On Reasoning about Finite Sets in Software Model Checking

Pavel Shved

Institute for System Programming, RAS

SYRCoSE
2 June 2010

Static Program Verification

Static Verification — checking programs against specific properties without executing them. Features:

+ all possible inputs are checked
+ certain methods can prove the program correct
− significant time and resource consumption
− expressiveness of checkable programs is limited

Checking programs with Reachability Verifiers

Considerable amount of properties to check against can be reduced to the reachability problem.

The reduction technique is known as program instrumentation:

1. modify the code to add transitions to the error state when violation of the property is detected
2. check reachability of the error state

Checking programs with Reachability Verifiers

Considerable amount of properties to check against can be reduced to the reachability problem.

The reduction technique is known as program instrumentation:

1. modify the code to add transitions to the error state when violation of the property is detected
2. check reachability of the error state

```
mutex_lock(&mtx);  →  if (mtx->locked==1)
                          goto ERROR;
                      mtx->locked = 1;
```

Checking programs with Reachability Verifiers

Considerable amount of properties to check against can be reduced to the reachability problem.

The reduction technique is known as program instrumentation:

1. modify the code to add transitions to the error state when violation of the property is detected
2. check reachability of the error state

```
mutex_lock(&mtx);   →   if (mtx->locked==1)
                            goto ERROR;
                        mtx->locked = 1;
                                    ↑
```

The way we modify the code is called **model of the property**

Finite sets in property models

Many properties can be formulated in terms of sets.
Let's dream how their models would look like if C language contained first-class set type...

Finite sets in property models

Many properties can be formulated in terms of sets.
Let's dream how their models would look like if C language contained first-class set type...

Entry point

```
Set locked = ∅;
```

Finite sets in property models

Many properties can be formulated in terms of sets.
Let's dream how their models would look like if C language contained first-class set type...

Entry point		`Set locked = ∅;`

`mutex_lock(&mtx);`	$\rightarrow$	`if (mtx ∈ locked)`
		`    goto ERROR;`
		`locked ∪= mtx;`

Finite sets in property models

Many properties can be formulated in terms of sets.
Let's dream how their models would look like if C language contained first-class set type...

Entry point		Set locked = $\emptyset$;

mutex_lock(&mtx);	$\rightarrow$	if (mtx $\in$ locked)
		    goto ERROR;
		locked $\cup$= mtx;

mutex_unlock(&mtx);	$\rightarrow$	if (mtx $\notin$ locked)
		    goto ERROR;
		locked $\setminus$= mtx;

Finite sets in property models

Many properties can be formulated in terms of sets.
Let's dream how their models would look like if C language contained first-class set type...

Entry point		`Set locked = ∅;`

`mutex_lock(&mtx);`	→	`if (mtx ∈ locked)`
		`    goto ERROR;`
		`locked ∪= mtx;`

`mutex_unlock(&mtx);`	→	`if (mtx ∉ locked)`
		`    goto ERROR;`
		`locked \= mtx;`

Exit		`if (locked != ∅)` `    goto ERROR;`

Finite sets in property models — memory allocation

Memory allocation can also utilize sets:

Entry point		`Set allocated = ∅;` `void* next_block = 1;`
`ptr=malloc(size);`	$\longrightarrow$	`allocated ∪= next_block;` `next_block += 1;`

`free(ptr);`	$\longrightarrow$	`if (ptr ∉ allocated)`
		`    goto ERROR;`
		`allocated \= ptr;`

Exit (check leaks)		`if (allocated != ∅)` `    goto ERROR;`

Finite sets in property models — kernel lists

Special list structure that shouldn't contain two equal pointers:

Entry point		`Set values = ∅;`

`list_add(&dev);`	$\rightarrow$	`if (&dev ∈ values)`
		`    goto ERROR;`
		`values ∪= &dev;`

`list_del(&dev);`	$\rightarrow$	`values \= &dev;`

CounterExample-Guided Abstraction Refinement

Solves reachability problem by iterative algorithm. We build an "abstraction" of the model of the program until it proves inreachability of `ERROR` (BLAST, SLAM, CPAchecker).

- Start with a coarse abstraction of the program (ART — Abstract Reachability Tree)
- Find a counterexample path if it exists
- Transitions along the path are collected
- A logical **path formula** is built
- Satisfiability check (by solvers) determines if the error location is feasible
- Craig interpolation yields linear constraints that prove it infeasible
- Abstraction is refined, utilizing these constraints

How we modify CEGAR to add reasoning about finite sets

Solves reachability problem by iterative algorithm. We build an "abstraction" of the model of the program until it proves inreachability of `ERROR` (BLAST, SLAM, CPAchecker).

