ON-THE-FLY DYNAMIC DEAD VARIABLE ANALYSIS

Eric Mercer
Joel Self

Software Model Checking Lab
Brigham Young University

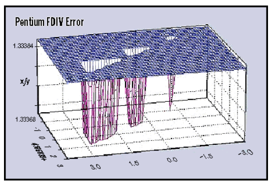
SPIN 2007
Berlin, Deutchland

Patriot Missile Disaster

February 25, 1991 a patriot missile failed to intercept an incoming Iraqi scud missile killing 28 solders in a military barrack.

Computer System Glitches

- 1991 Patriot missile clock drift



- 1994 Intel FPDIV
- 1996 Ariane 5 Rocket Explosion

- 2000 I Love You Virus
- 2004 BMW Engine Stall

- 2006 Utah voting machines fail
- 2006 Segway Destabilize

- 2007 Encryption broken Blu-Ray HD DVD

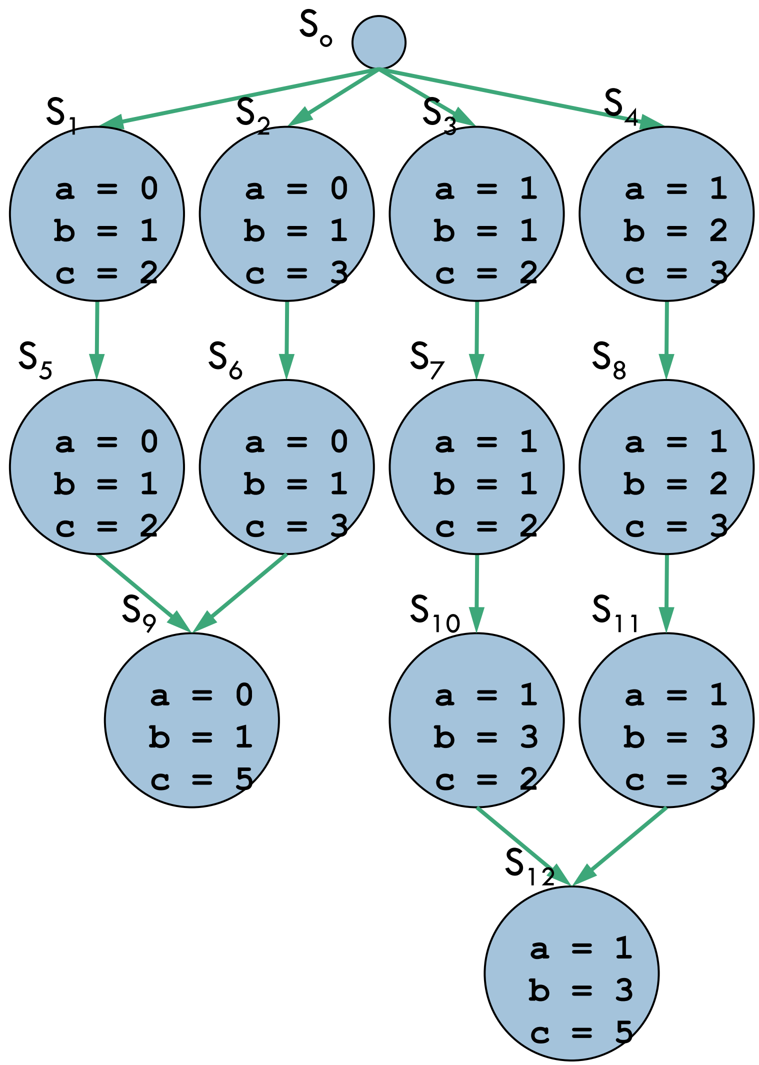
More Glitches

- 02/2007 F-22 Raptor system crash
- 02/2007 Subliminal slot machine messages
- 02/2007 Dow Jones plunge

What Can Be Done?

- Software errors may be small, but catastrophic
- Traditional testing will miss these small errors
- Model checking can help find subtle errors
- Model checking takes a system and a specification
  - Exhaustively enumerates all behaviors
  - Checks if behaviors meet or violate the specification
- Result is a proof

Example State Space




```
f(int a, int b, int c)
        if(a > 0) then
          b = 3;
   c = 5;
   print a,b,c; assert c != 7
end
```

