Model Checking

Edmund M. Clarke, Jr.
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213

Temporal Logic Model Checking

Specification Language: A propositional temporal logic.

Verification Procedure: Exhaustive search of the state space of the concurrent system to determine truth of specification.

- E. M. Clarke and E. A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *Logic of programs: workshop, Yorktown Heights, NY, May 1981*, volume 131 of *Lecture Notes in Computer Science*. Springer-Verlag, 1981.
- J.P. Quielle and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the Fifth International Symposium in Programming*, volume 137 of *Lecture Notes in Computer Science*. Springer-Verlag, 1981.

Why Model Checking?

Advantages:

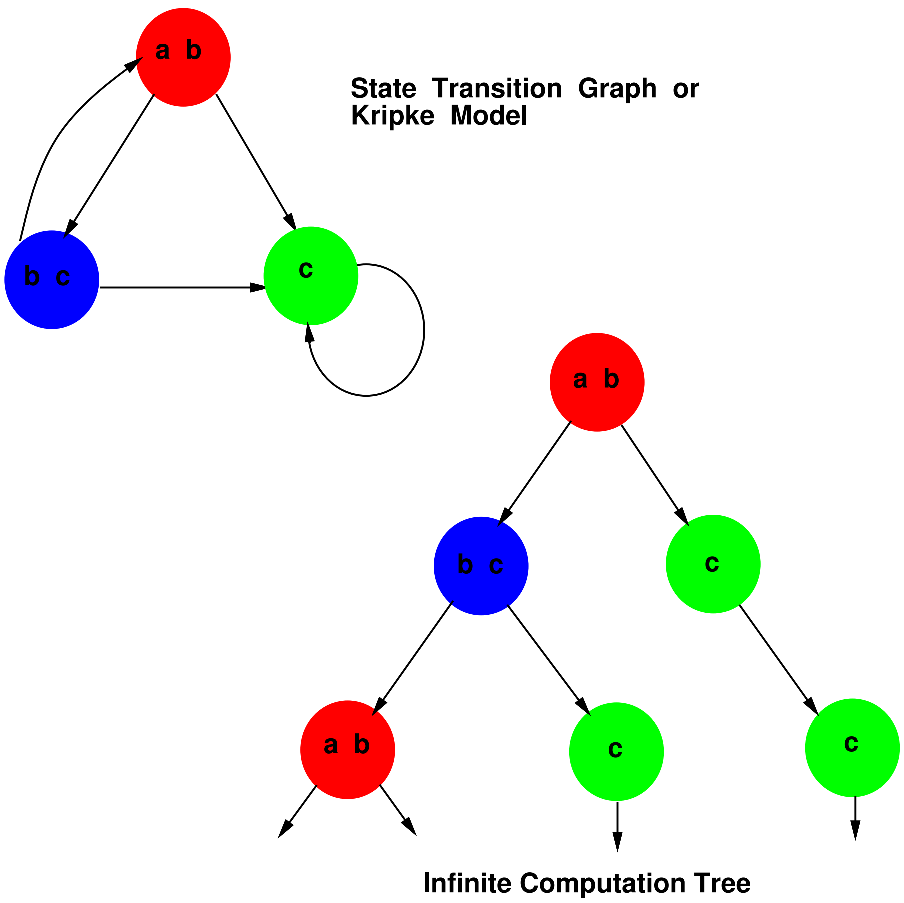
- No proofs!!!
- Fast
- Counterexamples
- No problem with partial specifications
- Logics can easily express many concurrency properties

Main Disadvantage: State Explosion Problem

- Too many processes
- In digital hardware terms: too many latches

Much progress recently!!

Model of Computation



(Unwind State Graph to obtain Infinite Tree)

Model of Computation (Cont.)

Formally, a Kripke structure is a triple $M = \langle S, R, L \rangle$, where

- S is the set of states,
- $R \subseteq S \times S$ is the transition relation, and
- $L : S \rightarrow \mathcal{P}(AP)$ gives the set of atomic propositions true in each state.

We assume that R is total (i.e., for all states $s \in S$ there exists a state $s' \in S$ such that $(s, s') \in R$).

A path in M is an infinite sequence of states, $\pi = s_0, s_1, \ldots$ such that for $i \geq 0$, $(s_i, s_{i+1}) \in R$.

We write π^i to denote the suffix of π starting at s_i.

Unless otherwise stated, all of our results apply only to finite Kripke structures.

Computation Tree Logics

Temporal logics may differ according to how they handle branching in the underlying computation tree.

In a linear temporal logic, operators are provided for describing events along a single computation path.

In a branching-time logic the temporal operators quantify over the paths that are possible from a given state.

The Logic CTL*

The computation tree logic CTL* combines both branching-time and linear-time operators.

In this logic a path quantifier can prefix an assertion composed of arbitrary combinations of the usual linear-time operators.

1. Path quantifier:
   - **A**—"for every path"
   - **E**—"there exists a path"
2. Linear-time operators:
   - **X**p—p holds next time.
   - **F**p—p holds sometime in the future
   - **G**p—p holds globally in the future
   - p**U**q—p holds until q holds

Path Formulas and State Formulas

The syntax of state formulas is given by the following rules:

- If $p \in AP$, then p is a state formula.
- If f and g are state formulas, then $\neg f$ and $f \vee g$ are state formulas.
- If f is a path formula, then $\mathbf{E}(f)$ is a state formula.

Two additional rules are needed to specify the syntax of path formulas:

- If f is a state formula, then f is also a path formula.
- If f and g are path formulas, then $\neg f$, $f \vee g$, $\mathbf{X} f$, and $\mathbf{U}\ f g$ are path formulas.

State Formulas (Cont.)

If f is a state formula, the notation $M, s \models f$ means that f holds at state s in the Kripke structure M.

Assume f_1 and f_2 are state formulas and g is a path formula. The relation $M, s \models f$ is defined inductively as follows:

1. $s \models p \quad \Leftrightarrow p \in L(s)$.
2. $s \models \neg f_1 \quad \Leftrightarrow s \not\models f_1$.
3. $s \models f_1 \vee f_2 \Leftrightarrow s \models f_1$ or $s \models f_2$.
4. $s \models \mathbf{E}(g) \quad \Leftrightarrow$ there exists a path π starting with s such that $\pi \models g$.

Path Formulas (Cont.)

If f is a path formula, $M, \pi \models f$ means that f holds along path π in Kripke structure M.

Assume g_1 and g_2 are path formulas and f is a state formula. The relation $M, \pi \models f$ is defined inductively as follows:

1. $\pi \models f \quad \Leftrightarrow s$ is the first state of π and $s \models f$.
2. $\pi \models \neg g_1 \quad \Leftrightarrow \pi \not\models g_1$.
3. $\pi \models g_1 \vee g_2 \Leftrightarrow \pi \models g_1$ or $\pi \models g_2$.
4. $\pi \models \mathbf{X} g_1 \quad \Leftrightarrow \pi^1 \models g_1$.
5. $\pi \models \mathbf{U}\ g_1 g_2 \Leftrightarrow$ there exists a $k \geq 0$ such that $\pi^k \models g_2$ and for $0 \leq j < k$, $\pi^j \models g_1$.

Standard Abbreviations

The customary abbreviations will be used for the connectives of propositional logic.

In addition, we will use the following abbreviations in writing temporal operators:

- $\mathbf{A}(f) \equiv \neg\,\mathbf{E}(\neg f)$
- $\mathbf{F}f \equiv \mathbf{U}\ true f$
- $\mathbf{G}f \equiv \neg\mathbf{F}\neg f$

CTL and LTL

CTL is a restricted subset of CTL* that permits only branching-time operators—each of the linear-time operators $\mathbf{G}$, $\mathbf{F}$, $\mathbf{X}$, and $\mathbf{U}$ must be immediately preceded by a path quantifier.

Example: $\mathbf{AG}(\mathbf{EF}\,p)$

LTL consists of formulas that have the form $\mathbf{A}\,f$ where f is a path formula in which the only state subformulas permitted are atomic propositions.

Example: $\mathbf{A}(\mathbf{FG}\,p)$

Expressive Power

It can be shown that the three logics discussed in this section have different expressive powers.

For example, there is no CTL formula that is equivalent to the LTL formula $\mathbf{A}(\mathbf{FG}\, p)$.

Likewise, there is no LTL formula that is equivalent to the CTL formula $\mathbf{AG}(\mathbf{EF}\, p)$.

The disjunction $\mathbf{A}(\mathbf{FG}\, p) \vee \mathbf{AG}(\mathbf{EF}\, p)$ is a CTL* formula that is not expressible in either CTL or LTL.

Basic CTL Operators

There are eight basic CTL operators:

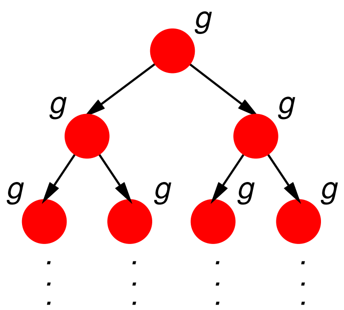
- **AX** and **EX**,
- **AG** and **EG**,
- **AF** and **EF**,
- **AU** and **EU**

Each of these can be expressed in terms of **EX**, **EG**, and **EU**:

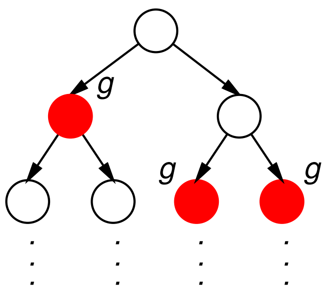
- $\mathbf{AX}\, f = \neg\, \mathbf{EX}(\neg f)$
- $\mathbf{AG}\, f = \neg\, \mathbf{EF}(\neg f)$
- $\mathbf{AF}\, f = \neg\, \mathbf{EG}(\neg f)$
- $\mathbf{EF}\, f = \mathbf{E}[true\ \mathbf{U}\ f]$
- $\mathbf{A}[f\ \mathbf{U}\ g] \equiv \neg\, \mathbf{E}[\neg g\ \mathbf{U}\ \neg f \wedge \neg g] \wedge \neg\, \mathbf{EG}\, \neg g$

Basic CTL Operators

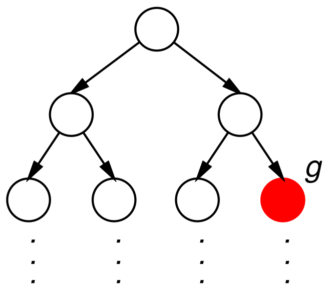
The four most widely used CTL operators are illustrated below. Each computation tree has the state s_0 as its root.



