The Yogi Project

Software property checking via verification and testing

Aditya V. Nori, Sriram K. Rajamani
Programming Languages and Tools
Microsoft Research India

What is Yogi?

- An industrial strength program verifier
- Philosophy: Synergize verification and testing
- Synergy [FSE '06], Dash [ISSTA '08], Smash [POPL '10], Bolt [submitted] algorithms to perform scalable analysis
- Engineered a number of optimizations for scalability
- Integrated with Microsoft's Static Driver Verifier (SDV) toolkit and used internally

Property checking

```
void f(int *p, int *q)
{
0:   *p = 4;
1:   *q = 5;
2:   assert (¬φ_error)
}
```

Question
Does the assertion hold for all possible inputs?

Must analysis: finds bugs, but can't prove their absence
May analysis: can prove the absence of bugs, but can result in false errors

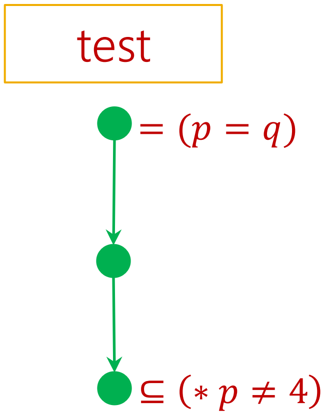
More generally, we are interested in the query

$$\langle \varphi_{pre} \overset{?}{\Rightarrow}_f \varphi_{error} \rangle$$

Must information

$$\langle T \overset{?}{\Rightarrow}_f (* \, p \neq 4)\rangle = yes$$

```
void f(int *p, int*q)
{
0:   *p = 4;
1:   *q = 5;
}
```



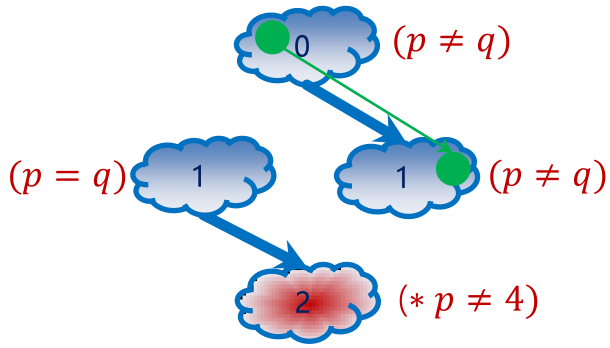
- Captures facts that are guaranteed to hold on particular executions of the program (*under-approximation*)
- Error condition is reachable by any input that satisfies $(p = q)$

May information

$\langle (p \neq q) \overset{?}{\Rightarrow}_f (*p \neq 4) \rangle = no$

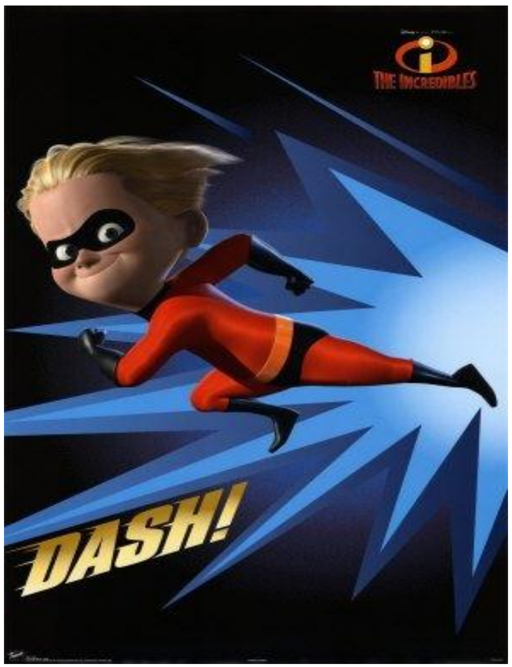
```
void f(int *p, int*q)
{
0:   *p = 4;
1:   *q = 5;
}
```

proof



- Captures facts that are true for all executions of the program (*over-approximation*)
- Proof can be obtained by keeping track of the predicates $(p = q)$ and $(*p \neq 4)$

Dash: Proofs from Tests

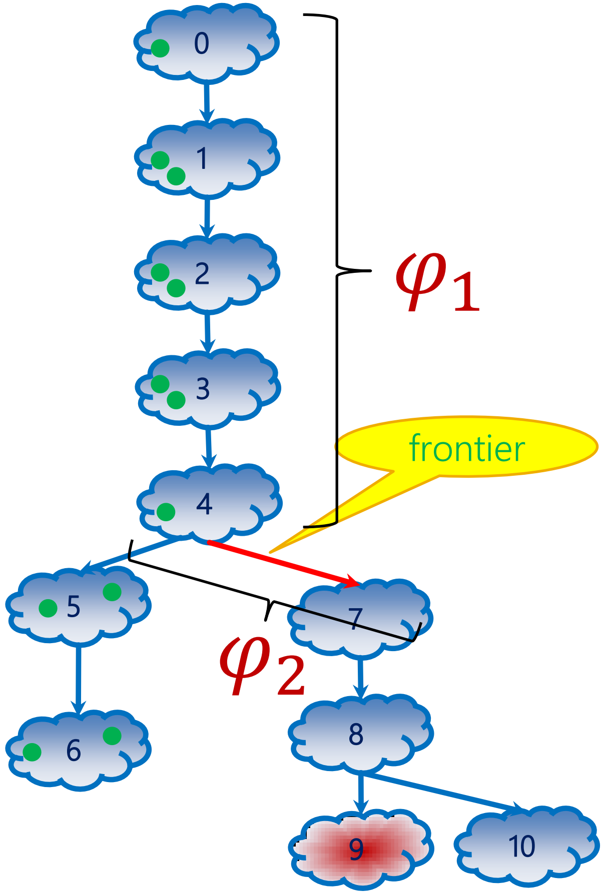


- Algorithm uses only test case generation operations
- Maintains two data structures:
  - A forest of reachable concrete states (tests)
    - Under-approximates executions of the program
  - A region graph (an abstraction)
    - Over-approximates all executions of the program
- Our goal: bug finding and proving
  - If a test reaches an error, we have found bug
  - If we refine the abstraction so that there is *no* path from the initial region to error region, we have a proof
- Key ideas
  - Frontier
  - WP_α uses only aliases α that are present along concrete tests that are executed

Key ideas

Step 1: Try to generate a test that crosses the frontier

- Perform symbolic simulation on the path until the frontier and generate a constraint φ_1
- Conjoin with the condition φ_2 needed to cross frontier
- Is $\varphi_1 \wedge \varphi_2$ satisfiable?