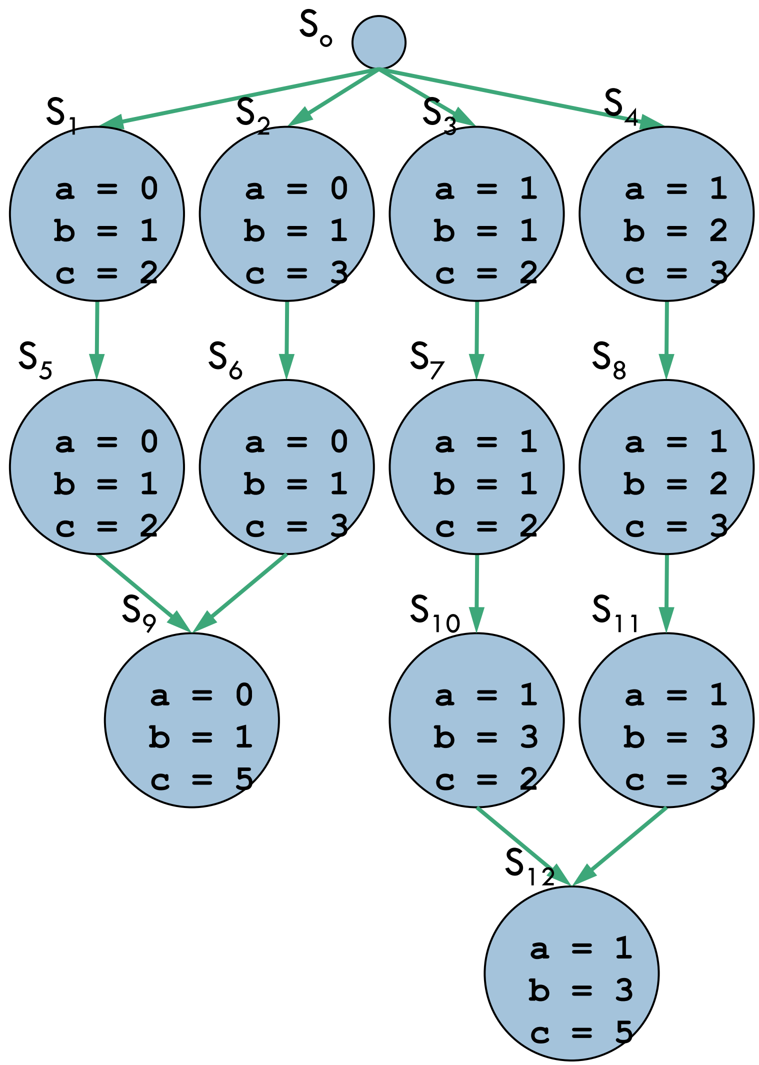
Problem With Model Checking

- Size of state space can be prohibitive
- 32 bit integer = 2^32 or around 4 billion values
- Data abstraction can help
- Represent many values with fewer values
- Dead variable analysis
- Precise data abstraction
- Requires no theorem prover
- Builds on known static analysis techniques

Dead Variable Analysis (DVA)

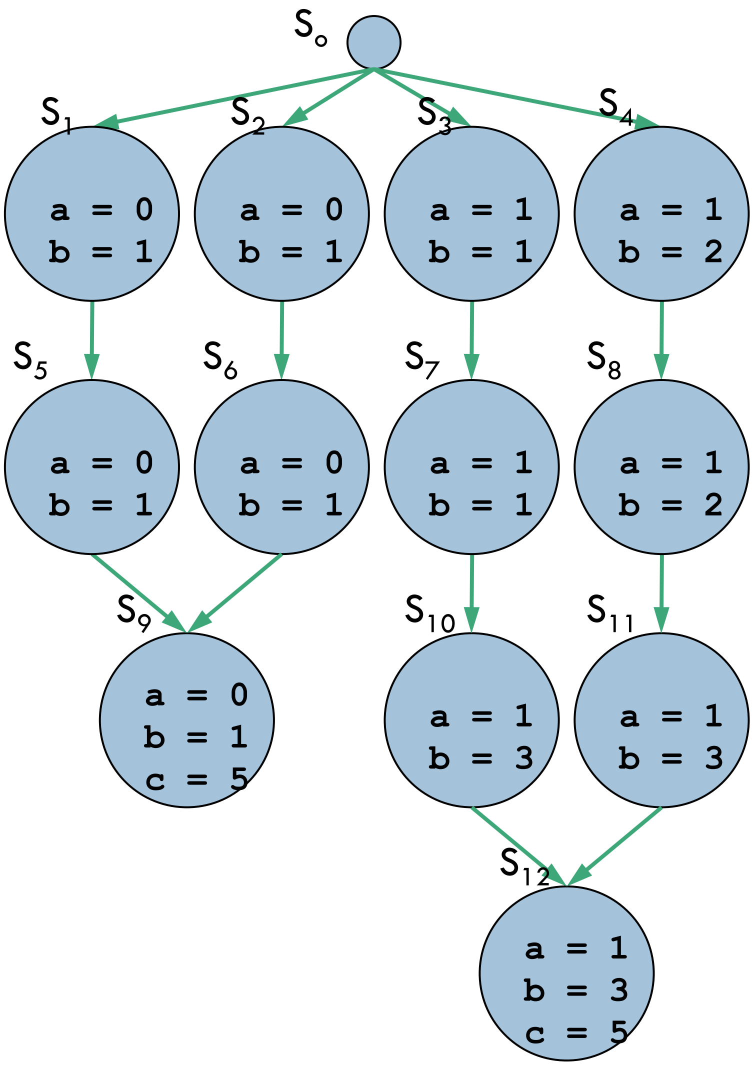
- Label variables live or dead at a location
  - Live = current valuation will be used in some future
  - Dead = current valuation will not be used in any future
- Dead variables do not affect program behavior
- We ignore these valuations
- Dead variable values do not distinguish states

Static Dead Variable Analysis (SDVA)