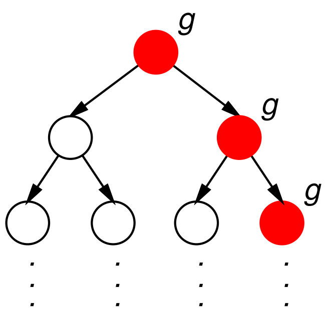
$M, s_0 \models \mathbf{AG}\, g$



$M, s_0 \models \mathbf{AF}\, g$



$M, s_0 \models \mathbf{EF}\, g$



$M, s_0 \models \mathbf{EG}\, g$

Typical CTL Formulas

- **EF**$(Started \wedge \neg Ready)$: it is possible to get to a state where *Started* holds but *Ready* does not hold.
- **AG**$(Req \Rightarrow$ **AF**$Ack)$: if a *Request* occurs, then it will be eventually *Acknowledged*.
- **AG**(**AF** $DeviceEnabled$): *DeviceEnabled* holds infinitely often on every computation path.
- **AG**(**EF** $Restart$): from any state it is possible to get to the *Restart* state.

Model Checking Problem

Let M be the state–transition graph obtained from the concurrent system.

Let f be the specification expressed in temporal logic.

Find all states s of M such that

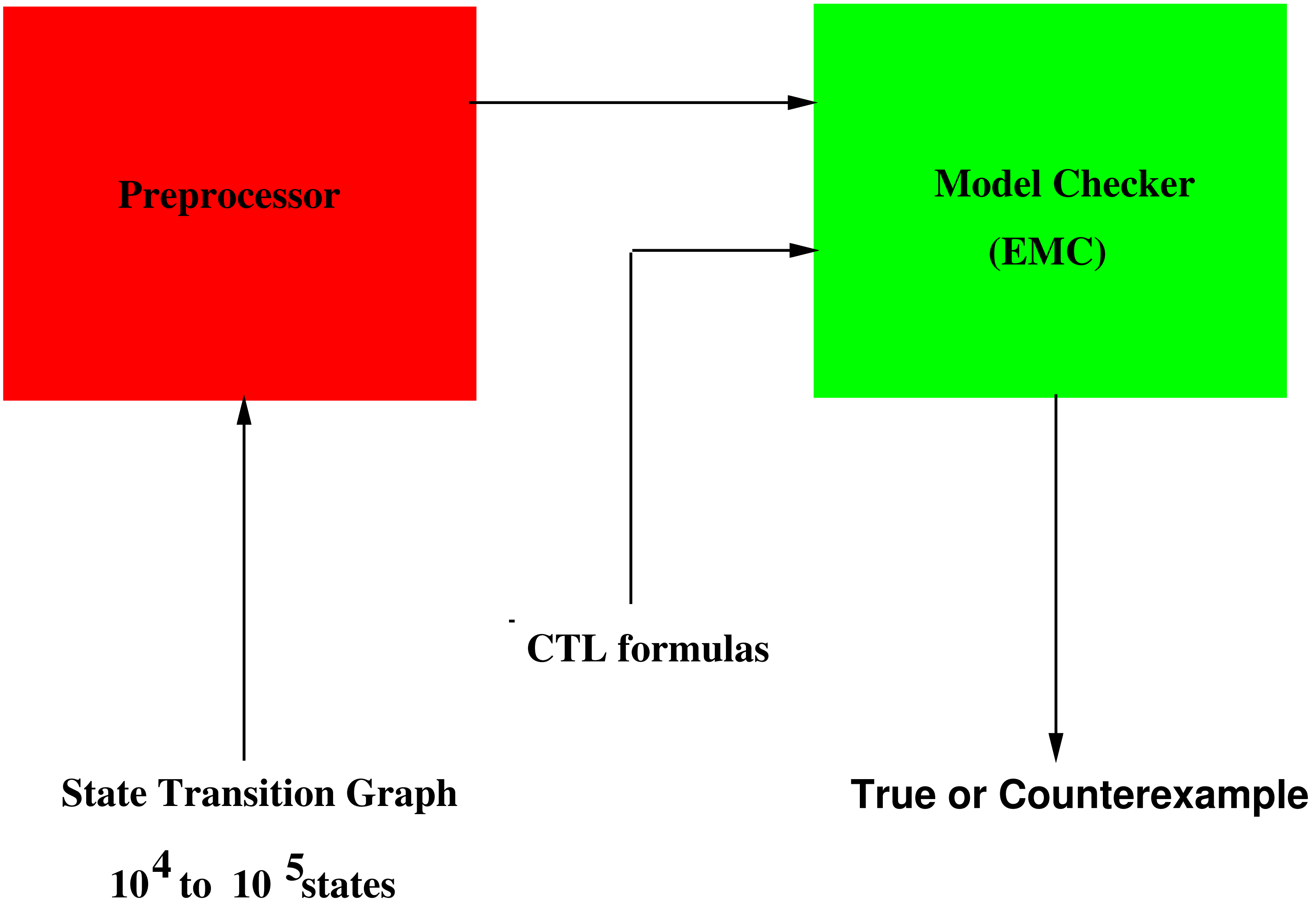
$$M, s \models f$$

and check if initial states are among these.

Efficient model checking algorithms exist for CTL.

- E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Programming Languages and Systems*, 8(2):pages 244–263, 1986.

Explicit Traversal



Symbolic Model Checking

Method used by most "industrial strength" model checkers:

- uses boolean encoding for state machine and sets of states.
- can handle much larger designs – hundreds of state variables.
- BDDs traditionally used to represent boolean functions.

Symbolic Model Checking with BDDs

Ken McMillan implemented a version of the CTL model checking algorithm using Binary Decision Diagrams in 1987.

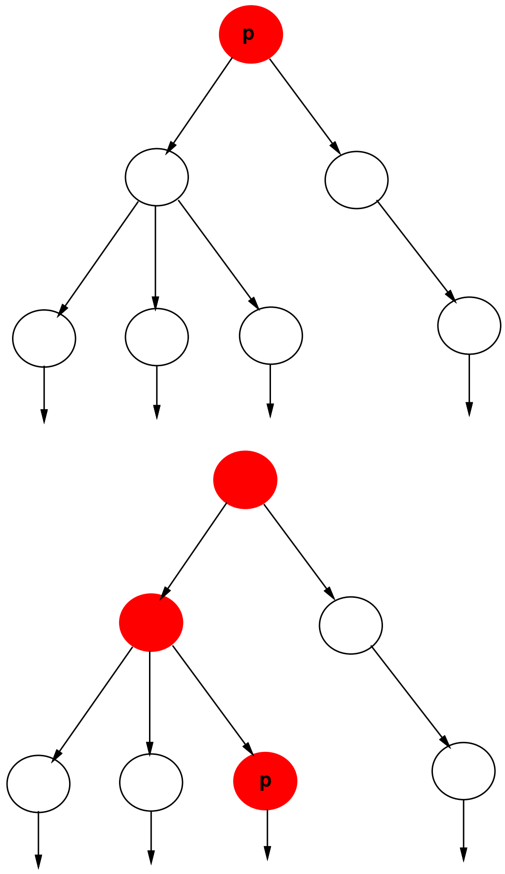
Carl Pixley independently developed a similar algorithm, as did the French researchers, Coudert and Madre.

BDDs enabled handling much larger concurrent systems. (usually, an order of magnitude increase in hardware latches!)

- J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):pages 142–170, 1992.
- K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

Fixpoint Algorithms

$$\mathbf{EF}\, p = p \vee \mathbf{EX}\, \mathbf{EF}\, p$$



Fixpoint Algorithms (cont.)

Key properties of $\mathbf{EF}\,p$:

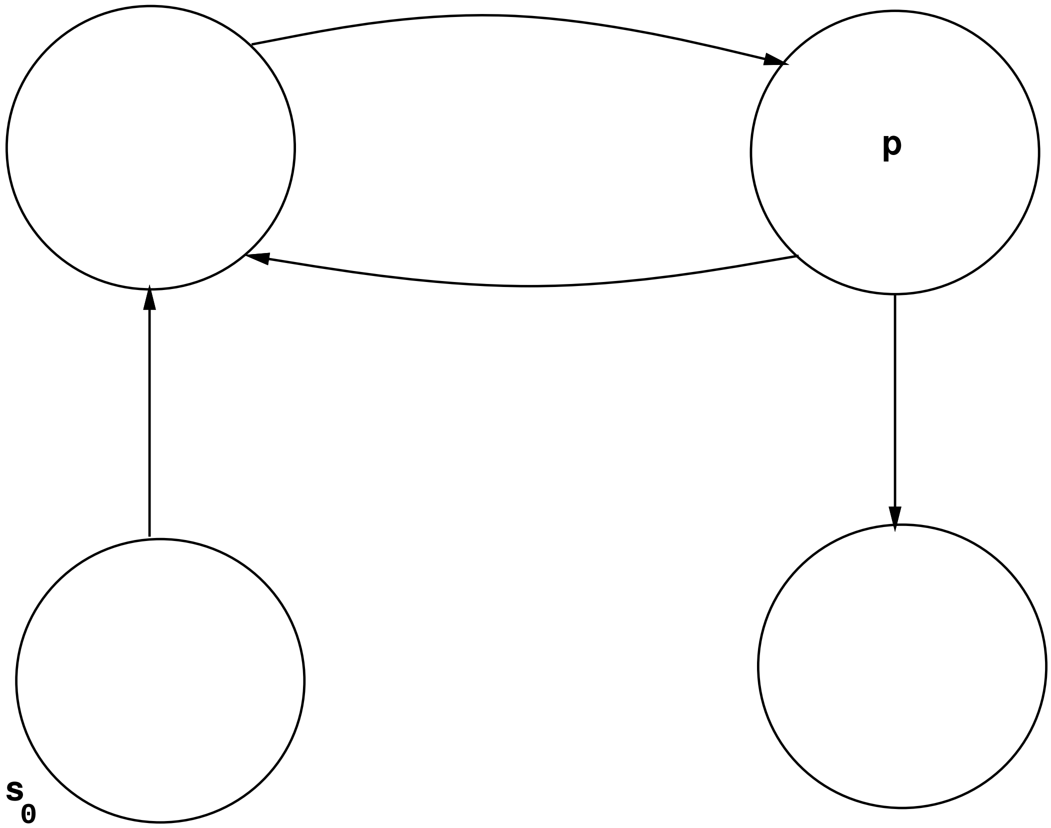
1. $\mathbf{EF}\,p = p \vee \mathbf{EX}\,\mathbf{EF}\,p$
2. $U = p \vee \mathbf{EX}\,U$ implies $\mathbf{EF}\,p \subseteq U$

We write $\mathbf{EF}\,p = \mathbf{Lfp}\ U.p \vee \mathbf{EX}\,U$.