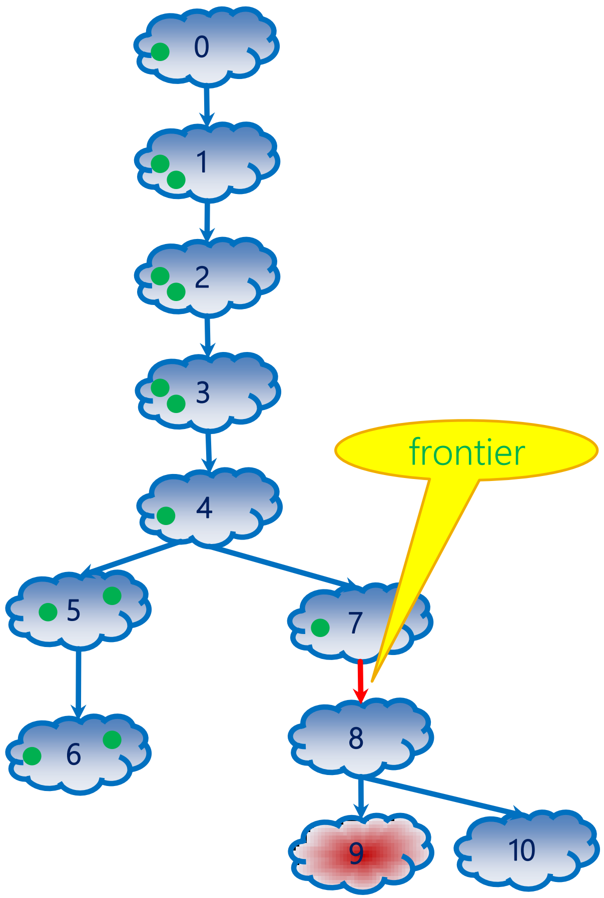


Key ideas

Step 1: Try to generate a test that crosses the frontier

- Perform symbolic simulation on the path until the frontier and generate a constraint φ_1
- Conjoin with the condition φ_2 needed to cross frontier
- Is $\varphi_1 \land \varphi_2$ satisfiable? [YES]

Step 2: run the test and extend the frontier

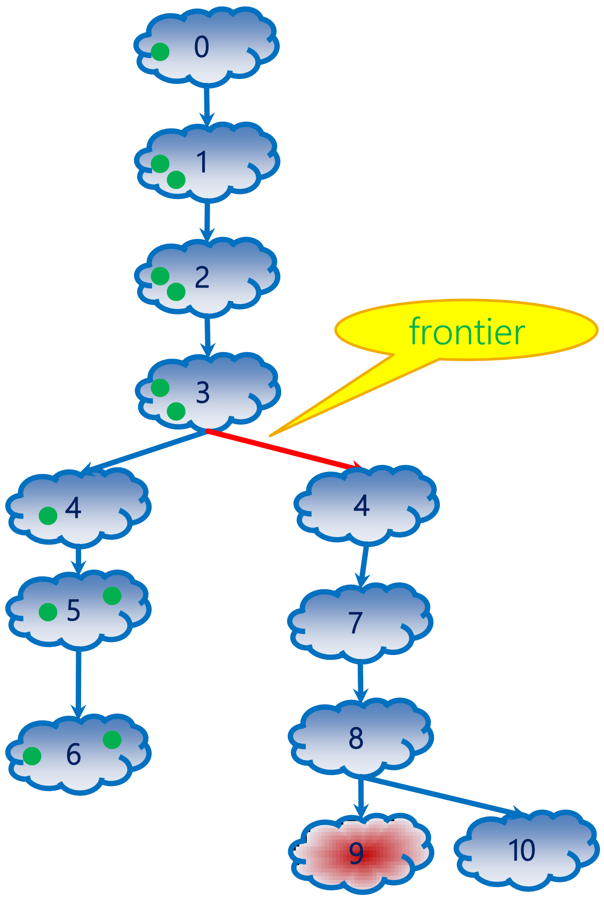


Key ideas

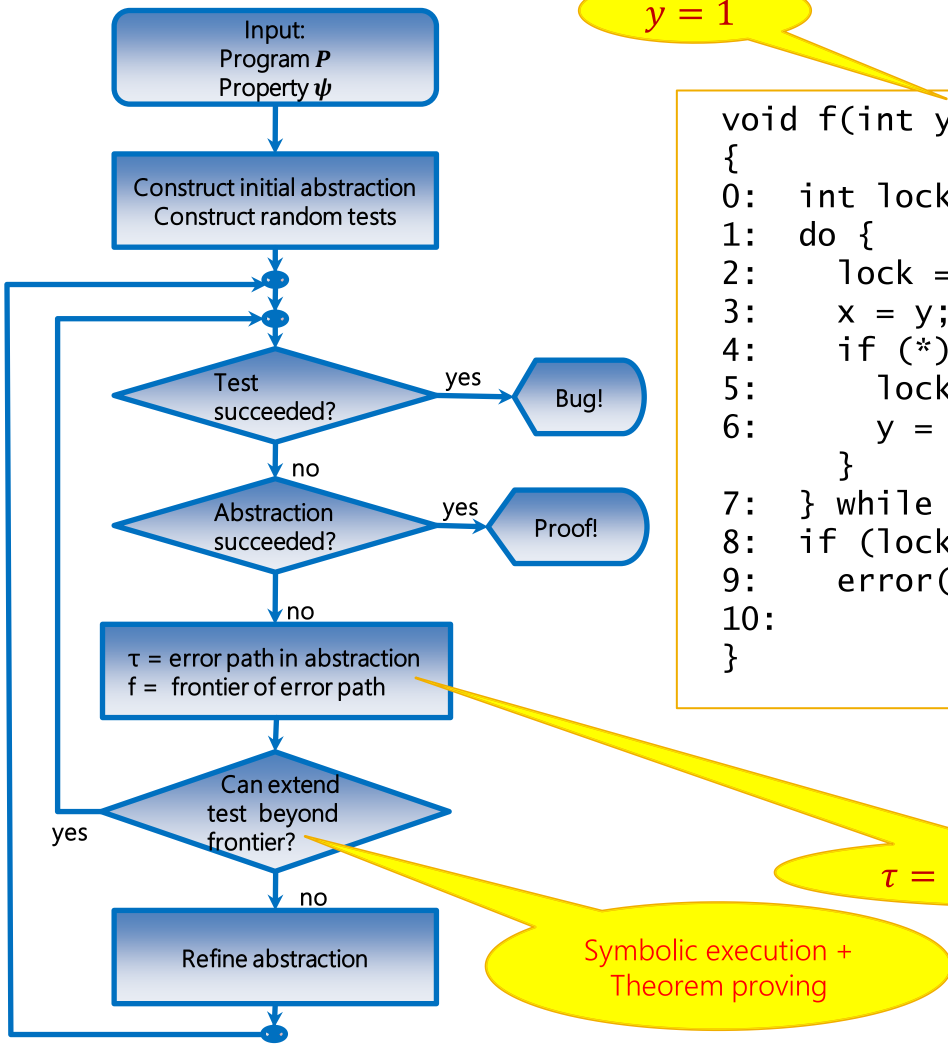
Step 1: Try to generate a test that crosses the frontier

- Perform symbolic simulation on the path until the frontier and generate a constraint φ_1
- Conjoin with the condition φ_2 needed to cross frontier
- Is $\varphi_1 \wedge \varphi_2$ satisfiable? [NO]

Step 2: use WP_α to refine so that the frontier moves back!