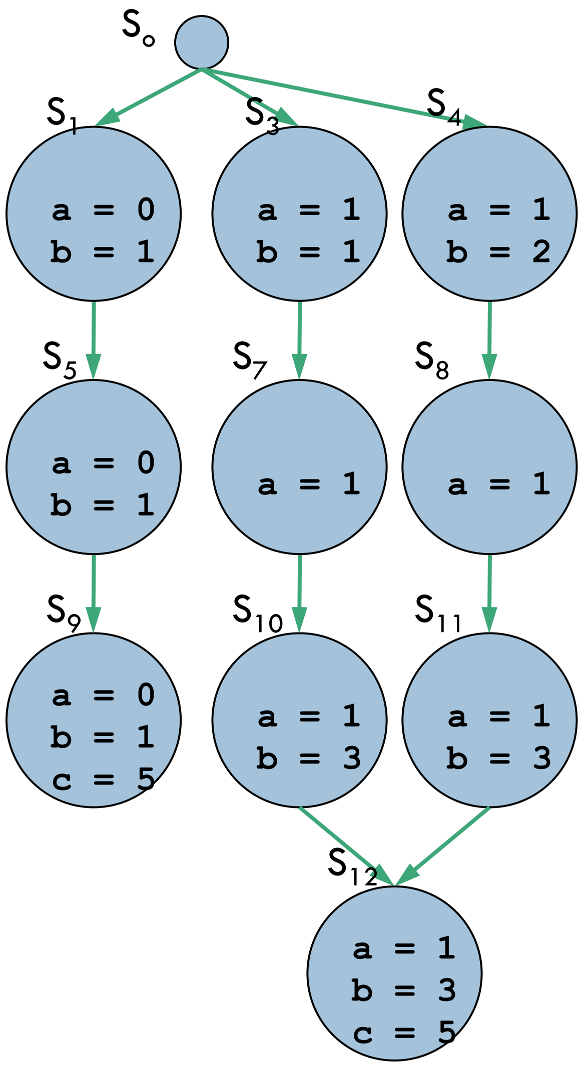

```
f(int a, int b, int c)
        if(a > 0) then
            b = 3;
    c = 5;
    print a, b, c;
end
```

Static Dead Variable Analysis



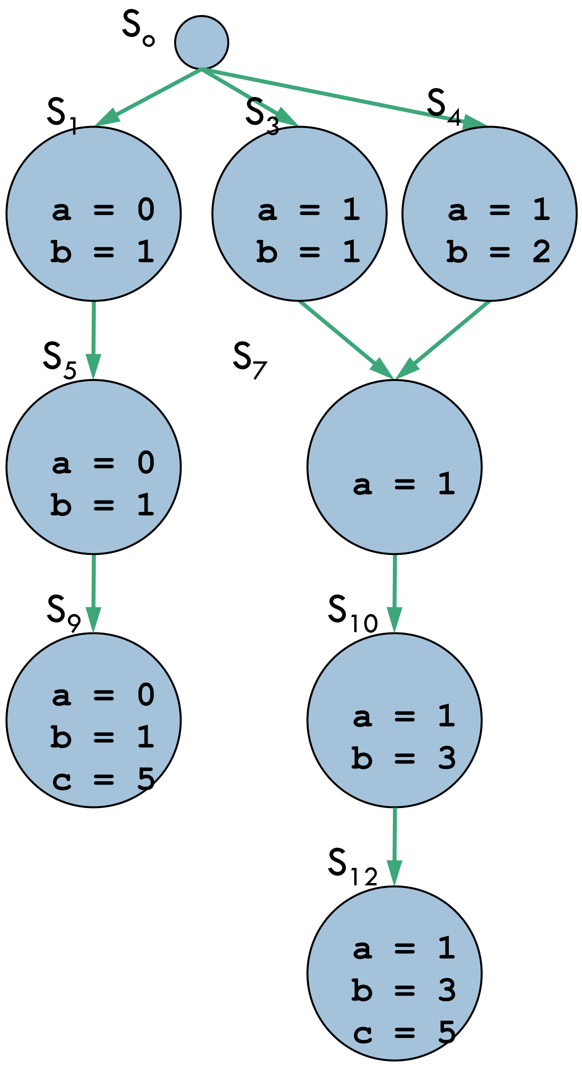

```
f(int a, int b, int c)
        if(a > 0) then
          b = 3;
    c = 5;
    print a, b, c;
end
```

Static Dead Variable Analysis




```
1: f(int a, int b, int c)
2:         if(a > 0) then
3:           b = 3;
4:    c = 5;
5:    print a, b, c;
6: end
```