How to compute $\mathbf{EF}\,p$:

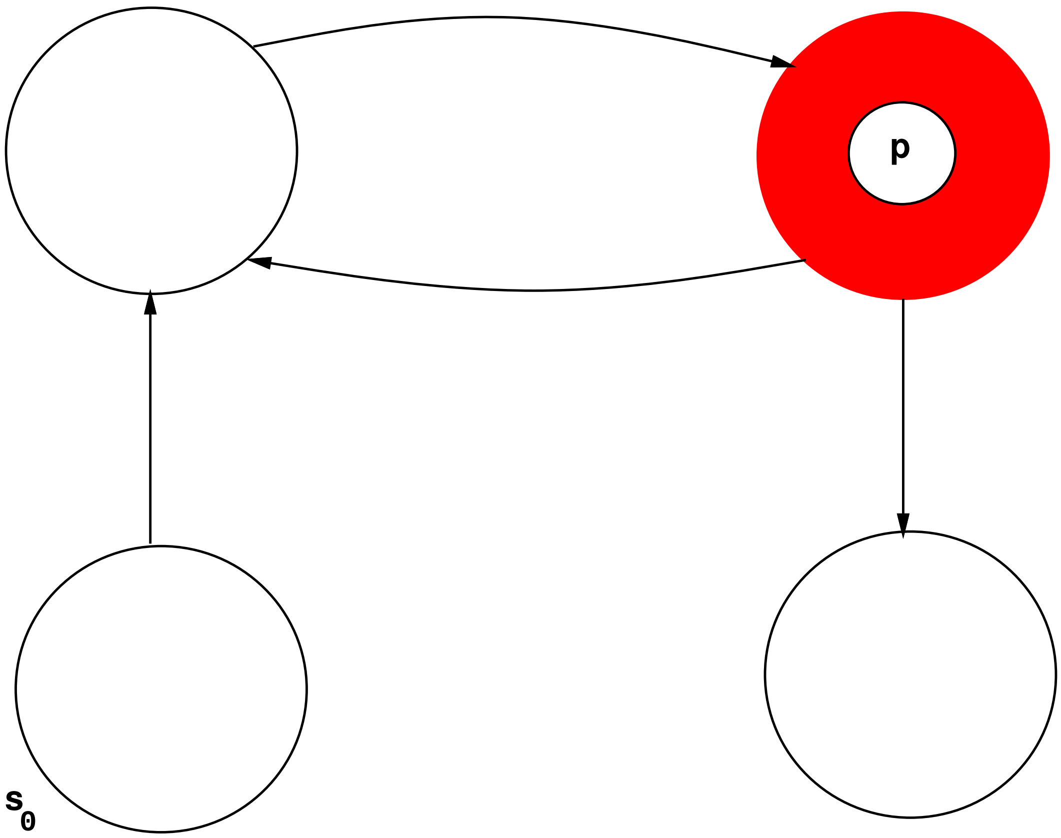
$$
\begin{aligned}
U_0 &= \mathbf{False} \\
U_1 &= p \vee \mathbf{EX}\,U_0 \\
U_2 &= p \vee \mathbf{EX}\,U_1 \\
U_3 &= p \vee \mathbf{EX}\,U_2 \\
&\vdots
\end{aligned}
$$

