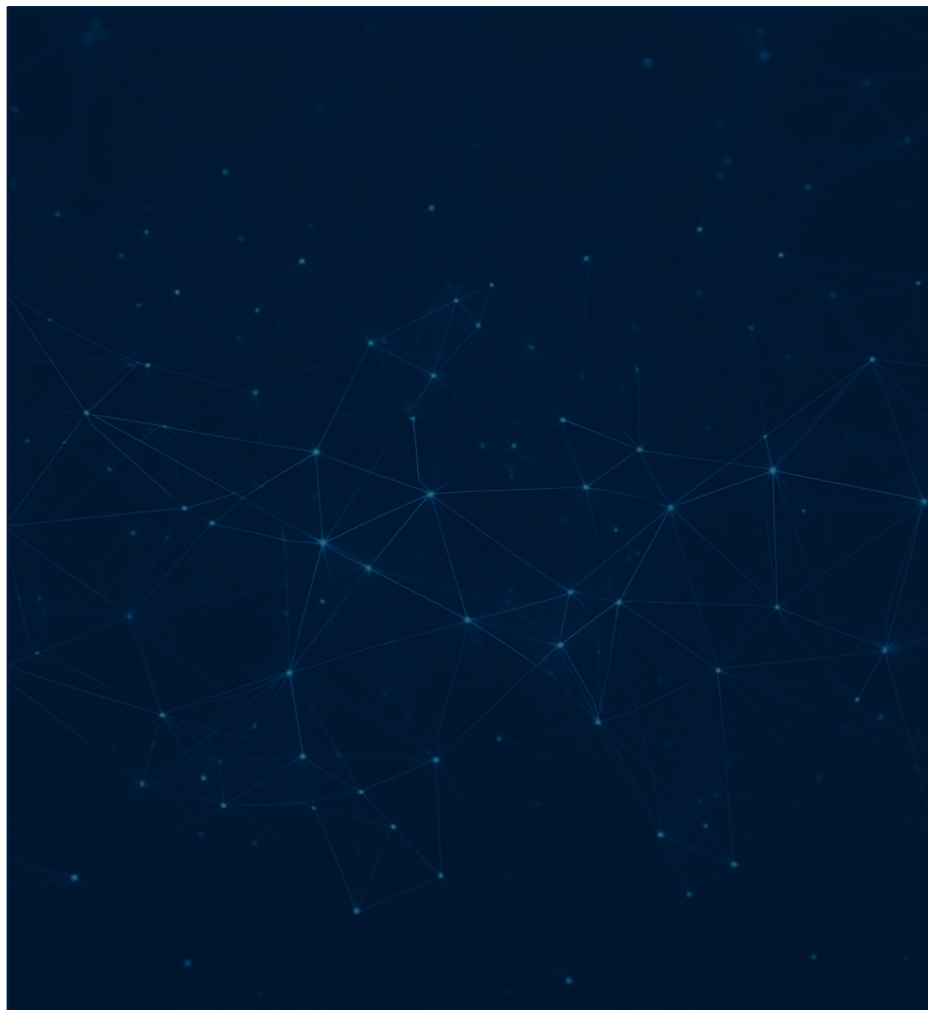


# Developing a BDD-Based CTL Model Checker

*CS 6840 Formal System Design*

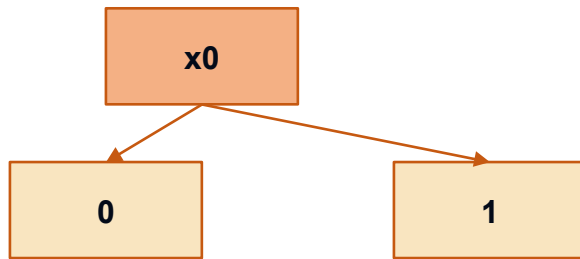
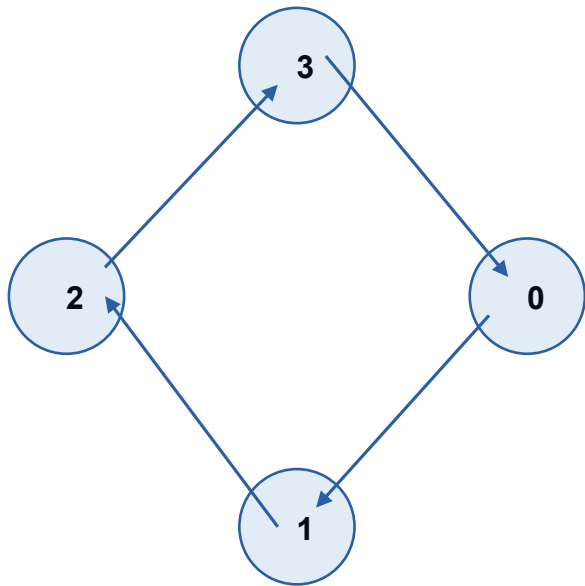
Kevin Bell  
August 12, 2025



