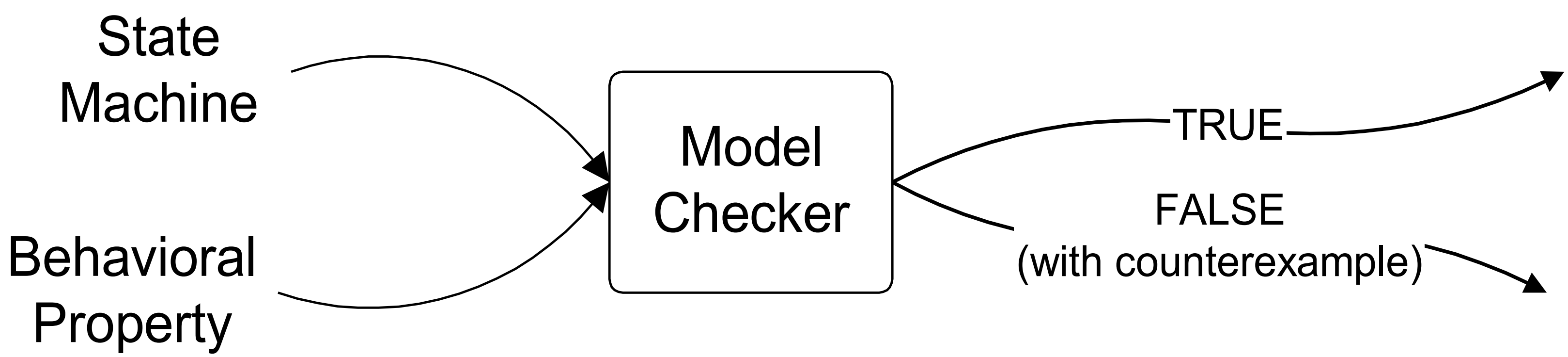
Symbolic Model Checking for Large Software Specifications

William Chan
Richard Anderson
Paul Beame
Steve Burns
Francesmary Modugno
David Jones (Boeing)
David Notkin
Jon D. Reese
William Warner (Boeing)

Motivation: circa 1998-2000

- How to increase confidence in correctness of safety-critical software?
- Existing techniques are limited to some degree
  - Inspection
  - Syntactic check
  - Simulation/testing
  - Theorem proving
- Symbolic model checking successful for industrial hardware
  - Effective also for software?
  - Many people's conjecture: No

Temporal-Logic Model Checking [Clarke & Emerson 81]



- Some properties expressible in temporal logics
  - Error states not reached (invariant)
    - Ex: AG ¬Err ⇐ Today's focus
  - Eventually ack for each request (liveness)
    - AG (Req → AF Ack)
  - Always possible to restart machine (possibility)
    - AG EF Restart

Two Approaches to Model Checking

- Explicit
  - Conventional state-space search: depth-first, breadth-first, etc.
  - Needs substantial manual abstraction and state reduction
- Symbolic
  - Can search huge state spaces (e.g. 10^{20})
  - Practical for many industrial hardware circuits
  - Provably bad for certain arithmetic operations.
  - Not believed to work well for software

Software Experts Said

- "The time and space complexity of the symbolic approach is affected…by the regularity of specification. Software requirements specifications lack this necessary regular structure…" [Heimdahl & Leveson 96]

And say…

- “[Symbolic model checking] works well for hardware designs with regular logical structures…However, it is less likely to achieve similar reductions in software specifications whose logical structures are less regular.” [Cheung & Kramer 99]

Model Checking Co-Inventor Says

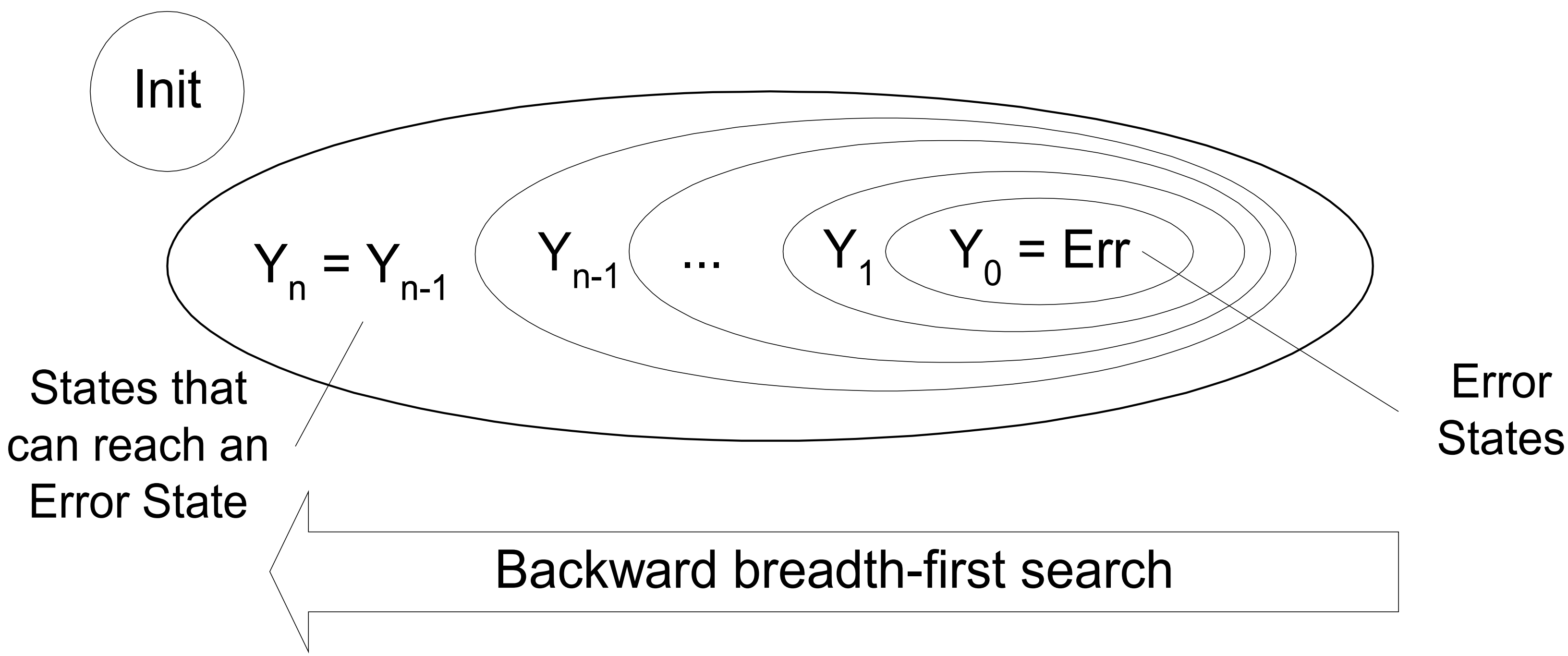
- “…[symbolic model checkers] are often able to exploit the regularity…in many hardware designs. Because software typically lacks this regularity, [symbolic] model checking seems much less helpful for software verification.” [Emerson 97]