$$M, s_0 \models \mathbf{EF}\, p\textbf{?}$$



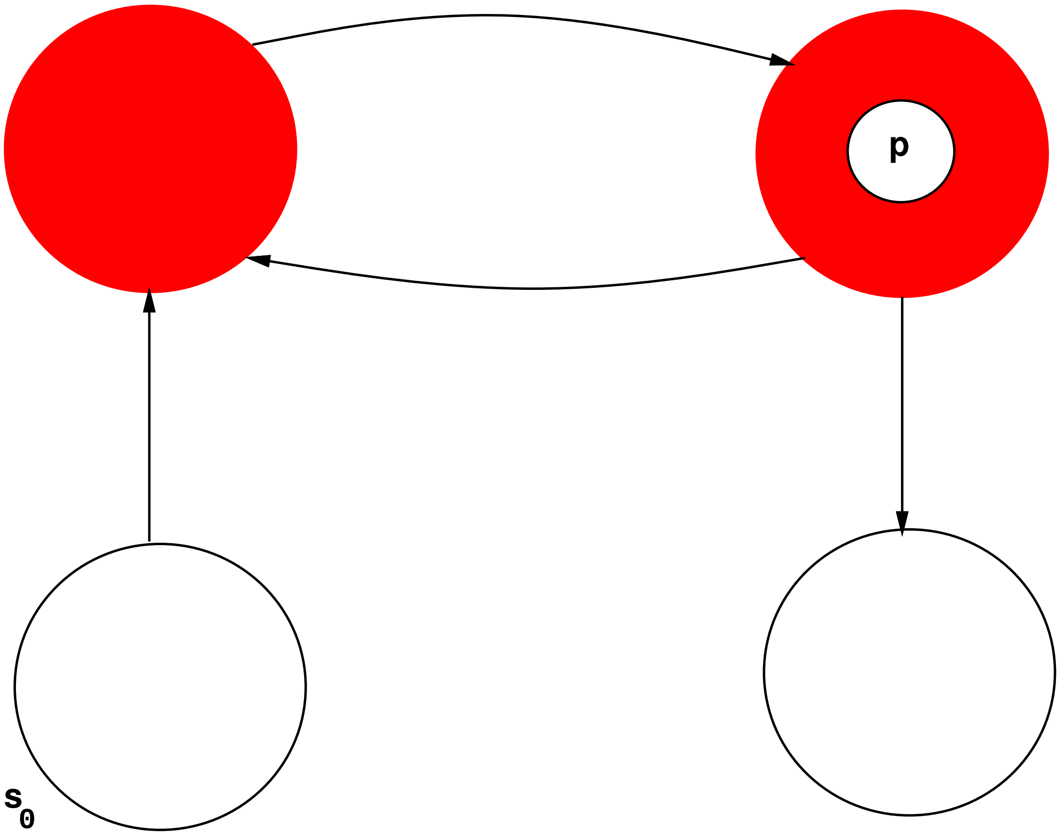
$$U_0 = \emptyset$$

$$M, s_0 \models \mathbf{EF}\, p?$$



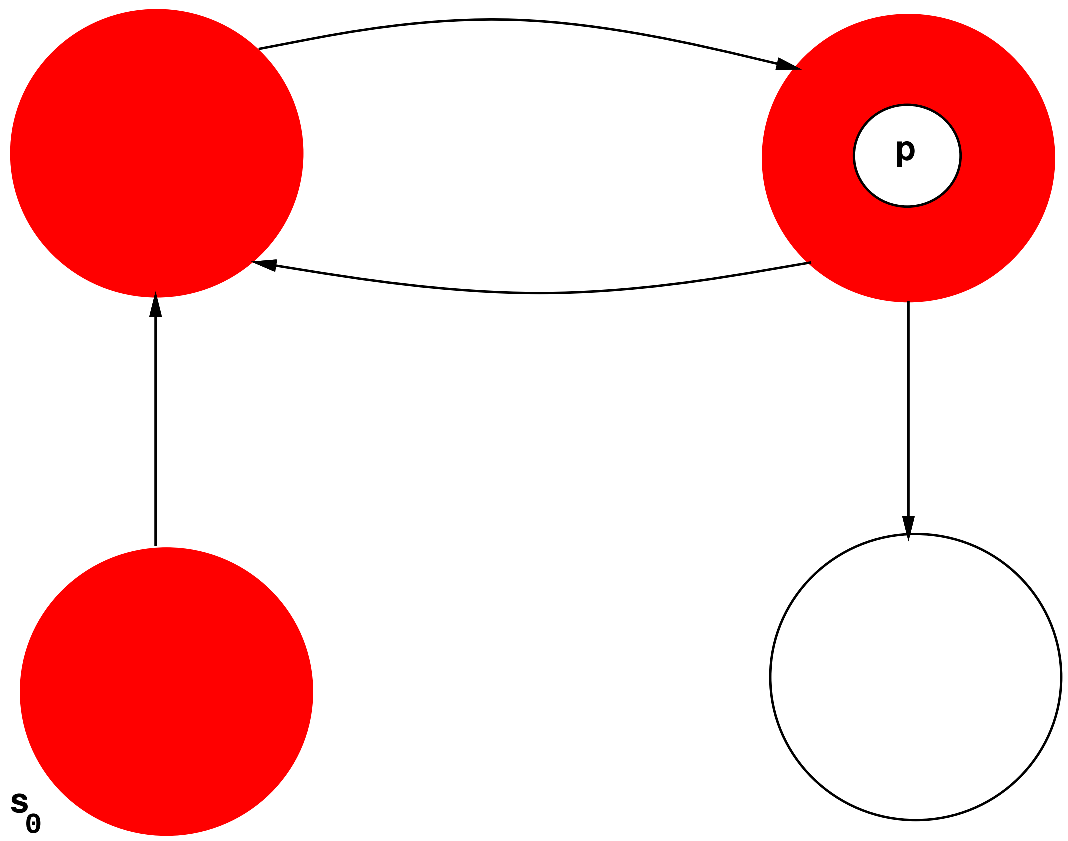
$$U_1 = p \vee \mathbf{EX}\, U_0$$

$$M, s_0 \models \mathbf{EF}\, p?$$



$$U_2 = p \vee \mathbf{EX}\, U_1$$

$$M, s_0 \models \mathbf{EF}\, p?$$



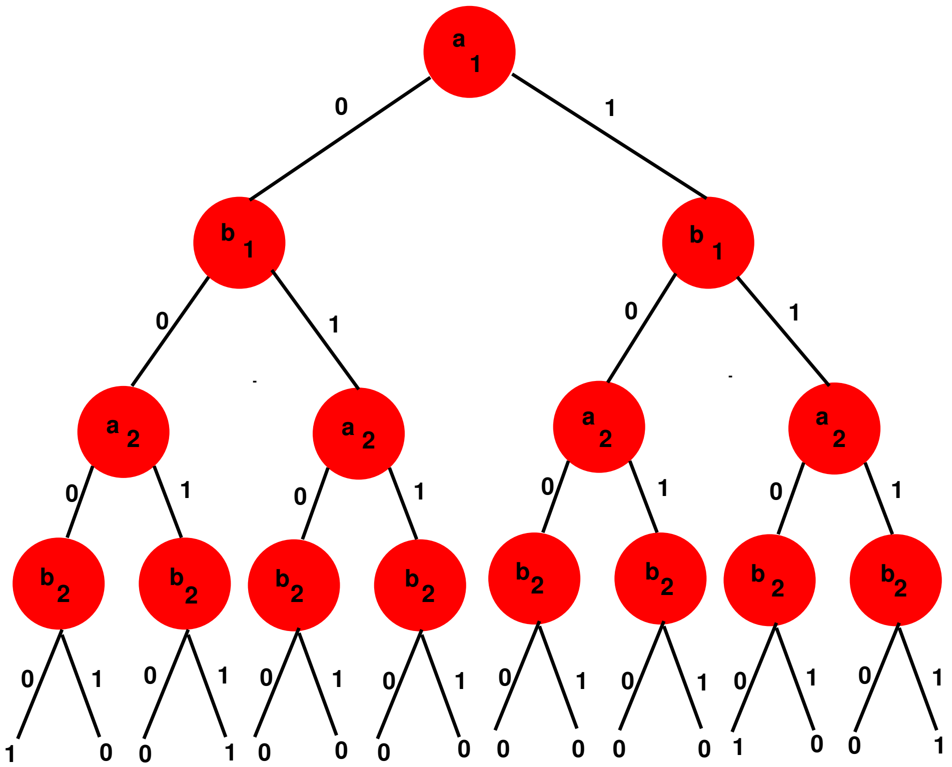
$$U_3 = p \vee \mathbf{EX}\, U_2$$

Ordered Binary Decision Trees and Diagrams

Ordered Binary Decision Tree for the two-bit comparator, given by the formula

$$f(a_1, a_2, b_1, b_2) = (a_1 \leftrightarrow b_1) \wedge (a_2 \leftrightarrow b_2),$$

is shown in the figure below:



From Binary Decision Trees to Diagrams

An Ordered Binary Decision Diagram (OBDD) is an ordered decision tree where

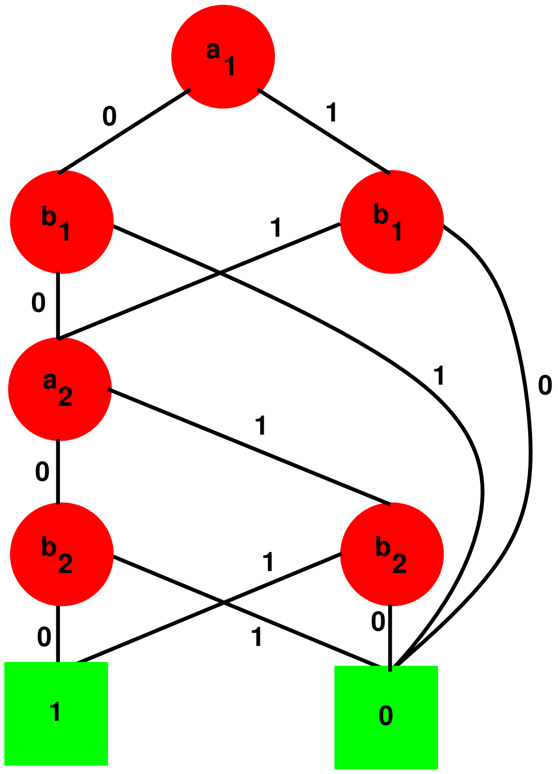
- All isomorphic subtrees are combined, and
- All nodes with isomorphic children are eliminated.

Given a parameter ordering, OBDD is unique up to isomorphism.

- R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

OBDD for Comparator Example

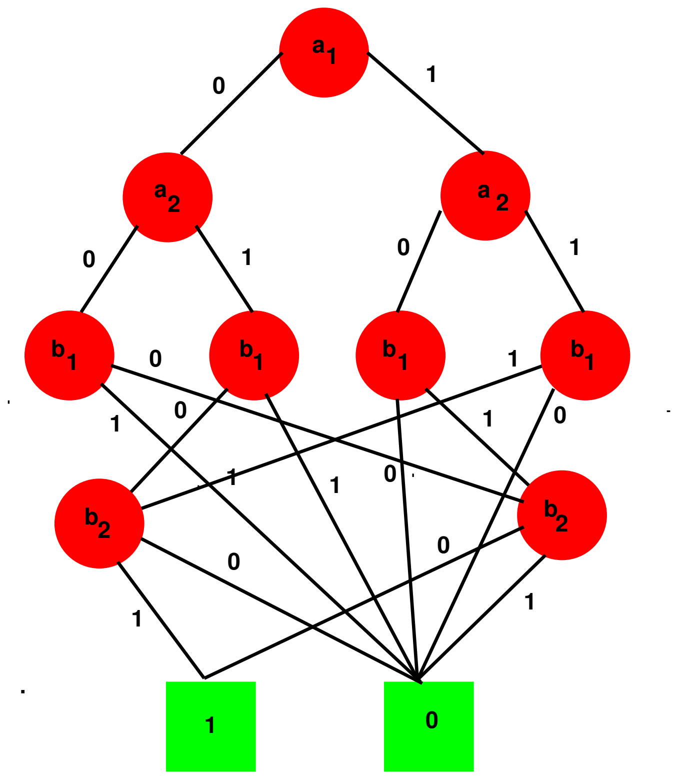
If we use the ordering $a_1 < b_1 < a_2 < b_2$ for the comparator function, we obtain the OBDD below:



Variable Ordering Problem

The size of an OBDD depends critically on the variable ordering.

If we use the ordering $a_1 < a_2 < b_1 < b_2$ for the comparator function, we get the OBDD below:



Variable Ordering Problem (Cont.)

For an n-bit comparator:

- if we use the ordering $a_1 < b_1 < \ldots < a_n < b_n$, the number of vertices will be $3n + 2$.
- if we use the ordering $a_1 < \ldots < a_n < b_1 \ldots < b_n$, the number of vertices is $3 \cdot 2^n - 1$.

Moreover, there are boolean functions that have exponential size OBDDs for any variable ordering.

An example is the middle output (n^{th} output) of a combinational circuit to multiply two n bit integers.

Logical operations on OBDD's

- Logical negation: $\neg f(a, b, c, d)$

  Replace each leaf by its negation

- Logical conjunction: $f(a, b, c, d) \wedge g(a, b, c, d)$
  - Use Shannon's expansion as follows,

    $$f \cdot g = \bar{a} \cdot (f|_{\bar{a}} \cdot g|_{\bar{a}}) + a \cdot (f|_a \cdot g|_a)$$

    to break problem into two subproblems. Solve subproblems recursively.
  - Always combine isomorphic subtrees and eliminate redundant nodes.
  - Hash table stores previously computed subproblems
  - Number of subproblems bounded by $|f| \cdot |g|$.

Logical operations (cont.)

- Boolean quantification: $\exists a : f(a, b, c, d)$
  - By definition,

$$\exists a : f = f|_{\bar{a}} \vee f|_a$$

  - $f(a, b, c, d)|_{\bar{a}}$: replace all a nodes by left sub-tree.
  - $f(a, b, c, d)|_a$: replace all a nodes by right sub-tree.

Using the above operations, we can build up OBDD's for complex boolean functions from simpler ones.

Symbolic Model Checking Algorithm

How to represent state-transition graphs with Ordered Binary Decision Diagrams:

Assume that system behavior is determined by n boolean state variables $v_1, v_2, \ldots, v_n$.

The Transition relation T will be given as a boolean formula in terms of the state variables:

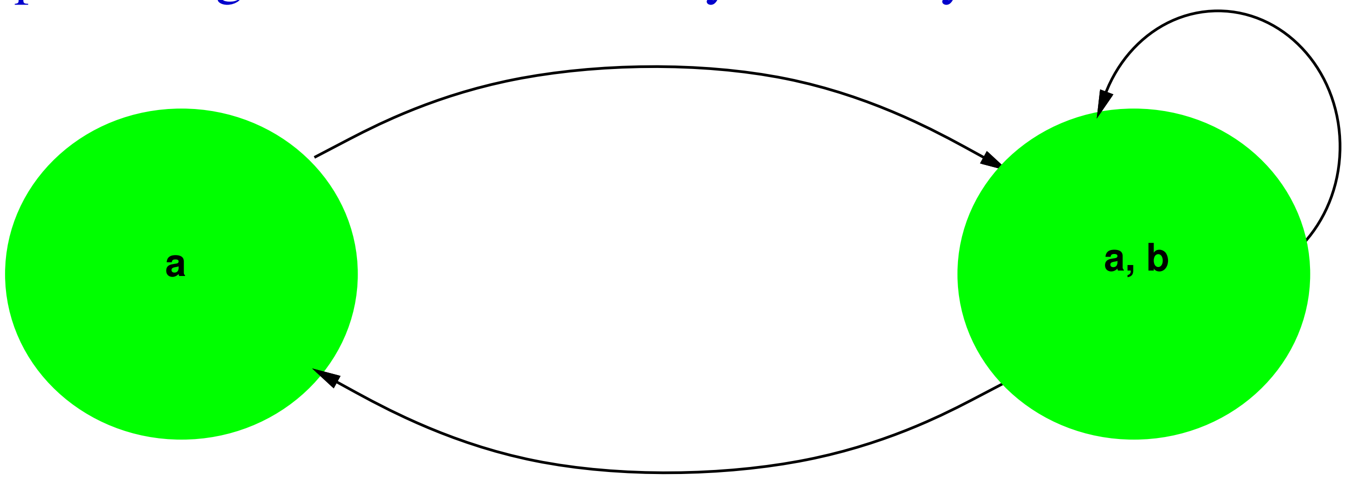
$$T(v_1, \ldots, v_n, v'_1, \ldots, v'_n)$$

where $v_1, \ldots v_n$ represents the current state and $v'_1, \ldots, v'_n$ represents the next state.

Now convert T to a OBDD!!

Symbolic Model Checking (cont.)

Representing transition relations symbolically:



Boolean formula for transition relation:

$$
\begin{array}{lccccccc}
 & (a & \wedge & \neg b & \wedge & a' & \wedge & b') \\
\vee & (a & \wedge & b & \wedge & a' & \wedge & b') \\
\vee & (a & \wedge & b & \wedge & a' & \wedge & \neg b')
\end{array}
$$

Now, represent as an OBDD!

Symbolic Model Checking (cont.)

Consider $f = \mathbf{EX}\, p$.