# Outline

- Motivation & Problem
- CTL & Temporal Operators
- Implementation & Design
- Benchmarks & Validation
- Lessons, Future Work & Conclusion

# Motivation & Problem



- **CTL checks temporal properties on transition systems**
- Explicit enumeration enumerates all reachable states, leading to state-space explosion
- BDDs encode state sets symbolically, compacting many states into a succinct graph  
*but require careful variable ordering to remain efficient*

# CTL & Temporal Operators

## Existential

### **EX $\phi$**

Exists next: some successor satisfies  $\phi$

### **EF $\phi$**

Exists eventually: some path eventually reaches  $\phi$

### **EG $\phi$**

Exists globally: some path stays in  $\phi$  forever

### **E[ $\phi$ U $\psi$ ]**

Exists until: along some path  $\phi$  holds until  $\psi$  holds

## Universal

### **AX $\phi$**

For all next: every successor satisfies  $\phi$

### **AF $\phi$**

For all eventually: along all paths  $\phi$  eventually holds

### **AG $\phi$**

For all globally: along all paths  $\phi$  holds globally

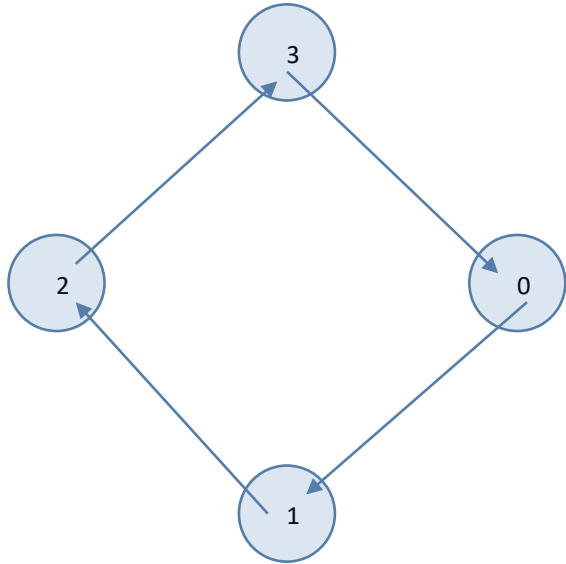
### **A[ $\phi$ U $\psi$ ]**

For all until: on every path  $\phi$  holds until  $\psi$  holds



EF p: a path exists where p eventually holds

# Transition System Representation



Ring topology example

Current Bits	Next Bits
s0	s0_next
s1	s1_next

## Labeling

Each state has a set of propositions (e.g.,  $p$ ,  $q$ ) that hold in that state.

## Transition relation

Encoded as a single BDD over current and next bits.

# Symbolic CTL Model Checker

- **Parse formulas with Lark grammar**
- Represent boolean connectives as BDD operations ( $\wedge$ ,  $\vee$ ,  $\neg$ )
- Use predecessor function for EX/AX
- Compute least/greatest fixpoints for EF/AF and EG/AG
- Evaluate until operators using iterative fixpoints

```
def eval(node):
    kind = node[0]
    if kind == 'atom': return ap_bdd(node[1])
    if kind == 'not': return ~eval(node[1])
    if kind == 'and': return eval(node[1]) &
eval(node[2])
    if kind == 'or': return eval(node[1]) |
eval(node[2])
    if kind == 'ex': return pre(eval(node[1]))
    if kind == 'ax': return ~pre(~eval(node[1]))
    if kind == 'ef': return least_fix(lambda Y:
eval(node[1]) | pre(Y))
    if kind == 'af': return least_fix(lambda Y:
eval(node[1]) | ~pre(~Y))
    if kind == 'eg': return greatest_fix(lambda
Y: eval(node[1]) & pre(Y))
    if kind == 'ag': return greatest_fix(lambda
Y: eval(node[1]) & ~pre(~Y))
    if kind == 'eu': return least_fix(lambda Y:
eval( $\psi$ ) | (eval( $\phi$ ) & pre(Y)))
```

