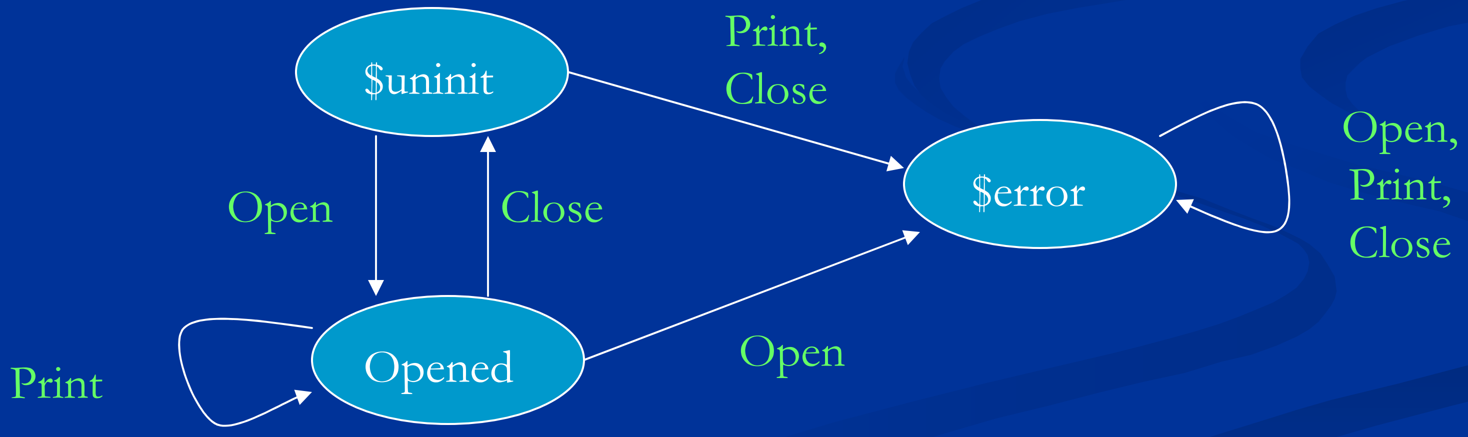
ESP - Path-Sensitive Program Verification in Polynomial Time

M. Das, S. Lerner, M. Seigle

PLDI '02

Partial program verification

- Verify that a program obeys a temporal safety property
  - e.g. correct file opening/closing behavior
- Property representable as DFA (FSM)



Why it’s hard:

- In a program, FSM may transition differently along different execution paths
- Path-insensitive dataflow analysis will merge and lose relevant information
- The program may satisfy the property, but we won't be able to determine this.

Example

```
void main(){
  if (dump)
      f = fopen(dumpFil, "w");
  if (p)
      x = 0;
  else
      x = 1;
  if (dump)
      fclose(f);
}
```

Path-insensitive dataflow analysis

```
void main(){
                 ----------------------   [ $uninit ]
  if (dump)
     f = fopen(dumpFil, "w");
                 ----------------------   [ $uninit, Opened ]
  if (p)
     x = 0;
  else
     x = 1;
                 ----------------------   [ $uninit, Opened ]
  if (dump)
     fclose(f);
                 ----------------------   [ $uninit, $error]
}
```

Path-sensitive analysis

```
void main(){
  if (dump)
      f = fopen(dumpFil, "w");
  if (p)
      x = 0;
  else
      x = 1;
  if (dump)
      fclose(f);
}
```

[**$uninit**]

[**$uninit**, ¬d]
[**Opened**, d]

[**$uninit**, ¬ d, ¬ p, x =1]
[**$uninit**, ¬ d, p, x = 0]
[**Opened**, d, ¬ p, x =1]
[**Opened**, d, p, x =0]

Only one of the two paths possible from each state

Moral of the story:

- Path-insensitive dataflow analysis is too imprecise
- But path-sensitive analysis is overkill and too expensive.
- The obvious solution: keep as much information as needed, no more, no less
  - the paper presents a heuristic for this

Main contributions of this paper

- An analysis framework that is **only as path-sensitive as needed** to verify a property
  - Including an inter-procedural version
- Insights into developing a verification system using property simulation that will scale to large programs (such as `gcc`)
  - This is ESP - Error detection via Scalable Program analysis

Property analysis

- An analysis framework that parametrizes how path-sensitive we choose to be.
- Includes path-insensitive and fully path-sensitive analyses as extremes.
- Essentially a normal dataflow analysis, with interesting things happening at the merge points.
  - path-insensitive - merge everything
  - path-sensitive - no merges
  - property simulation - merge only info "irrelevant" for the property being verified