Now, introduce state variables and transition relation:

$$f(\bar{v}) = \exists \bar{v}'[T(\bar{v}, \bar{v}') \wedge p(\bar{v}')]$$

Compute OBDD for relational product on right side of formula.

Symbolic Model Checking (cont.)

How to evaluate fixpoint formulas using OBDDs:

$$\mathbf{EF}\,p = \mathbf{Lfp}\,U.\,p \vee \mathbf{EX}\,U$$

Introduce state variables:

$$\mathbf{EF}\,p = \mathbf{Lfp}\,U.\,p(\bar{v}) \vee \exists \bar{v}'\,[T(\bar{v},\bar{v}') \wedge U(\bar{v}')]$$

Now, compute the sequence

$$U_0(\bar{v}), U_1(\bar{v}), U_2(\bar{v}), \ldots$$

until convergence.

Convergence can be detected since the sets of states $U_i(\bar{v})$ are represented as OBDDs.

Notable Examples

The following examples illustrate the power of model checking to handle industrial size problems.

They come from many sources, not just my research group.

- Edmund M. Clarke, Jeannette M. Wing, et al. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, December 1996.

Notable Examples–IEEE Futurebus$^+$

- In 1992 Clarke and his students at CMU used SMV to verify the cache coherence protocol in the IEEE Futurebus+ Standard.
- They constructed a precise model of the protocol and attempted to show that it satisfied a formal specification of cache coherence.
- They found a number of previously undetected errors in the design of the protocol.
- This was the first time that formal methods have been used to find errors in an IEEE standard.
- Although development started in 1988, all previous attempts to validate Futurebus+ were based on informal techniques.

Notable Examples–HDLC

- A High-level Data Link Controller (HDLC) was being designed at AT&T in Madrid.
- In 1996 researchers at Bell Labs offered to check some properties of the design. The design was almost finished, so no errors were expected.
- Within five hours, six properties were specified and five were verified, using the FormalCheck verifier.
- The sixth property failed, uncovering a bug that would have reduced throughput or caused lost transmissions.
- The error was corrected in a few minutes and formally verified.

Notable Examples–PowerPC 620 Microprocessor

- Richard Raimi and Jim Lear at Somerset used Motorola's Verdict model checker to debug a hardware laboratory failure.
- Initial silicon of PowerPC 620 microprocessor crashed during boot of an operating system.
- With run time in seconds, Verdict produced example of BIU deadlock causing the failure.
- Paper on this published at 1997 IEEE International Test Conference.

Problems with BDDs

- BDDs are a canonical representation. Often become too large.
- Selecting right variable ordering very important for obtaining small BDDs.
  - Often time consuming or needs manual intervention.
  - Sometimes, no space efficient variable ordering exists.

Next, we describe an alternative approach to symbolic model checking that uses SAT procedures.

Advantages of SAT Procedures

- SAT procedures also operate on boolean expressions but do not use canonical forms.
- Do not suffer from the potential space explosion of BDDs.
- Can handle functions with 100s to 1000s of variables.
- Very efficient implementations available.

Bounded Model Checking

- Bounded model checking uses a SAT procedure instead of BDDs.
- We construct boolean formula that is satisfiable iff there is a specific finite path of length k in underlying machine.
- We look for longer and longer paths by incrementing the bound k.
- After some number of iterations, we may conclude no such path exists and specification holds.
- For example, to verify safety properties, number of iterations is bounded by diameter of finite state machine.

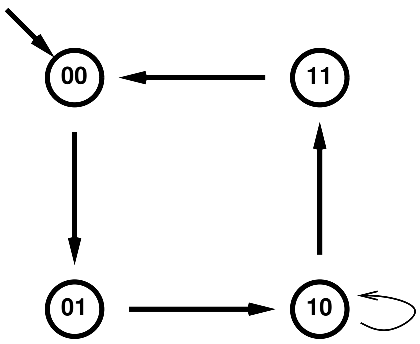
- Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In W. Rance Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Amsterdam, March 1999.

Main Advantages of SAT Approach

- Bounded model checking works quickly. This is due to depth first nature of SAT search procedures.
- It finds finite paths of minimal length. This helps user understand the example more easily.
- It uses much less space than BDD based approaches.
- Does not need manually selected variable order or costly reordering. Default splitting heuristics usually sufficient.

Example To Illustrate New Technique

Two-bit counter with an erroneous transition:



- Each state s is represented by two state variables $s[1]$ and $s[0]$.

- In initial state, value of the counter is 0. Thus,

$$I(s) = \neg s[1] \wedge \neg s[0].$$

- Let

$$inc(s, s') = (s'[0] \leftrightarrow \neg s[0]) \wedge (s'[1] \leftrightarrow (s[0] \oplus s[1])).$$

- Define

$$T(s, s') = inc(s, s') \vee (s[1] \wedge \neg s[0] \wedge s'[1] \wedge \neg s'[0])$$

- Have deliberately added erroneous transition!!

Example (Cont.)

- Suppose we want to know if counter will eventually reach state (11).
- Can specify the property by $\mathbf{AF}q$, where $q(s) = s[1] \wedge s[0]$.

  On all execution paths, there is a state where $q(s)$ holds.
- Equivalently, we can check if there is a path on which counter never reaches state (11).
- This is expressed by $\mathbf{EG}p$, where $p(s) = \neg s[1] \vee \neg s[0]$.

  There exists a path such that $p(s)$ holds globally along it.

Example (Cont.)

- In bounded model checking, we consider paths of length k.
- We start with $k = 0$ and increment k until a witness is found.
- Assume k equals 2. Call the states s_0, s_1, s_2.
- We formulate constraints on s_0, s_1, and s_2 in propositional logic.
- Constraints guarantee that (s_0, s_1, s_2) is a witness for $\mathbf{EG}p$ and, hence, a counterexample for $\mathbf{AF}q$.

Example (Cont.)

- First, we constrain (s_0, s_1, s_2) to be a valid path starting from the initial state.
- Obtain a propositional formula

$$[\![\, M \,]\!] = I(s_0) \wedge T(s_0, s_1) \wedge T(s_1, s_2).$$

Example (Cont.)

- Second, we constrain the shape of the path.
- The sequence of states s_0, s_1, s_2 can be a loop.
- If so, there is a transition from s_2 to the initial state s_0, s_1 or itself.
- We write ${}_lL = T(s_2, s_l)$ to denote the transition from s_2 to a state s_l where $l \in [0, 2]$.
- We define L as $\bigvee_{l=0}^{2} {}_lL$. Thus $\neg L$ denotes the case where no loop exists.

Example (Cont.)

- The temporal property $\mathbf{G}p$ must hold on (s_0, s_1, s_2).
- If no loop exists, $\mathbf{G}p$ does not hold and $[\![\ \mathbf{G}p\ ]\!]$ is $false$.
- To be a witness for $\mathbf{G}p$, the path must contain a loop (condition L, given previously).
- Finally, p must hold at every state on the path

$$[\![\ \mathbf{G}p\ ]\!] = p(s_0) \wedge p(s_1) \wedge p(s_2).$$

- We combine all the constraints to obtain the propositional formula

$$[\![\ M\ ]\!] \wedge ((\neg L \wedge false) \vee \bigvee_{l=0}^{2}({}_lL \wedge [\![\ \mathbf{G}p\ ]\!])).$$

Example (Cont.)

- In this example, the formula is satisfiable.
- Truth assignment corresponds to counterexample path (00), (01), (10) followed by self-loop at (10).
- If self-loop at (10) is removed, then formula is unsatisfiable.

General Translation

- Given M, f and k, we construct a propositional formula $[\![\ M, f\ ]\!]_k$.
- The variables $s_0, \ldots, s_k$ in $[\![\ M, f\ ]\!]_k$ are states on path π.
- Each s_i is a vector of state variables.
- If $[\![\ M, f\ ]\!]_k$ is satisfiable, then f is valid along π.

Complexity

- Size of $[\![\ M, f \ ]\!]_k$ is polynomial in size of f if common subformulas are shared.
- It is linear in k and in the size of formulas for T, I and $p \in \mathcal{A}$.
- Thus, existential bounded model checking can be reduced in polynomial time to SAT.

General Strategy

To construct $[\![M, f]\!]_k$:

- We define a propositional formula $[\![M]\!]_k$ that constrains $s_0, \ldots, s_k$ to be on a valid path π in M.
- We give the translation of LTL formula f to a propositional formula that constrains π to satisfy f.

Sequential Multiplier Example

	SMV$_1$		SMV$_2$		SATO		PROVER	
bit	sec	MB	sec	MB	sec	MB	sec	MB
0	919	13	25	79	0	0	0	1
1	1978	13	25	79	0	0	0	1
2	2916	13	26	80	0	0	0	1
3	4744	13	27	82	0	0	1	2
4	6580	15	33	92	2	0	1	2
5	10803	25	67	102	12	0	1	2
6	43983	73	258	172	55	0	2	2
7	>17h		1741	492	209	0	7	3
8				>1GB	473	0	29	3
9					856	1	58	3
10					1837	1	91	3
11					2367	1	125	3
12					3830	1	156	4
13					5128	1	186	4
14					4752	1	226	4
15					4449	1	183	5
sum	71923		2202		23970		1066	

Model Checking: 16x16 bit sequential shift and add multiplier with overflow flag and 16 output bits.

DME Example

	SMV$_1$		SMV$_2$		SATO $k=5$		PROVER $k=5$		SATO $k=10$		PROVER $k=10$	
cells	sec	MB	sec	MB	sec	MB	sec	MB	sec	MB	sec	MB
4	846	11	159	217	0	3	1	3	3	6	54	5
5	2166	15	530	703	0	4	2	3	9	8	95	5
6	4857	18	1762	703	0	4	3	3	7	9	149	6
7	9985	24	6563	833	0	5	4	4	15	10	224	8
8	19595	31		>1GB	1	6	6	5	16	12	323	8
9	>10h				1	6	9	5	24	13	444	9
10					1	7	10	5	36	15	614	10
11					1	8	13	6	38	16	820	11
12					1	9	16	6	40	18	1044	11
13					1	9	19	8	107	19	1317	12
14					1	10	22	8	70	21	1634	14
15					1	11	27	8	168	22	1992	15

Model Checking: Liveness for one user in the DME.

