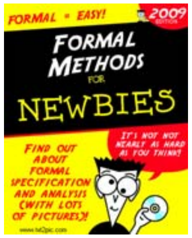
CISC422/853: Formal Methods in Software Engineering: Computer-Aided Verification



Topic 9: Optimization

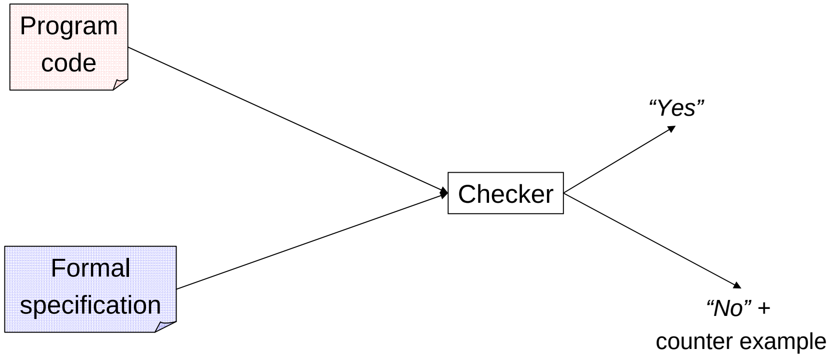
Juergen Dingel
March, 2009

Readings:
- Spin book, Chapter 9
- Handouts on control flow analysis and slicing posted on course web page

---

Where Do We Want to Be?

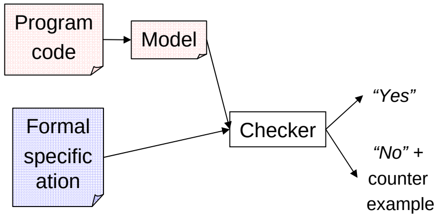
Software model checking: The Dream



How Could We Get There?

One class of approaches:

- Automatic model extraction
  - Bandera/Bogor (KSU)
  - ModEx/Spin (JPL)
  - Zing (MSR)
  - Automatic abstraction refinement
    - SLAM and SDV (MSR)
    - Blast (Berkeley and EPFL)
    - Magic (CMU)



To make this work, we need optimization!

Complexity and Optimization

- Size of $A_S \otimes A_{\neg P}$
  - R = # of reachable states in $A_S \otimes A_{\neg P}$
  - $R = R_S \cdot R_{\neg P}$ where
    - R_S = # of reachable states in A_S (typically: $10^9 \ldots 10^{11}$)
    - $R_{\neg P}$ = # of reachable states in $A_{\neg P}$ (typically: 1..4)
- Size of A_S
  - $R_S = R_{T1} \cdot \ldots \cdot R_{Tn} \sim R_T^n$
- Size of T
  - R_T = (# loc's in T) · $|dtype_1| \cdot \ldots \cdot |dtype_m|$ ~ (# loc's in T) · $|dtype|^m$
- Thus,
  - $R_S = \left(\text{(\# loc's in T)} \cdot |dtype|^m \right)^n$

R_S increases with
- # of processes n (exponentially)
- # of variables m
- size of data types
- size of process

Complexity and Optimization (Cont'd)

- Size of $A_S \otimes A_{\neg P}$
  - $R = R_S \cdot R_{\neg P} = \left((\#\text{ loc's in T}) \cdot |\text{dtype}|^m \right)^n \cdot R_{\neg P}$
- Reduce R by

reducing
- # of processes n (exponentially)
- # of variables m
- size of data type dtype
- size of process T
- size of specification P

user

using
- partial order reduction
- statement merging
- abstraction

checker/user

- Reduce memory requirement by
  - reducing size of state vector and/or seen set

Outline

- Reduce number of reachable states
  - slicing
  - partial order reduction & statement merging
- Reduce memory requirement
  - Reduce size of representation of state
    - slicing
    - compression
  - Reduce size of representation of Seen Set
    - bitstate hashing

Slicing: Motivating Example

Consider program P:

```
1: INPUT(n);
2: i := 1;
3: sum := 0;
4: prod := 1;
5: WHILE i ≤ n DO
6:    sum := sum+i;
7:    prod := prod*i;
8:    i := i+1
9: END
10: OUTPUT(sum);
11: OUTPUT(prod);
```

Suppose we are interested in proving ϕ:

$G(pc=10 \Rightarrow sum=\sum^{n}_{i=1} i)$

Then, P satisfies ϕ iff P' satisfies ϕ

Program P':

```
1: INPUT (n);
2: i := 1;
3: sum := 0;
4: prod := 1;
5: WHILE i ≤ n DO
6:    sum := sum+i;
7:    prod := prod*i;
8:    i := i+1
9: END
10: OUTPUT(sum);
11: OUTPUT(prod);
```

Statements in lines 4, 7, and 11 are irrelevant to value of sum in line 10!

Wouldn't it be nice, if …

- … given
  - a program P and
  - a line number n in P,

  we could compute and remove all statements in P that are irrelevant to the values of the variables in line n?
- This could substantially reduce
  - the number of reachable states and
  - the memory requirement (fewer variables)
- That's what slicing does! Sort of.

Definitions

Let P be a program

Definition: Slice
A *slice* S of P is an executable program that is obtained from P by deleting zero or more statements.

Definition: Slicing criterion
A *slicing criterion* consists of a pair (n, V) where n is a node in the control flow graph (CFG) of P and V is a subset of the variables in P

Definition: Slice with respect to criterion
A *slice S of P is called a slice wrt criterion (n, V),* if it contains the statement at node n and whenever P halts for a given input,
- S also halts for that input, and
- S computes the same values for the variables in V whenever the statement corresponding to the node n is executed

Definitions (Cont'd)

Definition: Minimal slice
A slice is called *minimal*, if no other slice for the same criterion contains fewer statements

Theorem: Minimal slices
1. Minimal slices are not necessarily unique
2. The problem of determining whether a given slice is minimal is undecidable

Proof:

```
1: x_new := 1;
2: C;
3: output(x_new);
```

is minimal slice of

```
1: x_new := 1;
2: C;
3: output(x_new);
```

iff C halts

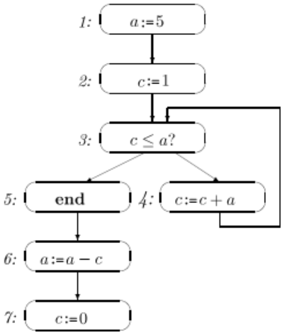
Slicing: Adjusting Expectations

- **Bad news:**
  - No general algorithm for computing minimal slices
- **The problem, intuitively:**
  - To compute minimal slices we'd have to be able to compute at compile-time the values of variables at certain locations
  - For programs with iteration or recursion this problem is as difficult as the halting problem
- **Instead:**
  - Compute only a (hopefully very good) conservative approximation to the minimal slice
    - **soundness:** the output of our slicing procedure will be a slice
    - **no optimality**: it may not be minimal
  - Use ideas from a static (compile-time) analysis technique called data flow analysis

Important Notion 0: Control Flow Graphs (CFGs)

- Graphical representation of paths through the program
- CGF(P) = (V, E) where
  - V set of locations in P
  - E ⊆ V× V, with $(l_1,l_2)\in$ E iff *"may be able to go from l_1 to l_2 in P"*
- **Example:**

```
C₁ ≡  1:  a:=5;
      2:  c:=1;
      3:  while c ≤ a do
      4:     c:=c + a
      5:  end;
      6:  a:=a − c;
      7:  c:=0
```



- **Note:** Some paths in CFG(P) may be infeasible, i.e., feasiblePaths(P) ⊆ CFG(P)

Important Notion 0: Control Flow Graphs (CFGs) (Cont'd)

- Compute CFG(P) by structural induction over P
- Suppose:

```
S ::= x:=e | S;S | if b then S else S | while b do S | …
```