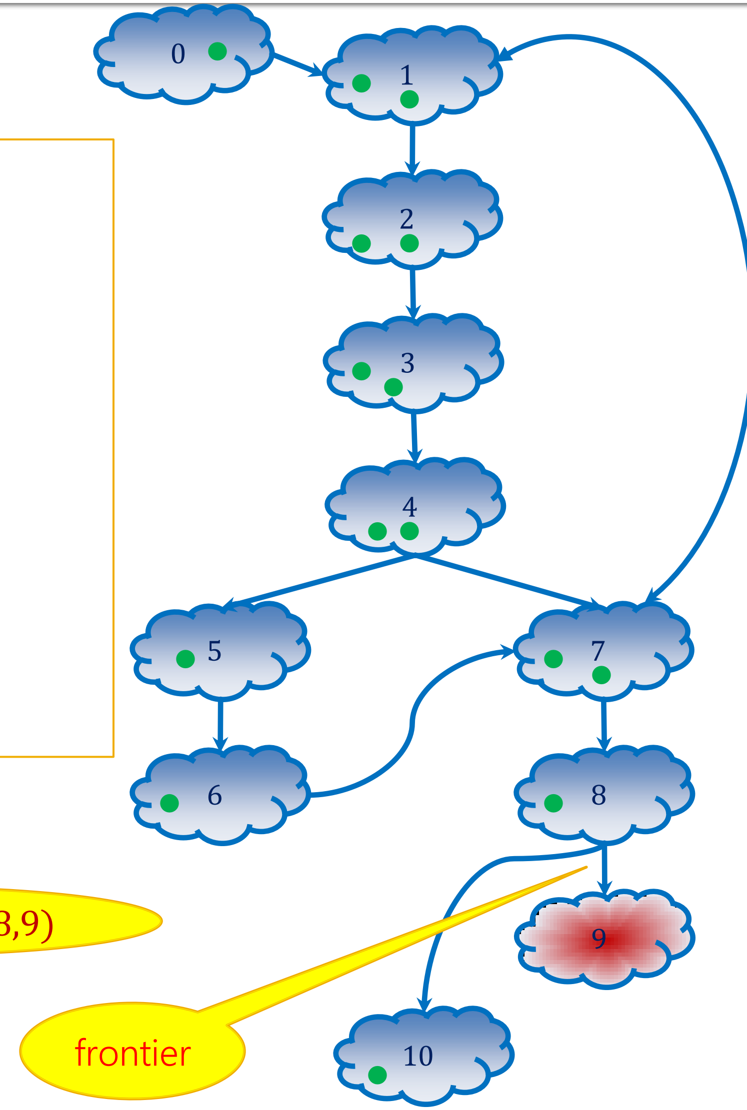


The Dash algorithm



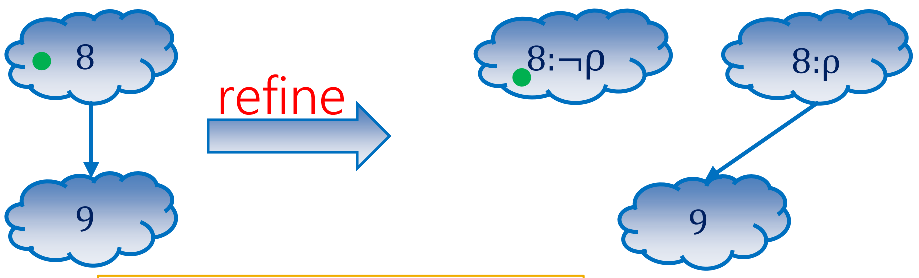
$y = 1$

```
void f(int y)
{
0:  int lock, x;
1:  do {
2:    lock = 1;
3:    x = y;
4:    if (*) {
5:      lock = 0;
6:      y = y+1;
      }
7:  } while (x != y);
8:  if (lock != 1)
9:    error();
10:
}
```