"Buggy" DME Example

	SMV$_1$		SMV$_2$		SATO		PROVER	
cells	sec	MB	sec	MB	sec	MB	sec	MB
4	799	11	14	44	0	1	0	2
5	1661	14	24	57	0	1	0	2
6	3155	21	40	76	0	1	0	2
7	5622	38	74	137	0	1	0	2
8	9449	73	118	217	0	1	0	2
9	segmentation		172	220	0	1	1	2
10	fault		244	702	0	1	0	3
11			413	702	0	1	0	3
12			719	702	0	2	1	3
13			843	702	0	2	1	3
14			1060	702	0	2	1	3
15			1429	702	0	2	1	3

Model Checking: Counterexample for liveness in a buggy DME implementation.

Queue Example

L	SMV_1		SMV_2		SMV_3		SMV_4		SATO		PROVER	
	sec	MB	sec	MB	sec	MB	sec	MB	sec	MB	sec	MB
12	18	10	4	55	11	17	7	51	60	7	9	2
13	44	13	6	60	29	20	11	56	68	8	11	2
14	109	19	11	70	37	27	20	65	287	12	15	2
15	291	31	18	86	82	40	36	81	102	10	19	2
16	711	55	43	196	207	66	80	197	411	6	6	2
17	2126	102	159	393	573	119	191	393	1701	16	45	3
18	6103	195	459	753	1857	223	422	754	302	14	58	3
19	23405	383	1491	920	5765	430	1101	817	1551	20	70	3
20	$>$17h			$>$1GB	30809	845	9136	977	1377	20	86	3
21						$>$1GB		$>$1GB	$>$40h		99	3
22											120	3
23											149	4
24											167	4

Invariant Checking: Comparison of two queue implementations.

- L is the length of the queues.
- $SMV_3 = SMV_1$ with invariant checking.
- $SMV_4 = SMV_2$ with invariant checking.

Combinational Circuit Example

Circuit	#ins	#outs	#gates	sec
Industry1	203	8	738	233
Industry2	317	232	15242	8790
Industry3	96	32	1032	210

Equivalence checking using SAT procedures: Three industrial examples from Fujitsu.

None could be done within a day using BDD based equvalence checkers!!

Model Checking with the Partial Order Reduction

- P. Godefroid. Using partial orders to improve automatic verification methods. In *Proc. 2nd Workshop on Computer Aided Verification*, volume 531 of *Lecture Notes in Computer Science*, pages 176–185, Rutgers, June 1990. Springer-Verlag.
- D. Peled. Combining partial order reductions with on-the-fly model-checking. In D. L. Dill, editor, *Proc. 1994 Workshop on Comput.-Aided Verification*, volume 818 of *LNCS*, pages 377–390. Springer-Verlag, June 1994.
- A. Valmari. A stubborn attack on state explosion. In *Proc. 2nd Workshop on Computer Aided Verification*, volume 531 of *Lecture Notes in Computer Science*, pages 156–165, Rutgers, June 1990. Springer-Verlag.

- G. J. Holzmann and D. Peled. An improvement in formal verification. In *Formal Description Techniques 1994*, pages 197–211, Bern, Switzerland, 1994. Chapman & Hall.

Asynchronous Computation

The interleaving model for asynchronous systems allows concurrent events to be ordered arbitrarily.

To avoid discriminating against any particular ordering, the events are interleaved in all possible ways.

The ordering between independent transitions is largely meaningless!!

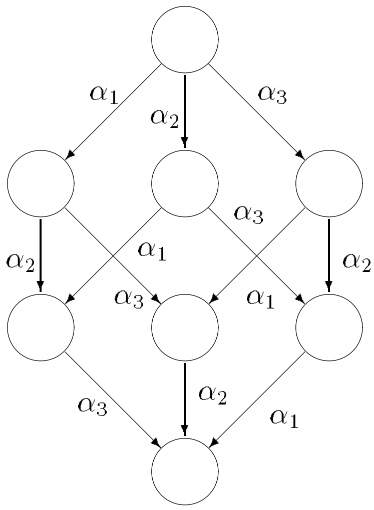
The State Explosion Problem

Allowing all possible orderings is a potential cause of the state explosion problem.

To see this, consider n transitions that can be executed concurrently.

In this case, there are $n!$ different orderings and 2^n different states (one for each subset of the transitions).

If the specification does not distinguish between these sequences, it is beneficial to consider only one with $n+1$ states.



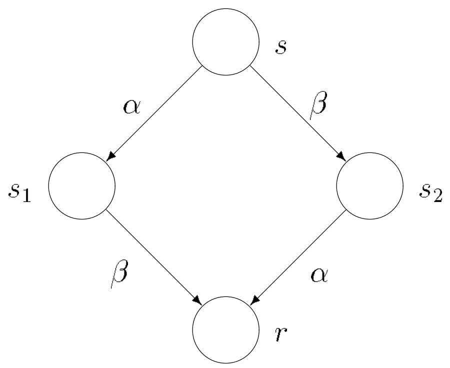
Partial Order Reduction

The partial order reduction is aimed at reducing the size of the state space that needs to be searched.

It exploits the commutativity of concurrently executed transitions, which result in the same state.

Thus, this reduction technique is best suited for asynchronous systems.

(In synchronous systems, concurrent transitions are executed simultaneously rather than being interleaved.)



Partial Order Reduction (Cont.)

The method consists of constructing a reduced state graph.

The full state graph, which may be too big to fit in memory, is never constructed.

The behaviors of the reduced graph are a subset of the behaviors of the full state graph.

The justification of the reduction method shows that the behaviors that are not present do not add any information.

Partial Order Reduction (Cont.)

The name partial order reduction comes from early versions of the algorithms that were based on the partial order model of program execution.

However, the method can be described better as model checking using representatives, since the verification is performed using representatives from the equivalence classes of behaviors.

- D. Peled. All from one, one for all: on model checking using representatives. In *Proc. 5th Workshop on Comput.-Aided Verification*, pages 409–423, 1993.

Modified Kripke Structures

The transitions of a system play a significant role in the partial order reduction.

The partial order reduction is based on the dependency relation that exists between the transitions of a system.

Thus, we modify the definition of a Kripke structure slightly.

A state transition system is a quadruple (S, T, S_0, L) where

- the set of states S, the set of initial states S_0, and the labeling function L are defined as for Kripke structures, and
- T is a set of transitions such that for each $\alpha \in T$, $\alpha \subseteq S \times S$.

A Kripke structure $M = (S, R, S_0, L)$ may be obtained by defining R so that

$$R(s, s') \Leftrightarrow \exists \alpha \in T[\, \alpha(s, s')\,].$$

Basic Definitions

A transition $\alpha \in T$ is enabled in a state s if there is a state s' such that $\alpha(s, s')$ holds.

Otherwise, α is disabled in s. The set of transitions enabled in s is $enabled(s)$.

A transition α is deterministic if for every state s there is at most one state s' such that $\alpha(s, s')$.

When α is deterministic we often write $s' = \alpha(s)$ instead of $\alpha(s, s')$.

Note:We will only consider deterministic transitions!

Basic Definitions (Cont.)

A path π from a state s_0 is a finite or infinite sequence

$$\pi = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \ldots$$

such that for every i, $\alpha_i(s_i, s_{i+1})$ holds.

Here, we do not require paths to be infinite. Moreover, any prefix of a path is also a path.

If π is finite, then the length of π is the number of transitions in π and will be denoted by $|\pi|$

Reduced State Graph

Goal is to reduce the number of states considered in model checking, while preserving the correctness of the property.

Will assume that a reduced state graph is first generated explicitly using depth-first search.

The model checking algorithm is then applied to the resulting state graph, which has fewer states and edges.

This speeds up the construction of the graph and uses less memory, thus resulting in a more efficient model checking algorithm.

- Actually, the reduction can be applied on-the-fly while doing the model checking.
- The DFS can also be replaced by breadth first search and combined with symbolic model checking.

Depth-First-Search Algorithm

```
hash(s0);
set on_stack(s0);
expand_state(s0);

procedure expand_state(s)
    work_set(s) := ample(s);
    while work_set(s) is not empty do
        let α ∈ work_set(s);
        work_set(s) := work_set(s) \ {α};
        s' := α(s);
        if new(s') then
            hash(s');
            set on_stack(s');
            expand_state(s');
        end if;
        create_edge(s, α, s');
    end while;
    set completed(s);
end procedure
```

Depth-First-Search Algorithm (Cont.)

The reduction is performed by modifying the standard DFS algorithm to construct the reduced state graph.

The search starts with an initial state s_0 (line 1) and proceeds recursively.

For each state s it selects only a subset $ample(s)$ of the enabled transitions $enabled(s)$ (in line 5).

The DFS explores only successors generated by these transitions (lines 6-16).

- The DFS algorithm constructs the reduced state graph directly.
- Constructing the full state graph and later reducing it would defy the purpose of the reduction.

Depth-First-Search Algorithm (Cont.)

When model checking is applied to the reduced state graph

- it terminates with a positive answer when the property holds for the original state graph.
- it produces a counterexample, otherwise

Note: The counterexample may differ from one obtained using the full state graph.

Ample Sets

In order to implement the algorithm we must find a systematic way of calculating $ample(s)$ for any given state s.

The calculation of $ample(s)$ needs to satisfy three goals:

1. When $ample(s)$ is used instead of $enabled(s)$, enough behaviors must be retained so DFS gives correct results.

2. Using $ample(s)$ instead of $enabled(s)$ should result in a significantly smaller state graph.

3. The overhead in calculating $ample(s)$ must be reasonably small.

Dependence and Independence

An independence relation $I \subseteq T \times T$ is a symmetric, antireflexive relation such that for $s \in S$ and $(\alpha, \beta) \in I$:

Enabledness If $\alpha,\ \beta \in enabled(s)$ then $\alpha \in enabled(\beta(s))$.

Commutativity $\alpha,\ \beta \in enabled(s)$ then $\alpha(\beta(s)) = \beta(\alpha(s))$.

The dependency relation D is the complement of I, namely

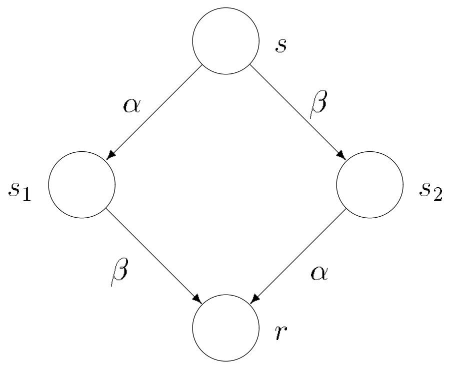
$$D = (T \times T) \setminus I.$$

Note

- The enabledness condition states that a pair of independent transitions do not disable one another.
- However, that it is possible for one to enable another.

