A Generic Approach to Symbolic Execution

Andrei Arusoaie, Dorel Lucanu, Vlad Rusu

INRIA
Lille, France
vlad.rusu@inria.fr

Faculty of Computer Science
Alexandru Ioan Cuza University, Iași, Romania
{andrei.arusoaie, dlucanu}@info.uaic.ro

Plan

What is Symbolic Execution?

- introduced in 1976 by James C. King;
- execute programs with *symbolic input*: e.g., x instead of 42;
- conditionals generate *branching executions*: *symbolic execution tree*;
- *path conditions* can be used to prune tree if found unsatisfiable;
- applications: test case generation, model checking, deductive verification, . . .

Example

read n;
if n $>$ 0 **then**
    print 'error';
end if;

Example

Symbolic execution when n $=$ n:

```
read n;
if n > 0 then
    print 'error';
end if;
```

Example

```
read n;
if n > 0 then
    print 'error';
end if;
```

Symbolic execution when n = n:

- path condition: $n > 0$, output: error
- path condition: $n \leq 0$, output:

Example: can this program print *error*?

```
class List {
   int a[10],size, capacity;
   void insert (int x) {
      if (size < capacity)
         a[size] = x; ++size;
   }

   void delete(int x){
   int i = 0;
   while(i < size-1 && a[i] ≠ x) i++;
   if (a[i] == x) {
      while (i < size - 1) {
         a[i] = a[i+1];
         i = i + 1;
      }
   size = size - 1;
   }
...
}
```

```
class OrderedList extends List {
   void insert(int x) {
      if (size < capacity) {
      int i = 0, k;
      while(i < size && a[i] ≤ x) i++;
         ++size; k = size - 1;
      while(k > i) {
         a[k] = a[k-1]; k = k - 1;
      }
      a[i] = x;
   }
}

void Main() {
   List l1 = new List();
   ... // initialise l1, read x
   List l2 = l1.copy();
   l1.insert(x); l1.delete(x);
   if (l2.eqTo(l1) == false)
      print("error");
}
```

Related Work

Many tools, highly optimised for *specific* languages:

- Java PathFinder
- PEX for C$\sharp$
- KLEE for LLVM ...

Related Work

Many tools, highly optimised for *specific* languages:

- Java PathFinder
- PEX for C♯
- KLEE for LLVM . . .

... but what happens when we change the (version) of the language?

A Nondeterminstic C Program (according to the C standard)

```
int r;
int f(int x) {
  return (r = x);
}
int main() {
  return f(a) + f(b); //a,b symbolic values
}
```

...nondeterministic behavior: `r` can be either a or b

A Nondeterminstic C Program (according to the C standard)

```
int r;
int f(int x) {
  return (r = x);
}
int main() {
  return f(a) + f(b); //a,b symbolic values
}
```

. . . nondeterministic behavior: `r` can be either a or b

KLEE returns: `path num explored here:  1`

- It is compiler dependent!

A Nondeterminstic C Program (according to the C standard)

```
int r;
int f(int x) {
  return (r = x);
}
int main() {
  return f(a) + f(b); //a,b symbolic values
}
```

... nondeterministic behavior: `r` can be either a or b

KLEE returns: `path num explored here:  1`

- It is compiler dependent!
- Symbolic execution has to be based on the language's complete formal semantics, according to the language's manual
- For C see [Ellison&Roșu, POPL'12]

Our contribution

Formal and *language-independent* framework for symbolic execution:

Our contribution

Formal and *language-independent* framework for symbolic execution:

- based on the *operational semantics* of programming languages

Our contribution

Formal and *language-independent* framework for symbolic execution:

- based on the *operational semantics* of programming languages

Our contribution

Formal and *language-independent* framework for symbolic execution:

- based on the *operational semantics* of programming languages
- automatically transforms language $\mathcal{L}$ into *symbolic language* $\mathcal{L}^{\mathfrak{s}}$:
  symbolic execution in $\mathcal{L} \triangleq$ concrete execution in $\mathcal{L}^{\mathfrak{s}}$;

Our contribution

Formal and *language-independent* framework for symbolic execution:

- based on the *operational semantics* of programming languages
- automatically transforms language $\mathcal{L}$ into *symbolic language* $\mathcal{L}^{\mathfrak{s}}$: symbolic execution in $\mathcal{L} \triangleq$ concrete execution in $\mathcal{L}^{\mathfrak{s}}$;

Our contribution

Formal and *language-independent* framework for symbolic execution:

- based on the *operational semantics* of programming languages
- automatically transforms language $\mathcal{L}$ into *symbolic language* $\mathcal{L}^{\mathfrak{s}}$: symbolic execution in $\mathcal{L} \triangleq$ concrete execution in $\mathcal{L}^{\mathfrak{s}}$;
- to each concrete program-execution in $\mathcal{L}$ there exists a *feasible* symbolic-program execution in $\mathcal{L}^{\mathfrak{s}}$ on the same path, *and reciprocally*;

Our contribution

Formal and *language-independent* framework for symbolic execution:

- based on the *operational semantics* of programming languages
- automatically transforms language $\mathcal{L}$ into *symbolic language* $\mathcal{L}^{\mathfrak{s}}$: symbolic execution in $\mathcal{L} \triangleq$ concrete execution in $\mathcal{L}^{\mathfrak{s}}$;
- to each concrete program-execution in $\mathcal{L}$ there exists a *feasible* symbolic-program execution in $\mathcal{L}^{\mathfrak{s}}$ on the same path, *and reciprocally*;

Our contribution

Formal and *language-independent* framework for symbolic execution:

- based on the *operational semantics* of programming languages
- automatically transforms language $\mathcal{L}$ into *symbolic language* $\mathcal{L}^{\mathfrak{s}}$: symbolic execution in $\mathcal{L} \triangleq$ concrete execution in $\mathcal{L}^{\mathfrak{s}}$;
- to each concrete program-execution in $\mathcal{L}$ there exists a *feasible* symbolic-program execution in $\mathcal{L}^{\mathfrak{s}}$ on the same path, *and reciprocally*;
- is implemented as a prototype tool in the $\mathbb{K}$ language definition framework, online at `http://k-framework.org`

Our contribution

Formal and *language-independent* framework for symbolic execution:

- based on the *operational semantics* of programming languages
- automatically transforms language $\mathcal{L}$ into *symbolic language* $\mathcal{L}^{\mathfrak{s}}$: symbolic execution in $\mathcal{L} \triangleq$ concrete execution in $\mathcal{L}^{\mathfrak{s}}$;
- to each concrete program-execution in $\mathcal{L}$ there exists a *feasible* symbolic-program execution in $\mathcal{L}^{\mathfrak{s}}$ on the same path, *and reciprocally*;
- is implemented as a prototype tool in the $\mathbb{K}$ language definition framework, online at `http://k-framework.org`

Our contribution

Formal and *language-independent* framework for symbolic execution:

- based on the *operational semantics* of programming languages
- automatically transforms language $\mathcal{L}$ into *symbolic language* $\mathcal{L}^{\mathfrak{s}}$: symbolic execution in $\mathcal{L} \triangleq$ concrete execution in $\mathcal{L}^{\mathfrak{s}}$;
- to each concrete program-execution in $\mathcal{L}$ there exists a *feasible* symbolic-program execution in $\mathcal{L}^{\mathfrak{s}}$ on the same path, *and reciprocally*;
- is implemented as a prototype tool in the $\mathbb{K}$ language definition framework, online at `http://k-framework.org`
- Restriction: requires distinction between *code* and *data*; only data is symbolic

IMP: a simple IMPerative language defined in $\mathbb{K}$ (syntax)

$$
\begin{aligned}
\mathit{Id} &::= \text{domain of identifiers} \\
\mathit{Int} &::= \text{domain of integer numbers (including operations)} \\
\mathit{Bool} &::= \text{domain of boolean constants (including operations)}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{AExp} ::= \ & \mathit{Int} && \mid \mathit{AExp}\ /\ \mathit{AExp}\ [\text{strict}] \\
& \mid \mathit{Id} && \mid \mathit{AExp} * \mathit{AExp}\ [\text{strict}] \\
& \mid (\mathit{AExp}) && \mid \mathit{AExp} + \mathit{AExp}\ [\text{strict}]
\end{aligned}
$$

$$
\begin{aligned}
\mathit{BExp} ::= \ & \mathit{Bool} \\
& \mid (\mathit{BExp}) && \mid \mathit{AExp} \texttt{ <= } \mathit{AExp}\ [\text{strict}] \\
& \mid \texttt{not}\ \mathit{BExp}\ [\text{strict}] && \mid \mathit{BExp}\ \texttt{and}\ \mathit{BExp}\ [\text{strict(1)}]
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Stmt} ::= \ & \texttt{skip} \mid \{\ \mathit{Stmt}\ \} \mid \mathit{Stmt}\ ;\ \mathit{Stmt} \mid \mathit{Id} \texttt{ := } \mathit{AExp}\ [\text{strict(2)}] \\
& \mid \texttt{while}\ \mathit{BExp}\ \texttt{do}\ \mathit{Stmt} \\
& \mid \texttt{if}\ \mathit{BExp}\ \texttt{then}\ \mathit{Stmt}\ \texttt{else}\ \mathit{Stmt}\ [\text{strict(1)}]
\end{aligned}
$$