- Code:

```
class Cfg {
  Node first, last;
  Cfg(Node f, Node l) {
    first = f;
    last = l
} }
```

```
class Stmt {…}
class Assign extends Stmt {…}
class Seq extends Stmt {…}
Class Cond extends Stmt {…}
Class While extends Stmt {…}
…
```

```
Cfg toCfg(Stmt s) {
  switch (P) {
    case "s ≡ x:=e ∈ Assign":
         Node n = new Node(x,e);
         return new Cfg(n,n);
    case "s ≡ s1;s2 ∈ Seq":
        Cfg cfg1 = toCfg(s1);
        Cfg cfg2 = toCfg(s2);
        link(cfg1.last, cfg2.first);
        return new Cfg(cfg1.first, cfg2.last);
    case "s ≡ if b then s1 else s2 ∈ Cond":
        …
    case "s ≡ while b do s ∈ Cond":
        …
    case …
} }
```

Slicing Linear IMPerative (LIMP) Programs

Suppose we want to slice the LIMP program P_1 below wrt criterion C = (6,{z}).

```
1:      y := 0;
2:      a := 1;
3:      x := w+1;
4:      z := 2;
5:      z := x+y;
6:      OUTPUT(z);
```

Definition: **Slice with respect to criterion**

A *slice S of P is called a slice wrt criterion (n, V),* if it contains the statement at node n and whenever P halts for a given input,

- S also halts for that input, and
- S computes the same values for the variables in V whenever the statement corresponding to the node n is executed

What is a slice of P_1 wrt C?

What did you do to compute it?

Is it minimal?

Important Notion 1: Directly Relevant Variables

Intuition: **Directly Relevant Variables $R^0_C(i)$**

- Let i be node in the CFG of a program P and let C = (n, V) be a slicing criterion.
- We say that a variable v is *directly relevant* at i wrt C, if the value of v right before execution of i may influence the value of at least one variable in V at node n.
- Variable v is *not directly relevant* at i wrt C, if the value of v right before execution of i can never influence the value of any of the variables in V at node n.

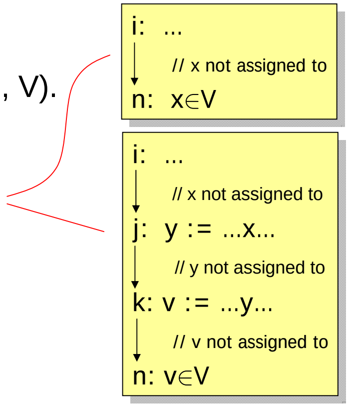
Directly Relevant Variables (Cont'd)

Let P be program with node i and C = (n, V).

$x \in R^0_C(i)$ if either

- p: $i \rightarrow^*_{CFG(P)} n \wedge$ *"x flows into"* some v ∈ V along p, or

"There is a data flow dependency between x at node i and one of the variables in V."



i: ...

// x not assigned to

n: x∈V

i: ...

// x not assigned to

j: y := ...x...

// y not assigned to

k: v := ...y...

// v not assigned to

n: v∈V

Important Notion 2: Directly Relevant Statements

Intuition: Directly Relevant Statements S^0_C

Given a program P with node i and a criterion C. The statement at node i is *directly relevant* in P wrt C, if

- i is an assignment x := e and x is directly relevant at least one successor of i wrt C.

Directly Relevant Statements Are All We Need

Theorem:

Given a LIMP program P and a criterion C = (n, V), the set of directly relevant statements S^0_C together with n forms a slice of P wrt. C

Program P_1		$R^0_C(i)$	S^0_C
1:	y := 0;	1:	1:
2:	a := 1;	2:	2:
3:	x := w+1;	3:	3:
4:	z := 2;	4:	4:
5:	z := x+y;	5:	5:
6:	**OUTPUT**(z);	6:	6:

C = (6, {z})

Slicing Branching LIMP (BLIMP) Programs

Suppose we want to slice the BLIMP program P_2 below wrt criterion C = (6,{z}).

```
1:      x := 1;
2:      IF x>0 THEN
3:              z := 1;
4:              w := 2
        ELSE
5:              z := z+y
6:      END;
7:      OUTPUT(z);
```

Definition: Slice with respect to criterion

A *slice S of P is called a slice wrt criterion (n, V),* if it contains the statement at node n and whenever P halts for a given input,

- S also halts for that input, and
- S computes the same values for the variables in V whenever the statement corresponding to the node n is executed

What is a slice of P_2 wrt C?

What did you do to compute it?

Is it minimal?

Important Notion 3: Relevant Branching Statements

Definition: Relevant Branching Statements B_C

- Let i be node in the CFG of a program P and let C = (n, V) be a slicing criterion.
- i is a relevant branching statement in P wrt C iff
  - i is the test node of a conditional

    IF b **THEN** P_1 **END** or **IF** b **THEN** P_1 **ELSE** P_2 **END**

    such that either P_1 or P_2 contain at least one directly relevant statement

Directly Relevant Variables: Revised

Let P be program with node i and C = (n, V).

$x \in R^0_C(i)$ if either

- p: $i \rightarrow^*_{CFG(P)} n \wedge$ *"x flows into"* some $v \in V$ along p, or
- p: $i \rightarrow^*_{CFG(P)} j \wedge j$ is test node b of relevant branching statement

  IF b **THEN** P_1 **ELSE** P_2 **END** or

  IF b **THEN** P_1 **END**

  such that x is read in b

"p: $i \rightarrow^*_{CFG(P)}$ condition node of relevant branching statement and value of x at i may determine truth of condition", i.e., *"there is a control flow dependency between x at node i and the statements inside the branching statement"*

```
i:  ...
 |      // x not assigned to
 v
n:  x∈V
```

```
i:  ...
 |      // x not assigned to
 v
j:  y := ...x...
 |      // y not assigned to
 v
k: v := ...y...
 |      // v not assigned to
 v
n: v∈V
```

```
i: ...
 |  // x not assigned to
 v
j:  IF ...x... THEN
  \
   v
 k:      y := ...
 k+1:  y∈Rc(k+1)
```

Directly Relevant Statements Are All We Need

and Relevant Branching Statements

Theorem:

Given a BLIMP program P and a criterion C = (n, V), the set of directly relevant statements S^0_C together with n form a slice of P wrt. C

Program P_2

```
1:      x := 1;
2:      IF x>0 THEN
3:              z := 1;
4:              w := 2
        ELSE
5:              z := z+y
6:      END;
7:      OUTPUT(z);
```

$R^0_C(i)$

```
1:
2:
3:
4:

5:
6:
7:
```

$S^0_C \cup B_C$

```
1:
2:
3:
4:

5:
6:
7:
```

Slicing Imperative (IMP) Programs

Suppose we want to slice the IMP program P_3 below wrt criterion C = (8,{z}).

```
1:      w := u+3;
2:      v := 1;
3:      WHILE w>0 DO
4:              y := x;
5:              z := y;
6:              t := t+1
7:      END;
8:      OUTPUT(z);
```

What is a slice of P_3 wrt C?

What did you do to compute it?

Is it minimal?

Relevant Branching Statements (Revised)

Definition: Relevant Branching Statements B_C

- Let i be node in the CFG of a program P and let C = (n, V) be a slicing criterion.
- i is a *relevant branching statement* in P wrt C iff
  - i is the test node of a conditional

    IF b **THEN** P_1 **END** or **IF** b **THEN** P_1 **ELSE** P_2 **END**

    such that either P_1 or P_2 contain at least one directly relevant statement, or
  - i is the test node of a iteration

    WHILE b **DO** P **END** or **REPEAT** P **UNTIL** b

    such that P contains at least one directly relevant statement

Slicing Imperative (IMP) Programs (Cont'd)

Suppose we want to slice the IMP programs P_4 below wrt criterion C = (8,{z}).

```
1:     w := u+3;
2:     v := 1;
3:     WHILE w>0 DO
4:          y := x;
5:          z := y;
6:          w := w+v
7:     END;
8:     OUTPUT(z);
```

What is a slice of P_4 wrt C?

