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

1. Introduction and Motivation
   - An Example
   - In General

2. Symbolic Execution by Language Transformation

3. Formal Properties of Symbolic Execution

4. Prototype Implementation

5. Conclusion
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

Symbolic execution when $n = n$:

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

Example: can this program print \textit{error}?

class \texttt{List} 
{
    int a[10], size, capacity;
    void \texttt{insert} (int x) 
    {
        if (size < capacity)
            a[size] = x; ++size;
    }
    void \texttt{delete} (int x) 
    {
        int i = 0;
        while (i < size-1 && a[i] \neq x) i++;
        if (a[i] == x) 
        {
            while (i < size - 1)
                {
                    a[i] = a[i+1];
                    i = i + 1;
                }
            size = size - 1;
        }
    }
}

class \texttt{OrderedList} extends \texttt{List} 
{
    void \texttt{insert} (int x) 
    {
        if (size < capacity)
        {
            int i = 0, k;
            while (i < size && a[i] \leq x) i++;
            ++size; k = size - 1;
            while (k > i) 
            {
                a[k] = a[k-1]; k = k - 1;
            }
            a[i] = x;
        }
    }
    void \texttt{Main}() 
    {
        List l1 = new \texttt{List}();
        ... // initialise l1, read x
        List l2 = l1.copy();
        l1.insert(x); l1.delete(x);
        if (l2.eqTo(l1) == false)
            print(\"error\");
    }
}
Many tools, highly optimised for *specific* languages:

- Java PathFinder
- PEX for C♯
- 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 Nondeterministic C Program (according to the C standard)

```c
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 Nondeterministic 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 Nondeterministic C Program (according to the C standard)

```c
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&Roşu, 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* \( \mathcal{L}^s \):
  - symbolic execution in \( \mathcal{L} \) \( \triangleq \) concrete execution in \( \mathcal{L}^s \);
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* $\mathcal{L}^s$:
  symbolic execution in $\mathcal{L} \triangleq$ concrete execution in $\mathcal{L}^s$;
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 $\mathcal{L}^s$: symbolic execution in $\mathcal{L} \triangleq$ concrete execution in $\mathcal{L}^s$;
- to each concrete program-execution in $\mathcal{L}$ there exists a feasible symbolic-program execution in $\mathcal{L}^s$ on the same path, and reciprocally;

Restriction: requires distinction between code and data; only data is symbolic.
**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* $\mathcal{L}^s$: symbolic execution in $\mathcal{L} \triangleq$ concrete execution in $\mathcal{L}^s$;
- to each concrete program-execution in $\mathcal{L}$ there exists a *feasible* symbolic-program execution in $\mathcal{L}^s$ on the same path, *and reciprocally*;

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

Restriction: requires distinction between code and data; only data is symbolic.
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* $\mathcal{L}_s$: symbolic execution in $\mathcal{L} \triangleq$ concrete execution in $\mathcal{L}_s$;
- to each concrete program-execution in $\mathcal{L}$ there exists a feasible symbolic-program execution in $\mathcal{L}_s$ on the same path, and reciprocally;
- is implemented as a prototype tool in the **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* \( \mathcal{L}^s \): symbolic execution in \( \mathcal{L} \triangleq \) concrete execution in \( \mathcal{L}^s \);
- to each concrete program-execution in \( \mathcal{L} \) there exists a *feasible* symbolic-program execution in \( \mathcal{L}^s \) 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](http://k-framework.org)
- Restriction: requires distinction between *code* and *data*; only data is symbolic
**IMP: a simple IMPerative language defined in K (syntax)**

\[
Id ::= \text{domain of identifiers}\\
Int ::= \text{domain of integer numbers (including operations)}\\
Bool ::= \text{domain of boolean constants (including operations)}\\
\]