- Start with a coarse abstraction of the program (ART — Abstract Reachability Tree)
- Find a counterexample path if it exists
- Transitions along the path are collected
- A logical **path formula** is built
- Satisfiability check (by solvers) determines if the error location is feasible
- Craig interpolation yields linear constraints that prove it infeasible
- Abstraction is refined, utilizing these constraints

The aim is to modify the marked parts of algorithm to allow reasoning about finite sets.

Supported operations with sets

The models in Linux Driver Verification project demonstrated demand for the following operations:

Operation	Effect
Set construction	
`S = SetEmpty()`	$S \leftarrow \emptyset$
`S = SetAdd(T,expr)`	$S \leftarrow T \cup \{x\}$; x — a value of expr
`S = SetDel(F,expr)`	$S \leftarrow F \setminus \{y\}$; y — a value of expr
Set checking	
`z = SetInTest(S,expr)`	$z \leftarrow x \in S$; x — a value of expr
`z = SetEmptyTest(S)`	$z \leftarrow S \neq \emptyset$

Supported operations with sets

The models in Linux Driver Verification project demonstrated demand for the following operations:

Operation	Effect
Set construction	
`S = SetEmpty()`	$S \leftarrow \emptyset$
`S = SetAdd(T,expr)`	$S \leftarrow T \cup \{x\}$; x — a value of expr
`S = SetDel(F,expr)`	$S \leftarrow F \setminus \{y\}$; y — a value of expr
Set checking	
`z = SetInTest(S,expr)`	$z \leftarrow x \in S$; x — a value of expr
`z = SetEmptyTest(S)`	$z \leftarrow S \neq \emptyset$

Set is described by its **constitution**, a sequence of construction operations that yield the set.

The proposed way to construct path formula

- **Presence of an element in a set**

  The symbolic formula for presence check is built recursively, based on sequence of construction operations.

  The formula $f(e, S)$, where e — expression checked for presence, and S — constitution of a set:

`S = SetEmpty()`	$f(e, S) \equiv \mathit{false}$
`S = SetAdd(T,x)`	$f(e, S) \equiv (e = x) \vee f(e, T)$
`S = SetDel(F,y)`	$f(e, S) \equiv (e \neq x) \wedge f(e, F)$

The proposed way to construct path formula

- **Presence of an element in a set**
  The symbolic formula for presence check is built recursively, based on sequence of construction operations.

  The formula $f(e, S)$, where e — expression checked for presence, and S — constitution of a set:

`S = SetEmpty()`	$f(e, S) \equiv \mathit{false}$
`S = SetAdd(T,x)`	$f(e, S) \equiv (e = x) \vee f(e, T)$
`S = SetDel(F,y)`	$f(e, S) \equiv (e \neq x) \wedge f(e, F)$

- **Emptiness check**
  Emptiness check is based on observation that, in an empty set, every element added should be later deleted

The proposed way to construct path formula

- **Presence of an element in a set**
  The symbolic formula for presence check is built recursively, based on sequence of construction operations.

  The formula $f(e, S)$, where e — expression checked for presence, and S — constitution of a set:

`S = SetEmpty()`	$f(e, S) \equiv false$
`S = SetAdd(T,x)`	$f(e, S) \equiv (e = x) \vee f(e, T)$
`S = SetDel(F,y)`	$f(e, S) \equiv (e \neq x) \wedge f(e, F)$

- **Emptiness check**
  Emptiness check is based on observation that, in an empty set, every element added should be later deleted
- **Abstraction refinement** piggybacks on existing refinement algorithms

Features of the proposed approach

- **Small ART size**
  Each set construction operation requires just one ART node.
- **More complex formulæ**
  The formulæ to interpolate and be checked for Satisfiability have larger CNF, and are more complex. Sometimes the trade-off of ART size for formulæ size decreases analysis time (see "Large Block Encoding" by Beyer et. al.).
- **Incapability to use set operations inside loops**
  If the value of an expression added to/removed from a set changes after the operation, the algorithm may yield incorrect result.

Known approaches to reasoning about finite sets

- **Implement in C via a standard data structure**
  For example, as a Hash Table.
  Shortcomings: the algorithm relies on verification of arrays, lists and modular arithmetic. These features of C language are outside of correctly verifiable subset.