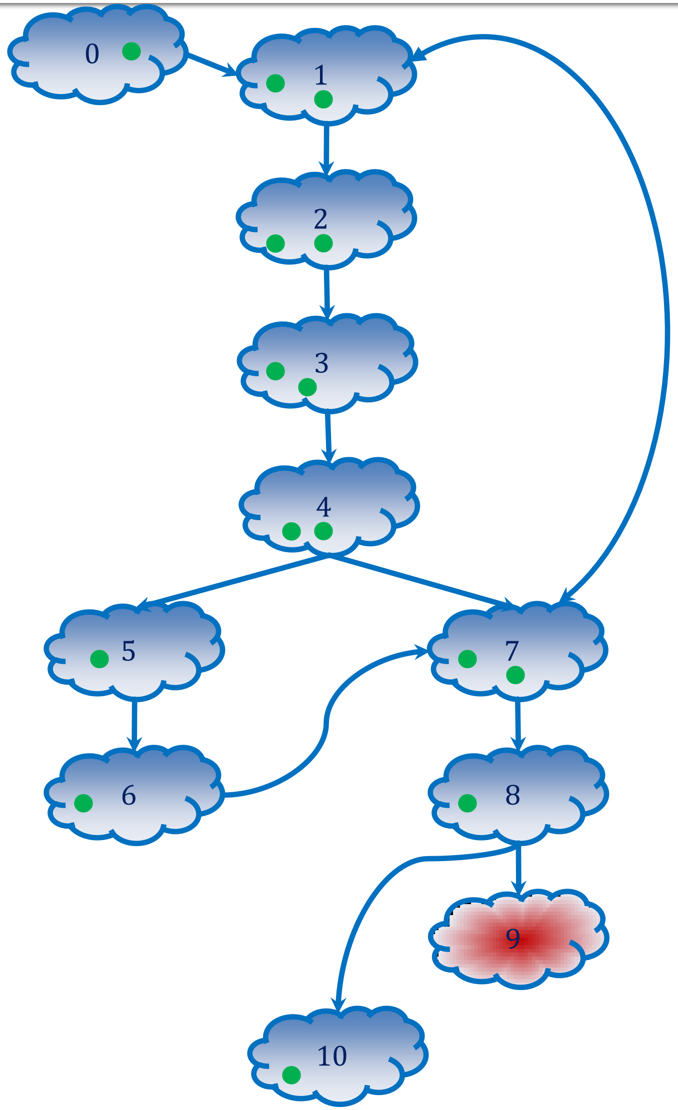


$\tau = (0,1,2,3,4,7,8,9)$

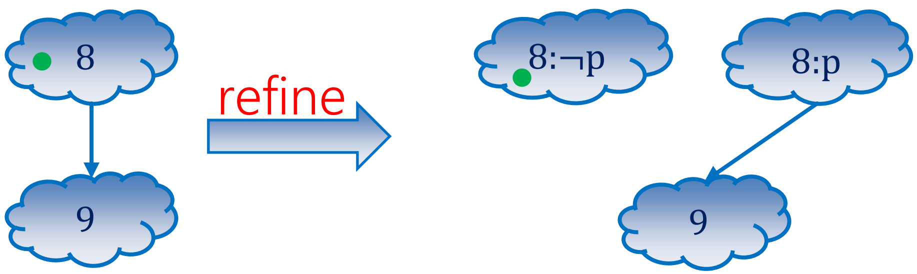
Refinement



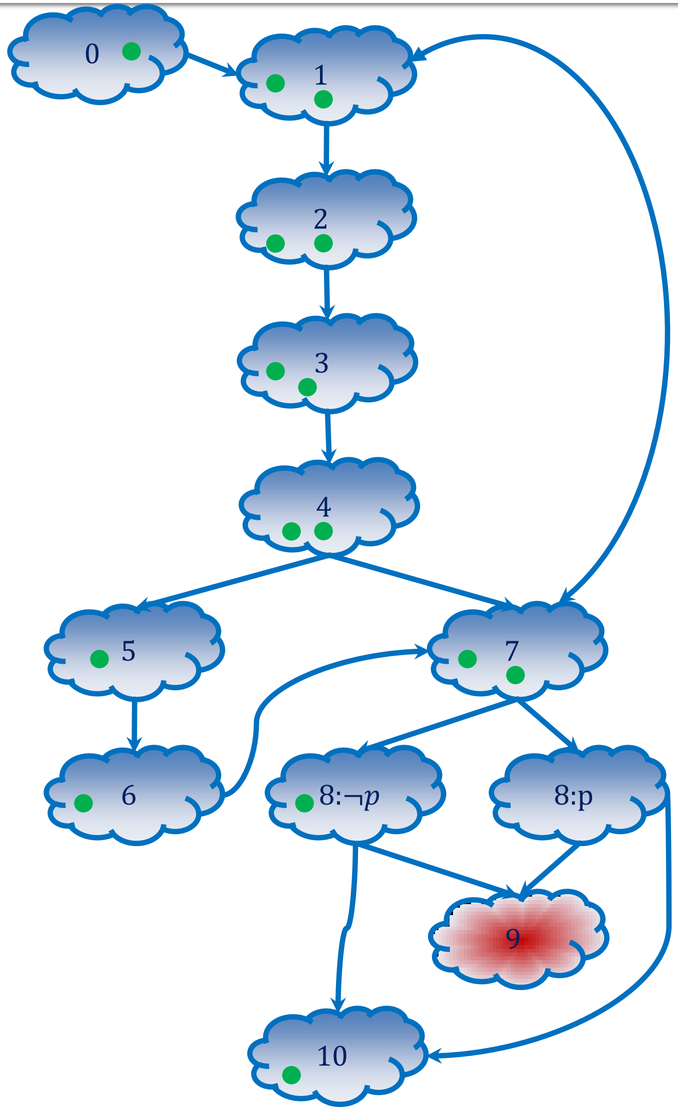
$\rho = (lock.state \mathrel{!=} L)$



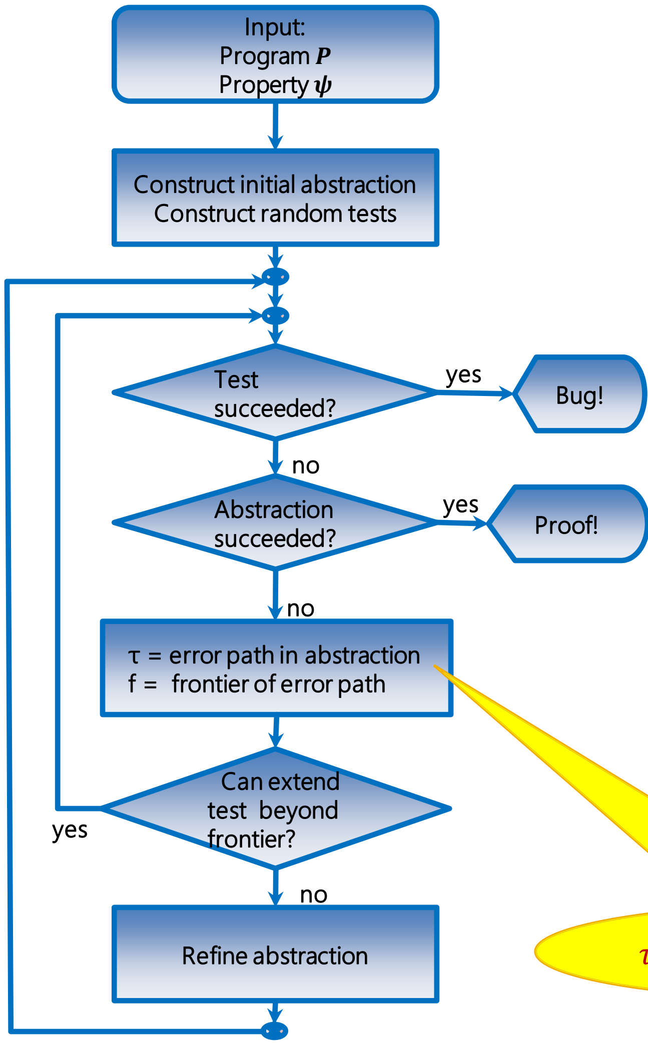
Refinement



p= $(lock.state \mathrel{!}= L)$

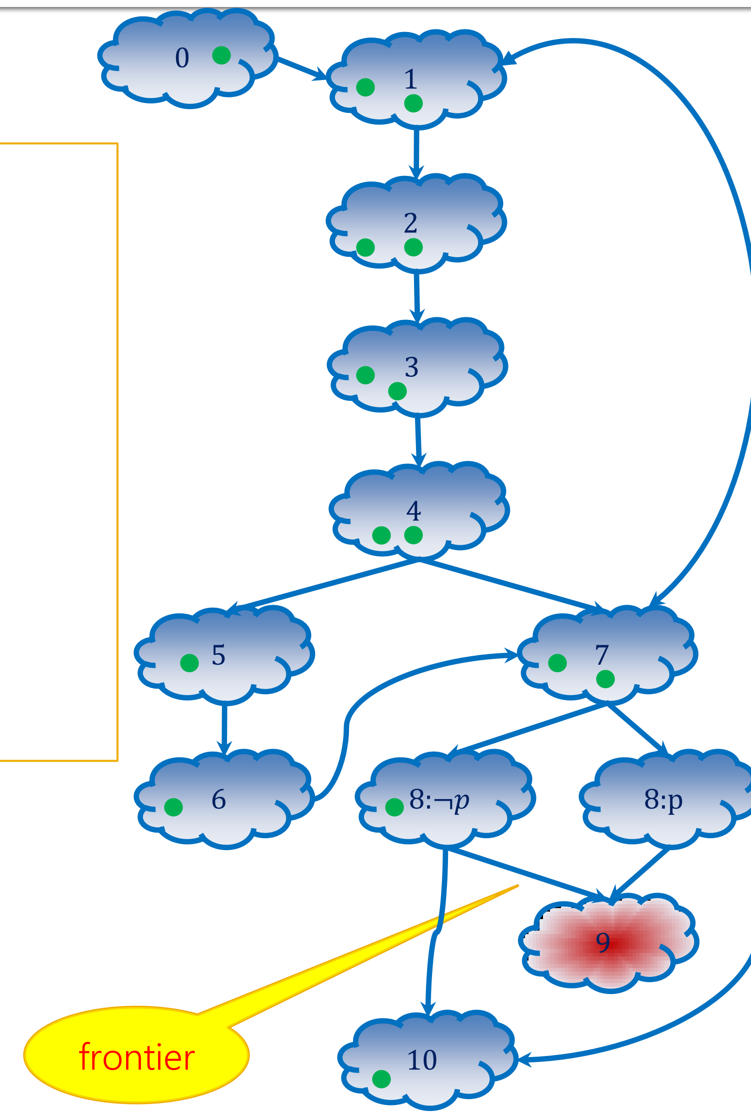


Another iteration



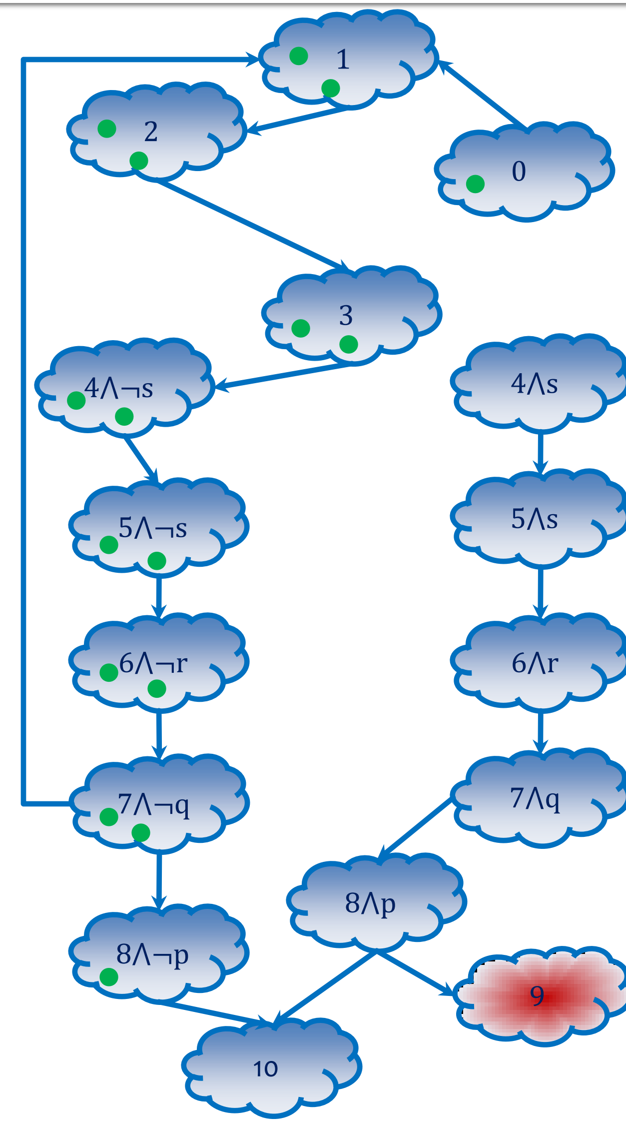

```
void f(int y)
{
0:  int lock, x;
1:  do {
2:    lock = 1;
3:    x = y;
4:    if (*) {
5:      lock = 0;
6:      y = y+1;
      }
7:  } while (x != y)
8:  if (lock != 1)
9:    error();
10:
}
```

$\tau = (0,1,2,3,4,7,<8,p>,9)$

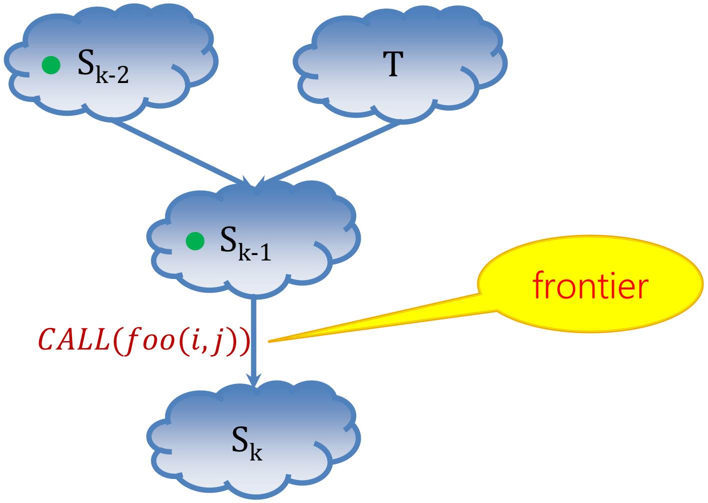


Correct, the program is …

```
void f(int y)
{
0:  int lock, x;
1:  do {
2:    lock = 1;
3:    x = y;
4:    if (*) {
5:      lock = 0;
6:      y = y+1;
      }
7:  } while (x != y)
8:  if (lock != 1)
9:    error();
10:
}
```

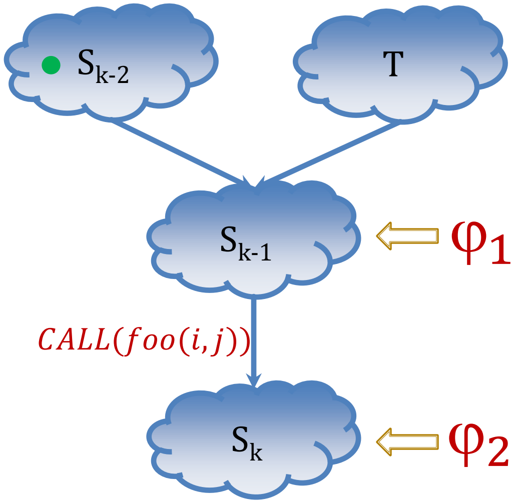


Interprocedural analysis