Static Dead Variable Analysis

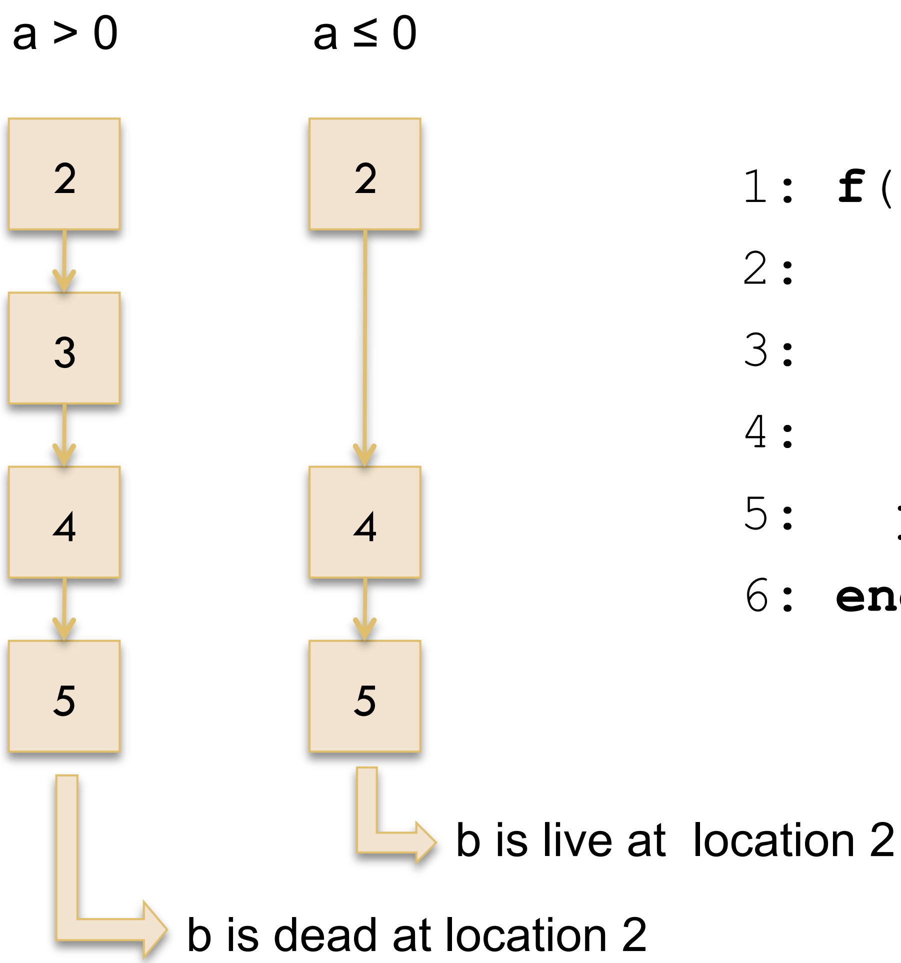



```
1: f(int a, int b, int c)
2:         if(a > 0) then
3:           b = 3;
4:    c = 5;
5:    print a, b, c;
6: end
```

Related Work

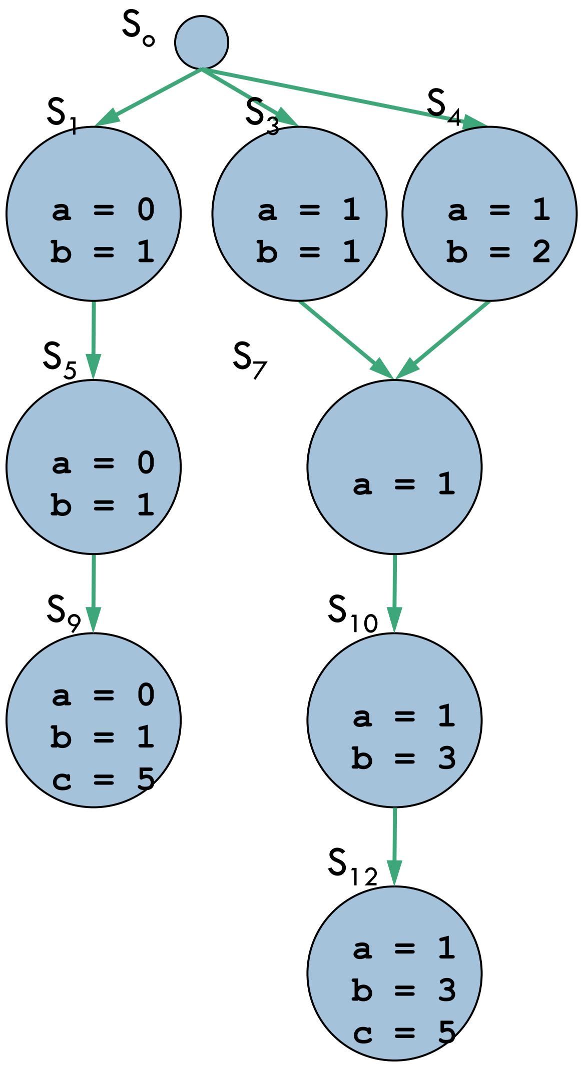
- M. Bozga and J. Fernandez and L. Ghirvu, *State Space Reduction Based on Live Variables Analysis*, 1999
- K. Yorav and O. Grumberg, *Static Analysis for State-Space Reductions Preserving Temporal Logics*, 2004
- M. S. Lewis and M. D. Jones, *A Dead Variable Analysis for Explicit Model Checking*, 2006