What did you do to compute it?

Is it minimal?

- Now, one backward pass over the program is not enough anymore to compute the slice!
- We may have to iterate:
  - Discover new dir. rel. var.
  - ⇒ discover new dir. rel. stmt
  - ⇒ discover new dir. rel.var.
  - ⇒ and so on…
- Until …
  - … a fixed point is reached!

Slicing Imperative (IMP) Programs (Cont'd)

Worst case:

```
0:     WHILE b DO
1:       x_1 := x_2;
2:       x_2 := x_3;
3:       x_3 := x_4;
…        …
n-1:     x_{n-1} := x_n
n:     END;
n+1:   OUTPUT(x_1);
```

- $O(n_v \times n_n \times n_e)$

Divide Slicing Into Two Phases: Phase 1

1. **Computation of B^0_C and S^0_C**
   - Compute directly relevant variables R^0_C, that is, compute variables that may influence variables in
     - the criterion, or
     - tests in relevant branching statements,

     while ignoring loops (back edges in the CFG)
   - Use R^0_C to compute B_C and S^0_C
   - Needed: Single backwards pass over the program

Divide Slicing Into Two Phases: Phase 2

2. **Computation of $B^{>0}_C$ and $S^{>0}_C$**
   - Compute relevant variables $R^{>0}_C$, that is, compute variables that may influence variables in
     - the criterion, or
     - tests in relevant branching statements

     while also considering loops (back edges in the CFG)
   - Use $R^{>0}_C$ to compute $B^{>0}_C$ and $S^{>0}_C$
   - Needed:
     - Fixed point iteration until $R^{>0}_C(i)$ stabilizes for all i
     - Iterated backwards pass over the program

Important Notion 4: Relevant Variables $R_C^{>0}$

Let P be program with node i and C = (n, V).

$x \in R_C^{>0}(i)$ if either

- p: $i \rightarrow^*_{CFG(P}$ n ∧ "x flows into" some v ∈ V along p, or
- p: $i \rightarrow^*_{CFG(P)}$ j ∧ j is test node of relevant branching statement

  IF b **THEN** P_1 **ELSE** P_2 **END** or

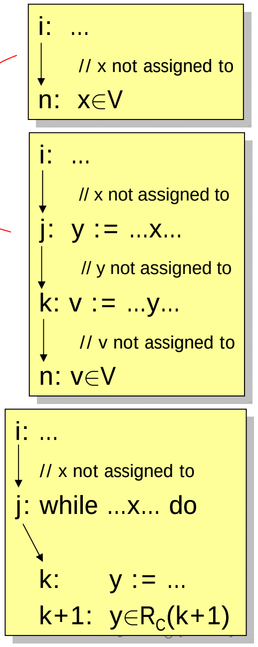
  IF b **THEN** P_1 **END** or

  WHILE b **DO** P **END** or

  REPEAT P **UNTIL** b

  such that x is read in b

$x \in R^0_C(i)$



```
i:  ...
 |      // x not assigned to
 v
n:  x∈V
```

```
i:  ...
 |      // x not assigned to
 v
j:  y := ...x...
 |      // y not assigned to
 v
k: v := ...y...
 |      // v not assigned to
 v
n: v∈V
```

```
i: ...
 |   // x not assigned to
 v
j: while ...x... do
  \
   v
   k:      y := ...
   k+1:  y∈R_C(k+1)
```

Important Notion 5: Relevant Statements $S_C^{>0}$

Definition: Relevant statements

Given a program P with node i and a criterion C.

The statement at node i is a relevant statement ($i \in S_C^{>0}$) iff

- i is an assignment to a variable that is relevant at a successor of i
- i is a relevant branching statement ($i \in B_C$)

Relevant Statements Are All We Need

Theorem:

Given a IMP program P and a criterion C = (n, V), the set of relevant statements $S^{>0}_C$ forms a slice of P wrt. C

Program P_4	$R^0_C(i)$	$R^{>0}_C(i)$	$S^{>0}_C \cup B_C$
1: w := u+3;	1:	1:	1:
2: v := 1;	2:	2:	2:
3: **WHILE** w>0 **DO**	3:	3:	3:
4: y := x;	4:	4:	4:
5: z := y;	5:	5:	5:
6: w := w+v;	6:	6:	6:
7: **END**;	7:	7:	7:
8: **OUTPUT**(z);	8:	8:	8:

C = (8, {z})

Slicing As Data Flow Analysis

- There are many data flow analyses
  - live/dead variables
  - reaching definitions
  - alias analysis, …
- All of them can be expressed in terms of data flow equations, that is, equations that describe the relevant information at each node
- We now want to do the same for slicing
- The need for a fixed point iteration (that allows the relevant information to properly propagate) during implementation will manifest itself in these equations in that the equations will be mutually recursive!

Directly Relevant Variables

Definition 1: **Directly Relevant Variables $R^0_C(i)$**

$$R^0_C(i) = \begin{cases} V, & \text{if } i{=}n \\ \{v \mid \exists j.\, i \to_{CFG} j \wedge (v \in R^0_C(j) \wedge v \notin Def(i)) \\ \quad \vee (v \in Use(i) \wedge Def(i) \cap R^0_C(j) \neq \emptyset)\}, & \text{otherwise} \end{cases}$$

$v \in R^0_C(i)$ iff

Case 1:

"v is criterion variable and i is criterion node"

Case 2:

"v directly relevant at successor j and not assigned to in i"

i: y := … or i: x>3
j: … v …

Case 3:

"v used in i and i defines (assigns) variable directly relevant at successor j"

i: x := … v …
j: … x …

Directly Relevant Statements

Definition 2: **Directly Relevant Statements S^0_C**

$$S^0_C = \{i \mid \exists j.\, i \to_{CFG} j \wedge Def(i) \cap R^0_C(j) \neq \emptyset\}$$

"Statement i is directly relevant at i if it defines (assigns) a variable which is directly relevant at a successor of i"

Example:

	Directly relevant variables wrt (6, {x, y})	Directly relevant statements wrt (6, {x, y})
1: w := 0;	1: {y, z}	1: w := 0;
2: **if** x=r **then**	2: {z, w, y}	2:
3: y := y+1;	3: {z, w, y}	3: y := y+1;
else		
4: z := 0;	4: {w, y}	4: z := 0;
5: x := z+w;	5: {z, w, y}	5: x := z+w;
6: **output**(x);	6: {x, y}	6:

Relevant Branch Statements and Relevant Variables

Definition 3: **Relevant Branch Statements B_C**

$$B_C = \{b \mid b \text{ is branch statement and has at least one node } i \in S^{>0}_C \text{ in its scope}\}$$

Definition 4: **Relevant Variables $R_C^{>0}$**

$$R^{>0}_C(i) = R^0_C(i) \cup \bigcup_{b \in B_C} R^0_{b,Use(b)}(i)$$

notice change in subscript here

"v is relevant at i wrt C if either

- *v is directly relevant at i wrt C, or*
  v is directly relevant at i wrt (b, Use(b)) for some relevant branch statement b"

Relevant Statements

Definition 5: **Relevant Statements $S_C^{>0}$**

$$S^{>0}_C = B_C \cup \{i \mid \exists j.\, i \to_{CFG} j \wedge Def(i) \cap R^{>0}_C(j) \neq \emptyset\}$$

"A statement is relevant at i if either

- *it is a relevant branching statement, or*
- *it defines a variable relevant at a successor j of i"*

Definition 1: Directly Relevant Variables $R^0_C(i)$

$$R^0_C(i) = \begin{cases} V, & \text{if } i{=}n \\ \{v \mid \exists j. i \rightarrow_{CFG} j \wedge (v \in R^0_C(j) \wedge v \notin Def(i)) \\ \quad \vee (v \in Use(i) \wedge Def(i) \cap R^0_C(j) \neq \emptyset)\}, & \text{otherwise} \end{cases}$$

- A **solution** to this system of equations will be
  - a fixed point, and
  - a slice.
- The **smallest solution** will be
  - the smallest fixed point, and
  - an approximation of the minimal slice

$$R^{>0}_C(i) = R^0_C(i) \cup \bigcup_{b \in B_C} R^0_{b,Use(b)}(i)$$

Definition 5: Relevant Statements $S_C^{>0}$

$$S^{>0}_C = B_C \cup \{i \mid \exists j. i \rightarrow_{CFG} j \wedge Def(i) \cap R^{>0}_C(j) \neq \emptyset\}$$

Fixed Point Equations and How to Solve Them

- **Example:**
  - Let G=(V,→) be graph with vertex m∈V
  - Let $R_m \subseteq V$ be the set of all vertices reachable from m
- **Describe reachability recursively:**
  - Let F be F : V → V such that F(X) = {m} ∪ X ∪ {n∈V | ∃o∈X. o→n}
- **Note:**
  - R_m is solution to

    X = F(X)

    i.e., R_m is fixed point of F.