Contributions

- Case Studies: successfully analyzed state-machine specifications of
  - TCAS II (aircraft collision avoidance system) [FSE 96, TSE 98]
  - Electrical power distribution (EPD) system on Boeing 777 [ICSE 99, TSE 00]
- Optimizations: obtained orders-of-magnitude speedup [ISSTA 98, ICSE 99, TSE 00]
  - Developed intuitions about efficiency
  - Enabled difficult analyses
- Extension: handle complicated arithmetic
  - Combine with a constraint-satisfaction engine [CAV 97]

Invariant Checking as Set Manipulations

- Compute $Y_{i+1} = Pre\ (Y_i) \cup Y_i$
- Check if $Y_n \cap Init = \emptyset$



Explicit vs. Implicit (Symbolic Sets)

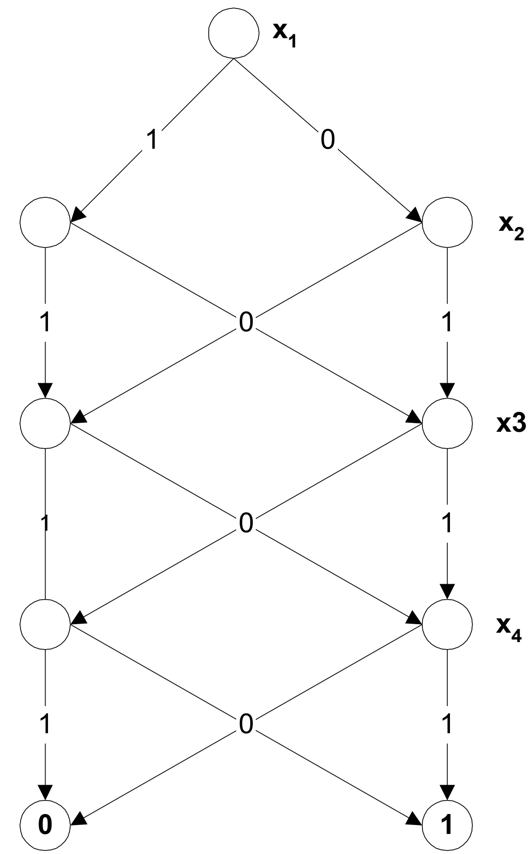
- All even numbers between 0 and 127
  - Explicit representation
    - 0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68, 70, 72, 74, 76, 78, 80, 82, 84, 86, 88, 90, 92, 94, 96, 98, 100, 102, 104, 106, 108, 110, 112, 114, 116, 118, 120, 122, 124, 126.
  - Implicit (symbolic) representation
    - $\neg x_0$ (x_0: least significant bit)
- Need efficient Boolean-function representation

Symbolic Model Checking [Burch et al. 90, Coudert et al. 89]

- Define Boolean state variables X
  - e.g., define x_{n-1}, x_{n-2}, …, x_0 for an n-bit integer.
- A state set becomes a Boolean function S(X)
  - e.g., x_0 for the set of n-bit even integers.
- Set operations ($\cap$,$\cup$)become Boolean operations ($\wedge$,$\vee$)
- Transition relation: R(X,X).
- Compute predecessors also using Boolean operations
  - $Pre(S) = \exists X'. S(X') \wedge R(X,X')$

Binary Decision Diagrams (BDDs) [Bryant 86]

- DAGs, evaluated like binary decision trees.
- Efficiency depends on BDD size
  - Usually small; some large hardware circuits can be handled
  - Some well-known limitations
    - e.g., exponential size for a > bc
  - Few theoretical results known
  - Performance unpredictable



Odd Parity

Symbolic Model Checking Ineffective for Software?

	Hardware	Software
Data	Simple	Complex
States	Finite	Infinite
Concurrency	Synchronous	Asynchronous
Strategy	Symbolic search	Abstraction and explicit search

This common view may be true for software like multi-threaded programs, but…

Consider Safety-Critical Software

- Most costly bugs in specification
- Use analyzable formal specification
  - State-machine specifications
  - Intuitive to domain experts like aircraft engineers
  - Statecharts [Harel 87], RSML [Leveson et al. 94], SCR [Parnas et al.], etc.

Model-Check the Spec!