Feasible Paths



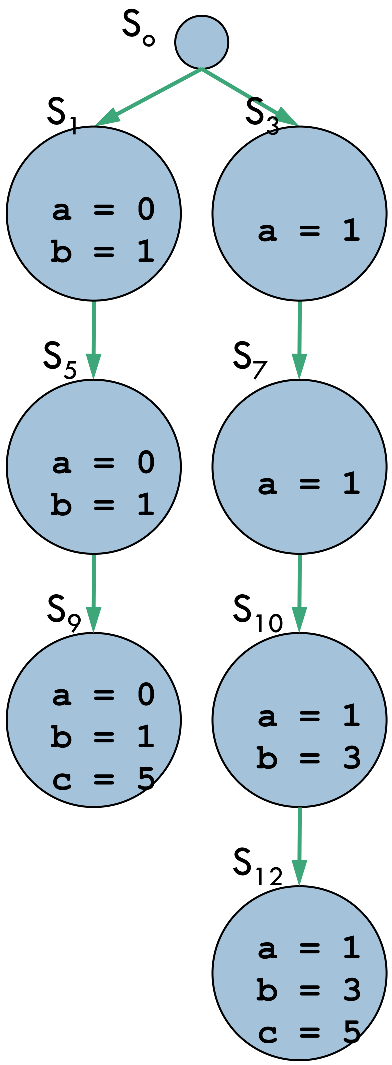

```
1: f(int a, int b, int c)
2:         if(a > 0) then
3:           b = 3;
4:    c = 5;
5:    print a, b, c;
6: end
```

Dynamic Dead Variable Analysis



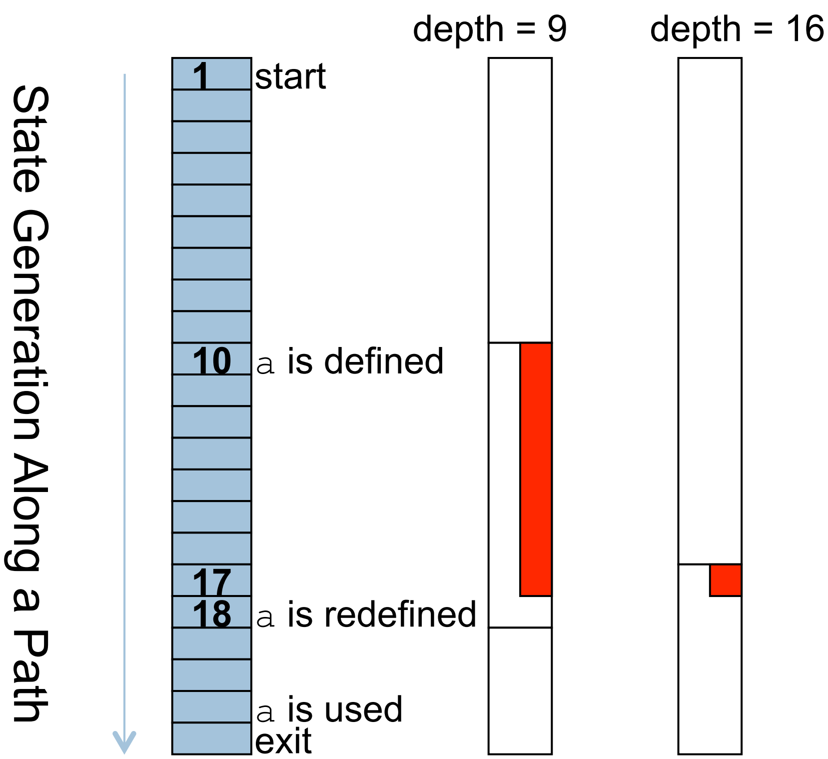

```
f(int a, int b, int c)
        if(a > 0) then
          b = 3;
   c = 5;
   print a, b, c;
end
```