The $\mathbb{K}$ semantics of IMP

- Configuration: $\langle\langle \mathit{Code}\rangle_{\mathsf{k}}\langle \mathit{Map}_{\mathit{Id},\mathit{Int}}\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}$

The $\mathbb{K}$ semantics of IMP

- Configuration: $\langle\langle Code\rangle_{\mathsf{k}}\langle Map_{Id,Int}\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}$
- $\mathbb{K}$ semantical rules:

$$\langle\langle I_1 + I_2 \ \cdots\rangle_{\mathsf{k}} \ \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle I_1 +_{Int} I_2 \ \cdots\rangle_{\mathsf{k}} \ \cdots\rangle_{\mathsf{cfg}}$$

The $\mathbb{K}$ semantics of IMP

- Configuration: $\langle\langle Code\rangle_{\mathsf{k}}\langle Map_{Id,Int}\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}$
- $\mathbb{K}$ semantical rules:

$$\langle\langle I_1 \texttt{ + } I_2 \ \cdots\rangle_{\mathsf{k}}\ \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle I_1 +_{Int} I_2 \ \cdots\rangle_{\mathsf{k}}\ \cdots\rangle_{\mathsf{cfg}}$$

$$\ldots$$

The $\mathbb{K}$ semantics of IMP

- Configuration: $\langle\langle Code\rangle_{\mathsf{k}}\langle Map_{Id,Int}\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}$
- $\mathbb{K}$ semantical rules:

$$\langle\langle I_1 + I_2 \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle I_1 +_{Int} I_2 \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}}$$

$$\ldots$$

$$\langle\langle \texttt{if}\ true\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle S_1\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}}$$

$$\langle\langle \texttt{if}\ false\ \texttt{then}\ S_1\ \texttt{else}\ S_2\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle S_2\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}}$$

The $\mathbb{K}$ semantics of IMP

- Configuration: $\langle\langle Code\rangle_{\mathsf{k}}\langle Map_{Id,Int}\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}$
- $\mathbb{K}$ semantical rules:

$$\langle\langle I_1 \texttt{ + } I_2 \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle I_1 +_{Int} I_2 \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}}$$

$$\ldots$$

$$\langle\langle \texttt{if } \mathit{true} \texttt{ then } S_1 \texttt{ else } S_2 \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle S_1\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}}$$

$$\langle\langle \texttt{if } \mathit{false} \texttt{ then } S_1 \texttt{ else } S_2\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle S_2\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}}$$

$$\langle\langle \texttt{while } B \texttt{ do } S \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \Rightarrow$$
$$\langle\langle \texttt{if } B \texttt{ then\{ } S \texttt{ ;while } B \texttt{ do } S \texttt{ \}else skip } \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}}$$

The $\mathbb{K}$ semantics of IMP

- Configuration: $\langle\langle Code\rangle_{\mathsf{k}}\langle Map_{Id,Int}\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}$
- $\mathbb{K}$ semantical rules:

$$\langle\langle I_1 \texttt{ + } I_2 \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle I_1 +_{Int} I_2 \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}}$$

$$\ldots$$

$$\langle\langle \texttt{if } true \texttt{ then } S_1 \texttt{ else } S_2 \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle S_1\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}}$$

$$\langle\langle \texttt{if } false \texttt{ then } S_1 \texttt{ else } S_2\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle S_2\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}}$$

$$\langle\langle \texttt{while } B \texttt{ do } S \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \Rightarrow$$
$$\langle\langle \texttt{if } B \texttt{ then\{ } S \texttt{ ;while } B \texttt{ do } S \texttt{ \}else skip} \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}}$$

$$\ldots$$

- ... + rewrite rules automatically generated from the [strict] annotations

Basic Ingredients of a Language Definition $\mathcal{L} = (\Sigma, \mathcal{T}, \mathcal{S})$

A language definition $\mathcal{L}$ is a triple $(\Sigma, \mathcal{T}, \mathcal{S})$, where