Known approaches to reasoning about finite sets

- **Implement in C via a standard data structure**
  For example, as a Hash Table.
  Shortcomings: the algorithm relies on verification of arrays, lists and modular arithmetic. These features of C language are outside of correctly verifiable subset.
- **Universal quantification trick**
  Branch unconditionally at each <u>construction</u> operation and track special characteristical variables.
  Shortcomings: exponential expansion of ART decreases analysis speed dramatically; only a subset of operations may be correctly verified.

Performance evaluation

Memory allocation correctness was verified by "trick" algorithm and the proposed one.

Leaks	Ignored		Checked	
Algorithm	Trick	**Proposed**	Trick	**Proposed**
1 chunk	1	1	1	1
2 chunks	4	3	5	4
3 chunks	41	6	52	10
4 chunks	443	17	540	X
5 chunks	1289	36	1553	80
6 chunks	> 2000	70	> 2000	X
7 chunks	> 2000	200	> 2000	X
8 chunks	> 2000	333	> 2000	X
9 chunks	> 2000	X	> 2000	X
10 chunks	> 2000	X	> 2000	X
15 chunks	> 2000	X	> 2000	X

(Time consumption in seconds. X - interpolation error)

Conclusion

Results of the work:

- A generic way to statically verify programs that contain finite sets operation was proposed
- Its limitations were described (no set operations within loops)
- The algorithm proposed was developed as a patch to BLAST tool
- The known and proposed solutions were evaluated, given BLAST platform as a basis

Conclusion

Results of the work:

- A generic way to statically verify programs that contain finite sets operation was proposed
- Its limitations were described (no set operations within loops)
- The algorithm proposed was developed as a patch to BLAST tool
- The known and proposed solutions were evaluated, given BLAST platform as a basis

Conclusion: the algorithm proposed has the same scalability as the known methods.

Thank you

http://linuxtesting.org/ $\rightarrow$ LDV Program

shved@ispras.ru

http://coldattic.info/shvedsky/pro/syrcose10

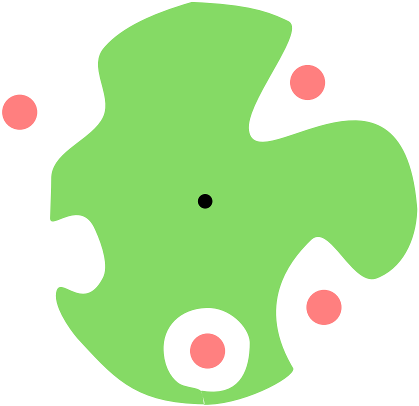
:-)

Counterexample-Guided Abstraction Refinement

Solves reachability problem by iterative algorithm. We build an "abstraction" of the model of the program until it proves inreachability of `ERROR` (BLAST, SLAM, CPAchecker).

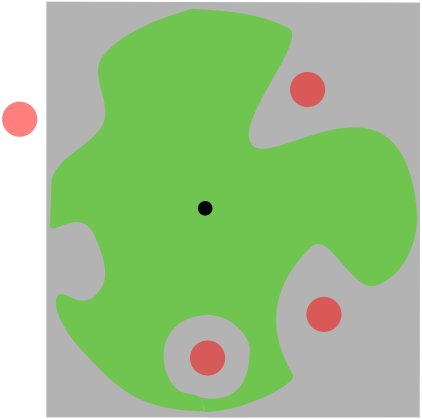
Counterexample-Guided Abstraction Refinement

Solves reachability problem by iterative algorithm. We build an "abstraction" of the model of the program until it proves inreachability of `ERROR` (BLAST, SLAM, CPAchecker).



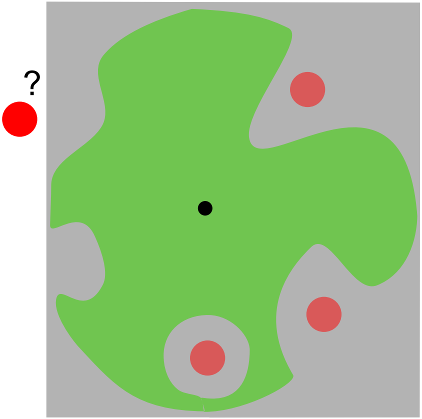
Counterexample-Guided Abstraction Refinement

Solves reachability problem by iterative algorithm. We build an "abstraction" of the model of the program until it proves inreachability of `ERROR` (BLAST, SLAM, CPAchecker).



