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
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
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;
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**
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 = ∅;

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;

Pavel Shved, ISPRAS

On Reasoning about Finite Sets in Software Checking
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 = ∅;

if (mtx ∈ locked)

goto ERROR;

locked ∪= mtx;

mutex_lock(&mtx); →

if (mtx ∉ locked)

goto ERROR;

locked \= mtx;

Exit

if (locked ≠ ∅)

goto ERROR;
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...

<table>
<thead>
<tr>
<th>Entry point</th>
<th>Set locked = ∅;</th>
</tr>
</thead>
<tbody>
<tr>
<td>mutex_lock(&amp;mtx);</td>
<td>if (mtx ∈ locked) goto ERROR;</td>
</tr>
<tr>
<td></td>
<td>locked ∪= mtx;</td>
</tr>
<tr>
<td>mutex_unlock(&amp;mtx);</td>
<td>if (mtx ∉ locked) goto ERROR;</td>
</tr>
<tr>
<td></td>
<td>locked = mtx;</td>
</tr>
</tbody>
</table>
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...

<table>
<thead>
<tr>
<th>Entry point</th>
<th>Set locked = ∅;</th>
</tr>
</thead>
<tbody>
<tr>
<td>mutex_lock(&amp;mtx); →</td>
<td>if (mtx ∈ locked) goto ERROR;</td>
</tr>
<tr>
<td></td>
<td>locked ∪= mtx;</td>
</tr>
<tr>
<td>mutex_unlock(&amp;mtx); →</td>
<td>if (mtx ∉ locked) goto ERROR;</td>
</tr>
<tr>
<td></td>
<td>locked = mtx;</td>
</tr>
<tr>
<td>Exit</td>
<td>if (locked != ∅) goto ERROR;</td>
</tr>
</tbody>
</table>
Memory allocation can also utilize sets:

**Entry point**

- `ptr = malloc(size);` → `allocated ∪= next_block;`
- `next_block += 1;`

**Free**

- `free(ptr);` → `if (ptr ∉ allocated)`
  - `goto ERROR;`
  - `allocated \= ptr;`

**Exit (check leaks)**

- `if (allocated != ∅)`
  - `goto ERROR;`
Special list structure that shouldn’t contain two equal pointers:

<table>
<thead>
<tr>
<th>Entry point</th>
<th>Set values = ∅;</th>
</tr>
</thead>
<tbody>
<tr>
<td>list_add(&amp;dev);</td>
<td>if (&amp;dev ∈ values) goto ERROR;</td>
</tr>
<tr>
<td></td>
<td>values ⊔= &amp;dev;</td>
</tr>
<tr>
<td>list_del(&amp;dev);</td>
<td>values ⊖= &amp;dev;</td>
</tr>
</tbody>
</table>
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.
The models in Linux Driver Verification project demonstrated demand for the following operations:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Set construction</strong></td>
<td></td>
</tr>
<tr>
<td>$S = \text{SetEmpty}()$</td>
<td>$S \leftarrow \emptyset$</td>
</tr>
<tr>
<td>$S = \text{SetAdd}(T, expr)$</td>
<td>$S \leftarrow T \cup {x}; \ x \leftarrow \text{a value of } expr$</td>
</tr>
<tr>
<td>$S = \text{SetDel}(F, expr)$</td>
<td>$S \leftarrow F \setminus {y}; \ y \leftarrow \text{a value of } expr$</td>
</tr>
<tr>
<td><strong>Set checking</strong></td>
<td></td>
</tr>
<tr>
<td>$z = \text{SetInTest}(S, expr)$</td>
<td>$z \leftarrow x \in S; \ x \leftarrow \text{a value of } expr$</td>
</tr>
<tr>
<td>$z = \text{SetEmptyTest}(S)$</td>
<td>$z \leftarrow S \neq \emptyset$</td>
</tr>
</tbody>
</table>
Supported operations with sets

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

<table>
<thead>
<tr>
<th>Operation</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Set construction</strong></td>
<td></td>
</tr>
<tr>
<td><code>S = SetEmpty()</code></td>
<td><code>S ← ∅</code></td>
</tr>
<tr>
<td><code>S = SetAdd(T, expr)</code></td>
<td><code>S ← T ∪ {x}; x — a value of expr</code></td>
</tr>
<tr>
<td><code>S = SetDel(F, expr)</code></td>
<td><code>S ← F \ {y}; y — a value of expr</code></td>
</tr>
<tr>
<td><strong>Set checking</strong></td>
<td></td>
</tr>
<tr>
<td><code>z = SetInTest(S, expr)</code></td>
<td><code>z ← x ∈ S; x — a value of expr</code></td>
</tr>
<tr>
<td><code>z = SetEmptyTest(S)</code></td>
<td><code>z ← S ≠ ∅</code></td>
</tr>
</tbody>
</table>

