change, assert_win_unchange, assert_window

OVL Assertions-Examples

TYPE	NAME	PORTS	DESCRIPTION
single cycle	assert_always	(clk, reset_n, test_expr)	test_expr must always hold
2 cycles	assert_always_on_edge	(clk, reset_n, sampling_event, test_expr)	test_expr is true immediately following the specified edge (edge_type: 0=no-edge, 1=pos, 2=neg, 3=any)
n cycles	assert_change	(clk, reset_n, start_event, test_expr)	test_expr must change within num_cks of start_event (action_on_new_start: 0=ignore, 1=restart, 2=error)

OVL

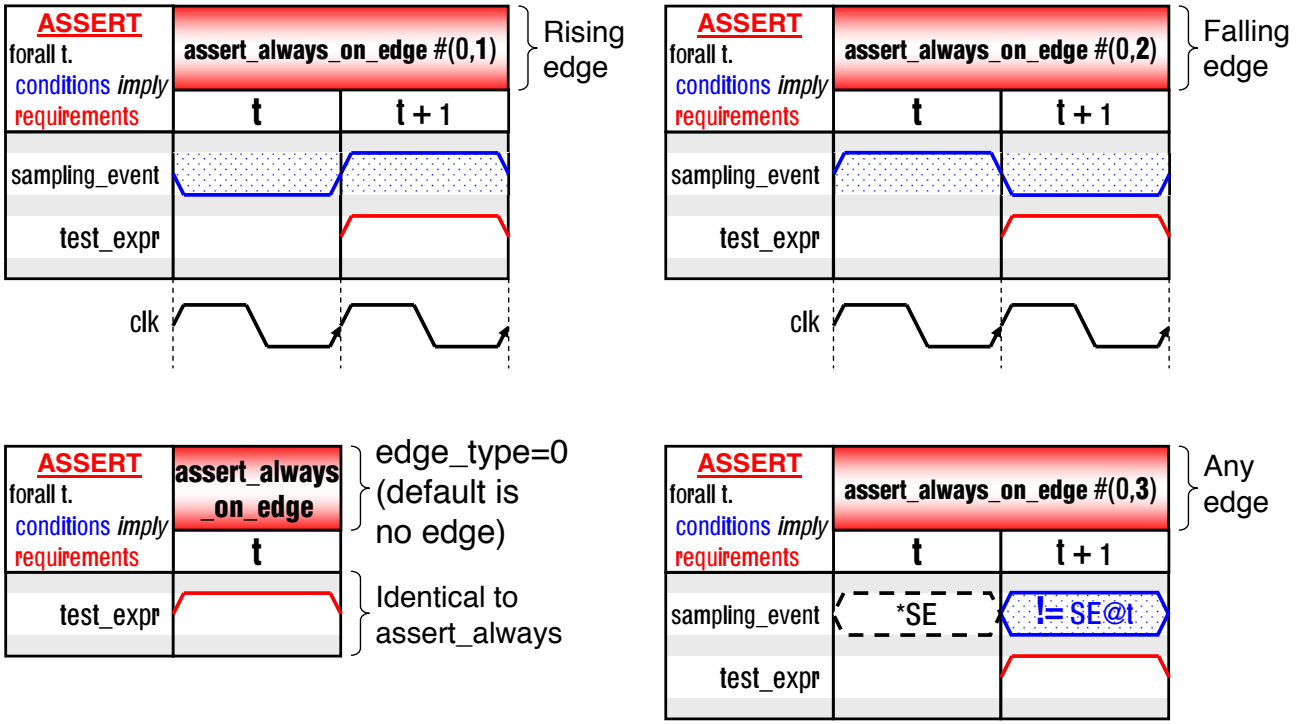
- OVL Assertions are used for property verification as well as constraint specification(environment modeling)
- OVL is just a layer for specifying properties
- The verification tool has to understand these assertions and then translate them into temporal formula of choice

OVL Timing Diagram - Example

assert_always_on_edge

```
#(severity_level, edge_type, property_type, msg, coverage_level)
u1 (clk, reset_n, sampling_event, test_expr)
```

`test_expr` is true immediately following the edge specified by the `edge_type` parameter — 2-Cycles