Potential Problems

Suppose that α and β commute:



It does not matter whether α is executed before β or vice versa in order to reach the state r from s.

It is tempting to select only one of the transitions originating from s.

This is not appropriate for the following reasons:

Problem 1: The checked property might be sensitive to the choice between the states s_1 and s_2, not only the states s and r.

Problem 2: The states s_1 and s_2 may have other successors in addition to r, which may not be explored if either is eliminated.

Visible and Invisible Transitions

Let $L : S \rightarrow 2^{AP}$ be the function that labels each state with a set of atomic propositions.

A transition $\alpha \in T$ is invisible with respect to $AP' \subseteq AP$ if for each pair $s, s' \in S$ such that $s' = \alpha(s)$,

$$L(s) \cap AP' = L(s') \cap AP'.$$

Thus, a transition is invisible when its execution from any state does not change the value of the propositional variables in AP'.

A transition is visible if it is not invisible.

Stuttering Equivalence

Stuttering refers to a sequence of identically labeled states along a path in a Kripke structure.

Let σ and ρ be two infinite paths:

$$\sigma = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \ldots \text{ and } \rho = r_0 \xrightarrow{\beta_0} r_1 \xrightarrow{\beta_1} \ldots$$

Then σ and ρ are stuttering equivalent, denoted $\sigma \sim_{st} \rho$, if there are two infinite sequences of integers

$$0 = i_0 < i_1 < i_2 < \ldots \text{ and } 0 = j_0 < j_1 < j_2 < \ldots$$

such that for every $k \geq 0$,

$$L(s_{i_k}) = L(s_{i_k+1}) = \ldots = L(s_{i_{k+1}-1}) =$$
$$L(r_{j_k}) = L(r_{j_k+1}) = \ldots = L(r_{j_{k+1}-1}).$$

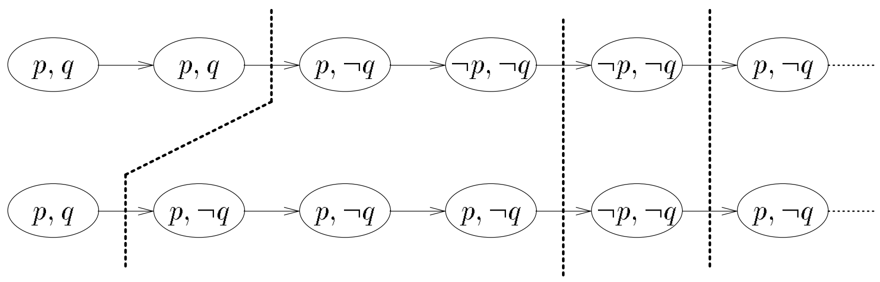
Stuttering equivalence can be defined similarly for finite paths.

Stuttering Equivalence (Cont.)

A finite sequence of identically labeled states is called a block.

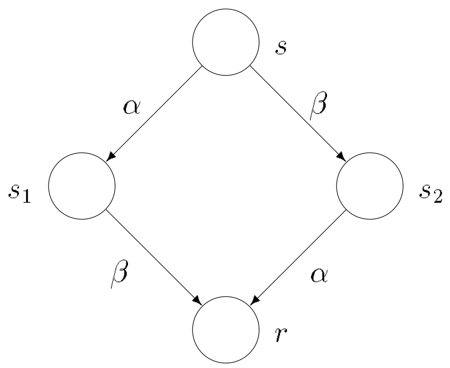
Intuitively, Two paths are stuttering equivalent if they can be partitioned into blocks, so states in the kth block of one are labeled the same as states in the kth block of the other.

Note: Corresponding blocks may have different lengths!



Stuttering Equivalence Example

Consider the diagram used to illustrate commutativity again.



Suppose that at least one transition, say α, is invisible, then $L(s) = L(s_1)$ and $L(s_2) = L(r)$.

Consequently,

$$s\,s_1\,r \sim_{st} s\,s_2\,r$$

Note: The paths $s\,s_1\,r$ and $s\,s_2\,r$ are stuttering equivalent!!

LTL and Stuttering Equivalence

An LTL formula $\mathbf{A}f$ is invariant under stuttering if and only if for each pair of paths π and π' such that $\pi \sim_{st} \pi'$,

$$\pi \models f \text{ if and only if } \pi' \models f.$$

We denote the subset of the logic LTL without the next time operator by LTL_{-X}.

Theorem. *Any* LTL_{-X} *property is invariant under stuttering.*

Stuttering Equivalent Structures

Without loss of generality, assume that M has initial state s_0 and that M' has initial state s'_0.

Then the two structures M and M' are stuttering equivalent if and only if

- For each path σ of M that starts in s_0 there is a path σ' of M' starting in s'_0 such that $\sigma \sim_{st} \sigma'$.
- For each path σ' of M' that starts in s'_0 there is a path σ of M starting in s_0 such that $\sigma' \sim_{st} \sigma$.

Corollary. *Let M and M' be two stuttering equivalent structures. Then, for every* LTL_{-X} *property* $\mathbf{A}f$

$$M,\, s_0 \models f \;\; \textit{if and only if} \;\; M',\, s'_0 \models \mathbf{A}f.$$

DFS Algorithm and Ample Sets

Commutativity and invisibility allow us to avoid generating some of the states when the specification is invariant under stuttering,

Based on this observation, it is possible to devise a systematic way of selecting an ample set for any given state.

The ample sets will be used by the DFS algorithm to construct a reduced state graph so that for every path not considered there is a stuttering equivalent path that is considered.

This guarantees that the reduced state graph is stuttering equivalent to the full state graph.

We say that state s is fully expanded when

$$ample(s) = enabled(s).$$

In this case, all of the successors of that state will be explored by the DFS algorithm.

Correctness of Reduction

Will state four conditions for selecting $ample(s) \subseteq enabled(s)$ so satisfaction of the LTL$_{-X}$ specifications is preserved.

The reduction will depend on the set of propositions AP' that appear in the LTL$_{-X}$ formula.

Condition C0 is very simple:

C0 $ample(s) = \emptyset$ if and only if $enabled(s) = \emptyset$.

Intuitively, if the state has at least one successor, then the reduced state graph also contains a successor for this state.

Correctness of Reduction (Cont.)

Condition **C1** is the most complicated constraint.

C1 Along every path in the full state graph that starts at s, the following condition holds:

A transition that is dependent on a transition in $ample(s)$ can not be executed without one in $ample(s)$ occurring first.

Note that Condition **C1** refers to paths in the full state graph.

Obviously, we need a way of checking that **C1** holds without actually constructing the full state graph.

Later, we will show how to restrict **C1** so that $ample(s)$ can be calculated based on the current state s.

Correctness of Reduction (Cont.)

Lemma. *The transitions in* $enabled(s) \setminus ample(s)$ *are all independent of those in* $ample(s)$.

Proof:

Let $\gamma \in enabled(s) \setminus ample(s)$.

Suppose that $(\gamma, \delta) \in D$, where $\delta \in ample(s)$.

Since γ is enabled in s, there is a path starting with γ in the full graph.

But then a transition dependent on some transition in $ample(s)$ is executed before a transition in $ample(s)$.

This contradicts condition **C1**. $\square$

Correctness of Reduction (Cont.)

If we always choose the next transition from $ample(s)$, we will not omit any paths that are essential for correctness.

Condition **C1** implies that such a path will have one of two forms:

- The path has a prefix $\beta_0\beta_1\ldots\beta_m\alpha$, where $\alpha \in ample(s)$ and each β_i is independent of all transitions in $ample(s)$ including α.
- The path is an infinite sequence of transitions $\beta_0\beta_1\ldots$ where each β_i is independent of all transitions in $ample(s)$.

Correctness of Reduction (Cont.)

If along a sequence of transitions $\beta_0\beta_1 \ldots \beta_m$ executed from s,

If none of the transitions in $ample(s)$ have occurred, then all the transitions in $ample(s)$ remain enabled.

This is because each β_i is independent of the transitions in $ample(s)$ and, therefore, cannot disable them.

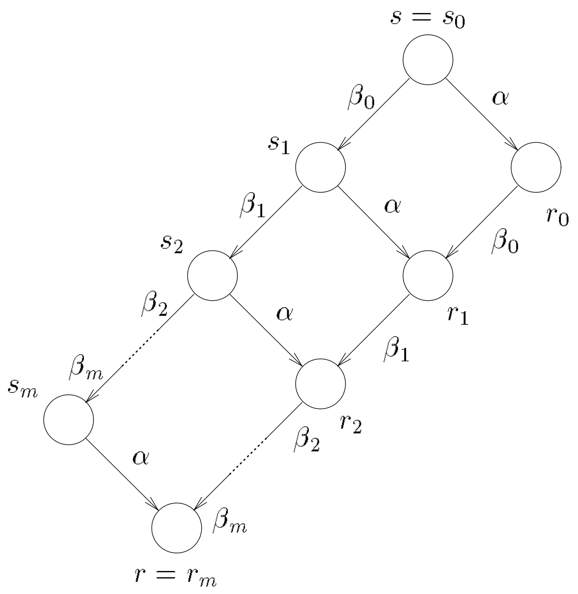
Correctness of Reduction (Cont.)

In the first case, assume that the sequence of transitions $\beta_0\beta_1 \ldots \beta_m\alpha$ reaches a state r.

This sequence will not be considered by the DFS algorithm.

By applying the enabledness and commutativity conditions m times, we can construct a sequence $\alpha\beta_0\beta_1 \ldots \beta_m$, that also reaches r.

Thus, if the reduced state graph does not contain the sequence $\beta_0\beta_1 \ldots \beta_m\alpha$ that reaches r, we can construct from s another sequence that reaches r.



Another Correctness Condition

Consider the two sequences of states:

- $\sigma = s_0 s_1 \ldots s_m r$ generated by $\beta_0 \beta_1 \ldots \beta_m \alpha$, and
- $\rho = s r_0 r_1 \ldots r_m$ generated by $\alpha \beta_0 \beta_1 \ldots \beta_m$.

In order to discard σ, we want σ and ρ to be stuttering equivalent.

This is guaranteed if α is invisible, since then $L(s_i) = L(r_i)$ for $0 \leq i \leq m$.

Thus, the checked property will not be able to distinguish between the two sequences above.

C2 If s is not fully expanded, then every $\alpha \in ample(s)$ is invisible.

- This condition is called invisibility.

Correctness of Reduction (Cont.)

Now consider the case in which an infinite path $\beta_0\beta_1\beta_2\ldots$ that starts at s does not include any transition from $ample(s)$.

