Branching vs. Linear Time: Final Showdown

Moshe Y. Vardi *

Rice University

*Acknowledging Orna Kupferman and the Intel FV team
Final Verification Today

Verification as debugging: Failure of verification identifies bugs.

- Both specifications and programs attempt to formalize informal requirements.
- Verification contrasts two independent formalizations.
- Failure of verification reveals inconsistency between the two formalizations.

Model checking: uncommonly effective debugging tool

- a systematic exploration of the design state space
- good at catching difficult “corner cases”
Designs are Labeled Graphs

**Key Idea:** Designs can be represented as transition systems (finite-state machines)

**Transition System:** $M = (W, I, R, F, \pi)$

- $W$: states
- $I \subseteq W$: initial states
- $R \subseteq W \times W$: transition relation
- $F \subseteq W$: fair states
- $\pi : W \rightarrow \text{Powerset}(Prop)$: Observation function

**Fairness:** An assumption of “reasonableness” – restrict attention to computations that visit $F$ infinitely often, e.g., “the channel will be up infinitely often.”
Specifications

Linear-Time Specifications: properties of computations.

Examples:

- “No two processes can be in the critical section at the same time.” – safety
- “Every request is eventually granted.” – liveness
- “Every continuous request is eventually granted.”
  – liveness
- “Every repeated request is eventually granted.” – liveness
Linear Temporal Logic

**Linear Temporal logic (LTL):** logic of temporal sequences

*Main feature:* time is implicit

- *next* $\varphi$: $\varphi$ holds in the next state.
- *eventually* $\varphi$: $\varphi$ holds eventually
- *always* $\varphi$: $\varphi$ holds from now on
- $\varphi$ *until* $\psi$: $\varphi$ holds until $\psi$ holds.
Examples

• always not (CS₁ and CS₂): mutual exclusion (safety)

• always (Request implies eventually Grant): liveness

• always (Request implies Request until Grant): liveness

• always eventually Request implies eventually Grant: liveness
Linear vs. Branching

- **Linear time**: a system generates a set of computations

- **Specs**: describe computations

- **Branching time**: a program generates a computation tree

- **Specs**: describe computation trees
Program Equivalence

- $P_1$:

- $P_2$:

- Linear Time: $P_1 \equiv P_2$

- Branching Time: $P_1 \not\equiv P_2$
Temporal Logics

*Specs:* Every request is eventually granted

- **Linear** (*LTL*): always (Request implies eventually Grant)

- **Branching** (*CTL*): ∀ always (Request implies ∀ eventually Grant)
LTL vs. CTL: The Long Debate

- **Pnueli**: 1977
- **Lamport**: “‘Sometimes’ is sometimes ‘Not Never’”, 1980
- **Emerson and Clarke**: 1981
- **Ben-Ari, Pnueli, and Manna**: 1983
- **Pnueli**: 1985
- **Emerson and Lei**: “‘Branching-time logic strikes back’”, 1985
- **Emerson and Halpern**: “‘Sometimes’ and ‘Not Never’ Revisited”, 1986

**Conclusion**: Philosophically, a draw.
LTL vs. CTL: Expressiveness

Caveat: Linear and branching logics are incomparable.

- **LTL**: eventually always \( P \) – in every computation \( P \) is ultimately true.

- **CTL**: \( (\forall \text{ eventually } \forall \text{ always } P) \) – \( P \) will stabilize at true within a bounded amount of time.

General Assessment:

- Interesting CTL-LTL: “small”

- Interesting LTL-CTL: “large”
LTL vs. CTL: Complexity

Model-Checking Problem: Does $T$ satisfy $\varphi$?

$|T| = n$, $|\varphi| = m$

Time Complexity:

- **CTL**: $O(nm)$ [CES’86]
- **LTL**: $O(n2^m)$ (PSPACE-complete) [LP’86,SC’85]

Conclusions:

- Low complexity in $|T|$  
- **CTL** exponentially easier than **LTL**
Pragmatics

**Folk Wisdom:** *CTL* is less expressive than *LTL*, but *CTL* is superior to *LTL* computationally.

**Model Checking in practice:** *CTL* usage dominates

- *CTL*: SMV, VIS, RuleBase, CheckOff, Motorola
- *Linear Time*: Cadence’s SMV, FormalCheck, SPIN, Intel

**Note:** Linear Time ≠ LTL!
CTL vs. LTL: A Fresh Perspective

- Expressiveness
- Computational Complexity
- Compositionality
- Pragmatics
Expressiveness

IBM’s Experience:

- IBM J. of Research and Development: *Formal Verification Made Easy*, 1997
  
  “We found only simple CTL equations to be intuitively comprehensible; nontrivial CTL equations are hard to understand and prone to error.”