	Hardware	Spec	Multi-threaded Code
Data	Simple	Simple (except arithmetic)	Often complex
States	Finite	Finite (except arithmetic)	Possibly infinite
Concurrency	Synchronous	Synchronous	Asynchronous

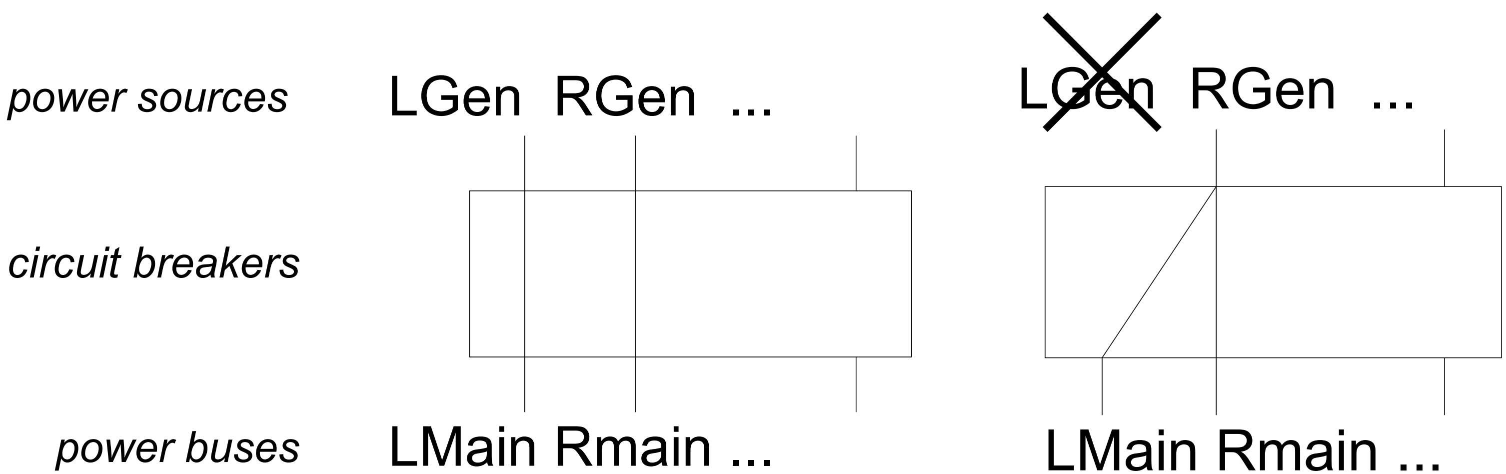
- Symbolic model checking good for such specs?
- Develop more intuitions about efficiency? Optimize analyses?
- How to handle arithmetic?

Case Study 1: TCAS II

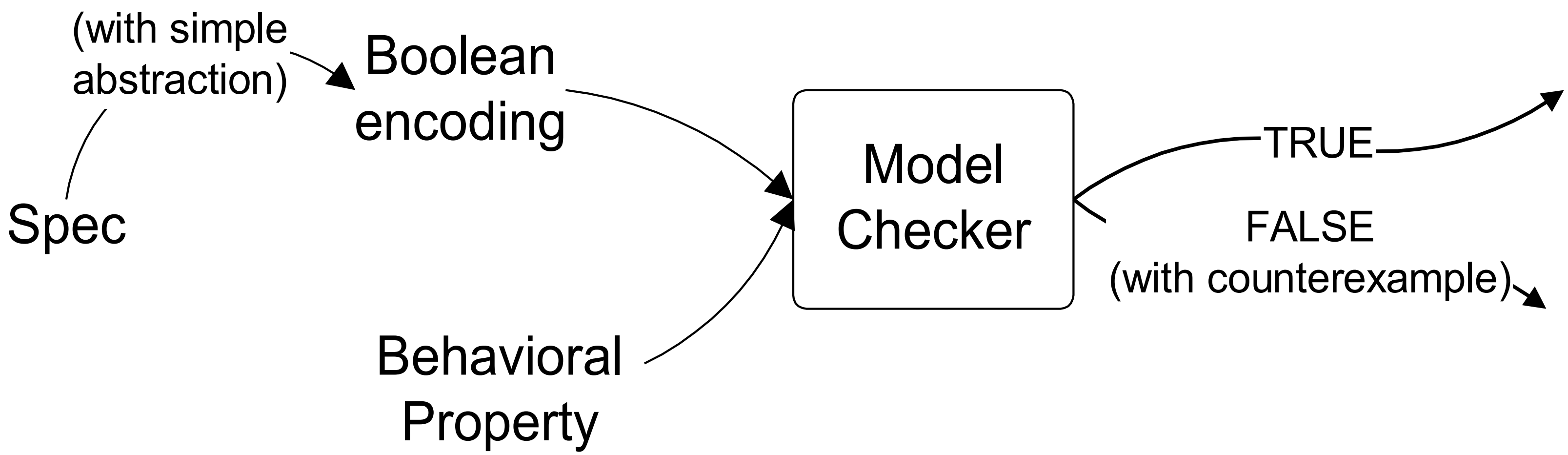
- Traffic Alert and Collision Avoidance System
  - Reduce mid-air collisions
    - Warns pilots of traffic
    - Issues resolution advisories
  - Required on most commercial aircraft
  - “One of the most complex systems on commercial aircraft.”
- 400-page specification reverse-engineered from pseudo-code
- Written in RSML by Leveson et al., based on statecharts

Case Study 2: EPD System

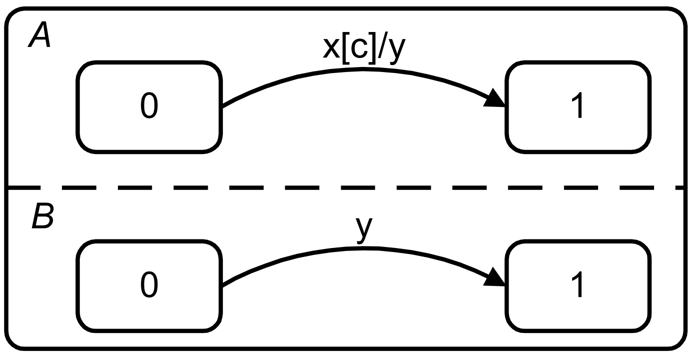
- Electrical Power Distribution system used on Boeing 777
- Distribute power from sources to buses via circuit breakers
  - Tolerate failures in power sources and circuit breakers
- Prototype specification in statecharts
- Analysis joint with Jones and Warner of Boeing



