Ghost code in action: Automated verification of a symbolic interpreter using Why3

Benedikt Becker, Claude Marché
Inria Saclay & LRI, Université Paris-Saclay, France

Gallium seminar, Inria Paris
May 27, 2019

Submitted to VSTTE 2019

Context: Project CoLiS (Correctness of Linux Scripts)

Collaboration between IRIF (Université Paris), LINKS (Inria Lille), and Toccata (Inria Saclay)

Questions

Can the execution of a given Shell script (Debian maintainer script) fail?
Does it play well with the other maintainer scripts?
(They are executed as root!)

Approach

Symbolic execution with tree constraints to represent the file system

http://colis.irif.fr/
https://github.com/colis-anr/

Context: Project CoLiS

Achievements

- a trustworthy parser for POSIX shell: Morbig, Morsmall
- CoLiS language: “Shell with sane semantics”
  - verified concrete interpreter
  - verified symbolic execution engine

[JFLA 2016, VSTTE 2017]

- in progress: symbolic specifications of Linux utilities

- verification with Why3, a platform for deductive program verification

http://why3.lri.fr/

This seminar

1. Sketch of the symbolic correctness properties
2. Concrete semantics of IMP and concrete execution
3. Symbolic execution of IMP
4. Formalisation of symbolic correctness properties
   and proof techniques
5. Application to Debian maintainer scripts in the CoLiS project

(6. *And no fancy symbolic execution techniques ...*)

Example program p_0

```
y := x - y - 1;
if y ≠ 0 then
  x := y - 3
else
  y := x - 3
```

Concrete execution

Concrete state: variable environment $\Gamma : PVar \rightarrow \mathbb{Z}$

A partial mapping from program variables to integers.

- executing a program in an initial state results in a possibly changed result state

$$\texttt{interp}\,(x \mapsto 2, y \mapsto 0)\,(p_0) = (x \mapsto -2, y \mapsto 1)$$
$$\texttt{interp}\,(x \mapsto 2, y \mapsto 1)\,(p_0) = (x \mapsto 2, y \mapsto -1)$$

(Reminder: p_0 = `y := x - y - 1; if y then x := y - 3 else y := x - 3`)

Concrete execution

Concrete state: variable environment $\Gamma : PVar \nrightarrow \mathbb{Z}$

A partial mapping from program variables to integers.

- executing a program in an initial state results in a possibly changed result state

$$\texttt{interp}\,(x \mapsto 2, y \mapsto 0)\,(p_0) = (x \mapsto -2, y \mapsto 1)$$

$$\texttt{interp}\,(x \mapsto 2, y \mapsto 1)\,(p_0) = (x \mapsto 2, y \mapsto -1)$$

$$\texttt{interp}\,(x \mapsto 2)\,(p_0)\ \texttt{raises}\ \text{UnboundVar}$$

(Reminder: p_0 = `y := x - y - 1; if y then x := y - 3 else y := x - 3`)

Concrete execution

Concrete state: variable environment $\Gamma : PVar \rightharpoonup \mathbb{Z}$

A partial mapping from program variables to integers.

- executing a program in an initial state results in a possibly changed result state

$$\texttt{interp}\,(x \mapsto 2, y \mapsto 0)\,(p_0) = (x \mapsto -2, y \mapsto 1)$$

$$\texttt{interp}\,(x \mapsto 2, y \mapsto 1)\,(p_0) = (x \mapsto 2, y \mapsto -1)$$

$$\texttt{interp}\,(x \mapsto 2)\,(p_0)\ \texttt{raises}\ \mathsf{UnboundVar}$$

$$\texttt{interp}\,(x \mapsto -1)\,(\texttt{while}\ x\ \mathsf{do}\ x \mathrel{\texttt{:=}} x - 1\ \mathsf{done}) = \ldots$$

(Reminder: p_0 = `y := x - y - 1;` **`if`** `y` **`then`** `x := y - 3` **`else`** `y := x - 3`)

Correctness properties of a concrete interpreter

Completeness A concrete interpreter is complete if
"it produces any result specified by the semantics."

Soundness An interpreter is sound if
"any result corresponds to the semantics."