By Condition **C2** all transitions in $ample(s)$ are invisible. Let α be such a transition.

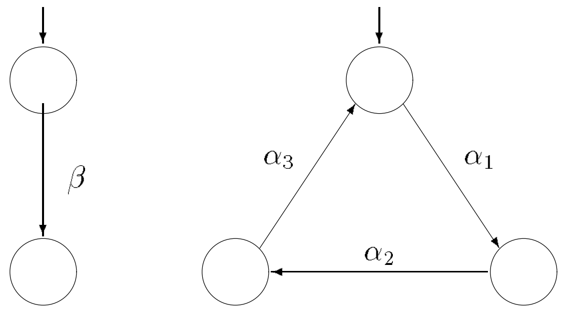
Then the path generated by the sequence $\alpha\beta_0\beta_1\beta_2\ldots$ is stuttering equivalent to the one generated by $\beta_0\beta_1\beta_2\ldots$.

Again, even though the path $\beta_0\beta_1\beta_2\ldots$ is not in the reduced state graph, a stuttering equivalent path is included.

Problem with Correctness Conditions

C1 and **C2** are not yet sufficient to guarantee that the reduced state graph is stuttering equivalent to the full state graph.

In fact, there is a possibility that some transition will actually be delayed forever because of a cycle in the constructed state graph.



Assume that β is independent of the transitions α_1, α_2 and α_3 and that α_1, α_2 and α_3 are interdependent.

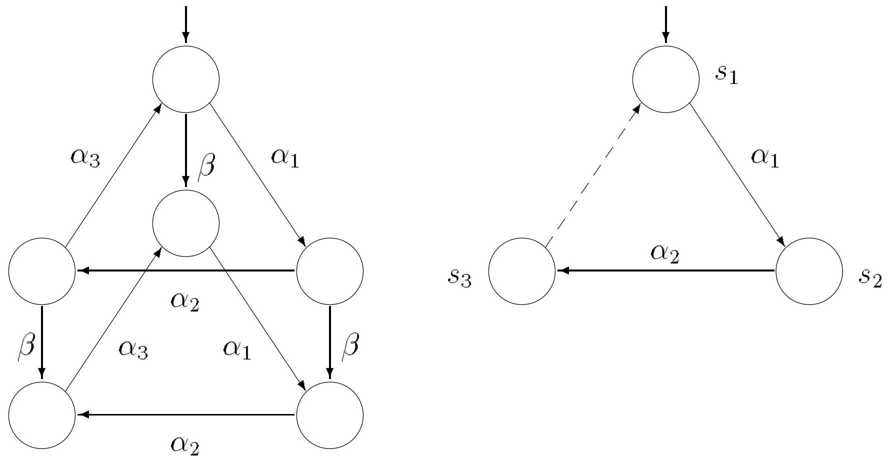
The process on the left can execute the β exactly once.

Assume there is one proposition p, which is changed from true to false by β, so that β is visible.

The process on the right performs the invisible transitions α_1, α_2 and α_3 repeatedly in a loop.

Problem with Correctness Condition (Cont.)

The full state graph of the system is shown below:



Starting with the initial state s_1, we can select $ample(s_1) = \{\alpha_1\}$ and generate $s_2 = \alpha_1(s_1)$.

Next, we can select $ample(s_2) = \{\alpha_2\}$ and generate $s_3 = \alpha_2(s_2)$.

When we reach s_3, we can select $ample(s_3) = \{\alpha_3\}$ and close the cycle (s_1, s_2, s_3).

(Easy to see that **C0**, **C1** and **C2** are satisfied at each step.)

But, the reduced state graph does not contain any sequences where p is changed from true to false!!

Cycle Closing Condition

Note that at each state on the cycle (s_1, s_2, s_3, s_1) β is deferred to a possible future state.

When the cycle is closed, the construction terminates, and transition β is ignored!!

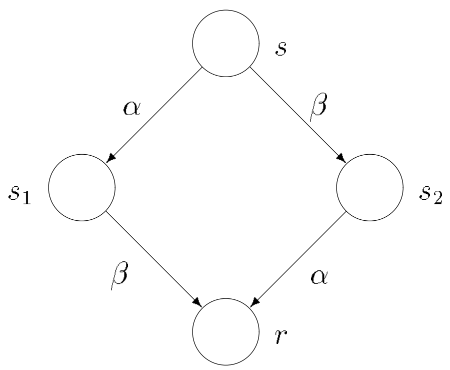
To prevent this situation from occurring we need one more condition:

C3 A cycle is not allowed if it contains a state in which some transition α is enabled, but is never included in $ample(s)$ for any state s on the cycle.

Condition **C3** is called the Cycle closing condition.

Problem 1 Again

Consider the diagram used to illustrate commutativity of actions.



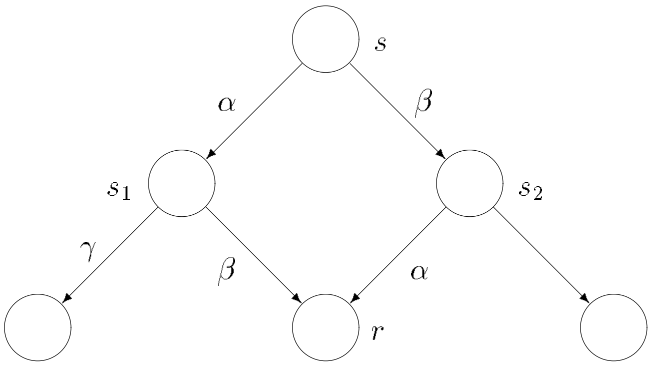
Assume that the DFS reduction algorithm chooses β as $ample(s)$ and does not include state s_1 in the reduced graph.

By Condition **C2**, β must be invisible; thus s, s_2, r and s, s_1, r are stuttering equivalent.

Since we are only interested in stuttering invariant properties, we can't distinguish between the two sequences.

Problem 2 Again

Assume that there is a transition γ enabled from s_1.



Note that γ cannot be dependent on β. Otherwise, the sequence α, γ violates **C1**.

Thus, γ is independent of β. Since it is enabled in s_1, it must also be enabled in state r.

Assume that γ, when executed from r, results in state r' and when executed from s_1 results in state s_1'.

Since β is invisible, the two state sequences s, s_1, s_1' and s, s_2, r, r' are stuttering equivalent.

Therefore, properties that are invariant under stuttering will not distinguish between the two.

Heuristics for Ample Sets

We assume that the concurrent program is composed of processes and that each process has a program counter.

- $pc_i(s)$ will denote the program counter of a process P_i in a state s.
- $pre(\alpha)$ is a set of transitions that includes the transitions whose execution may enable α.
- $dep(\alpha)$ is the set of transitions that are dependent on α.
- T_i is the set of transitions of process P_i.
- $T_i(s) = T_i \cap enabled(s)$ denotes the set of transitions of P_i that are enabled in the state s.
- $current_i(s)$ is the set of transitions of P_i that are enabled in some state s' such that $pc_i(s') = pc_i(s)$.

Heuristics for Ample Sets (Cont.)

We now describe the dependency relation for the different models of computation.

- Pairs of transitions that share a variable, which is changed by at least one of them, are dependent.
- Pairs of transitions belonging to the same process are dependent.
- Two send transitions that use the same message queue are dependent. Similarly, two receive transitions are dependent.

Note that a transition that involves handshaking or rendezvous communication as in CSP or ADA can be treated as a joint transition of both processes. Therefore, it depends on all of the transitions of both processes.

Heuristics for Ample Sets (Cont.)

An obvious candidate for $ample(s)$ is the set $T_i(s)$ of transitions enabled in s for some process P_i.

- Since the transitions in $T_i(s)$ are interdependent, an ample set for s must include either all of the transitions or none of them.
- To construct an ample set for the current state s, we start with some process P_i such that $T_i(s) \neq \emptyset$.
- We want to check whether $ample(s) = T_i(s)$ satisfies Condition **C1**.
- There are two cases in which this selection might violate **C1**.
- In both of these cases, some transitions independent of those in $T_i(s)$ are executed, eventually enabling a transition α that is dependent on $T_i(s)$.
- The independent transitions in the sequence cannot be in T_i, since all the transitions of P_i are interdependent.

Heuristics for Ample Sets (First Case)

In the first case, α belongs to some other process P_j.

A necessary condition for this to happen is that $dep(T_i(s))$ includes a transition of process P_j.

By examining the dependency relation, this condition can be checked effectively.

Heuristics for Ample Sets (Second Case)

In the second case, α belongs to P_i. Suppose that the transition $\alpha \in T_i$ which violates **C1** is executed from a state s'.

- The transitions executed on the path from s to s' are independent of $T_i(s)$ and hence, are from other processes.
- Therefore, $pc_i(s') = pc_i(s)$. So α must be in $current_i(s)$.
- In addition, $\alpha \notin T_i(s)$, otherwise it does not violate **C1**.
- Thus, $\alpha \in current_i(s) \setminus T_i(s)$.
- Since α is not in $T_i(s)$, it is disabled in s.
- Therefore, a transition in $pre(\alpha)$ must be included in the sequence from s to s'.

Thus, a necessary condition is that $pre(current_i(s) \setminus T_i(s))$ includes transitions of processes other than P_i.

This condition can also be checked effectively.

If all else fails . . .

In both cases we discard $T_i(s)$ and try the transitions $T_j(s)$ of another process j as a candidate for $ample(s)$.

Note: We take a conservative approach discarding some ample sets even though at run-time **C1** might not actually be violated.

Future Research Directions

Need to be able to handle systems with 5000 state variables routinely. This is an order of magnitude over the number that we can handle now.

- Develop automatic abstraction techniques. Abstraction is the most important technique for dealing with the state explosion problem.
- Devise more powerful compositional reasoning techniques. Currently, this usually requires considerable manual intervention.
- Find good techniques for reasoning about parameterized systems (e.g. the number of processes is a parameter).
- Develop practical symmetry reduction techniques. Many hardware and software systems are either symmetric or almost symmetric.
- Combine Model Checking and Theorem Proving. Work is just beginning on this problem. I think this will be necessary for softare verification.

Future Research Directions

- SAT procedures (like the Davis-Putnam procedure) have many advantages over BDDs. Develop model checking algorithms that use SAT procedures.
- Need practical model checking algorithms for reasoning about real-time systems, hybrid systems, and security protocols.
- Develop techniques for extracting finite state synchronization skeletons from concurrent programs. This seems feasible, but there has been little work on it until recently.
- Develop model checking algorithms for infinite state systems (including software with complex data types). This may require several of the other techniques in the list.