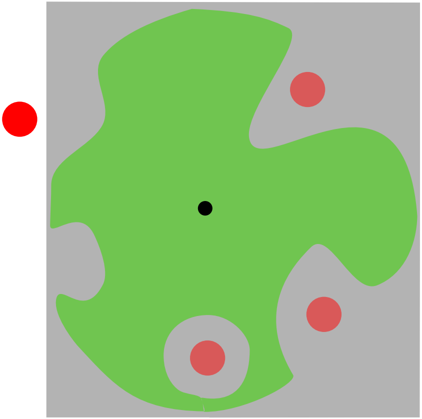
Counterexample-Guided Abstraction Refinement

Solves reachability problem by iterative algorithm. We build an "abstraction" of the model of the program until it proves inreachability of `ERROR` (BLAST, SLAM, CPAchecker).



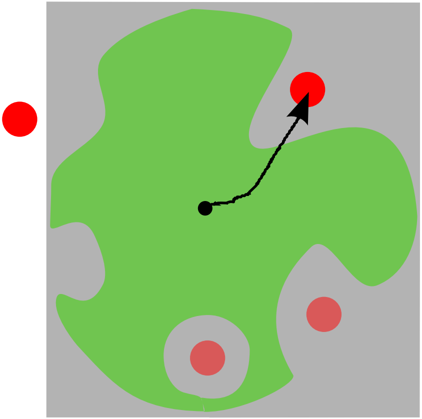
Counterexample-Guided Abstraction Refinement

Solves reachability problem by iterative algorithm. We build an "abstraction" of the model of the program until it proves inreachability of `ERROR` (BLAST, SLAM, CPAchecker).



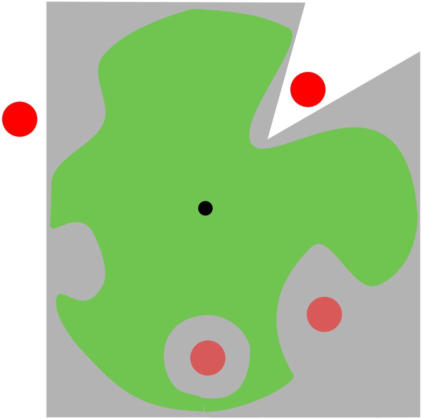
Counterexample-Guided Abstraction Refinement

Solves reachability problem by iterative algorithm. We build an "abstraction" of the model of the program until it proves inreachability of ERROR (BLAST, SLAM, CPAchecker).



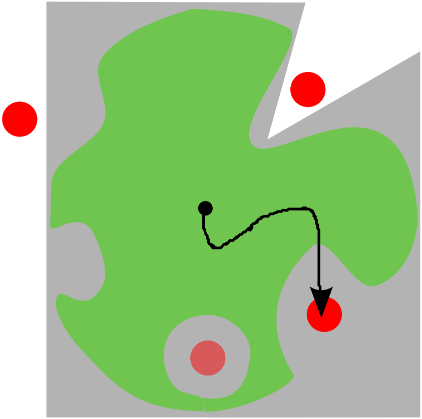
Counterexample-Guided Abstraction Refinement

Solves reachability problem by iterative algorithm. We build an "abstraction" of the model of the program until it proves inreachability of `ERROR` (BLAST, SLAM, CPAchecker).



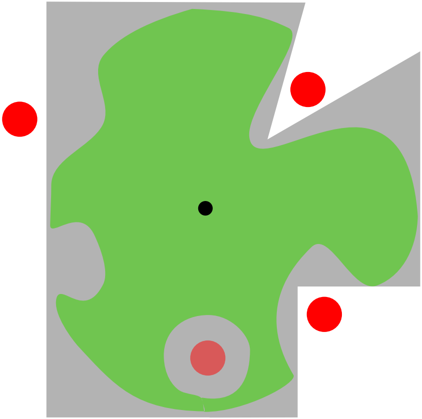
Counterexample-Guided Abstraction Refinement

Solves reachability problem by iterative algorithm. We build an "abstraction" of the model of the program until it proves inreachability of `ERROR` (BLAST, SLAM, CPAchecker).