Model Check the Specifications



Translation to SMV




```
VAR
   A: {0,1};
   x: boolean;
   y: boolean;
ASSIGN
   init (A):= 0;
   next (A):= case
     A=0 & x & c : 1;
     1 : A;
     esac;

   ...
```

Analyses and Results

- Used and modified SMV [McMillan 93]

	TCAS II	EPD System
State space	230 bits, 10^{60} states	90 bits, 10^{27} states
Prior verification	inspection, static analysis	simulation
Problems we found	inconsistent outputs, safety violations, etc.	violations of fault tolerance

- Optimizations crucial for successful model checking

Some Formulas Checked

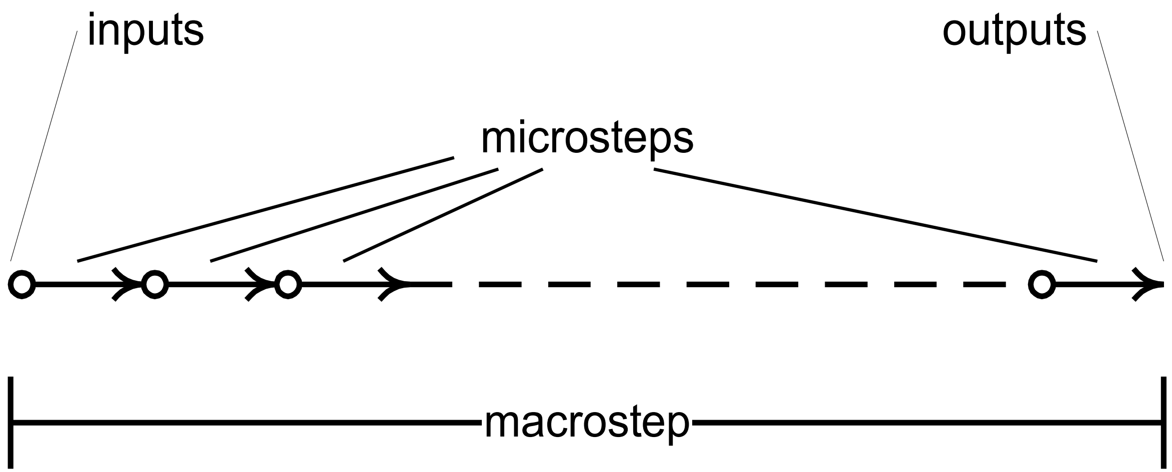
- TCAS II
  - Descent inhibition
    - $AG\ (Alt < 1000 \rightarrow \neg Descend)$
  - Output agreement
    - $AG\ \neg(GoalRate \geq 0 \wedge Descend)$
- EPD system
  - $AG\ (NoFailures \rightarrow (LMain \wedge RMain \wedge LBackup \wedge RBackup))$
  - $AG\ (AtMostOneFailure \rightarrow (LMain \wedge RMain))$
  - $AG\ (AtMostTwoFailures \rightarrow (LBackup \vee RBackup))$

A Counterexample Found

- A single failure can cause a bus to lose power
  1. Power-up sequence; normal operation
  2. A circuit breaker fails
  3. Other circuit breakers reconfigured to maintain power
  4. User changes some inputs
  5. The first circuit breaker recovers
  6. User turns off a generator
  7. A bus loses power

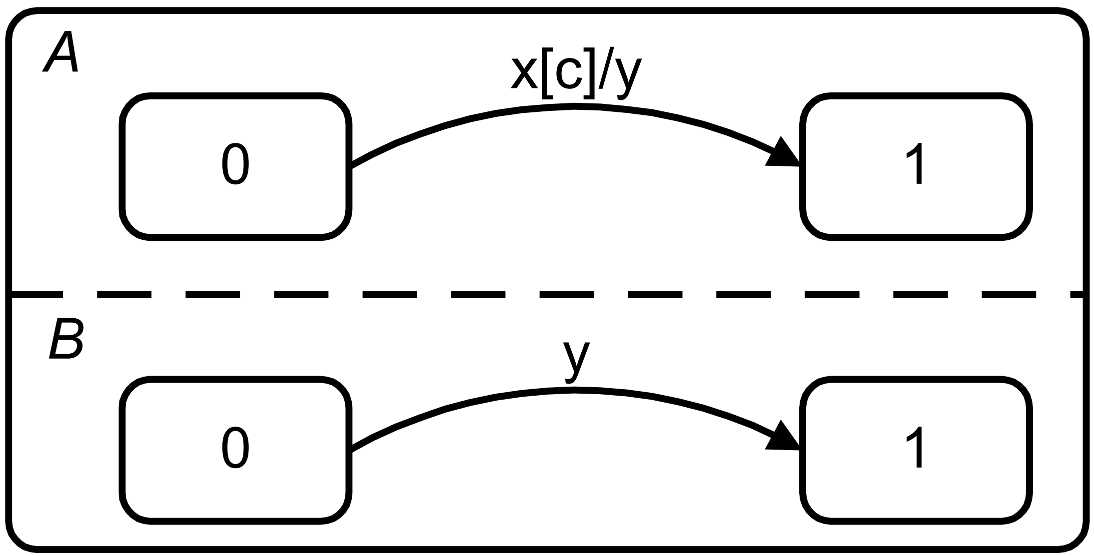
This error does not exist in onboard system

Environmental Model



- Synchrony hypothesis
  - No new inputs within macrostep
  - Macrostep encoded as a sequence of transitions
  - Statecharts, Esterel [Berry & Gonthier 92], Lustre [Halbwachs et al. 92], etc.