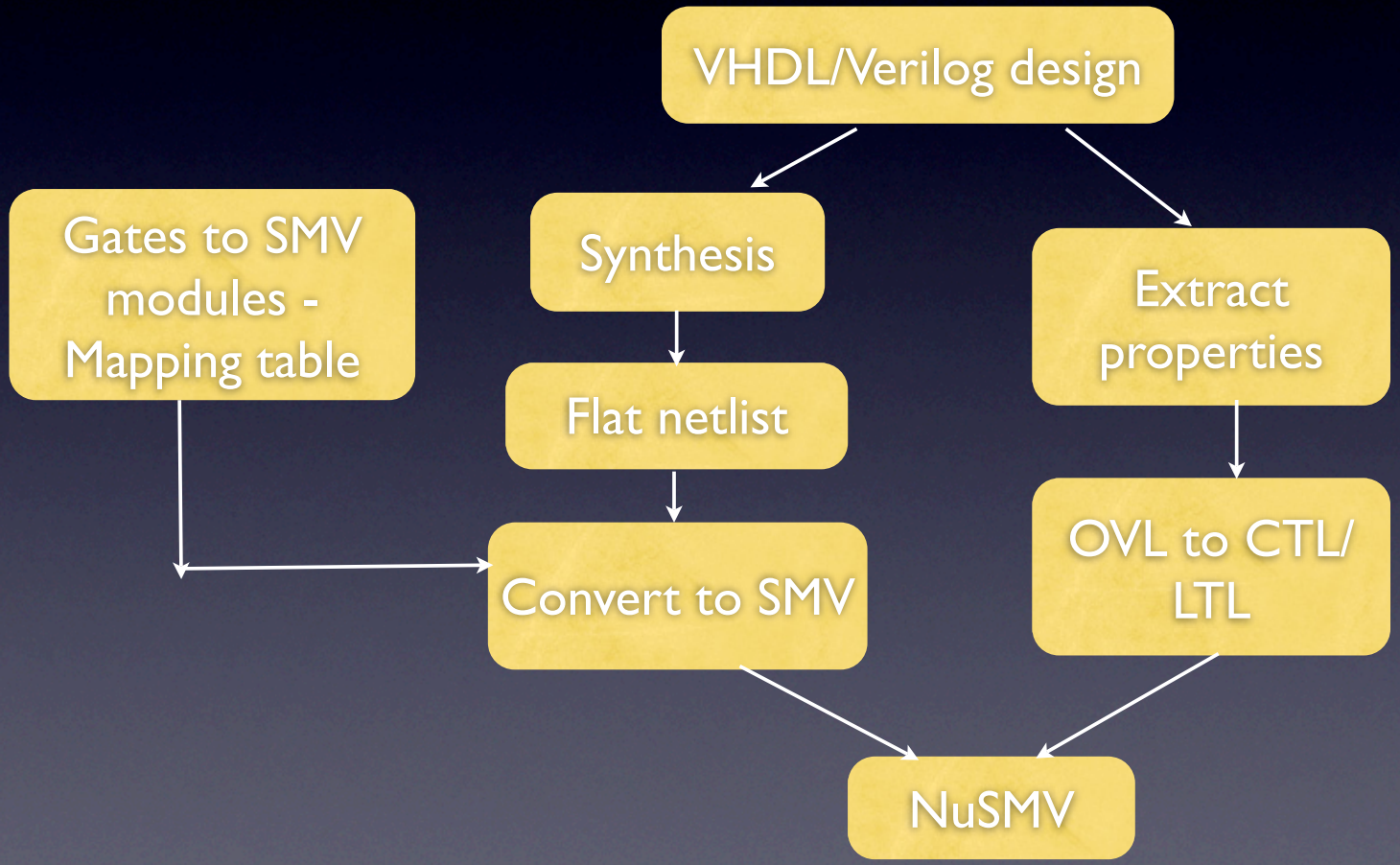
Using CTL/LTL based verification

- There are a number of issues to be solved before we can directly translate OVL to CTL/LTL
  - Presence of multiple clocks
  - Presence of positive and negative edge triggered logic
  - Support for BMC (for assertions like assert_change specifying num_cks)

A simple case study

- Methodology to use NuSMV with VHDL/ Verilog designs
- Restricted to designs with one global clock (logic uses only one edge of the clock)
- Uses synthesis tools along with verification engines
- Properties specified in OVL

Tool flow


VHDL/Verilog design
Gates to SMV modules - Mapping table
Synthesis
Flat netlist
Extract properties
OVL to CTL/ LTL
Convert to SMV
NuSMV

Conclusion

- LTL is better than CTL for specifying temporal properties of FSMs
- Many different libraries in use for specifying properties and constraints
- Designers can use these with minimal effort

References

- Moshe Y. Vardi, *Branching vs Linear time : Final Showdown,* Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, 2001, pp. 1 - 22

- http://www.accellera.org/activities/ovl/