    Intuitively, *"R_m is closed under F".*
  - But, F has more than one fixed point! Which are the others?
- **So:**
  - Computing R_m is equivalent to finding the smallest fixed point of F

Fixed Point Equations and How to Solve Them (Cont'd)

- **Theorem:**

  Whenever
  - F is a monotone function, i.e., X ⊆ F(X) for all X
  - "Solution space" finite (in example, V is largest potential solution)

  then,
  - fixed point of F can be found through "fixed point" iteration

```
X0 = smallest value in solution space;
i = 0;
Repeat
      i = i+1;
      Xi = F(Xi-1)
until Xi = Xi-1;
output Xi;
```

- **Back to example:** What is correct initial value to compute R_m?

Definition 1: Directly Relevant Variables $R^0_C(i)$

$$R^0_C(i) = \begin{cases} V, & \text{if } i{=}n \\ \{v \mid \exists j. i \rightarrow_{CFG} j \wedge (v \in R^0_C(j) \wedge v \notin Def(i)) \end{cases}$$

- Slice is smallest fixed point to this set of equations
- **Question:**

  Can use fixed point iteration to compute approximation of minimal slice?
- **Answer:**

  Yes, because
  - all functions involved are monotone
  - solution space is finite

$$S^{>0}_C = B_C \cup \{i \mid \exists j. i \rightarrow_{CFG} j \wedge Def(i) \cap R^{>0}_C(j) \neq \emptyset\}$$

Slicing Algorithm

```
1.  input program P and criterion C=(n, V)
2.  $R_C^{>0}(n) := V$ and mark n as relevant
3.  forall $i \in CFG(P)$ with $i \neq n$
    1.  $R_C^{>0}(i) := \emptyset$ and mark i as not relevant
4.  WL := $pred_{CFG(P)}(n)$
5.  while WL $\neq \emptyset$ do
    1.  i, WL := head(WL)
    2.  compute $R_C^{>0}(i)$ using $R_C^{>0}(j)$ for all $j \in succ_{CFG(P)}(i)$:
        case i of
            ▪ skip or print: $R_C^{>0}(i) := R_C^{>0}(i) \cup \bigcup_{j \in succ(i)} R_C^{>0}(j)$
            ▪ assignment x:=e:
                ▪ if $x \in R_C^{>0}(j)$ for at least one $j \in succ_{CFG(P)}(i)$, then
                    $R_C^{>0}(i) := (R_C^{>0}(i) - \{x\}) \cup read(e)$
                    mark i as relevant
                ▪ else, $R_C^{>0}(i) := R_C^{>0}(i) \cup \bigcup_{j \in succ(i)} R_C^{>0}(j)$
            ▪ test node b of if b then C end, or if b then C1 else C2 end
                ▪ $R_C^{>0}(i) := R_C^{>0}(i) \cup \bigcup_{j \in succ(i)} R_C^{>0}(j)$
                ▪ if at least one relevant statement in C, then
                  $R_C^{>0}(i) := R_C^{>0}(i) \cup read(b)$ and mark i as relevant
            ▪ …
    3.  If Step 2) changed $R_C^{>0}(i)$, then WL := WL + $pred_{CFG(P)}(i)$
6.  output relevant statements in P
```

Closing Words on Slicing

- Slicing has first been proposed in by Mark Weiser in 1979
- Complexity: O($n_v \times n_n \times n_e$)
- Sophisticated graph-based data structures (program dependence graphs) have since been devised for the implementation of slicers

---

Closing Words on Slicing (Cont'd)

- Many different versions and extensions of slicing have since been proposed
  - Backward slicing (as discussed):
    - determine which statements may influence the criterion
    - **uses**: e.g., debugging
  - Forward slicing:
    - determine which statements may be influenced by the criterion
    - **uses:** e.g., impact analysis
  - Dynamic slicing:
    - take program input into account to increase precision of slice
  - Slicing in the presence of:
    - procedures/methods, inheritance, references and aliasing, concurrency

---

Closing Words on Slicing (Cont'd)

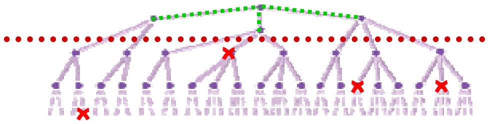
- Slicing has found many applications in all areas in which it's useful to reduce program size
  - E.g., program understanding, maintenance, analysis, debugging
- Most advanced commercial software development tools support some form of slicing (e.g., CodeSurfer from Grammatech, **`www.grammatech.com/products/codesurfer/index.html`**)
- Spin also implements slicing

More Optimizations to Come

- Reduce size of state representation
  - (Static) State compression
    - Huffman encoding
    - Collapse compression
- Reduce size of representation of “seen set”
  - Bit state hashing
- Reduce size of state space
  - Partial order reduction
  - Statement merging
- But, first: To something completely different

Optimizations: Possible Consequences

- Consider depth-bounded search again:



⇒ search incomplete

⇒ may overlook bugs

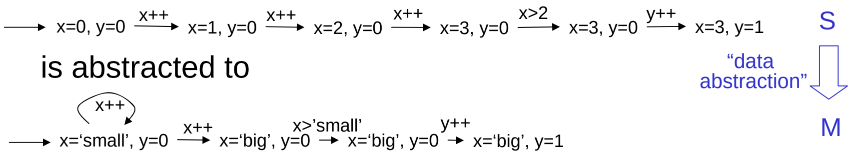
⇒ analysis result may be a “false positive”

Definition: False positive analysis results
A “No violations found” analysis of system S is a false positive iff S contains violations

---

Optimizations: Possible Consequences (Cont’d)

- Suppose the following iFSM



$\longrightarrow$ x=0, y=0 $\xrightarrow{x++}$ x=1, y=0 $\xrightarrow{x++}$ x=2, y=0 $\xrightarrow{x++}$ x=3, y=0 $\xrightarrow{x>2}$ x=3, y=0 $\xrightarrow{y++}$ x=3, y=1 S

is abstracted to

“data abstraction”

$\longrightarrow$ x=‘small’, y=0 (x++) $\xrightarrow{x++}$ x=‘big’, y=0 $\xrightarrow{x>\text{'small'}}$ x=‘big’, y=0 $\xrightarrow{y++}$ x=‘big’, y=1 M

⇒ S and M don’t satisfy the same properties (Examples?)

⇒ analysis of M reports violations that are not violations in S

⇒ analysis result of M may be a “false negative”

Definition: False negative analysis results
An analysis of system S returning “Violation found” with counter example e is a false negative iff e does not constitute a violation

---

Optimizations: Initial Summary

	Depth-bounded Search	Data Abstraction	Slicing	State Compression	Bitstate Hashing	Partial Order Reduction
Reduce size of state space						
Reduce size of states						
Reduce size of seen set						
Precision when used w/ MC?						