Key idea
Perform a recursive **Dash** query on the called procedure and use the result to either generate a test or compute WP_{α}

Interprocedural analysis



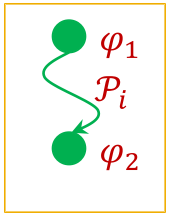
Dash$\langle \varphi_1 \overset{?}{\Rightarrow}_{foo} \varphi_2 \rangle$

- pass: perform refinement
- fail: generate test

Procedure summaries

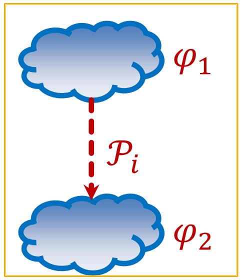
- A *must summary* for a procedure $\mathcal{P}_i$ is of the form $(\varphi_1, \varphi_2) \in \overset{must}{\Longrightarrow}_{\mathcal{P}_i}$
- $\forall t \in \varphi_2 \,.\, \exists s \in \varphi_1 \,.\, t$ can be obtained by executing $\mathcal{P}_i$ from an initial state s

must summary



- A $\neg$*may summary* for a procedure $\mathcal{P}_i$ is of the form $(\varphi_1, \varphi_2) \in \overset{\neg may}{\Longrightarrow}_{\mathcal{P}_i}$
- $\forall s \in \varphi_1 \; \forall t \in \varphi_2 \,.\, t$ cannot be obtained by executing $\mathcal{P}_i$ starting in state s