A few details

- State carried in analysis is *symbolic state*
- Two components:
  - abstract state $\subseteq$ D, where D = set of states in the property FSM
  - execution state (as normal)
- S = domain of all symbolic states
- Analysis computes dataflow facts from the domain 2^S

A few details (2)

- Key is filtering function used at merge points:
  - $\alpha : 2^S \to 2^S$
- $\alpha_{cs}(ss) = ss$
  - gives path-sensitive analysis
- $\alpha_{df}(ss) = \{\cup_{s \in ss} as(s), \sqcup_{s \in ss} es(s)]\}$
  - gives path-insensitive dataflow analysis

A few details (3)

- Property simulation merges all those symbolic states that have the same property state
- $\alpha_{as} = \{[\{d\}, \sqcup_{s \in ss[d]} es(s)] \mid d \in D \ \& \ ss[d] \neq \emptyset\}$
- Notation:
  - $ss[d] = \{ s \mid s \in ss \ \& \ d \in as(s) \}$
  - “set of all s in ss containing d”
- Example
- Will see limitations of this heuristic soon

Path-sensitive analysis

```
void main(){
  if (dump)
      f = fopen(dumpFil, "w");
  if (p)
      x = 0;
  else
      x = 1;
  if (dump)
      fclose(f);
}
```

[**$uninit**]

[**$uninit**, ¬d]
[**Opened**, d]

[**$uninit**, ¬ d, ¬ p, x =1]

[**$uninit**, ¬ d, p, x = 0]

[**Opened**, d, ¬ p, x =1]

[**Opened**, d, p, x =0]

Property simulation

```
void main(){
  if (dump)
      f = fopen(dumpFil, "w");
  if (p)
      x = 0;
  else
      x = 1;
  if (dump)
      fclose(f);
}
```

[**$uninit**]

[**$uninit**, ¬d]
[Opened, d]

No changes to property state

[**$uninit**, ¬ d] [**Opened**, d]

Only one of the two paths possible from each state

A few details (4)

- Not all branches are possible from a particular symbolic state
  - Analysis exploits this by using a theorem prover to attempt to determine whether path is feasible from a given symbolic state
- Complexity $O(H\ |E|\ |D|\ (T + J + Q))$ where
  - H is the lattice height
  - E is the number of edges in CFG
  - D is the number of property states
  - T is the cost of one call to the flow function (includes deciding branch feasibility), J is join, Q is deciding equality on execution states.

Property Analysis

- Instantiation to constant propagation with property simulation – $O(V^2 |E| |D|)$
  - V = number of variables
- Can obtain an inter-procedural analysis using the framework by Reps, Horwitz and Sagiv
  - the algorithm is context-sensitive for property states only (insensitive for execution states).

But property simulation is no magic bullet

```
if (dump)
  flag = 1;
else
  flag = 0;
if (dump)
  f = fopen(...);
if (flag)
  fclose(f);
```

We lose information

```
if (dump)
  flag = 1;
else
  flag = 0;
if (dump)
  f = fopen(...);
if (flag)
  fclose(f);
```

Property state stays same here, so analysis won't save correlation between flag and dump

Property states will be $uninit and Opened

Potential error here!

The authors' response

- This is not a common example
- Property simulation matches "the behavior of a careful programmer"
  - Programmers use variables to maintain a correlation between a given property state and the corresponding execution states
  - Property simulation models this

ESP

- Want to use property simulation to verify large programs like gcc (140,000 LOC)
- Main insight: analysis is not monolithic
  - and different parts can be run at different levels of precision, flow-sensitivity, etc.

Stateful Values

- e.g. file handles
- programmer supplies a specification for the safety property:
  - FSM
  - Mapping from source code patterns to FSM transitions and to stateful value creation

C code pattern	Transition	Creation?
`e = fopen(...)`	Open	Yes
`fclose(e)`	Close	No
`fprintf(e, _ )`	Print	No

Value flow analysis

- First step is value flow analysis to discover which stateful values are affected at relevant function calls
  - flow-insensitive, context-sensitive
- Note they disallow properties that correlate the states of multiple values
  - so can analyze one stateful value at a time
    - cf. gcc, 15 files instead of 2^15 possibilities!

ESP analysis – the steps:

- CFG construction
- Value flow alnalysis
- Abstract CFG construction
  - essentially combines 2 steps above
- Various computations to optimize analysis
  - alias set computation for stateful values
  - mod set (things that can be ignored by property simulation)
- Property simulation

Experimental results

- Used to verify correctness of calls to `fprintf` in gcc
- Initially, 15 files created based on user flags
  - for each file handle, core code analyzed twice – with this file open, and with this file closed and user flag set to false.
- Analysis verifies the correctness of all 646 calls to `fprintf`
- Running time – average 72.9 s, max 170 s (for one file handle)
- Memory usage – average 49.7 MB, max 102 MB