Optimizations: Initial Summary (Cont'd)

	Depth-bounded Search	Data Abstraction	Slicing	State Compression	Bitstate Hashing	Partial Order Reduction
Reduce size of state space	X					
Reduce size of states						
Reduce size of seen set						
Precision when used w/ MC?	incomplete (false positives possible)					

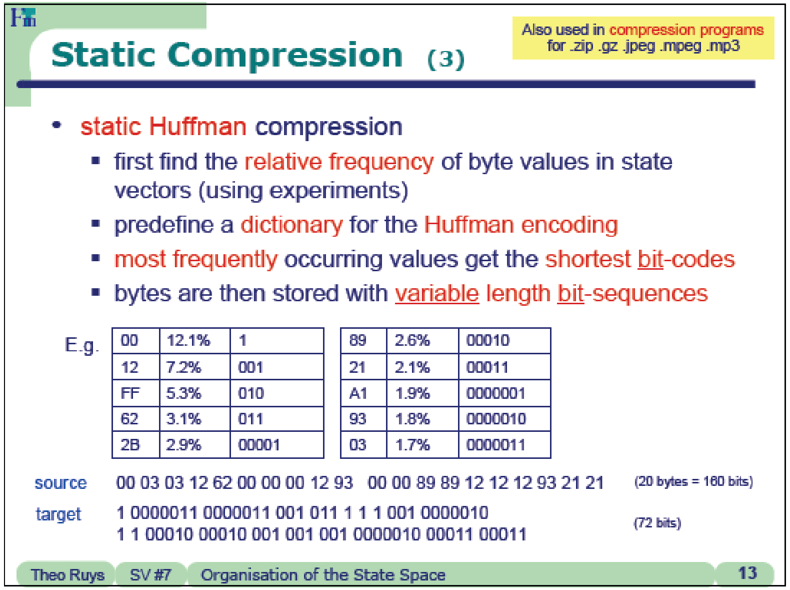
Optimizations: Initial Summary (Cont'd)

	Depth-bounded Search	Data Abstraction	Slicing	State Compression	Bitstate Hashing	Partial Order Reduction
Reduce size of state space	X	X				
Reduce size of states		X				
Reduce size of seen set						
Precision when used w/ MC?	incomplete (false positives possible)	lossy (false negatives possible)				

Optimizations: Initial Summary (Cont'd)

	Depth-bounded Search	Data Abstraction	Slicing	State Compression	Bitstate Hashing	Partial Order Reduction
Reduce size of state space	X	X	X			
Reduce size of states		X	X			
Reduce size of seen set						
Precision when used w/ MC?	incomplete (false positives possible)	lossy (false negatives possible)	precise			

State Compression



Static Compression (3)

Also used in compression programs for .zip .gz .jpeg .mpeg .mp3

- static Huffman compression
  - first find the relative frequency of byte values in state vectors (using experiments)
  - predefine a dictionary for the Huffman encoding
  - most frequently occurring values get the shortest bit-codes
  - bytes are then stored with variable length bit-sequences

E.g.

00	12.1%	1
12	7.2%	001
FF	5.3%	010
62	3.1%	011
2B	2.9%	00001

89	2.6%	00010
21	2.1%	00011
A1	1.9%	0000001
93	1.8%	0000010
03	1.7%	0000011

source 00 03 03 12 62 00 00 00 12 93 00 00 89 89 12 12 12 93 21 21 (20 bytes = 160 bits)

target 1 0000011 0000011 001 011 1 1 1 001 0000010
1 1 00010 00010 001 001 001 0000010 00011 00011 (72 bits)

Theo Ruys SV #7 Organisation of the State Space 13

State Compression (Cont’d)

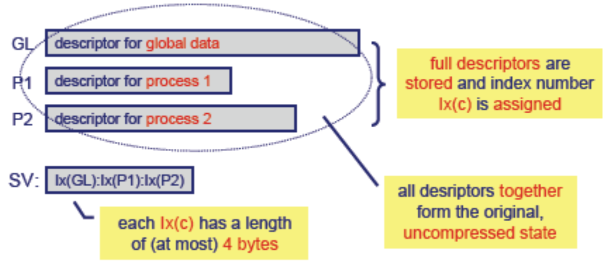
Collapse Compression (1)

- Observation
  - number of distinct system states grows very fast
  - despite the fact that each process and each data object can typically reach only a small number of distinct states
  - explosion of the number of reachable states is caused by the large number of ways in which local states of individual components can be combined.
  - replicating a complete description of all local components is therefore an inherently wasteful technique

State Compression (Cont’d)

Collapse Compression (2)

- Possible organisation of components
  - global component: all global data
  - one component for each process, recording its control state together with the state of all its local variables



full descriptors are stored and index number Ix(c) is assigned

all desriptors together form the original, uncompressed state

each Ix(c) has a length of (at most) 4 bytes

State Compression (Cont’d)

Collapse Compression (3)

- Example:

Given the following tables with partial components:

	globals
01	00 01 39 89 90 12 30 40
02	12 38 45 24 91 30 29 80
03	21 84 92 23 78 63 54 12
04	31 00 29 56 23 91 80 22
05	49 23 24 89 12 34 26 19
06	79 23 11 91 53 91 23 78

	P1
01	00 01 11
02	12 38 45
03	78 63 23

	P2
01	31 00 91 80 22
02	49 23 34 26 19
03	79 23 78 26 19

	P3
01	30 40
02	12 38
03	54 12
04	31 22
05	49 23

Now the state: $\overbrace{79\ 23\ 11\ 91\ 53\ 91\ 23\ 78}^{\text{globals}}\ \overbrace{12\ 38\ 45}^{\text{P1}}\ \overbrace{79\ 23\ 78\ 26\ 19}^{\text{P2}}\ \overbrace{12\ 38}^{\text{P3}}$

is represented by: 06 02 03 02

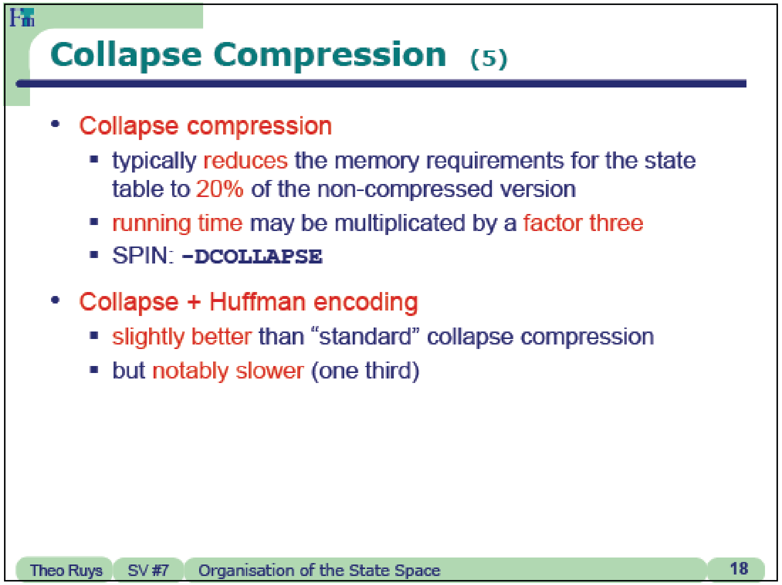
State Compression (Cont’d)

Collapse Compression (4)

- Huffman encoding and collapse compression can even be combined:
  - first perform a training run, which does not need to be complete (e.g. using bitstate hashing)
  - gather statistics on the state descriptors, i.e. the frequency of the distinct components
  - use the statistics to build a custom Huffman dictionary for the components