Synchronization in Statecharts



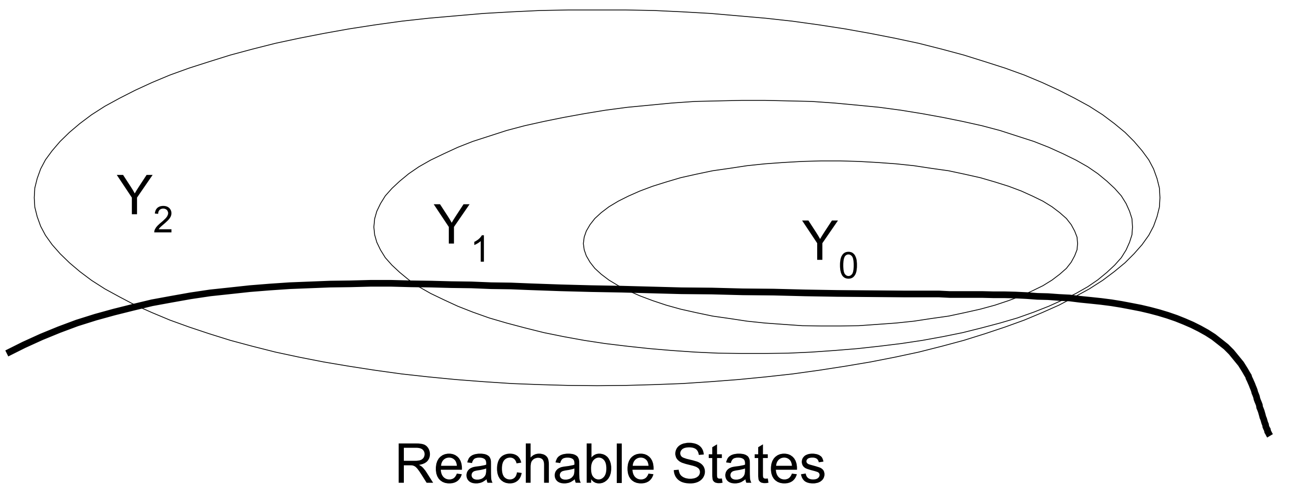
- Event-driven
- Label: trigger[guard]/action

Forward vs. Backward Search

- Generally unclear which is better
- Forward search
  - Often good for low-level hardware.
  - But always bad for us; large BDDs
- Focus on backward search

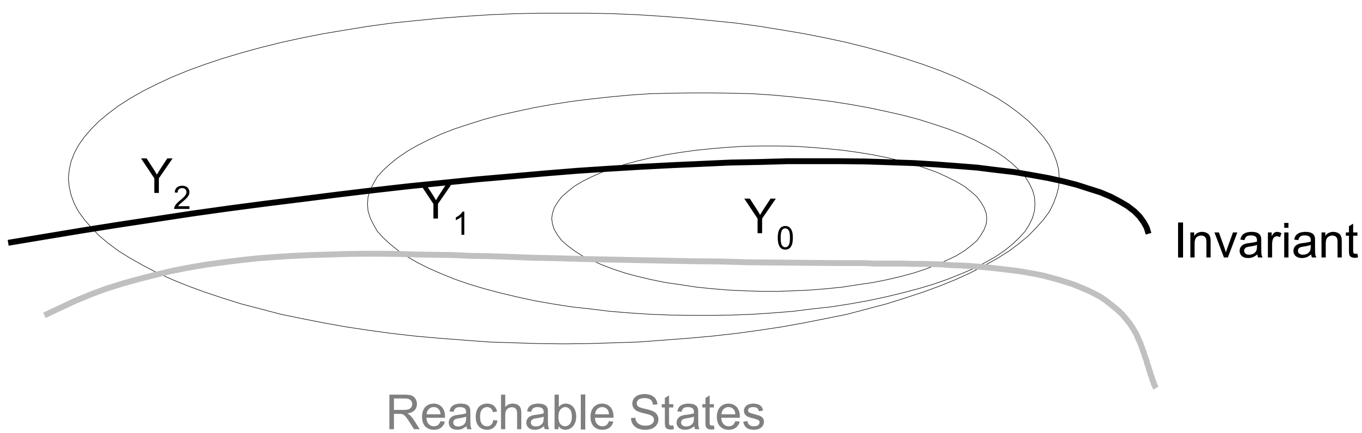
A Disadvantage of Backward Search

- Visiting unreachable states

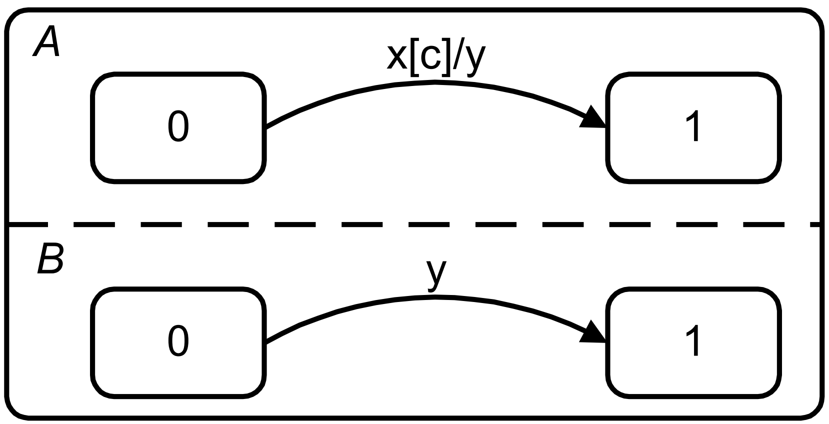


Use Known Invariants for Pruning

- Need known invariants that are
  - small as BDDs and
  - effective in reducing BDD size