\[
AExp ::= Int \mid AExp / AExp \quad \text{[strict]}\\
\mid Id \mid AExp \ast AExp \quad \text{[strict]}\\
\mid (AExp) \mid AExp + AExp \quad \text{[strict]}\\
\]

\[
BExp ::= Bool\\
\mid (BExp) \quad \mid AExp \leq AExp \quad \text{[strict]}\\
\mid \text{not } BExp \quad \text{[strict]} \mid BExp \text{ and } BExp \quad \text{[strict(1)]}\\
\]

\[
Stmt ::= \text{skip} \mid \{ Stmt \} \mid Stmt ; Stmt \mid Id := AExp \quad \text{[strict(2)]}\\
\mid \text{while } BExp \text{ do } Stmt\\
\mid \text{if } BExp \text{ then } Stmt \text{ else } Stmt \quad \text{[strict(1)]}\\
\]
The $K$ semantics of IMP

- Configuration: $\langle\langle Code \rangle_K \langle Map_{Id,Int} \rangle_{env} \rangle_{cfg}$
The $\mathcal{K}$ semantics of IMP

- Configuration: $\langle\langle \text{Code} \rangle_k \langle \text{MapId,Int} \rangle_{\text{env}} \rangle_{\text{cfg}}$
- $\mathcal{K}$ semantical rules:
  \[
  \langle\langle l_1 + l_2 \ldots \rangle_k \ldots \rangle_{\text{cfg}} \Rightarrow \langle\langle l_1 + \text{Int} \ l_2 \ldots \rangle_k \ldots \rangle_{\text{cfg}}
  \]
The $K$ semantics of IMP

- **Configuration**: $\langle\langle \text{Code}\rangle_k\langle Map_{\text{Id},\text{Int}}\rangle_{\text{env}}\rangle_{\text{cfg}}$
- **$K$ semantical rules**:
  
  $\langle\langle l_1 + l_2 \ldots \rangle_k \ldots \rangle_{\text{cfg}} \Rightarrow \langle\langle l_1 + \text{Int} \ l_2 \ldots \rangle_k \ldots \rangle_{\text{cfg}}$

  ...

  ... + rewrite rules automatically generated from the [strict] annotations
The $\mathbb{K}$ semantics of IMP

- **Configuration:** $\langle\langle \text{Code} \rangle_k \langle \text{Map}_{Id,\text{Int}} \rangle_{\text{env}} \rangle_{\text{cfg}}$

- **$\mathbb{K}$ semantical rules:**
  
  $\langle\langle l_1 + l_2 \ldots \rangle_k \ldots \rangle_{\text{cfg}} \Rightarrow \langle\langle l_1 + \text{Int} \ l_2 \ldots \rangle_k \ldots \rangle_{\text{cfg}}$

  $\ldots$

  $\langle\langle \text{if true then } S_1 \text{ else } S_2 \ldots \rangle_k \ldots \rangle_{\text{cfg}} \Rightarrow \langle\langle S_1 \rangle_k \ldots \rangle_{\text{cfg}}$

  $\langle\langle \text{if false then } S_1 \text{ else } S_2 \rangle_k \ldots \rangle_{\text{cfg}} \Rightarrow \langle\langle S_2 \rangle_k \ldots \rangle_{\text{cfg}}$
The $K$ semantics of IMP

- **Configuration:** $\langle\langle \text{Code}\rangle_k \langle\text{Map}_{Id,\text{Int}}\rangle_{\text{env}}\rangle_{\text{cfg}}$
- **$K$ semantical rules:**

  $\langle\langle l_1 + l_2 \cdots \rangle_k \cdots \rangle_{\text{cfg}} \Rightarrow \langle\langle l_1 + \text{Int} \; l_2 \cdots \rangle_k \cdots \rangle_{\text{cfg}}$

  $\cdots$

  $\langle\langle \text{if true then } S_1 \text{ else } S_2 \cdots \rangle_k \cdots \rangle_{\text{cfg}} \Rightarrow \langle\langle S_1 \rangle_k \cdots \rangle_{\text{cfg}}$