Dynamic Dead Variable Analysis



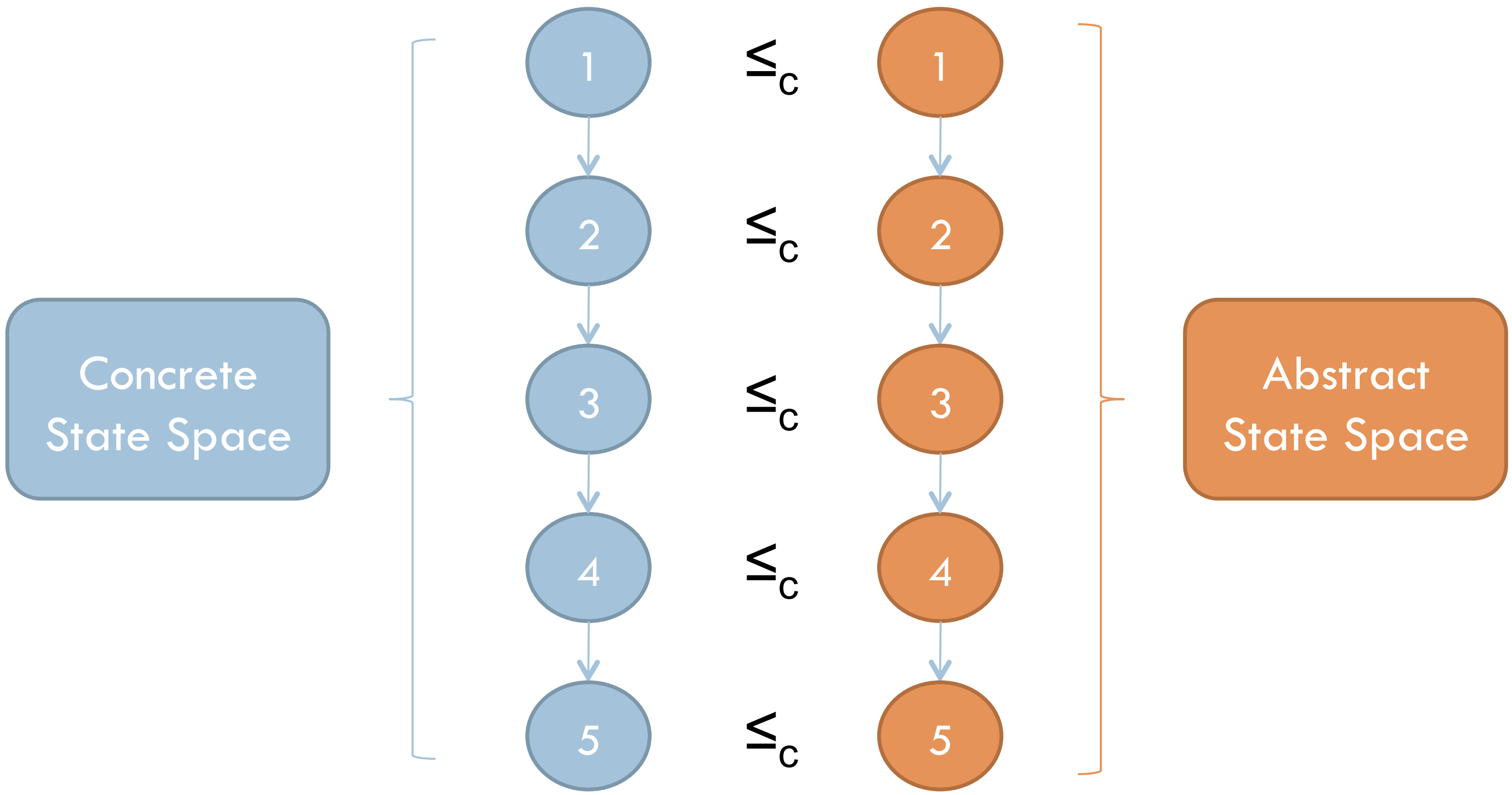

```
f(int a, int b, int c)
        if(a > 0) then
          b = 3;
   c = 5;
   print a, b, c;
end
```