¬may summary



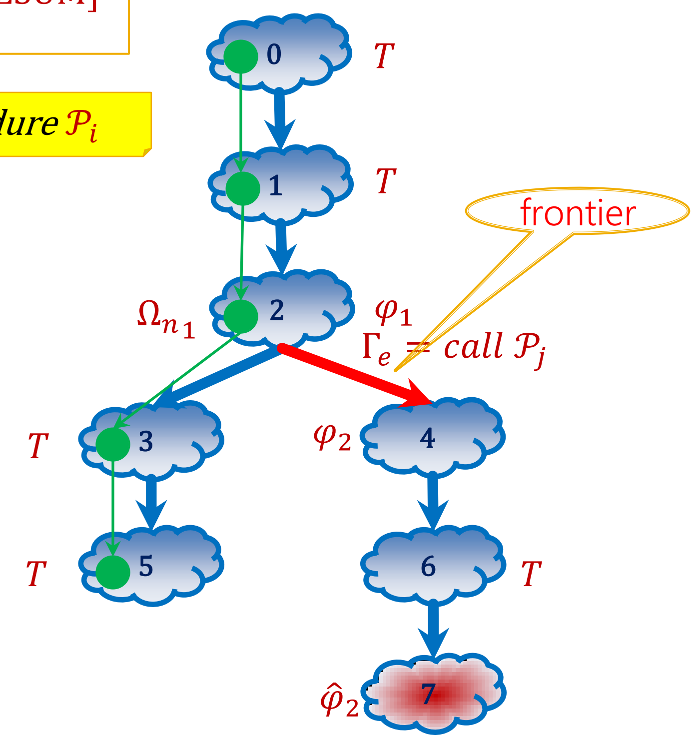
Compositional may-must analysis

$$\frac{\varphi_1 \in \Pi_{n_1} \quad \varphi_2 \in \Pi_{n_2} \quad \varphi_1 \cap \Omega_{n_1} \neq \emptyset \quad \varphi_2 \cap \Omega_{n_2} = \emptyset \\ e = (n_1, n_2) \in E_{\mathcal{P}_i} \textit{ is a call to procedure } \mathcal{P}_j \\ (\hat{\varphi}_1, \hat{\varphi}_2) \in \overset{must}{\Longrightarrow}_{\mathcal{P}_j} \quad \Omega_{n_1} \supseteq \hat{\varphi}_1 \quad \theta \subseteq \hat{\varphi}_2 \quad \varphi_2 \cap \theta \neq \emptyset}{\Omega_{n_2} := \Omega_{n_2} \cup \theta} \text{[MUST – POST – USESUM]}$$

must summary

- $\hat{\varphi}_1 \subseteq \Omega_{n_1}$
- $\mathcal{P}_j$
- $(\hat{\varphi}_2 \supseteq \theta) \wedge (\varphi_2 \cap \theta \neq \emptyset)$

procedure $\mathcal{P}_i$

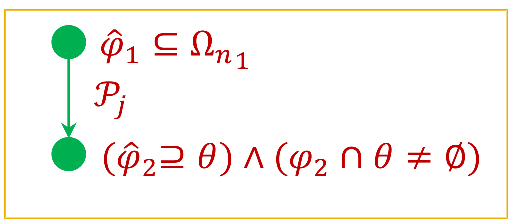


- Check if frontier (n_1, n_2) can be extended by a *must summary* $(\hat{\varphi}_1, \hat{\varphi}_2)$
- If yes, grow Ω_{n_2} with $\theta \subseteq \hat{\varphi}_2$

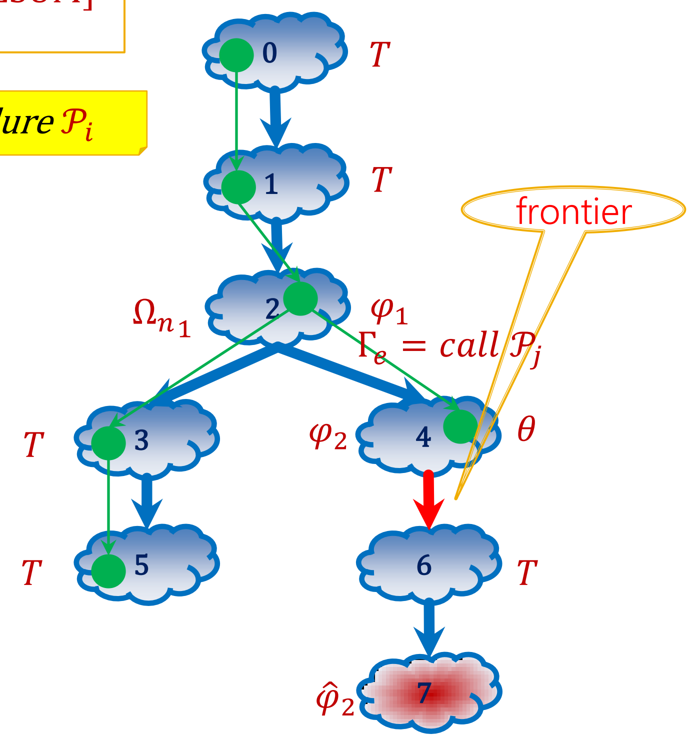
Compositional may-must analysis

$$\frac{\varphi_1 \in \Pi_{n_1} \quad \varphi_2 \in \Pi_{n_2} \quad \varphi_1 \cap \Omega_{n_1} \neq \emptyset \quad \varphi_2 \cap \Omega_{n_2} = \emptyset \\ e = (n_1, n_2) \in E_{\mathcal{P}_i} \text{ is a call to procedure } \mathcal{P}_j \\ (\hat{\varphi}_1, \hat{\varphi}_2) \in \overset{must}{\Longrightarrow}_{\mathcal{P}_j} \quad \Omega_{n_1} \supseteq \hat{\varphi}_1 \quad \theta \subseteq \hat{\varphi}_2 \quad \varphi_2 \cap \theta \neq \emptyset}{\Omega_{n_2} := \Omega_{n_2} \cup \theta} \text{[MUST − POST − USESUM]}$$

must summary



procedure $\mathcal{P}_i$

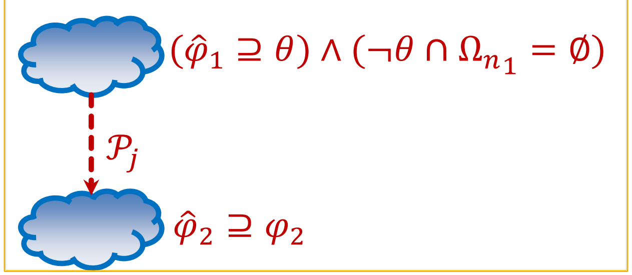


- Check if frontier (n_1, n_2) can be extended by a *must summary* $(\hat{\varphi}_1, \hat{\varphi}_2)$
- If yes, grow Ω_{n_2} with $\theta \subseteq \hat{\varphi}_2$