  
  “CTL is difficult to use for most users and requires a new way of thinking about hardware.”

Facts:

- *Sugar, RuleBase*’s spec language, tries to hide away CTL

- In partice, users write “linear” CTL formulas.
Example

- **LTL:**
  - next eventually P
  - eventually next P

  Both formulas assert that P holds in the *strict* future.

- **CTL:**
  - $\forall$ next $\forall$ eventually P
  - $\forall$ eventually $\forall$ next P

  Are these formulas equivalent? What do they say? How do they relate to the LTL formulas?
Algorithmic Foundations

Basic Graph-Theoretic Problems:

● Reachability: Is there a finite path from $I$ to $F$?

● Fair Reachability: Is there an infinite path from $I$ that goes through $F$ infinitely often.

Note: These paths may correspond to error traces, e.g., deadlock and livelock.
CTL Model Checking

Basic Algorithm:

- Iterated reachability analysis (i.e., reachability and fair reachability)

- Simple recursion on structure of formulas, e.g., $\forall$ always $\exists$ eventually $P$ involves a reachability computation followed by a fair-reachability computation.

- Computational complexity is linear in size of design and size of spec.
Automata on Infinite Words

Büchi Automaton: \( A = (\Sigma, S, S_0, \rho, F) \)

- **Alphabet**: \( \Sigma \)
- **States**: \( S \)
- **Initial states**: \( S_0 \subseteq S \)
- **Transition relation**: \( \rho \subseteq S \times \Sigma \times S \)
- **Accepting states**: \( F \subseteq S \)

**Input word**: \( a_0, a_1, \ldots \)

**Run**: \( s_0, s_1, \ldots \)

- \( s_0 \in S_0 \)
- \( (s_i, a_i, s_{i+1}) \in \rho \) for \( i \geq 0 \)

**Acceptance**: \( F \) visited infinitely often
Temporal Logic vs. Automata

**Paradigm:** Compile high-level logical specifications into low-level finite-state language

**The Compilation Theorem:** [V.-Wolper]

Given an LTL formula $\varphi$, one can construct an automaton $A_\varphi$ such that a computation $\sigma$ satisfies $\varphi$ if and only if $\sigma$ is accepted by $A_\varphi$. Furthermore, the size of $A_\varphi$ is at most exponential in the length of $\varphi$.

**Example:**

- always eventually P:

- eventually always P
LTL Model Checking

The following are equivalent:

- \( M \) satisfies \( \varphi \)
- all computations in \( L(M) \) satisfy \( \varphi \)
- \( L(M) \subseteq L(A_{\varphi}) \)
- \( L(M \parallel A_{\neg \varphi}) = \emptyset \)

**Bottom Line:** To check that \( M \) satisfies \( \varphi \), compose \( M \) with \( A_{\neg \varphi} \) and check whether the composite system has a reachable (fair) path. Verification reduces to *reachability* or *fair reachability*.

**Intuition:** \( A_{\neg \varphi} \) is a “watchdog” for “bad” behaviors. A reachable (fair) path means a bad behavior.
Computational Complexity

**Worst case:** linear in the size of the design space and exponential in the size of the specification.

**Real life:** Specification is given in the form of a list of properties $\varphi_1, \ldots, \varphi_n$. It suffices to check that $M$ satisfies $\varphi_i$ for $1 \leq i \leq n$.