Symbolic execution

Symbolic state $(\sigma \mid C)$

- symbolic variable environment $\sigma : PVar \nrightarrow SVar$, a partial map from program variables to *symbolic variables*
- constraint C on symbolic variables

$$\texttt{sym-interp}\,(x \mapsto v_1, y \mapsto v_2 \mid v_1 = 2 \wedge v_2 = 0)\,(p_0) = \\ (x \mapsto v_4, y \mapsto v_3 \mid v_4 = -2 \wedge v_3 = 1)$$

(Reminder: p_0 = `y := x - y - 1;` **`if`** `y` **`then`** `x := y - 3` **`else`** `y := x - 3`)

Symbolic execution

Symbolic state $(\sigma \mid C)$

- symbolic variable environment $\sigma : PVar \rightharpoonup SVar$,
  a partial map from program variables to *symbolic variables*
- constraint C on symbolic variables

$$\texttt{sym-interp}\,(x \mapsto v_1, y \mapsto v_2 \mid v_1 = 2 \land v_2 = 0)\,(p_0) =$$
$$(x \mapsto v_4, y \mapsto v_3 \mid v_4 = -2 \land v_3 = 1)$$ Normal

$$\texttt{sym-interp}\,(x \mapsto v_1 \mid v_1 = 2 \land v_2 = 0)\,(p_0) =$$
$$(x \mapsto v_1, y \mapsto v_2 \mid v_1 = 2 \land v_2 = 0)$$ UnboundVar

(Reminder: p_0 = `y := x - y - 1; if y then x := y - 3 else y := x - 3`)

Symbolic execution

Symbolic state $(\sigma \mid C)$

- symbolic variable environment $\sigma : \mathit{PVar} \nrightarrow \mathit{SVar}$, a partial map from program variables to *symbolic variables*
- constraint C on symbolic variables

- symbolic states describes an (infinite) set of concrete states

$$\texttt{sym-interp}\,(x \mapsto v_1, y \mapsto v_2 \mid v_1 - v_2 - 1 \neq 0)\,(p_0) =$$
$$(x \mapsto v_4, y \mapsto v_3 \mid v_1 - v_2 - 1 \neq 0 \wedge v_3 = v_1 - v_2 - 1 \wedge v_4 = v_3 - 3)_{\mathsf{Normal}}$$

(Reminder: p_0 = `y := x - y - 1;` **`if`** `y` **`then`** `x := y - 3` **`else`** `y := x - 3`)

Symbolic execution

Symbolic state $(\sigma \mid C)$

- symbolic variable environment $\sigma : PVar \nrightarrow SVar$, a partial map from program variables to *symbolic variables*
- constraint C on symbolic variables

- symbolic states describes an (infinite) set of concrete states
- symbolic result state sets capture different execution paths

$$\begin{array}{l}\texttt{sym-interp}\,(x \mapsto v_1, y \mapsto v_2 \mid \top)\,(p_0) = \\ \quad \{(x \mapsto v_4, y \mapsto v_3 \mid v_3 = v_1 - v_2 - 1 \wedge v_3 \neq 0 \wedge v_4 = v_3 - 3)_{\mathsf{Normal}}, \\ \quad\; (x \mapsto v_1, y \mapsto v_4 \mid v_3 = v_1 - v_2 - 1 \wedge v_3 = 0 \wedge v_4 = v_1 - 3)_{\mathsf{Normal}}\}\end{array}$$

(Reminder: p_0 = `y := x - y - 1;` **`if`** `y` **`then`** `x := y - 3` **`else`** `y := x - 3`)

Handling of branching language constructs

How to execute conditionals?

Execute all branches.

While loops?

- unroll loop iterations
- problem: makes symbolic execution generally *non-terminating*
- (simplest) solution: limit the number of loop iterations

Handling of loops: example

Example program p_1

```
y := 1;
while x > 1 do y := y * x; x := x - 1 done
```

Handling of loops: example

Example program p_1

```
y := 1;
while x > 1 do y := y * x; x := x - 1 done
```

Symbolic execution with loop limit $N = 2$

$$\texttt{sym-interp}_N\,(x \mapsto v_1 \mid \top)\,(p_1) = \\ \{(x \mapsto v_1, y \mapsto v_2 \mid v_2 = 1 \wedge v_1 \leq 1)_{\text{Normal}}$$