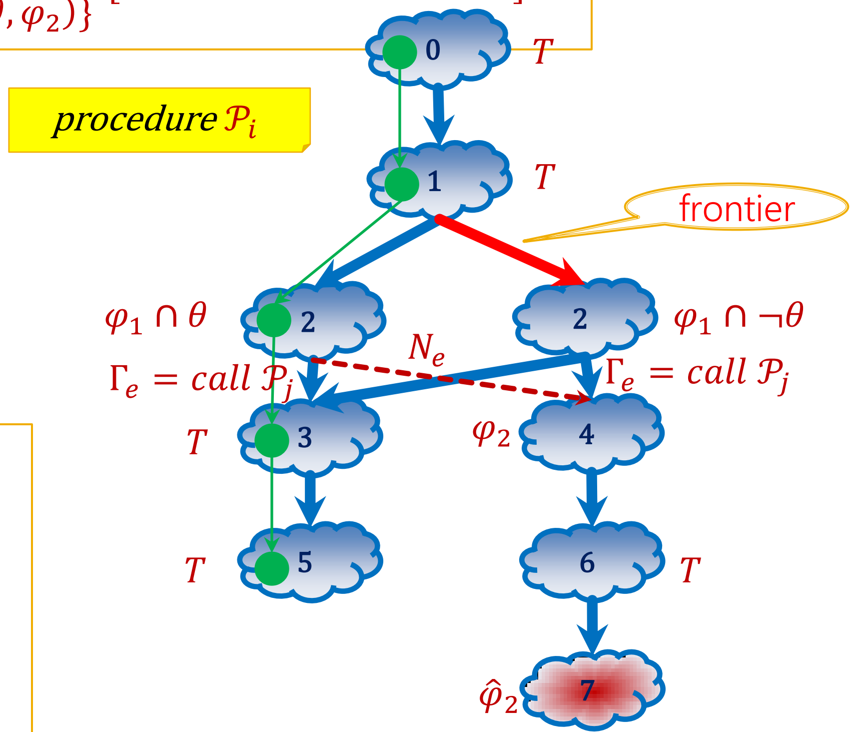
Compositional may-must analysis

$$\frac{\varphi_1 \in \Pi_{n_1} \quad \varphi_2 \in \Pi_{n_2} \quad \varphi_1 \cap \Omega_{n_1} \neq \emptyset \quad \varphi_2 \cap \Omega_{n_2} = \emptyset \\ e = (n_1, n_2) \in E_{\mathcal{P}_i} \textit{ is a call to procedure } \mathcal{P}_j \\ \langle \hat{\varphi}_1, \hat{\varphi}_2 \rangle \in \overset{\neg may}{\Longrightarrow}_{\mathcal{P}_j} \quad \varphi_2 \subseteq \hat{\varphi}_2 \quad \theta \subseteq \hat{\varphi}_1 \quad \neg\theta \cap \Omega_{n_1} = \emptyset}{\Pi_{n_1} := (\Pi_{n_1} \setminus \{\varphi_1\}) \cup \{\varphi_1 \cap \theta, \varphi_1 \cap \neg\theta\} \quad N_e := N_e \cup \{(\varphi_1 \cap \theta, \varphi_2)\}} \text{[NMAY} - \text{PRE} - \text{USESUM]}$$

¬may summary



- Check if frontier (n_1, n_2) can be refined by a $\neg may\ summary\ \ (\hat{\varphi}_1, \hat{\varphi}_2)$
- If yes, use $\theta \subseteq \hat{\varphi}_1$ to refine the abstraction
- If both $must$ and $\neg may$ summaries are not available, analyze procedure $\mathcal{P}_j$
  - $yes \Rightarrow must\ summary$ for $\mathcal{P}_j$
  - $no \Rightarrow \neg may\ summary$ for $\mathcal{P}_j$



Optimizations

- Engineering for making Yogi robust, scalable and industrial strength
- Several of the implemented optimizations are folklore
  - Very difficult to design tools that are bug free ⇒ evaluating optimizations is hard!
  - Our empirical evaluation gives tool builders information about what gains can be realistically expected from optimizations
  - Details in ICSE '10
- Vanilla implementation of algorithms:
  - (`flpydisk`, `CancelSpinLock`) took 2 hours
- Algorithms + engineering + optimizations:
  - (`flpydisk`, `CancelSpinLock`) took less than 1 second!

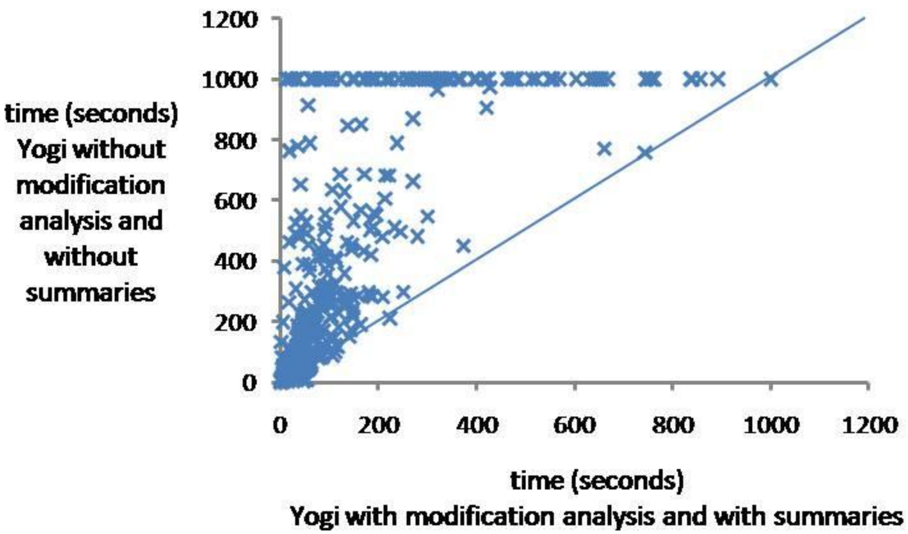
Evaluation setup

- Benchmarks:
  - 30 WDM drivers and 83 properties (2490 runs)
  - Anecdotal belief: most bugs in the tools are usually caught with this test suite

Empirical results (Summaries)

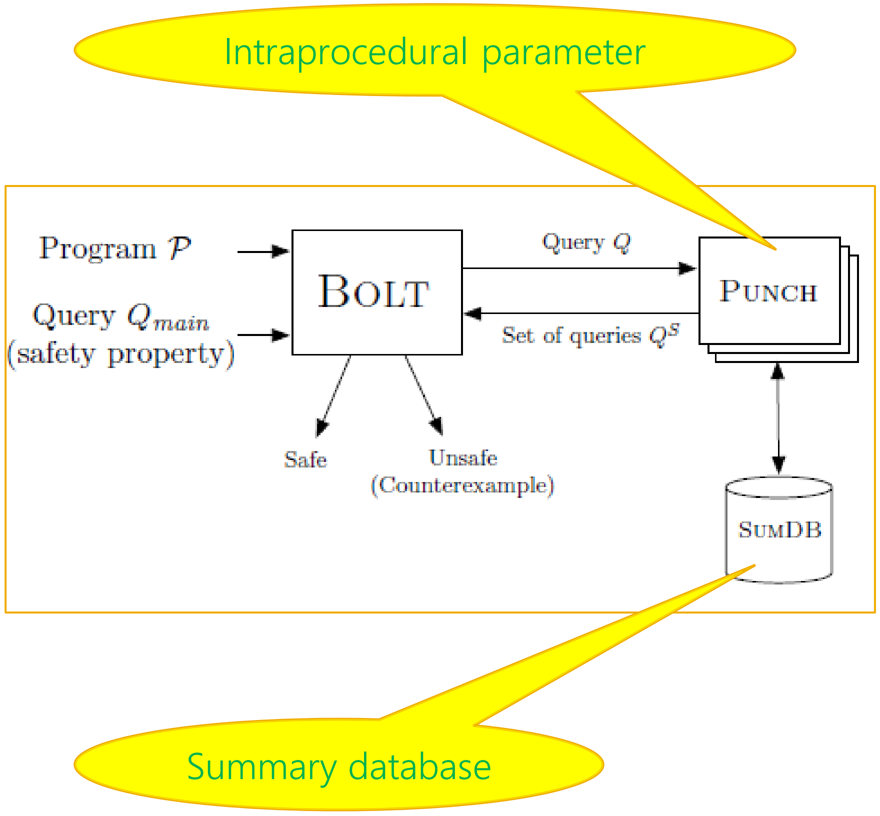
Summaries	Total time (minutes)	#defects	#timeouts
yes	2160	241	77
no	3780	236	165