**Moral:** There is life after exponential explosion.

**The real problem:** too many design states – symbolic methods needed
CTL vs. LTL: Comparison

- **Invalid Comparison**: worst case of an inexpressive logic against worst case of an expressive logic

- **Valid Comparison**: competitive analysis – compare performance of CTL and LTL model checkers on formulas that are in both logics
  
  - always eventually P
  - $\forall$ always $\forall$ eventually P

**Empirical Claim**: On formulas in $\text{LTL} \cap \text{CTL}$, CTL and LTL model checkers behave similarly, and if they don’t, you can make them (see work by Bloem-Ravi-Somenzi in CAV’99 and by Maidl in FOCS’00).
Compositional Verification

State Explosion:

- $T = T_1 \parallel \ldots \parallel T_k$

- $|T| = |T_1| \cdot \ldots \cdot |T_k|$

\[ \begin{array}{ll}
P_1 \text{ satisfies } \psi_1 \\
P_2 \text{ satisfies } \psi_2 \\
C(\psi, \psi_1, \psi_2)
\end{array} \quad \begin{array}{l}
P_1 \parallel P_2 \text{ satisfies } \psi
\end{array} \]

- $P_1 \parallel P_2$: composition of $P_1$ and $P_2$

- $C(\psi, \psi_1, \psi_2)$: logical condition relating $\psi$, $\psi_1$, and $\psi_2$

Advantage: apply model checking only to the underlying modules, which have smaller state spaces.
Assume-Guarantee Verification

$M$ guarantees $\psi$ assuming $\varphi$ – $\langle \varphi \rangle M \langle \psi \rangle$: for an arbitrary $M'$, if $M \parallel M' \models \varphi$, then $M \parallel M' \models \psi$

\[
\begin{align*}
\langle \text{true} \rangle M_1 \langle \varphi_1 \rangle \\
\langle \text{true} \rangle M_2 \langle \varphi_2 \rangle \\
\langle \varphi_2 \rangle M_1 \langle \psi_1 \rangle \\
\langle \varphi_1 \rangle M_2 \langle \psi_2 \rangle
\end{align*}
\]

\[
\langle \text{true} \rangle M_1 \parallel M_2 \langle \psi_1 \land \psi_2 \rangle
\]

**Fact:** Checking $\langle \varphi \rangle M \langle \psi \rangle$ is exponential in $\varphi$ for both $CTL$ and $LTL$ [KV’95]
It Gets Worse!

**CTL is too weak:**

- **Crucial:** Assumptions have to be strong enough to ensure guarantee; **LTL** assumptions may be needed for a **CTL** guarantee.

- **But:** The combination of a **CTL** guarantee and an **LTL** assumption involves a *doubly exponential* cost in computational complexity.

**In practice**

- **CTL-based** model checkers do not support compositional reasoning

- Verifiers engage in unsafe reasoning when using **CTL-based** model checkers because assumptions are *always* needed.

Ken McMillan: “In compositional reasoning use **LTL**” *(Cadence’s SMV uses linear time).*
Pragmatics

The linear-time view has numerous other advantages:

- **Refinement:** \( L(T_{imp}) \subseteq L(T_{spec}) \) – linear view
- **Abstraction:** \( L(T_{conc}) \subseteq L(T_{abst}) \) – linear view
- **Dynamic validation:** only linear view available
- **Counterexamples:** validators want traces
- **Bounded Model Checking:** Search linear counterexamples of predetermined size size.
What about Concurrency Theory?

**But:** CTL characterizes bisimulation!

**So what?**

- **Bisimulation is about structure**
  \[ \forall \text{ next } \forall \text{ eventually } P \text{ vs. } \forall \text{ eventually } \forall \text{ next } P \]

- **Model checking is about behavior**
  \[ \text{next eventually } P \text{ vs. } \text{eventually next } P \]

- **Difference between** \( ab + ac \) **and** \( a(b + c) \) **become clear in a state-based model, in which deadlock is modeled explicitly**
Is LTL The Answer?

**Question**: “Ok, ok. You made your point. Can we finish the talk and go with *LTL* then?”

**Answer**: “Not so fast. Let us reconsider compositional reasoning.”
Compositional Reasoning Revisited

**Crucial Points:**

- *Assume-guarantee* reasoning is the *prevalent* way of reasoning about complicated systems – you *always* need assumptions.

- When trying to check that “$M$ guarantees $\psi$ assuming $\varphi$”, you can weaken $\psi$, but you have to make $\varphi$ as strong as needed.