Handling of loops: example

Example program p_1

```
y := 1;
while x > 1 do y := y * x; x := x - 1 done
```

Symbolic execution with loop limit $N = 2$

$$
\begin{aligned}
&\texttt{sym-interp}_N\,(x \mapsto v_1 \mid \top)\,(p_1) = \\
&\;\{(x \mapsto v_1, y \mapsto v_2 \mid v_2 = 1 \land v_1 \leq 1)_{\mathsf{Normal}} \\
&\;\;(x \mapsto v_3, y \mapsto v_4 \mid v_2 = 1 \land v_1 = 2 \land v_3 = 1 \land v_4 = 2)_{\mathsf{Normal}}
\end{aligned}
$$

Handling of loops: example

Example program p_1

```
y := 1;
while x > 1 do y := y * x; x := x - 1 done
```

Symbolic execution with loop limit $N = 2$

$$
\begin{aligned}
&\texttt{sym-interp}_N\,(x \mapsto v_1 \mid \top)\,(p_1) = \\
&\;\{(x \mapsto v_1, y \mapsto v_2 \mid v_2 = 1 \wedge v_1 \leq 1)_{\mathsf{Normal}} \\
&\;\;(x \mapsto v_3, y \mapsto v_4 \mid v_2 = 1 \wedge v_1 = 2 \wedge v_3 = 1 \wedge v_4 = 2)_{\mathsf{Normal}} \\
&\;\;(x \mapsto v_5, y \mapsto v_6 \mid v_2 = 1 \wedge v_1 = 3 \wedge v_5 = 1 \wedge v_6 = 6)_{\mathsf{Normal}}
\end{aligned}
$$

Handling of loops: example

Example program p_1

```
y := 1;
while x > 1 do y := y * x; x := x - 1 done
```

Symbolic execution with loop limit $N = 2$

$$
\begin{aligned}
&\texttt{sym-interp}_N\,(x \mapsto v_1 \mid \top)\,(p_1) = \\
&\ \{(x \mapsto v_1, y \mapsto v_2 \mid v_2 = 1 \wedge v_1 \leq 1)_{\mathsf{Normal}} \\
&\ \ (x \mapsto v_3, y \mapsto v_4 \mid v_2 = 1 \wedge v_1 = 2 \wedge v_3 = 1 \wedge v_4 = 2)_{\mathsf{Normal}} \\
&\ \ (x \mapsto v_5, y \mapsto v_6 \mid v_2 = 1 \wedge v_1 = 3 \wedge v_5 = 1 \wedge v_6 = 6)_{\mathsf{Normal}} \\
&\ \ (x \mapsto v_5, y \mapsto v_6 \mid v_2 = 1 \wedge v_1 = 2 \wedge v_5 > 1 \wedge v_6 = 6)_{\mathsf{Incomplete}}\}
\end{aligned}
$$

Correctness properties of symbolic execution

Definition: Over-approximation – I "covers all concrete executions"

A symbolic execution is an over-approximation, if

"a concrete execution in a state that corresponds to the initial symbolic state results in a concrete state that corresponds to one of the result states."

Definition: Under-approximation – I "no useless result states"

A symbolic execution is an under-approximation, if

"every concrete state corresponding to a result state is the result of the concrete execution in a concrete state corresponding to the initial state."

(Also called coverage and precision)

[Arusoaie, 2015]

This seminar

1. Sketch of the symbolic correctness properties
2. Concrete semantics of IMP and concrete execution
3. Symbolic execution of IMP
4. Formalisation of symbolic correctness properties
   and proof techniques
5. Application to Debian maintainer scripts in the CoLiS project

The IMP language

Syntax

$$
\begin{array}{rcll}
\bar{n} & \in & \bar{\mathbb{Z}} & \text{— \textit{Integer literal}} \\
x & \in & \mathit{PVar} & \text{— \textit{Program variable}} \\
e & ::= & \bar{n} \mid x \mid e - e & \text{— \textit{Expression}} \\
i & ::= & \texttt{skip} & \text{— \textit{Instructions}} \\
 & \mid & x \texttt{ := } e & \text{— \textit{Assignment}} \\
 & \mid & i \texttt{ ; } i & \text{— \textit{Sequence}} \\
 & \mid & \texttt{if } e \texttt{ then } i \texttt{ else } i & \text{— \textit{Conditional}} \\
 & \mid & \texttt{while } e \texttt{ do } i & \text{— \textit{Loop}}
\end{array}
$$