# Explicit CTL Model Checker

- **ExplicitTransitionSystem** stores adjacency sets
- Predecessor function computes all predecessors via set comprehension
- Boolean connectives operate on Python sets
- Least/greatest fixpoints iterate until the set of states stabilizes
- Returns states satisfying a formula; initial states  $\subseteq$  result implies satisfaction

```
def eval(node):
    kind = node[0]
    if kind == 'atom': return {s for s in S if
ap in L(s)}
    if kind == 'not': return S - eval(node[1])
    if kind == 'and': return eval(node[1]) &
eval(node[2])
    if kind == 'or': return eval(node[1]) |
eval(node[2])
    if kind == 'ex': return pre(eval(node[1]))
    if kind == 'ax': return S - pre(S -
eval(node[1]))
    if kind == 'ef': return least_fix(lambda Y:
eval(node[1]) | pre(Y))
    if kind == 'af': return least_fix(lambda Y:
eval(node[1]) | (S - pre(S - Y)))
```

# Example Usage & Validation

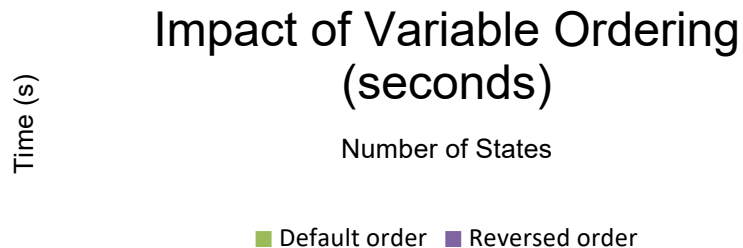
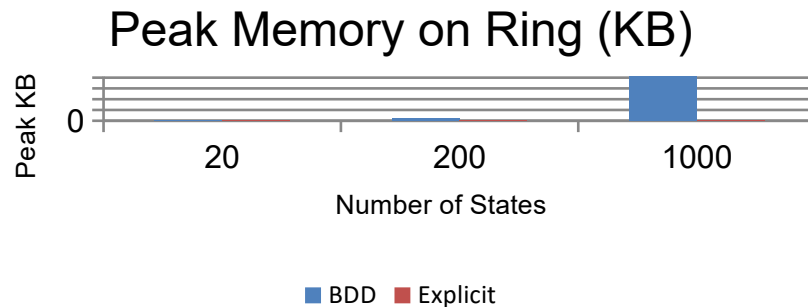
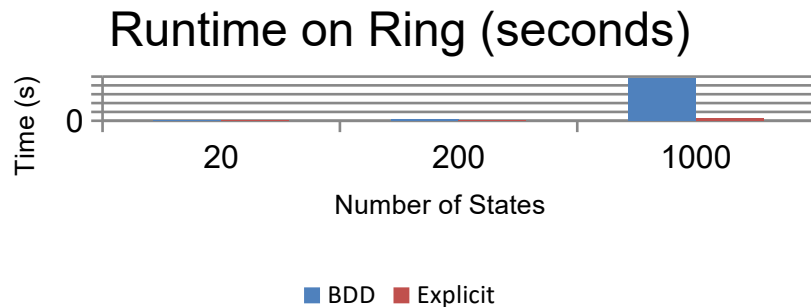
**A minimal script builds small systems and queries both backends.**

The results agree across all formulas, serving as a quick sanity check.

Formula	BDD	Explicit
EF p	True	True
AG p	False	False
AF p	True	True
EG q	False	False
EX p	True	True
AX p	True	True
E[q U p]	True	True
A[q U p]	False	False



# Benchmark Results



**Explicit checker handles larger rings significantly faster and uses far less memory.**

Reversing BDD variable order yields noticeable speedups on chains.

# Testing & Verification

**16 unit tests ensure correctness across both backends.**

Each test builds a small system and checks whether a CTL formula holds starting from the initial state.

Test Name	Formula	Result
test_ef_p	EF p	True
test_ag_p_false	AG p	False
test_af_p_true	AF p	True
test_eg_q_false	EG q	False
test_eu_q_until_p_true	E[q U p]	True
test_au_q_until_p_false	A[q U p]	False
test_ex_p_true	EX p	True
test_ax_q_false	AX q	False

# Lessons Learned & Future Work



## **Variable ordering dramatically affects BDD sizes and performance**

Symbolic methods excel when state spaces are large but regular

Python's dd library and Lark parser streamline prototyping



## **Add fairness constraints and support additional logics**

Experiment with dynamic variable reordering strategies

Scale to larger industrial benchmarks and compare with other tools

# Conclusion

**We delivered a complete CTL model checker with both symbolic and explicit backends.**

Our benchmarks reveal that while symbolic methods can compress large state spaces, explicit enumeration often outperforms them on small examples.

Rigorous testing and documentation ensure correctness and reproducibility.

*This project lays a foundation for further exploration in formal verification and model checking.*

**Thank you!**