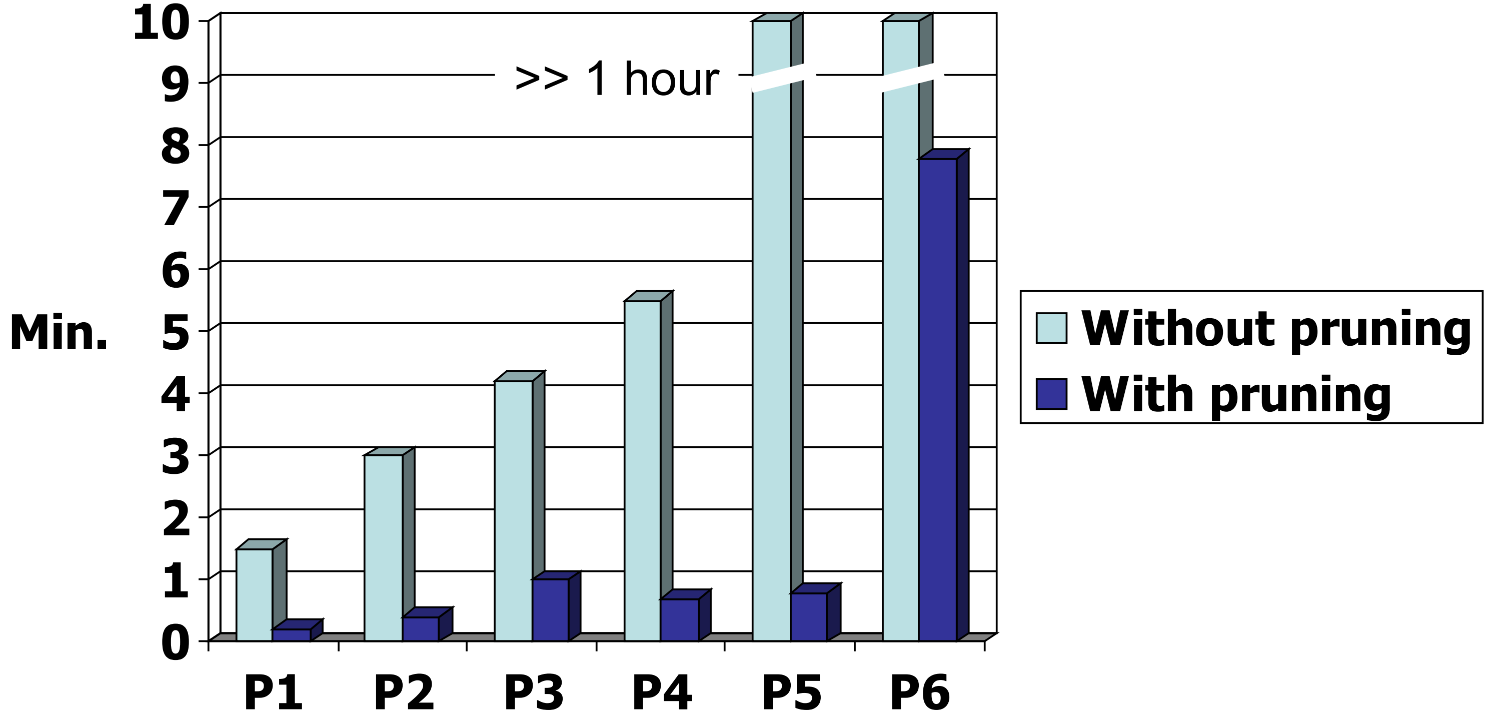


Optimization 1: Mutual Exclusion of Transitions



- Many “concurrent” transitions are sequential
  - Determine using static analysis
- Use this to prune backward search