(Condition e is true when non-zero)

Three semantic judgments

Expression e:

Variable environment:
$\Gamma \in PVar \rightarrow\!\!\!\!+ \mathbb{Z}$

$$e/_{\Gamma} \Downarrow \alpha$$

Expression behaviour:
α ::= Normal $\mathbb{Z}$ | UnboundVar

Instruction i:

Optional loop limit
$N \in \mathbb{N} \cup \{\infty\}$

$$i/_{\Gamma} \Downarrow^{N} \beta/_{\Gamma'}$$

Instruction behaviour:
β ::= Normal | UnboundVar | Incomplete

`while e do i`:

Loop counter:
$n \in \mathbb{N}$

$$e, i/_{\Gamma} \Downarrow^{n/N} \beta/_{\Gamma'}$$

- Loop limit to model incompleteness of symbolic execution

Semantic rules – expressions $e /_\Gamma \Downarrow \alpha$

Literal
$$\bar{n} /_\Gamma \Downarrow \textsf{Normal}\ n$$

Var
$$\frac{x \in \mathrm{dom}(\Gamma) \qquad \Gamma[x] = n}{x /_\Gamma \Downarrow \textsf{Normal}\ n}$$

Var-err
$$\frac{x \notin \mathrm{dom}(\Gamma)}{x /_\Gamma \Downarrow \textsf{UnboundVar}}$$

Sub
$$\frac{e_1 /_\Gamma \Downarrow \textsf{Normal}\ n_1 \qquad e_2 /_\Gamma \Downarrow \textsf{Normal}\ n_2}{e_1 - e_2 /_\Gamma \Downarrow \textsf{Normal}\ (n_1 - n_2)}$$

Sub-err-1
$$\frac{e_1 /_\Gamma \Downarrow \textsf{UnboundVar}}{e_1 - e_2 /_\Gamma \Downarrow \textsf{UnboundVar}}$$

Sub-err-2
$$\frac{e_1 /_\Gamma \Downarrow \textsf{Normal}\ n_1 \qquad e_2 /_\Gamma \Downarrow \textsf{UnboundVar}}{e_1 - e_2 /_\Gamma \Downarrow \textsf{UnboundVar}}$$

- (only) unbound variables cause abnormal behaviour UnboundVar
- abnormal behaviour is propagated through binary operations

Semantic rules – instructions $i/_\Gamma \Downarrow^N \beta/_{\Gamma'}$

Skip
$$\texttt{skip}/_\Gamma \Downarrow^N \mathsf{Normal}/_\Gamma$$

While
$$\frac{e, i/_\Gamma \Downarrow^{0/N} \beta/_{\Gamma'}}{\texttt{while}\ e\ \texttt{do}\ i/_\Gamma \Downarrow^N \beta/_{\Gamma'}}$$

Assign
$$\frac{e/_\Gamma \Downarrow \mathsf{Normal}\ n}{x \texttt{ := } e/_\Gamma \Downarrow^N \mathsf{Normal}/_{\Gamma[x\leftarrow n]}}$$

Assign-err
$$\frac{e/_\Gamma \Downarrow \mathsf{UnboundVar}}{x \texttt{ := } e/_\Gamma \Downarrow^N \mathsf{UnboundVar}/_\Gamma}$$

Seq
$$\frac{i_1/_\Gamma \Downarrow^N \mathsf{Normal}/_{\Gamma_1} \qquad i_2/_{\Gamma_1} \Downarrow^N \beta/_{\Gamma_2}}{i_1\texttt{;}i_2/_\Gamma \Downarrow^N \beta/_{\Gamma_2}}$$

Seq-err
$$\frac{i_1/_\Gamma \Downarrow^N \beta/_{\Gamma_1} \qquad \beta \neq \mathsf{Normal}}{i_1\texttt{;}i_2/_\Gamma \Downarrow^N \beta/_{\Gamma_1}}$$

Cond-true
$$\frac{e/_\Gamma \Downarrow \mathsf{Normal}\ n \qquad n \neq 0 \qquad i_1/_\Gamma \Downarrow^N \beta/_{\Gamma'}}{\texttt{if}\ e\ \texttt{then}\ i_1\ \texttt{else}\ i_2/_\Gamma \Downarrow^N \beta/_{\Gamma'}}$$