  $\langle\langle \text{if false then } S_1 \text{ else } S_2 \rangle_k \cdots \rangle_{\text{cfg}} \Rightarrow \langle\langle S_2 \rangle_k \cdots \rangle_{\text{cfg}}$

  $\langle\langle \text{while } B \text{ do } S \cdots \rangle_k \cdots \rangle_{\text{cfg}} \Rightarrow$

  $\langle\langle \text{if } B \text{ then}\{ S \; ; \text{while } B \text{ do } S \} \text{ else skip } \cdots \rangle_k \cdots \rangle_{\text{cfg}}$
The $K$ semantics of IMP

- **Configuration:** $\langle\langle Code \rangle_k \langle MapId,Int \rangle_{env} \rangle_{cfg}$
- **$K$ semantical rules:**

  $\langle\langle l_1 + l_2 \cdots \rangle_k \cdots \rangle_{cfg} \Rightarrow \langle\langle l_1 + Int l_2 \cdots \rangle_k \cdots \rangle_{cfg}$

  ...$\langle\langle if \ true \ then \ S_1 \ else \ S_2 \ \cdots \rangle_k \ \cdots \rangle_{cfg} \Rightarrow \langle\langle S_1 \rangle_k \ \cdots \rangle_{cfg}$

  $\langle\langle if \ false \ then \ S_1 \ else \ S_2 \rangle_k \ \cdots \rangle_{cfg} \Rightarrow \langle\langle S_2 \rangle_k \ \cdots \rangle_{cfg}$

  $\langle\langle while \ B \ do \ S \ \cdots \rangle_k \ \cdots \rangle_{cfg} \Rightarrow$

  $\langle\langle if \ B \ then\{S ; while \ B \ do \ S \} else \ skip \ \cdots \rangle_k \ \cdots \rangle_{cfg}$

  ...

  ... + 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

- $\Sigma$ is an algebraic signature
  - includes a sub-signature $(\Sigma^{Data})$ of data sorts ($Int, Bool, \ldots$)
  - sort $Cfg$ for configurations
  - sub-signature for programs ($AExp, BExp, Stmt, \ldots$)

- $\mathcal{T}$ is a $\Sigma$-model
  - $\mathcal{D}$ the sub model of data ($\Sigma^{Data}$-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

1. Introduction and Motivation
   - An Example
   - In General

2. Symbolic Execution by Language Transformation

3. Formal Properties of Symbolic Execution

4. Prototype Implementation

5. Conclusion
Symbolic Execution by Language Transformation $\mathcal{L} \rightarrow \mathcal{L}^5$

If $\mathcal{L} = (\Sigma, \mathcal{T}, \mathcal{S})$ then $\mathcal{L}^5 = (\Sigma^5, \mathcal{T}^5, \mathcal{S}^5)$, where

- $\Sigma^5 = \Sigma + \text{infinite set of symbolic Values } V^5$
- $\mathcal{D}^5 = \text{algebra of symbolic expressions over } \mathcal{D} \cup V^5$
- $\mathcal{T}^5 = \text{the model freely generated by } \mathcal{D}^5$
- $\mathcal{S}^5$ automatically obtained from rules $\mathcal{S}$ by
  - left-linearisation
  - replacement of data sub terms in lhs by fresh variables
  - collect path condition
- $\mathcal{S}^5$ generates a transition system $\Rightarrow_{\mathcal{S}^5}$ on $\mathcal{T}_{\text{Cfg}}^5$
Rule transformations

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

\[
\langle \langle \text{if } \text{true} \text{ then } S_1 \text{ else } S_2 \ldots \rangle_k \ldots \rangle_{\text{cfg}} \Rightarrow \langle \langle S_1 \ldots \rangle_k \ldots \rangle_{\text{cfg}}
\]
Rule transformations