**Corollary 1:** Your spec language for *assumptions* needs to be as expressive as your hardware modeling language.

**Crucial Point:**

- Your *assume-guarantee* reasoning is not *sound*, unless you guarantee your assumptions – danger of *false positives*.

**Corollary 2:** Your spec language needs to be as expressive as your hardware modeling language.

**Fact:** *LTL* is too weak – cannot express finite-state machines.
Beyond Naive Hardware Modeling

**Assumptions:** abstracted hardware

- Replace gorry detail by nondeterminism

- Eliminate possible runs by using fairness

**Note:** Nondeterministic FSMs with fairness conditions are Büchi automata, which express $\omega$-regularity (more expressive than LTL).

**Question:** Can we make Büchi automata into a spec language?
What Is Logic?

Features of Logic:

- Closure under *Boolean connectives*: if $\varphi$ and $\psi$ are formulas, then $\varphi \land \psi$, $\varphi \rightarrow \psi$ are formulas.

- Closure under *substitution*: atomic propositions can be replaced by formulas; if always $p$ and eventually $q$ are formulas, then always eventually $q$ is a formula.
Extended Temporal Logic

ETL:

- Start with Büchi automata where the labels are atomic propositions
- Close under Boolean connectives (compositionality)
- Close under substitutions (re-usability)

Note: Closure under Boolean connectives and substitutions is not necessary for expressiveness. FormalCheck does not have it.

Example:
ETL: Pros and Cons

Advantages:

- Expressive enough for assume-guarantee reasoning

  Pnueli, 1986: “In order to perform compositional specification and verification, it is necessary to have the full power of ETL.”

- Formalism (FSMs) is very familiar to hardware designers

- Worst-case complexity same as LTL.

Disadvantages:

- Nesting of machines is conceptually difficult

- No experimental validation (yet)

- Complementation is known to be difficult

Bottom Line: More research needed
Other Formalisms

- **μ-calculus:**
  - One temporal connective (next) plus fixpoint operators
  - Unreadable: always eventually P

\[(gfp X)(lfp Y)(X \land next(P \lor Y))\]

- **QPTL:**
  - LTL plus propositional quantifiers
  - Example:

\[(\exists X)(X \land always(X \leftrightarrow next\neg X) \land always(X \rightarrow P))\]

- **Complexity:** nonelementary (unbounded stack of exponentials)!
A Pragmatic Proposal

*Competing demands on real languages:*

- **Expressiveness:** supports compositional reasoning
- **Usability:** can be used by verification engineers
- **Closure:** supports specification libraries
- **Implementability:** feasible implementation
- **History:** consistency with prior experience of users
FTL: ForSpec Temporal Logic

*ForSpec*: Intel’s new formal specification language

**key features:**

- linear-time logic, with fully $\omega$-regularity

- rich set of operations of Boolean and arithmetical operations

- time windows ($P$ until $[10, 15] Q$)

- regular events

\[
\text{always}((\text{req}, (\neg\text{ack})^*, \text{ack}) \text{ triggers} \\
(\text{true}^+, \text{grant}, (\neg\text{rel})^*, \text{rel}))
\]

- universal propositional quantification

- hardware-oriented features (*multiple clocks* and *resets*)
Did We Waste 20 Years on CTL?

Absolutely not!

- Usefulness of model checking demonstrated

- Symbolic reachability and fair reachability algorithms

- CTL model checkers as back-end for linear-time model checkers (*Cadence’s SMV* and *Intel’s ForSpec*)

- CTL is useful in checking correct modeling, e.g., $\forall$ always $\exists$ true says that there is a fair path from every state.

- Branching time is appropriate in game-theoretic settings, e.g., AI planning and controller synthesis.
Conclusions

• In spite of 20 years of research, this issue has not been resolved yet

• $CTL$ is clearly not adequate as a spec language

• $LTL$ is better, but has weaknesses

• $FTL$ is a strong industrial contender

My bottom line:

• Let’s close the linear-time vs. branching time debate: linear time won!

• Let’s re-open the linear-time vs. linear-time debate (e.g., $FTL$ vs. $FormalCheck$ vs. $ITL$).

• Let’s develop linear-time model checking technology.