Cond-false
$$\frac{e/_\Gamma \Downarrow \mathsf{Normal}\ 0 \qquad i_2/_\Gamma \Downarrow^N \beta/_{\Gamma'}}{\texttt{if}\ e\ \texttt{then}\ i_1\ \texttt{else}\ i_2/_\Gamma \Downarrow^N \beta/_{\Gamma'}}$$

Cond-err
$$\frac{e/_\Gamma \Downarrow \mathsf{UnboundVar}}{\texttt{if}\ e\ \texttt{then}\ i_1\ \texttt{else}\ i_2/_\Gamma \Downarrow^N \mathsf{UnboundVar}/_\Gamma}$$

- abnormal behaviour is propagated from expressions and sub-instructions

Semantic rules – loops

$$e, i/_\Gamma \Downarrow^{n/N} \beta/_{\Gamma'}$$

While-limit

$$\frac{n = N}{e, i/_\Gamma \Downarrow^{n/N} \text{Incomplete}/_\Gamma}$$

While-false

$$\frac{n < N \qquad e/_\Gamma \Downarrow \text{Normal } 0}{e, i/_\Gamma \Downarrow^{n/N} \text{Normal}/_\Gamma}$$

While-test-err

$$\frac{n < N \qquad e/_\Gamma \Downarrow \text{UnboundVar}}{e, i/_\Gamma \Downarrow^{n/N} \text{UnboundVar}/_\Gamma}$$

While-body-err

$$\frac{n < N \qquad e/_{\Gamma_1} \Downarrow \text{Normal } m \qquad m \neq 0 \qquad i/_{\Gamma_1} \Downarrow^N \beta/_{\Gamma_2} \qquad \beta \neq \text{Normal}}{e, i/_{\Gamma_1} \Downarrow^{n/N} \beta/_{\Gamma_2}}$$

While-loop

$$\frac{n < N \qquad e/_{\Gamma_1} \Downarrow \text{Normal } m \qquad m \neq 0 \qquad i/_{\Gamma_1} \Downarrow^N \text{Normal}/_{\Gamma_2} \qquad e, i/_{\Gamma_2} \Downarrow^{(n+1)/N} \beta/_{\Gamma_3}}{e, i/_{\Gamma_1} \Downarrow^{n/N} \beta/_{\Gamma_3}}$$

- loop terminates normally when the condition is false
- when the condition is true and loop body executes without error, the loop execution continues with increased counter
- unbounded loops with $N = \infty$

A sound, concrete interpreter in Why3

```
let env = Env.empty ()

let rec interp_ins (i : ins) : unit diverges
  ensures { i/(old env) ⇓∞ Normal/env }
  raises { UnboundVar → i/(old env) ⇓∞ UnboundVar/env }
= match i with
  | Skip → ()
  | Assign x e → Env.set env x (interp_exp env e)
  | Seq i1 i2 → interp_ins i1; interp_ins i2
  | If e i1 i2 →
    if interp_exp env e ≠ 0
    then interp_ins i1 else interp_ins i2
  | While e i →
    let ghost env0 = env.model in
    let ghost ref n = 0 in
    while interp_exp env e ≠ 0 do
     invariant { ∀ β/Γ'.
       e,i/env ⇓n/∞ β/Γ' →
       e,i/env0 ⇓0/∞ β/Γ'
     }
     interp_ins i;
     n ← n + 1
    done
  end
```

- using an imperative, global variable environment env
- abnormal behaviour as exceptions
- soundness stated in post-conditions
- unbound loops, no incomplete behaviour
- loop invariant: if the loop terminates when starting in the current evaluation state, the loop terminates with the same result when starting in the initial evaluation state
- all 22 VCs proven automatically

This seminar

1. Sketch of the symbolic correctness properties
2. Concrete semantics of IMP and concrete execution
3. Symbolic execution of IMP
4. Formalisation of symbolic correctness properties
   and proof techniques
5. Application to Debian maintainer scripts in the CoLiS project

Symbolic execution context of IMP

Symbolic state

$(\sigma \mid C)$ with symbolic variable environment $\sigma PVar \nrightarrow SVar$