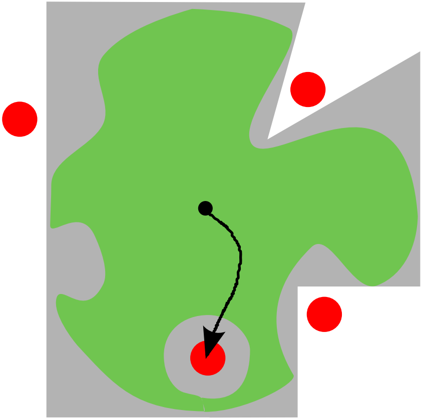
Counterexample-Guided Abstraction Refinement

Solves reachability problem by iterative algorithm. We build an "abstraction" of the model of the program until it proves inreachability of `ERROR` (BLAST, SLAM, CPAchecker).



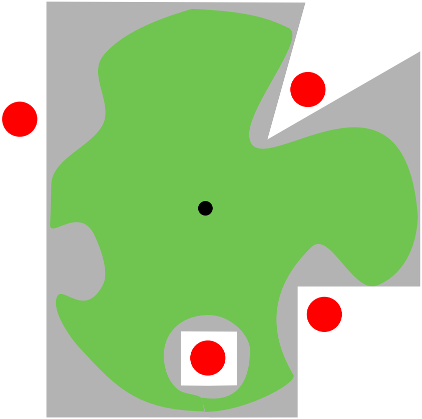
Counterexample-Guided Abstraction Refinement

Solves reachability problem by iterative algorithm. We build an "abstraction" of the model of the program until it proves inreachability of `ERROR` (BLAST, SLAM, CPAchecker).



Counterexample-Guided Abstraction Refinement

Solves reachability problem by iterative algorithm. We build an "abstraction" of the model of the program until it proves inreachability of `ERROR` (BLAST, SLAM, CPAchecker).



How predicates are analyzed

Path formula undergoes Craig interpolation at certain cut-points

Sample program:

```
int main()
{
  int x=0;
  int y=5;
  //cut-point
  if (x>1){
    error();
  }
}
```

How predicates are analyzed

Path formula undergoes Craig interpolation at certain cut-points

Sample program:

```
int main()
{
  int x=0;
  int y=5;
  //cut-point
  if (x>1){
    error();
  }
}
```

$x = 0 \wedge$

How predicates are analyzed

Path formula undergoes Craig interpolation at certain cut-points

Sample program:

```
int main()
{
  int x=0;
  int y=5;
  //cut-point
  if (x>1){
    error();
  }
}
```

$$\frac{x = 0 \wedge \\ y = 5 \wedge}{x > 1}$$

How predicates are analyzed

Path formula undergoes Craig interpolation at certain cut-points

Sample program:

```
int main()
{
  int x=0;
  int y=5;
  //cut-point
  if (x>1){
    error();
  }
}
```

$x = 0 \wedge$
$y = 5 \wedge$

$x > 1$

Solver checks if SAT?

How predicates are analyzed

Path formula undergoes Craig interpolation at certain cut-points

Sample program:

```
int main()
{
  int x=0;
  int y=5;
  //cut-point
  if (x>1){
    error();
  }
}
```

$x = 0 \wedge$

$y = 5 \wedge$

$x > 1$

Solver result:
UNSAT

How predicates are analyzed

Path formula undergoes Craig interpolation at certain cut-points

Sample program:

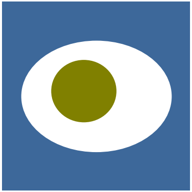
```
int main()
{
  int x=0;
  int y=5;
  //cut-point
  if (x>1){
    error();
  }
}
```

$x = 0 \land$

$y = 5 \land$

$x > 1$

Solver result:
UNSAT



How predicates are analyzed

Path formula undergoes Craig interpolation at certain cut-points

Sample program:

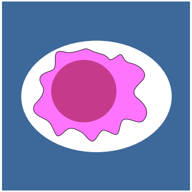
```
int main()
{
  int x=0;
  int y=5;
  //cut-point
  if (x>1){
    error();
  }
}
```

$x = 0 \wedge$

$y = 5 \wedge$

$x > 1$

Solver result:
UNSAT



Path formula undergoes Craig interpolation at certain cut-points

Sample program:

```
int main()
{
   int x=0;
   int y=5;
   //cut-point
   if (x>1){
      error();
   }
}
```

$$\frac{x = 0 \wedge y = 5 \wedge}{x > 1} \qquad \frac{x \leq 0 \wedge}{x > 1}$$

Solver result:
UNSAT