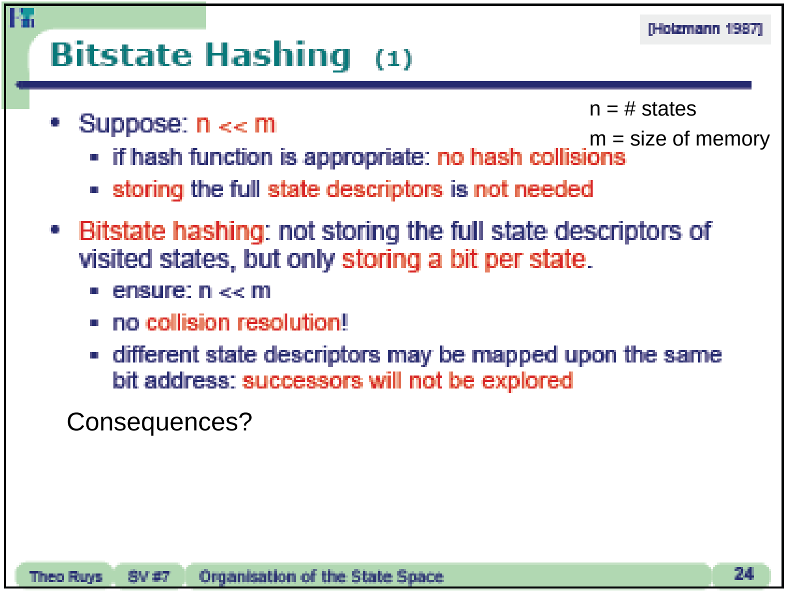
    again, assigning the most frequently occurring values the shortest bit-codes.
  - use the Huffman indices in the final verification run

State Compression (Cont'd)



Collapse Compression (5)

- Collapse compression
  - typically reduces the memory requirements for the state table to 20% of the non-compressed version
  - running time may be multiplicated by a factor three
  - SPIN: **`-DCOLLAPSE`**
- Collapse + Huffman encoding
  - slightly better than "standard" collapse compression
  - but notably slower (one third)



[Holzmann 1987]

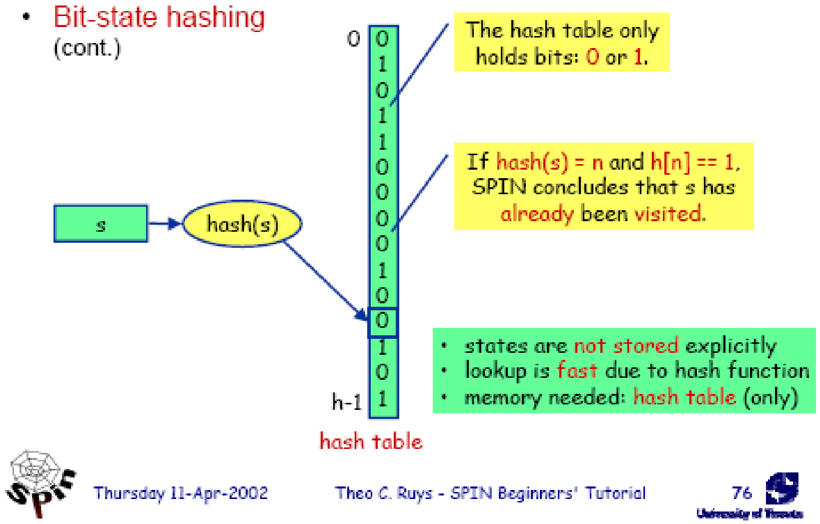
Bitstate Hashing (1)

n = # states

m = size of memory

- Suppose: n << m
  - if hash function is appropriate: no hash collisions
  - storing the full state descriptors is not needed
- Bitstate hashing: not storing the full state descriptors of visited states, but only storing a bit per state.
  - ensure: n << m
  - no collision resolution!
  - different state descriptors may be mapped upon the same bit address: successors will not be explored

Consequences?

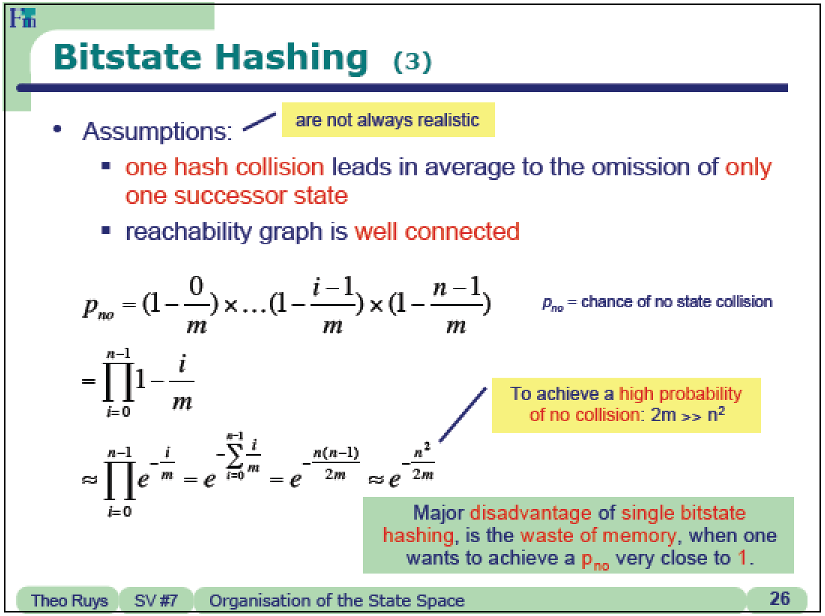


- Bit-state hashing (cont.)

The hash table only holds bits: 0 or 1.

If hash(s) = n and h[n] == 1, SPIN concludes that s has already been visited.

- states are not stored explicitly
- lookup is fast due to hash function
- memory needed: hash table (only)



Bitstate Hashing (3)

- Assumptions: are not always realistic
  - one hash collision leads in average to the omission of only one successor state
  - reachability graph is well connected

$$P_{no} = (1-\frac{0}{m})\times\ldots(1-\frac{i-1}{m})\times(1-\frac{n-1}{m})$$

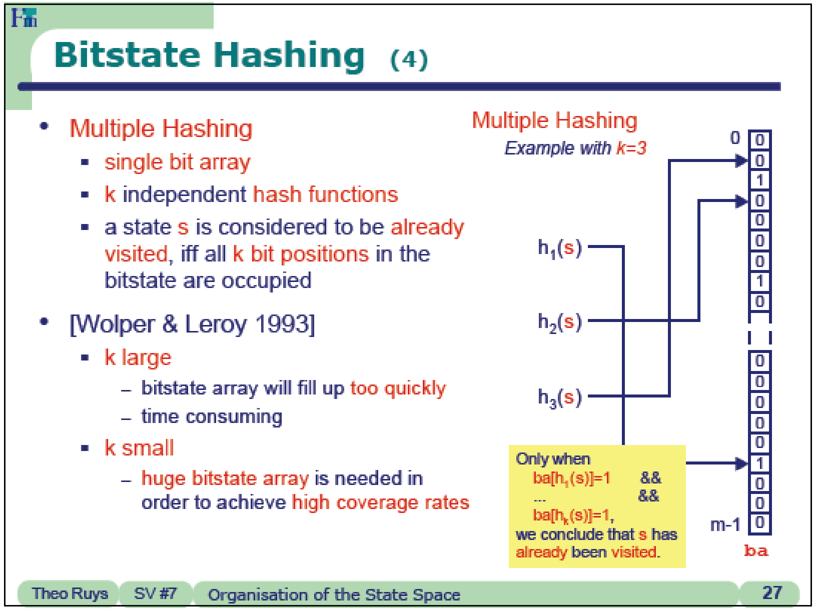
P_{no} = chance of no state collision

$$= \prod_{i=0}^{n-1} 1-\frac{i}{m}$$

$$\approx \prod_{i=0}^{n-1} e^{-\frac{i}{m}} = e^{-\sum_{i=0}^{n-1}\frac{i}{m}} = e^{-\frac{n(n-1)}{2m}} \approx e^{-\frac{n^2}{2m}}$$

To achieve a high probability of no collision: $2m >> n^2$

Major disadvantage of single bitstate hashing, is the waste of memory, when one wants to achieve a p_{no} very close to 1.