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:

- $S = \text{SetEmpty}()$  \quad $f(e, S) \equiv false$
- $S = \text{SetAdd}(T, x)$  \quad $f(e, S) \equiv (e = x) \lor f(e, T)$
- $S = \text{SetDel}(F, y)$  \quad $f(e, S) \equiv (e \neq x) \land f(e, F)$
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:
  
  - $S = \text{SetEmpty()} \implies f(e, S) \equiv \text{false}$
  - $S = \text{SetAdd}(T, x) \implies f(e, S) \equiv (e = x) \lor f(e, T)$
  - $S = \text{SetDel}(F, y) \implies f(e, S) \equiv (e \neq x) \land f(e, F)$

- **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:
  
  - $S = \text{SetEmpty}()$  \hspace{1cm} $f(e, S) \equiv \text{false}$
  - $S = \text{SetAdd}(T, x)$  \hspace{1cm} $f(e, S) \equiv (e = x) \lor f(e, T)$
  - $S = \text{SetDel}(F, y)$  \hspace{1cm} $f(e, S) \equiv (e \neq x) \land f(e, F)$

- **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 construction operation and track special characteristical variables.
  Shortcomings: exponential expansion of ART decreases analysis speed dramatically; only a subset of operations may be correctly verified.
Memory allocation correctness was verified by “trick” algorithm and the proposed one.

<table>
<thead>
<tr>
<th>Leaks</th>
<th>Ignored</th>
<th>Checked</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Trick</td>
<td>Proposed</td>
</tr>
<tr>
<td>1 chunk</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2 chunks</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>3 chunks</td>
<td>41</td>
<td>6</td>
</tr>
<tr>
<td>4 chunks</td>
<td>443</td>
<td>17</td>
</tr>
<tr>
<td>5 chunks</td>
<td>1289</td>
<td>36</td>
</tr>
<tr>
<td>6 chunks</td>
<td>&gt; 2000</td>
<td>70</td>
</tr>
<tr>
<td>7 chunks</td>
<td>&gt; 2000</td>
<td>200</td>
</tr>
<tr>
<td>8 chunks</td>
<td>&gt; 2000</td>
<td>333</td>
</tr>
<tr>
<td>9 chunks</td>
<td>&gt; 2000</td>
<td>X</td>
</tr>
<tr>
<td>10 chunks</td>
<td>&gt; 2000</td>
<td>X</td>
</tr>
<tr>
<td>15 chunks</td>
<td>&gt; 2000</td>
<td>X</td>
</tr>
</tbody>
</table>

(Time consumption in seconds. X - interpolation error)
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.
http://linuxtesting.org/ → LDV Program
shved@ispras.ru
http://coldattic.info/shvedsky/pro/syrcose10
:-)
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).
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).
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).
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).
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).
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).
Solves reachability problem by iterative algorithm. We build an “abstraction” of the model of the program until it proves inreachability of \texttt{ERROR} (BLAST, SLAM, CPAchecker).
Solves reachability problem by iterative algorithm. We build an “abstraction” of the model of the program until it proves inreachability of \texttt{ERROR} (BLAST, SLAM, CPAchecker).
Path formula undergoes Craig interpolation at certain cut-points

Sample program:

```c
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:

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

\[ x = 0 \land \]

Pavel Shved, ISPRAS

On Reasoning about Finite Sets in Software Checking
How predicates are analyzed

Path formula undergoes Craig interpolation at certain cut-points

Sample program:

```c
int main ()
{
    int x=0;  \( x = 0 \wedge \)
    int y=5;  \( y = 5 \wedge \)
    //cut-point  \( x \geq 1 \)
    if (x>1){
        error();
    }
}
```
Path formula undergoes Craig interpolation at certain cut-points

Sample program:

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

\[ x = 0 \land y = 5 \land x > 1 \]

Solver checks if SAT?
Path formula undergoes Craig interpolation at certain cut-points

Sample program:

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

Solver result: UNSAT
Path formula undergoes Craig interpolation at certain cut-points

Sample program:

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

Solver result: UNSAT
Path formula undergoes Craig interpolation at certain cut-points

Sample program:

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

Solver result: UNSAT
How predicates are analyzed

Path formula undergoes Craig interpolation at certain cut-points

Sample program:

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

Solver result: UNSAT