Overall Effects on TCAS II



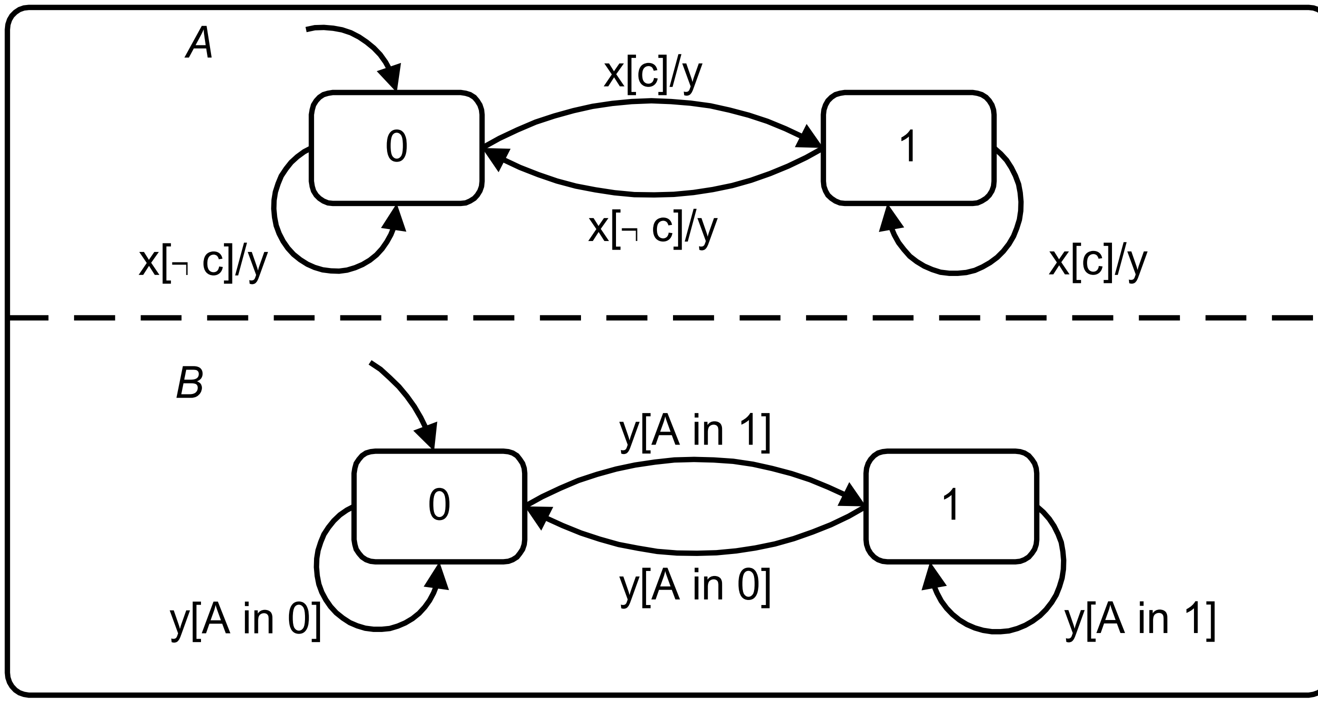
Initial EPD Analyses Failed

- Even though it has fewer states than TCAS II

	TCAS II	EPD System
State space	230 bits, 10^{60} states	90 bits, 10^{27} states

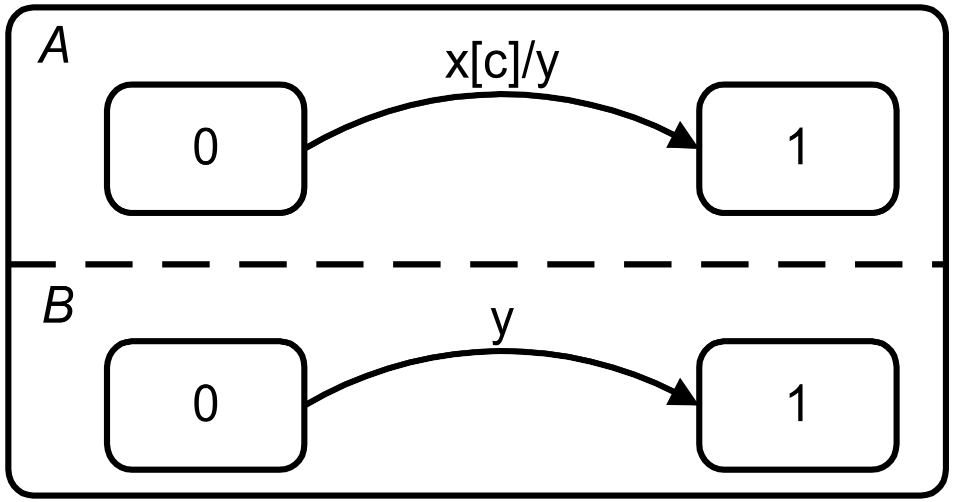
- Main difference in synchronization

Oblivious Synchronization (used in TCAS II)



- y signals completion of machine A
  - Macrostep length: 2
  - x $\rightarrow$ y $\rightarrow$ stable

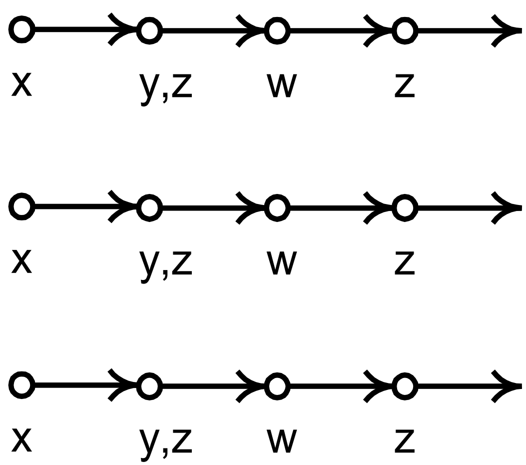
Non-Oblivious Synchronization (used in EPD)



- y signals state change in machine A
- Macrostep length: 1 or 2
  - $x \rightarrow y \rightarrow$ stable
  - $x \rightarrow$ stable