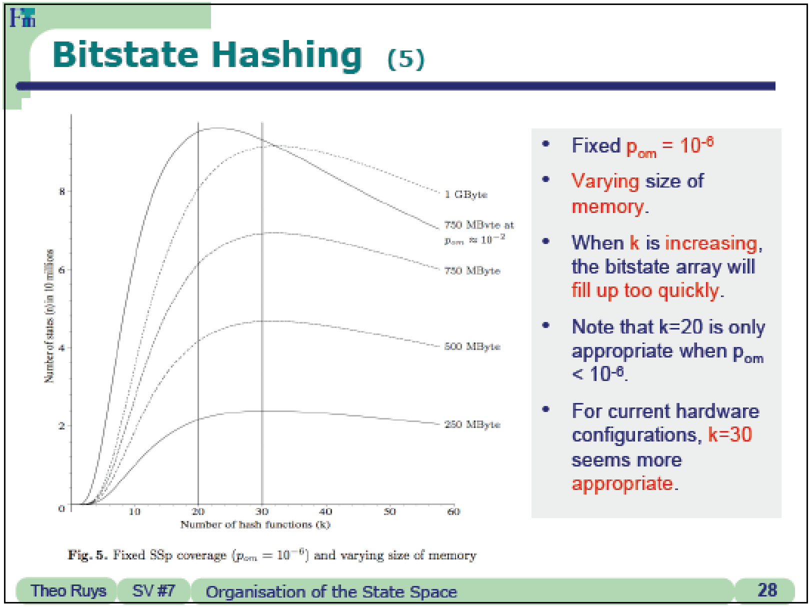


Bitstate Hashing (4)

- Multiple Hashing
  - single bit array
  - k independent hash functions
  - a state s is considered to be already visited, iff all k bit positions in the bitstate are occupied
- [Wolper & Leroy 1993]
  - k large
    - bitstate array will fill up too quickly
    - time consuming
  - k small
    - huge bitstate array is needed in order to achieve high coverage rates

Multiple Hashing
Example with k=3

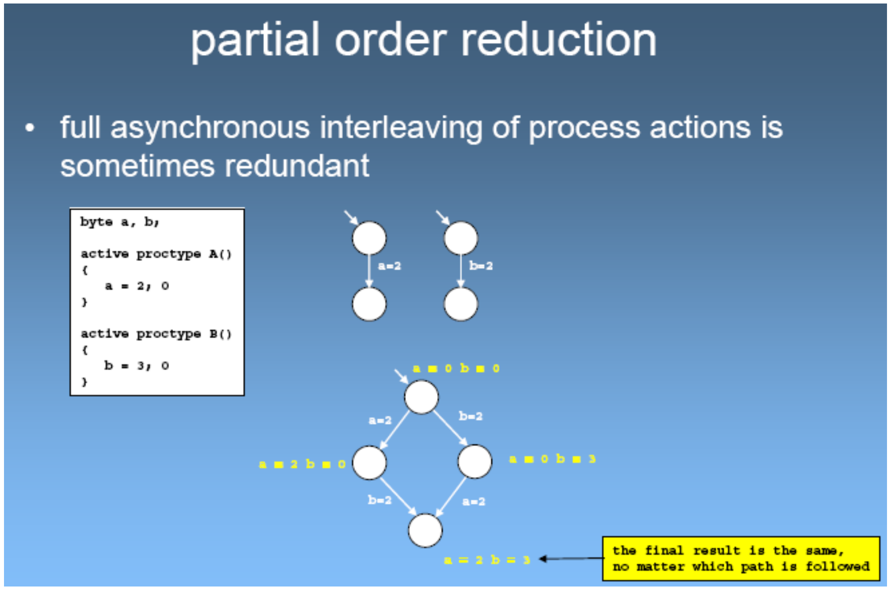
Only when
ba[h_1(s)]=1 &&
... &&
ba[h_k(s)]=1,
we conclude that s has already been visited.



Bitstate Hashing (5)

Fig. 5. Fixed SSp coverage ($p_{om} = 10^{-6}$) and varying size of memory

- Fixed $p_{om} = 10^{-6}$
- Varying size of memory.
- When k is increasing, the bitstate array will fill up too quickly.
- Note that k=20 is only appropriate when $p_{om} < 10^{-6}$.
- For current hardware configurations, k=30 seems more appropriate.



partial order reduction

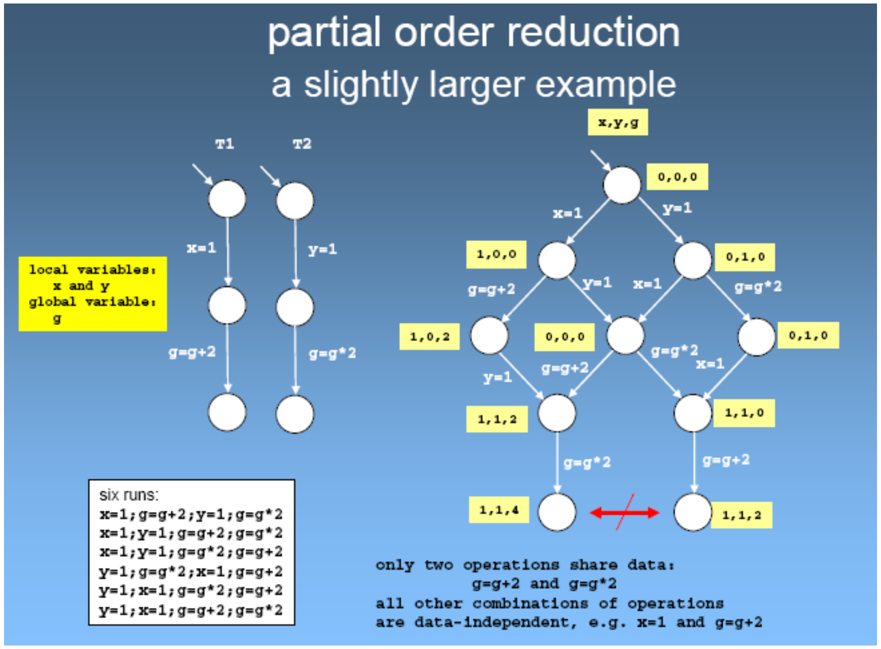
- full asynchronous interleaving of process actions is sometimes redundant

```
byte a, b;

active proctype A()
{
   a = 2; 0
}

active proctype B()
{
   b = 3; 0
}
```

the final result is the same,
no matter which path is followed



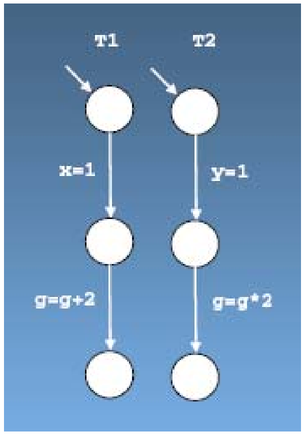
partial order reduction
a slightly larger example

local variables:
x and y
global variable:
g

six runs:
x=1;g=g+2;y=1;g=g*2
x=1;y=1;g=g+2;g=g*2
x=1;y=1;g=g*2;g=g+2
y=1;g=g*2;x=1;g=g+2
y=1;x=1;g=g*2;g=g+2
y=1;x=1;g=g+2;g=g*2

only two operations share data:
g=g+2 and g=g*2
all other combinations of operations
are data-independent, e.g. x=1 and g=g+2