Consider the following rule for \textit{if} from the IMP semantics:

\[
\langle\langle \text{if true then } S_1 \text{ else } S_2 \ldots \rangle_k \ldots \rangle_{\text{cfg}} \Rightarrow \langle\langle S_1 \ldots \rangle_k \ldots \rangle_{\text{cfg}}
\]

Replace \textit{true} with a variable \textit{B}, and add the condition \(B = \text{true}\):

\[
\langle\langle \text{if } B \text{ then } S_1 \text{ else } S_2 \ldots \rangle_k \ldots \rangle_{\text{cfg}} \Rightarrow \langle\langle S_1 \ldots \rangle_k \ldots \rangle_{\text{cfg}} \text{ when } B = \text{true}
\]
Rule transformations

Consider the following rule for \textit{if} from the IMP semantics:

\[
\langle\langle \text{if} \ true \ then \ S_1 \ else \ S_2 \ \cdots \rangle_{\text{cfg}} \Rightarrow \langle\langle S_1 \ \cdots \rangle_{\text{cfg}}
\]

Replace \textit{true} with a variable \(B\), and add the condition \(B = true\):

\[
\langle\langle \text{if} \ \ B \ then \ S_1 \ else \ S_2 \ \cdots \rangle_{\text{cfg}} \Rightarrow \langle\langle S_1 \ \cdots \rangle_{\text{cfg}} \text{ when } B = true
\]

Now \(B\) matches on all terms of sort \textit{Bool}, including symbolic expressions. Example: \(n >_{\text{Int}} 0\)

\[\]
Rule transformations

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

- \[ \langle \langle \text{if } \text{true} \text{ then } S_1 \text{ else } S_2 \cdots \rangle_k \cdots \rangle_{cfg} \Rightarrow \langle \langle S_1 \cdots \rangle_k \cdots \rangle_{cfg} \]

Replace *true* with a variable *B*, and add the condition *B = true*:

- \[ \langle \langle \text{if } B \text{ then } S_1 \text{ else } S_2 \cdots \rangle_k \cdots \rangle_{cfg} \Rightarrow \langle \langle S_1 \cdots \rangle_k \cdots \rangle_{cfg} \text{ when } B = true \]

Now *B* matches on all terms of sort *Bool*, including symbolic expressions

Example: \( n >_{\text{Int}} 0 \)

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

1. Introduction and Motivation
   - An Example
   - In General

2. Symbolic Execution by Language Transformation

3. Formal Properties of Symbolic Execution

4. Prototype Implementation

5. Conclusion
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

1. Introduction and Motivation
   - An Example
   - In General

2. Symbolic Execution by Language Transformation

3. Formal Properties of Symbolic Execution

4. Prototype Implementation

5. Conclusion
Symbolic execution in \( K \)

- Our prototype is incorporated in the \( K \) framework
Prototype Implementation

Symbolic execution in $\mathbb{K}$

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

Symbolic execution in $\mathbb{K}$

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

Symbolic execution in K

- Our prototype is incorporated in the K framework
- Generates $L^s$ automatically from $L$
  - `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");
    }
}
Prototype Implementation

Pattern search

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

$ krun lists.kool -search -cIN="e1 e2 x"
   -pattern="<T> <out> error </out> B:Bag </T>"

Solution 1, State 50:
<path-condition>
    e1 = x ∧ Bool ¬ Bool(e1 = e2)
</path-condition>
...

• l1 = [e2, e1] and l2 = [e1, e2]
Prototype Implementation

Pattern search

```csharp
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="e1 e2 x"
  -pattern="<T> <out> error </out> B:Bag </T>"

  Search results:
  No search results

- NB: this is not verification!
Plan

1. Introduction and Motivation
   • An Example
   • In General

2. Symbolic Execution by Language Transformation

3. Formal Properties of Symbolic Execution

4. Prototype Implementation

5. Conclusion
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 K framework web page: http://k-framework.org

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