Previous DDVA Work

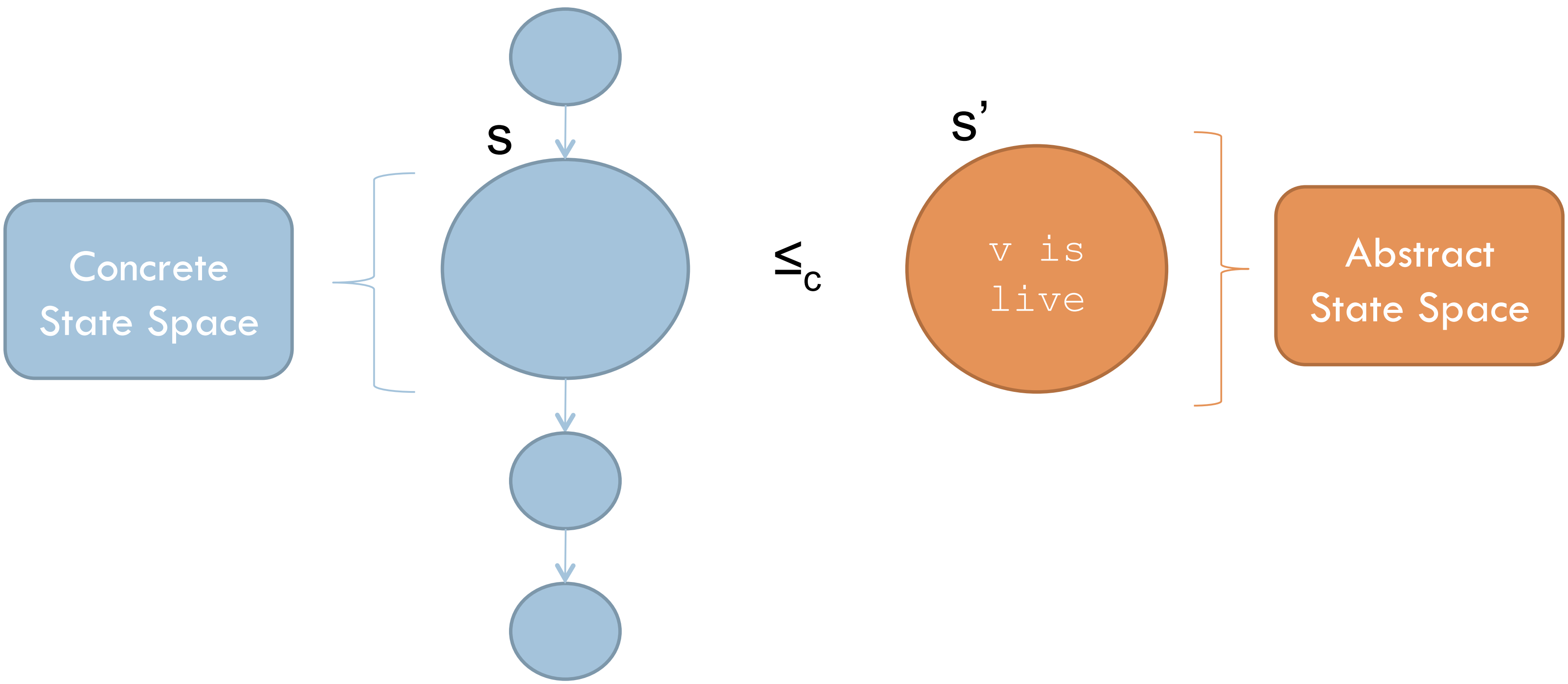


- Original DDVA
  - Uses forward analysis
  - Results dependent on depth bound
  - Does not compute maximal reduction
  - Does not handle loops

DVA Maximal Reduction



DVA Maximal Reduction

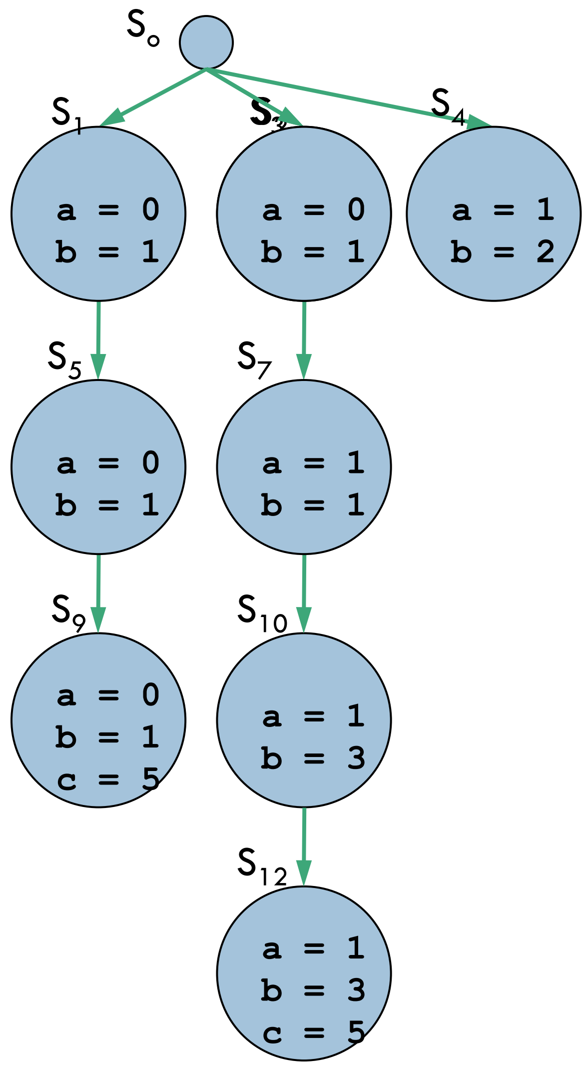


- Only live if exists concrete trace that requires it to be live

Maximal DDVA Implementation

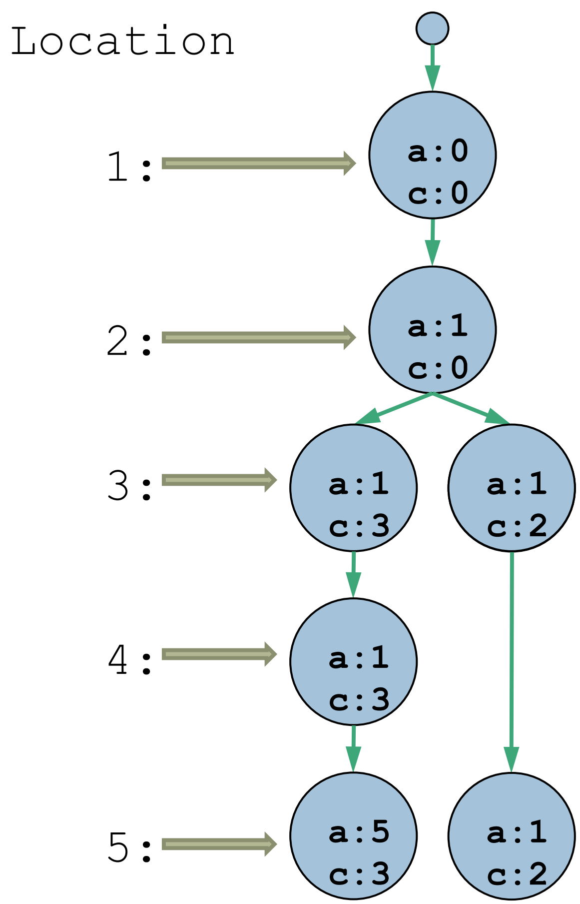
1. Take a full trace through the system
2. Apply *dead* to states in trace to find dead variables
3. Mark dead variables
4. Re-store states in *Visited*
5. Resume model checking