- Σ is an algebraic signature
  - includes a sub-signature (Σ^{Data}) of data sorts $(\mathit{Int}, \mathit{Bool}, \ldots)$
  - sort *Cfg* for configurations
  - sub-signature for programs $(\mathit{AExp}, \mathit{BExp}, \mathit{Stmt}, \ldots)$
- $\mathcal{T}$ is a Σ-model
  - $\mathcal{D}$ the sub model of data (Σ^{Data}-model)
  - $\mathcal{T}$ is the model freely generated by $\mathcal{D}$
- $\mathcal{S}$ is a set of rules $l \Rightarrow r$ *when* b
  - l and r are configuration terms with variables
  - b the condition (constraint)
- $\mathcal{S}$ defines a transition system $\Rightarrow_{\mathcal{S}}$ on $\mathcal{T}_{Cfg}$

Plan

Symbolic Execution by Language Transformation $\mathcal{L} \to \mathcal{L}^{\mathfrak{s}}$

If $\mathcal{L} = (\Sigma, \mathcal{T}, \mathcal{S})$ then $\mathcal{L}^{\mathfrak{s}} = (\Sigma^{\mathfrak{s}}, \mathcal{T}^{\mathfrak{s}}, \mathcal{S}^{\mathfrak{s}})$, where

- $\Sigma^{\mathfrak{s}} = \Sigma$ + infinite set of symbolic Values $V^{\mathfrak{s}}$
- $\mathcal{D}^{\mathfrak{s}}$ = algebra of symbolic expressions over $\mathcal{D} \cup V^{\mathfrak{s}}$
  $\mathcal{T}^{\mathfrak{s}}$ = the model freely generated by $\mathcal{D}^{\mathfrak{s}}$
- $\mathcal{S}^{\mathfrak{s}}$ automatically obtained from rules $\mathcal{S}$ by
  - left-linearisation
  - replacement of data sub terms in lhs by fresh variables
  - collect path condition
- $\mathcal{S}^{\mathfrak{s}}$ generates a transition system $\Rightarrow_{\mathcal{S}^{\mathfrak{s}}}$ on $\mathcal{T}^{\mathfrak{s}}_{Cfg^{\mathfrak{s}}}$

Rule transformations

Consider the following rule for *if* from the IMP semantics:

- $\langle\langle \texttt{if}\ \mathit{true}\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \cdots\rangle_{\mathsf{k}}\ \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle S_1\ \cdots\rangle_{\mathsf{k}}\ \cdots\rangle_{\mathsf{cfg}}$

Rule transformations

Consider the following rule for *if* from the IMP semantics:

- $\langle\langle \texttt{if}\ \mathit{true}\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \cdots\rangle_{\mathsf{k}}\ \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle S_1\ \cdots\rangle_{\mathsf{k}}\ \cdots\rangle_{\mathsf{cfg}}$

Replace *true* with a variable B, and add the condition $B = \mathit{true}$:

- $\langle\langle \texttt{if}\ B\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \cdots\rangle_{\mathsf{k}}\ \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle S_1\ \cdots\rangle_{\mathsf{k}}\ \cdots\rangle_{\mathsf{cfg}}$ *when* $B = \mathit{true}$

Rule transformations

Consider the following rule for *if* from the IMP semantics:

- $\langle\langle \texttt{if}\ \mathit{true}\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \cdots\rangle_{\mathsf{k}}\ \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle S_1\ \cdots\rangle_{\mathsf{k}}\ \cdots\rangle_{\mathsf{cfg}}$

Replace *true* with a variable B, and add the condition $B = \mathit{true}$:

- $\langle\langle \texttt{if}\ B\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \cdots\rangle_{\mathsf{k}}\ \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle S_1\ \cdots\rangle_{\mathsf{k}}\ \cdots\rangle_{\mathsf{cfg}}$ *when* $B = \mathit{true}$

Now B matches on all terms of sort *Bool*, including symbolic expressions
Example: $n >_{Int} 0$

Rule transformations

Consider the following rule for *if* from the IMP semantics:

- $\langle\langle \texttt{if}\ \mathit{true}\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \cdots\rangle_{\mathsf{k}}\ \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle S_1\ \cdots\rangle_{\mathsf{k}}\ \cdots\rangle_{\mathsf{cfg}}$

Replace *true* with a variable B, and add the condition $B = \mathit{true}$:

- $\langle\langle \texttt{if}\ B\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \cdots\rangle_{\mathsf{k}}\ \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle S_1\ \cdots\rangle_{\mathsf{k}}\ \cdots\rangle_{\mathsf{cfg}}$ *when* $B = \mathit{true}$

Now B matches on all terms of sort *Bool*, including symbolic expressions
Example: $n >_{\mathit{Int}} 0$

The rule's condition after matching (e.g. $n >_{\mathit{Int}} 0 = \mathit{true}$) is added to the current path condition.