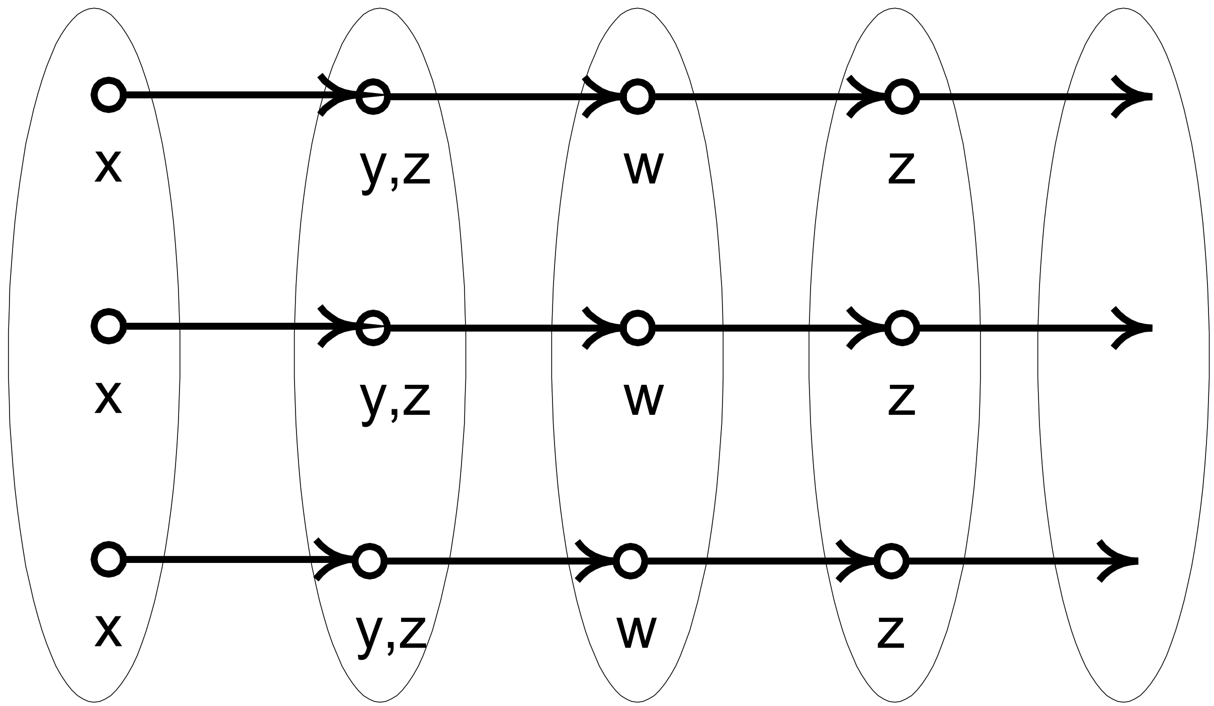
Oblivious Synchronization: General Case

- Event sequence always identical
  - Thus, every macrostep has the same length



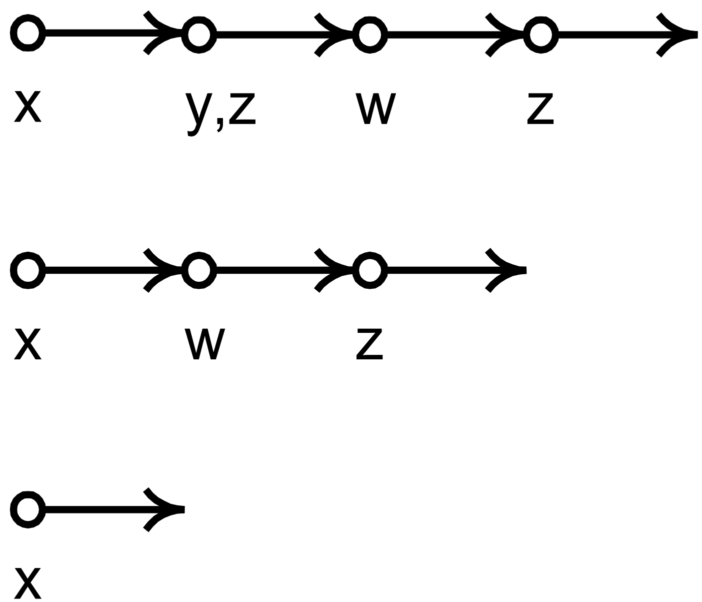
Backward Search for Oblivious Synchronization

- Yields small BDDs



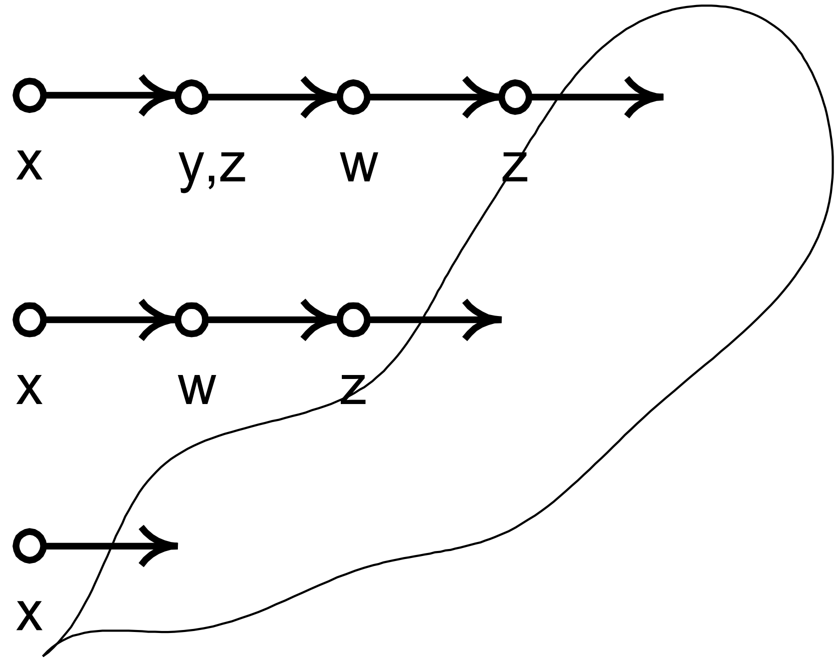
Non-Oblivious Synchronization: General Case

- Macrosteps may have different lengths.



Backward Search for Non-Oblivious Synchronization

- Larger BDDs



Optimization 2: Restoring Regularity in State Sets [ICSE 99]

- Automatic semantics-preserving transformation:

- Add stuttering states.
- Preserve most properties, e.g., invariants and eventualities. [Lamport 83, Browne et al. 89]

New Backward Search

- ✓ Make every macrostop equal in length. Smaller BDDs.
- ✗ Increase # states and # state variables.
- ✗ Increase # iterations to reach fixed points.

Other Optimizations [ISSTA 98]

- Partition transition relation in various ways.
  - Use multiple BDDs for transition relation.
- Abstract automatically be dependency analysis.
  - Remove part of system that can't affect result.
- Improve counter-example search.
  - Avoid work in forward search.

Arithmetic

- TCAS II spec contains nontrivial arithmetic.
  - Variables assumed discrete and bounded.
  - Encoded as bits.
  - Ok for linear constraints (e.g., a+b> c).
  - But nonlinear constraints abstracted away.
- *BMD [Bryant & Chen] and HDD [Clarke & Zhao 95] do not help.
  - Good for ab
  - But not ab> c.

Why Nonlinear Constraints Hard?

Recall: *Pre(S)* = *X . S(X) R(X,X).*

- Assume *X* has n numeric variables.
- Project 2*n*-dimensional regions on *n*-dimensional space.
- Hard if *S* and *R* nonlinear.

May not need to solve the general problem. Assume:

- All numeric variables are inputs,
- and unconstrained at beginning of macrostep

Different Approach

- Represent each constraint as a BDD variable.
  - Overly conservative if infeasible constraints not detected.
- Detect infeasible paths using constraint-satisfaction engine (black box).
- Prune infeasible paths from BDD ("filtering").

Some Lessons Learned

- Focus on restricted models that people care about.
- Exploit high-level knowledge to improve analysis.
  - Synchronization, environmental assumptions, etc.
  - In addition to low-level BDD tricks.
- Combine static analysis and symbolic model checking.
- Help understand system behaviors.
  - In addition to verification/falsification.

How General are the Techniques?

- Optimizations specific to events, macrosteps, and the synchrony hypothesis:
  - Maybe applicable to synchronous programming languages.
- Combining forward static analysis and backward symbolic search:
  - Seems promising.
- Constraint-satisfaction approach:
  - Applicable if environment not constrained.