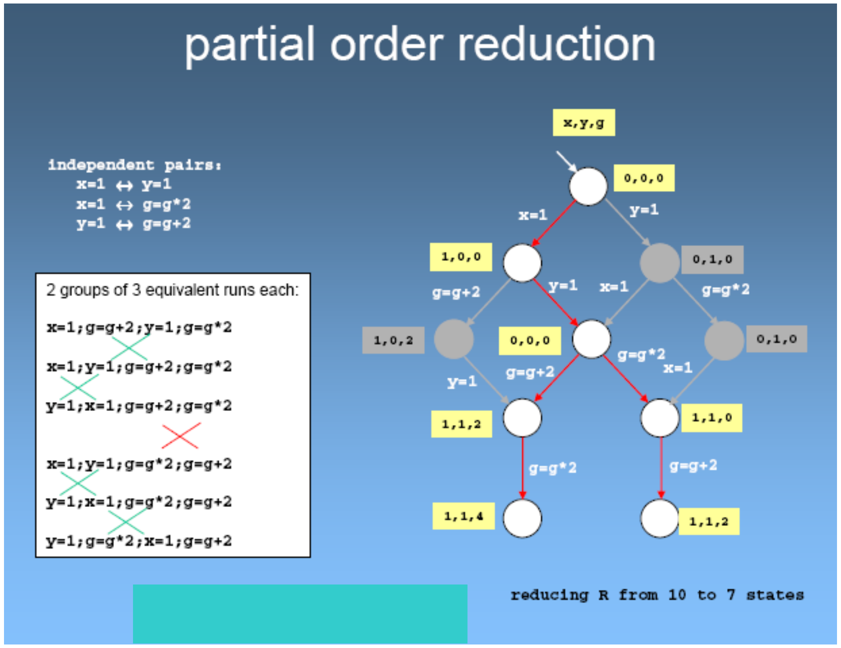
Control and Data Dependence



	x=1	y=1	g=g+2	g=g*2
x=1		Indep	Control	Indep
y=1	Indep		Indep	Control
g=g+2	Control	Indep		Data
g=g*2	Indep	Control	Data	

Runs that differ only in the order of independent actions can be considered equivalent

---



partial order reduction

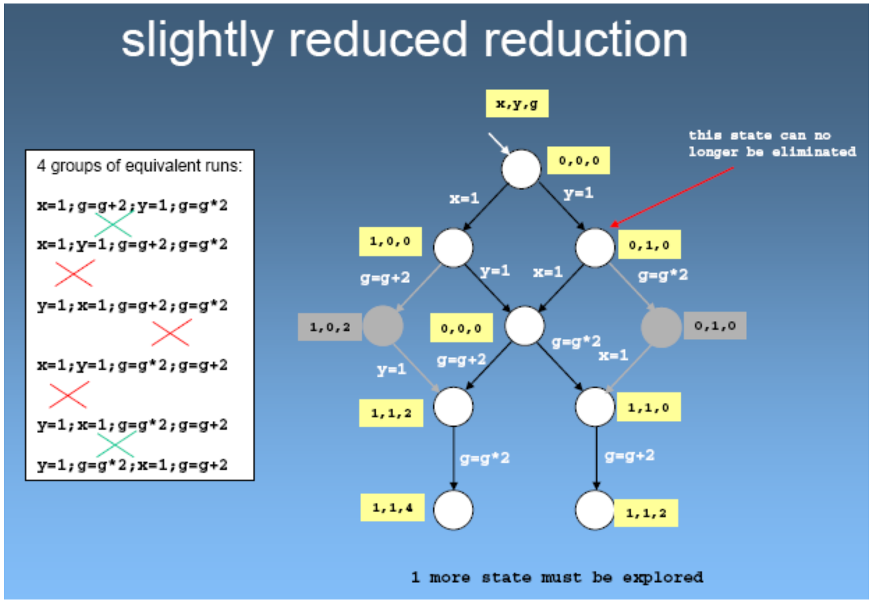
independent pairs:
x=1 ↔ y=1
x=1 ↔ g=g*2
y=1 ↔ g=g+2

2 groups of 3 equivalent runs each:

```
x=1;g=g+2;y=1;g=g*2
x=1;y=1;g=g+2;g=g*2
y=1;x=1;g=g+2;g=g*2

x=1;y=1;g=g*2;g=g+2
y=1;x=1;g=g*2;g=g+2
y=1;g=g*2;x=1;g=g+2
```

reducing R from 10 to 7 states



slightly reduced reduction

4 groups of equivalent runs:

```
x=1;g=g+2;y=1;g=g*2
x=1;y=1;g=g+2;g=g*2

y=1;x=1;g=g+2;g=g*2

x=1;y=1;g=g*2;g=g+2

y=1;x=1;g=g*2;g=g+2
y=1;g=g*2;x=1;g=g+2
```

this state can no longer be eliminated

1 more state must be explored

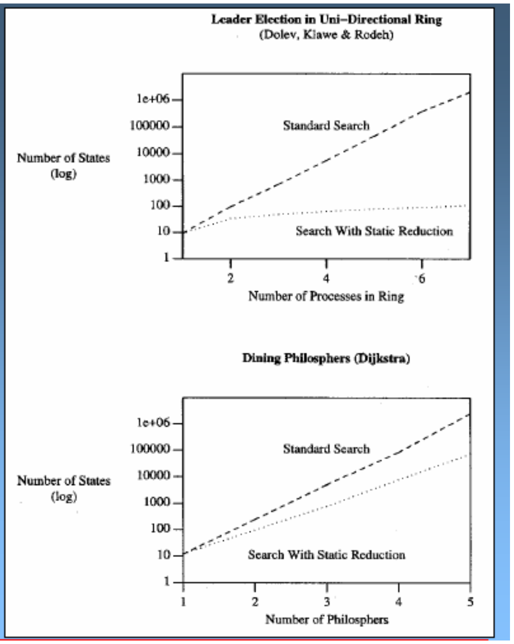
Independent Transitions

- Two transitions are independent at state s if
  - both are enabled at s
  - the execution of neither can disable the other
  - the combined effect of both transitions is independent of the relative order of execution
- Two transitions are strongly independent if they are independent at every state where both are enabled
- Spin (and other model checkers) use a syntactic condition checkable at compile-time to conservatively approximate strongly independent transitions
  - no overhead at run-time
  - reduction preserves all safety and liveness properties
  - even this conservative reduction can still lead to an exponential reduction in the size of the reachable state space

effect of partial order reduction

best case

worst case



Partial Order Reduction May Cause Incompleteness

- POR not compatible with
  - LTL’s next time operator X
  - rendezvous message passing and weak fairness
  - a small set of language constructs in some cases such as _last, enabled, remote references
- Spin’s analysis will be sound, but may be incomplete in these cases
- Spin will automatically detect incompatibility and either issue a warning or abort search

---

statement merging (default mode)

form of partial order reduction

E.g., when x, y, z are process-local

a sequence of unconditionally safe, non-blocking, transitions:

x = 1;
x = y+z;

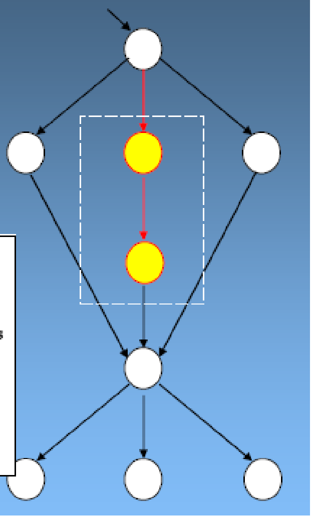
predictably produces a non-interleaved run of states in the global graph

the intermediate states in such sub-graphs are redundant and can be omitted

we can accomplish that effect by merging sequences of unconditionally safe transitions into a single transition (similar to d_step)

savings in memory and time

default in Spin
(can be disabled with spin -a -o3 …)



Optimizations: Final Summary

	Depth-bounded Search	Data Abstraction	Slicing	State Compression	Bitstate Hashing	Partial Order Reduction
Reduce size of state space	X	X	X			X
Reduce size of states		X	X	X		
Reduce size of seen set					X	
Precision when used w/ MC?	incomplete (false positives possible)	lossy (false negatives possible)	precise	precise	lossy (false positives possible)	precise (except for LTL w/ X)