42%



Current research

- Bolt: a generic framework that uses MapReduce style parallelism to scale top-down analysis



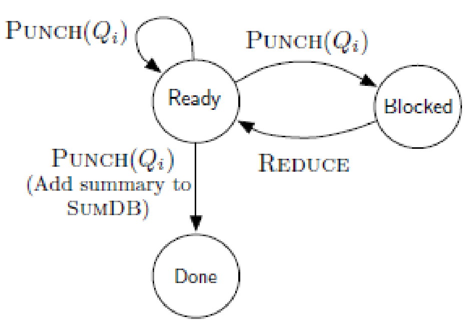
Example

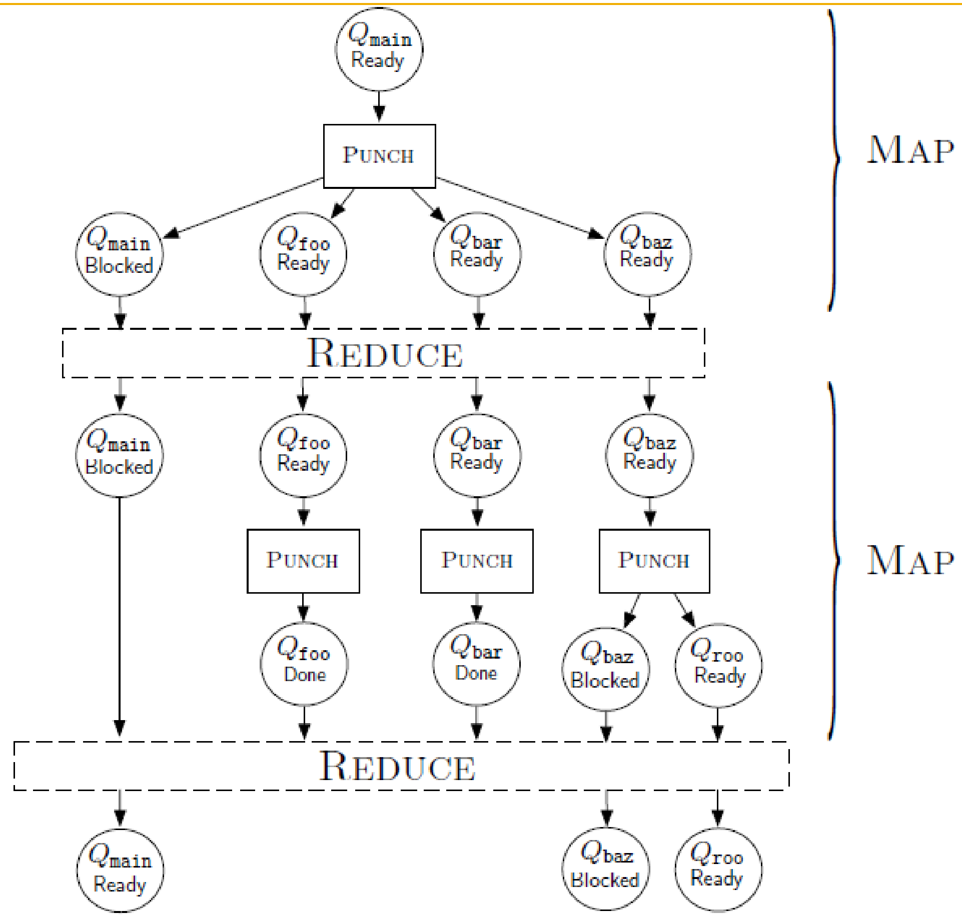
```
int foo(int p_foo);
int bar(int p_bar);
int baz(int p_baz);

main(int i, int j){
  int x, y;
  if (j > 0)
    x = foo(i);
  else if (j > -10)
    x = bar(i);
  else
    x = baz(j);

  y = x + 5;
  assert(y > 0);
}
```

(a)

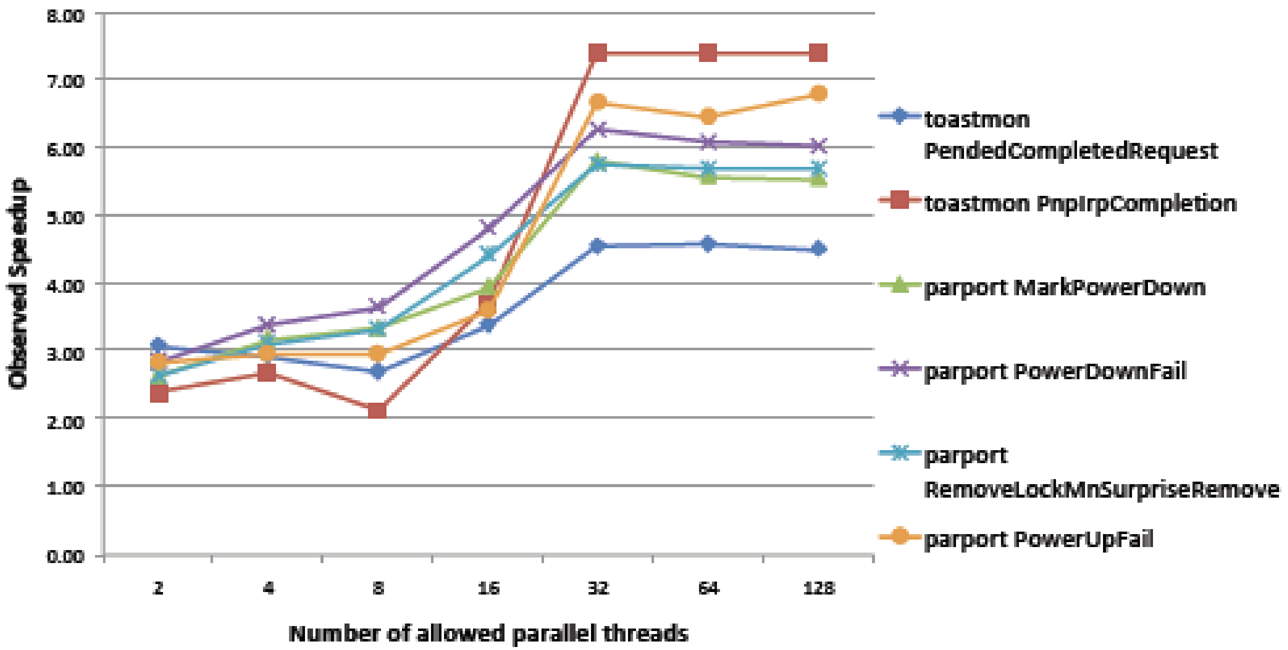




Empirical results

~Linear speedup!

Statistic	
Total time taken (sequential)	26 hours
Total time taken (parallel)	7 hours
Average observed speedup	3.71x
Maximum observed speedup	7.41x



Questions?

PLDI 2012 tutorial

http://research.microsoft.com/yogi/pldi2012.aspx