Symbolic expression

$$se ::= n \mid v \mid se - se$$

Constraint

$$C ::= \top \mid se = se \mid se \neq se \mid C \wedge C \mid \exists v. C$$

Natural extension of σ to expressions

$\sigma(e) = se$ when $\mathrm{vars}(e) \subseteq \mathrm{dom}(\sigma)$

Set of symbolic states with behaviour

$$(\sigma \mid C)_\beta \in \Sigma$$

A symbolic interpreter for IMP

Function signature:

Initial symbolic state

Set of symbolic result states with behaviour: $(\sigma \mid C)_\beta \in \Sigma$

$$\texttt{\textbf{val} sym-interp}_N\,(\sigma \mid C)(i)\ \texttt{:}\ \Sigma$$

Finite loop limit $\in \mathbb{N}$

A symbolic interpreter for IMP

Function signature:

Initial symbolic state

Set of symbolic result states with behaviour: $(\sigma \mid C)_\beta \in \Sigma$

$$\texttt{val sym-interp}_N(\sigma \mid C)(i) \;:\; \Sigma$$

Finite loop limit $\in \mathbb{N}$

Symbolic execution of assignment – I

```
let rec sym-interp_N(σ | C)(i) =
  match i with ...
  | Assign x e →
    try
      let se = σ(e) in
      let v = fresh_var () in
      let σ′ = σ[x ← v] in
      let C′ = C ∧ v = se in
      {(σ′ | C′)_Normal}
    with UnboundVar → {(σ | C)_UnboundVar} end
```

A symbolic interpreter for IMP

Symbolic execution of assignment – II

```
let rec sym-interp_N(σ | C)(i) =
  match i with ...
  | Assign x e →
    try
      let se = σ(e) in
      let v = fresh_var () in
      let σ′ = σ[x ← v] in
      let C′ = match σ(x) with
        | None → C ∧ v = se end in
        | Some v′ → ∃v′.C ∧ v = se in
      {(σ′ | C′)_Normal}
    with Unbound_var → {(σ | C)_UnboundVar} end
```

Existential quantification of a shadowed variable

A symbolic interpreter for IMP

Symbolic execution of assignment – III

val quantify-existentially (v) (C) : $\mathit{Constraint}$

Existential quantification, or quantifier elimination to reduce constraint size

let rec sym-interp$_N(\sigma \mid C)(i)$ =
  match i **with** ...
  | Assign x e $\rightarrow$
    try
      let se = $\sigma(e)$ **in**
      let v = fresh_var () **in**
      let σ' = $\sigma[x \leftarrow v]$ **in**
      let C' = **match** $\sigma(x)$ **with**
        | None $\rightarrow$ $C \wedge v = se$ **end** **in**
        | Some v' $\rightarrow$ quantify-existentially v' $(C \wedge v = se)$ **in**
      $\{(\sigma' \mid C')_{\text{Normal}}\}$
    with Unbound_var $\rightarrow$ $\{(\sigma \mid C)_{\text{UnboundVar}}\}$ **end**

A symbolic interpreter for IMP

Symbolic execution of conditionals

$$
\begin{array}{l}
\textbf{let rec } \texttt{sym-interp}_N(\sigma \mid C)(i) \texttt{ =} \\
\quad \textbf{match } i \textbf{ with } \ldots \\
\quad \mid \mathsf{If}\ e\ i_1\ i_2 \ \rightarrow \\
\qquad \textbf{try} \\
\qquad\quad \textbf{let } se \texttt{ = } \sigma(e) \textbf{ in} \\
\qquad\quad \textbf{let } \Sigma \texttt{ = } \texttt{(* then-branch *)} \\
\qquad\qquad \texttt{sym-interp}_N\,(\sigma \mid C \wedge se = 0)\,(i_1) \textbf{ in} \\
\qquad\quad \textbf{let } \Sigma' \texttt{ = } \texttt{(* else-branch *)} \\
\qquad\qquad \texttt{sym-interp}_N\,(\sigma \mid C \wedge se \neq 0)\,(i_2) \textbf{ in} \\
\qquad\quad \Sigma \cup \Sigma' \\
\qquad \textbf{with } \texttt{UnboundVar} \ \rightarrow \ \{(\sigma \mid C)_{\texttt{UnboundVar}}\} \textbf{ end}
\end{array}
$$

State explosion!