Maximal Dead Variable Analysis




```
f(int a, int b, int c)
        if(a > 0) then
          b = 3;
    c = 5;
    print a, b, c;
end
```

Non-Determinism




```
1: a = get_input();
2: c = get_input();
3: if c > 2 then
4:      a = 5;
5: print a, b, c;
```

Results Tables

easy3

Analysis	Explore Depth	States Generated	Total Time	User Time	Memory Used (MB)	Abstraction Time	Re-store Rate
None	N/A	34640	0m12.764s	0m9.969s	34.50	0.0s	N/A
Static	N/A	15814	0m06.605s	0m5.336s	33.80	0.001s	N/A
Original best	2	15814	0m10.765s	0m9.313s	34.46	3.792s	N/A
Original worst	2	15814	0m10.765s	0m9.313s	34.46	3.792s	N/A
Maximal	N/A	10330	0m08.105s	0m7.002s	25.5312	2.017s	1.021

littleBranch

Analysis	Explore Depth	States Generated	Total Time	User Time	Memory Used (MB)	Abstraction Time	Re-store Rate
None	N/A	864	0m0.442s	0m0.272s	30.90	0.0s	N/A
Static	N/A	721	0m0.405s	0m0.236s	31.40	0.0010s	N/A
Original best	6	658	0m0.344s	0m0.280s	31.43	0.0740s	N/A
Original worst	2	721	0m0.340s	0m0.268s	31.43	0.0492s	N/A
Maximal	N/A	530	0m0.223s	0m0.176s	23.79	0.0138s	1.391

multiBranch

Analysis	Explore Depth	States Generated	Total Time	User Time	Memory Used (MB)	Abstraction Time	Re-store Rate
None	N/A	294515	1m49.170s	1m28.146s	87.10	N/A	N/A
Static	N/A	217454	1m21.780s	1m06.084s	74.87	0.002s	N/A
Original best	16	176651	1m41.458s	1m27.673s	75.79	42.67s	N/A
Original worst	5	217478	2m10.965s	1m55.179s	83.46	46.35s	N/A
Maximal	N/A	145440	2m36.640s	2m25.453s	57.99	7.51s	1.06

Results Tables

lexer

Analysis	Explore Depth	States Generated	Total Time	User Time	Memory Used (MB)	Abstraction Time	Re-store Rate
None	N/A	262843	1m28.391s	1m10.220s	66.90	0.0s	N/A
Static	N/A	226169	1m17.633s	1m01.876s	66.32	0.002s	N/A
Original best	2	225370	1m51.479s	1m34.442s	71.30	31.66s	N/A
Original worst	3	226172	1m53.866s	1m36.554s	71.13	33.46s	N/A
Maximal	N/A	74024	1m45.560s	1m39.382s	37.69	4.898s	1.151

Robot

Analysis	Explore Depth	States Generated	Total Time	User Time	Memory Used (MB)	Abstraction Time	Re-store Rate
None	N/A	35865	0m12.838s	0m10.205s	35.70	0.0s	N/A
Static	N/A	27940	0m10.377s	0m8.229s	35.60	0.002s	N/A
Original best	2	27940	0m18.675s	0m16.641s	36.21	7.947s	N/A
Original worst	2	27940	0m18.675s	0m16.641s	36.21	7.947s	N/A
Maximal	N/A	27784	0m11.494s	0m09.525s	29.21	0.552s	1.28

bintree

Analysis	Explore Depth	States Generated	Total Time	User Time	Memory Used (MB)	Abstraction Time	Re-store Rate
None	N/A	157828	1m00.608s	0m49.811s	66.50	0.0s	N/A
Static	N/A	154084	1m01.061s	0m50.387s	68.40	0.005s	N/A
Original best	6	150964	2m14.807s	2m03.864s	73.74	72.09s	N/A
Original worst	2	154084	2m07.356s	1m56.635s	71.47	64.87s	N/A
Maximal	N/A	103839	1m07.530s	1m00.068s	52.62	16.34s	1.012

Conclusions

- Our algorithm generates DVA maximally reduced state spaces on-the-fly
- Uses less memory
- Requires no user specified depth bound
- Does well on models with loops
- Takes more time on some models
- Due to chained hash table and *contains* relation

Future Work

- Modify to work on multi-procedural programs
- Remove need for chained hash table and *contains* relation
- Adapt to other search algorithms
- Other static analysis techniques for precise abstraction?

Questions?

DVA Maximal Reduction

- Maximum state space reduction from a DVA

For every reachable trace in the concrete state space there exists an abstract trace such that the states s_i in the concrete trace are contained in states s_i' in the abstract trace and s_i' is in the abstract state space.

For all live variables in all states in the abstract state space, there exists a state in a reachable trace in the concrete state space where that variable is live in that state.

Static Vs. Dynamic

- Static dead variable analysis (SDVA) helps
- SDVA does not use dynamic run-time information
  - Variable valuations
  - Considers infeasible paths
- Dynamic Dead Variable Analysis (DDVA)
  - Uses variable valuation info
  - Only considers feasible paths
  - Finds more dead variables