Plan

Relation between Concrete and (Feasible) Symbolic Executions

Feasible symbolic execution: its path condition is satisfiable

- *Coverage*: for each concrete execution there is a *feasible* symbolic execution, which takes the same path in program's control flow graph;
- *Precision*: for each *feasible* symbolic execution there is a concrete execution, which takes the same path in program's control flow graph.

Plan

Symbolic execution in $\mathbb{K}$

- Our prototype is incorporated in the $\mathbb{K}$ framework

Symbolic execution in $\mathbb{K}$

- Our prototype is incorporated in the $\mathbb{K}$ framework
- Generates $\mathcal{L}^{\mathfrak{s}}$ automatically from $\mathcal{L}$
  - `kompile imp.k --backend symbolic`

Symbolic execution in $\mathbb{K}$

- Our prototype is incorporated in the $\mathbb{K}$ framework
- Generates $\mathcal{L}^{\mathfrak{s}}$ automatically from $\mathcal{L}$
  - `kompile imp.k --backend symbolic`
- Run programs with both concrete and symbolic input values

Symbolic execution in $\mathbb{K}$

- Our prototype is incorporated in the $\mathbb{K}$ framework
- Generates $\mathcal{L}^{\mathfrak{s}}$ automatically from $\mathcal{L}$
    - `kompile imp.k --backend symbolic`
- Run programs with both concrete and symbolic input values
- Other features: initial path condition, symbolic execution tree, stepper, bound, pattern search;

Example: can this program print *error*?

```
class List {
   int a[10],size, capacity;
   void insert (int x) {
      if (size < capacity)
         a[size] = x; ++size;
   }

   void delete(int x){
   int i = 0;
   while(i < size-1 && a[i] ≠ x) i++;
   if (a[i] == x) {
      while (i < size - 1) {
         a[i] = a[i+1];
         i = i + 1;
      }
   size = size - 1;
   }
...
}
```

```
class OrderedList extends List {
   void insert(int x) {
      if (size < capacity) {
      int i = 0, k;
      while(i < size && a[i] ≤ x) i++;
         ++size; k = size - 1;
      while(k > i) {
         a[k] = a[k-1]; k = k - 1;
      }
      a[i] = x;
   }
}

void Main() {
   List l1 = new List();
   ... // initialise l1, read x
   List l2 = l1.copy();
   l1.insert(x); l1.delete(x);
   if (l2.eqTo(l1) == false)
      print("error");
}
```

Pattern search

```
void Main() {
   List l1 = new List();
   ... // initialise l1, read x
   List l2 = l1.copy();
   l1.insert(x); l1.delete(x);
   if (l2.eqTo(l1) == false)
      print("error");
}
```

- `$ krun lists.kool -search -cIN="`e_1 e_2 x`"`
  `-pattern="<T> <out>` *error* `</out> B:Bag </T>"`

  Solution 1, State 50:

  <path-condition>

  $e_1 = x \wedge_{Bool} \neg_{Bool}(e_1 = e_2)$

  </path-condition>

  ...
- `l1 = [e2, e1] and l2 = [e1, e2]`

Pattern search

```
void Main() {
   List l1 = new OrderedList();
   ... // initialise l1, read x
   List l2 = l1.copy();
   l1.insert(x); l1.delete(x);
   if (l2.eqTo(l1) == false)
      print("error");
}
```

- $ krun lists.kool -search -cIN="e_1 e_2 x"
  -pattern="<T> <out> *error* </out> B:Bag </T>"

  Search results:

  No search results

- NB: this is not verification!

Plan

Conclusion & Future work

Conclusion

- A language independent framework for symbolic execution
- Relation between concrete and symbolic execution
- A prototype implementation

Conclusion & Future work

Conclusion

- A language independent framework for symbolic execution
- Relation between concrete and symbolic execution
- A prototype implementation

Future work

- Program verification (Reachability Logic)

Thank you!

The $\mathbb{K}$ framework web page: http://k-framework.org

Paper examples: http://fmse.info.uaic.ro/tools/Symbolic