- combinatoric explosion of result states
- (simplest) corrective: state pruning

A symbolic interpreter for IMP

Symbolic execution of conditionals with state pruning

```
val maybe_sat (C : Constraint) : 𝔹
  ensures { result = False → ∄ρ. ρ ⊨ C }

let rec sym-interp_N(σ | C)(i) =
  match i with ...
  | If e i₁ i₂ →
    try
      let se = σ(e) in
      let Σ = (* then-branch *)
        if maybe_sat (C ∧ se = 0)
        then sym-interp_N (σ | C ∧ se = 0) (i₁)
        else ∅ in
      let Σ′ = (* else-branch *)
        if maybe_sat (C ∧ se ≠ 0)
        then sym-interp_N (σ | C ∧ se ≠ 0) (i₂)
        else ∅ in
      Σ ∪ Σ′
    with UnboundVar → {(σ | C)_UnboundVar} end
```

Semi-decidable satisfiability predicate for constraints

Prune branches with inconsistent constraints

Symbolic execution properties: Over-approximation

Given a symbolic execution

$$\texttt{sym-interp}_N(\sigma \mid C)(i) = \Sigma$$

Definition: Over-approximation – I "covers all concrete executions"

Symbolic execution *over-approximates* the concrete execution, if

"a concrete execution in a state that corresponds to the initial symbolic state results in a concrete state that corresponds to one of the result states."

Constraint interpretations

Interpretation $\rho : SVar \nrightarrow \mathbb{Z}$

- partial map from symbolic variables to values

Solution

- interpretation ρ is a *solution* of C, $\rho \models C$, iff.

$$\mathrm{vars}(C) \subseteq \mathrm{dom}(\rho) \land \begin{cases} \top & \text{when } C = \top \\ \rho(se_1) = \rho(se_2) & \text{when } C = (se_1 = se_2) \\ \rho(se_1) \neq \rho(se_2) & \text{when } C = (se_1 \neq se_2) \\ \rho \models C_1 \land \rho \models C_2 & \text{when } C = C_1 \land C_2 \\ \exists\, n.\, \rho[v \leftarrow n] \models C_1 & \text{when } C = \exists\, v.C_1 \end{cases}$$

Concrete states and symbolic states

- composition of an interpretation with a symbolic environment, $\rho \circ \sigma$, is a concrete environment
- Γ is an instance $\Gamma \in \mathit{Inst}(\sigma \mid C)$, when there exists a solution $\rho \models C$ such that $\Gamma = \rho \circ \sigma$.

Symbolic execution properties: Over-approximation

Given a symbolic execution

$$\texttt{sym-interp}_N(\sigma \mid C)(i) = \Sigma$$

Definition: Over-approximation – II

Symbolic execution *over-approximates* the concrete execution, if

for any

- ▷ instance of the initial symbolic state, and $\forall\ \Gamma \in \mathit{Inst}(\sigma \mid C)$
- ▷ corresponding concrete evaluation result $\forall\ \beta, \Gamma'.\ i/_{\Gamma} \Downarrow^{N} \beta/_{\Gamma'}$

there exists

- ▷ a symbolic result state with the same behaviour $\exists\ (\sigma' \mid C')_{\beta} \in \Sigma$
- ▷ that has the concrete evaluation result as instance. $\Gamma' \in \mathit{Inst}(\sigma' \mid C')$

Symbolic execution properties: Over-approximation

Given a symbolic execution

$$\texttt{sym-interp}_N(\sigma \mid C)(i) = \Sigma$$

Definition: Over-approximation – III

Symbolic execution *over-approximates* the concrete execution, if

$$\forall\ \rho, \beta, \Gamma'.\ \rho \models C \rightarrow i/_{\rho\circ\sigma} \Downarrow^N \beta/_{\Gamma'} \rightarrow$$
$$\exists\ \sigma', C', \rho'.\ (\sigma' \mid C')_\beta \in \Sigma \wedge \rho' \models C' \wedge \Gamma' = \rho' \circ \sigma'$$

Symbolic execution properties: Under-approximation

Given a symbolic execution

$$\texttt{sym\_interp\_ins}_N(\sigma \mid C)(i) = \Sigma$$

Definition: Under-approximation – I "no useless result states"

Symbolic execution *under-approximates* the concrete execution, if

"every concrete state corresponding to a result state is the result of the concrete execution in a concrete state corresponding to the initial state."

Symbolic execution properties: Under-approximation

Given a symbolic execution

$$\texttt{sym\_interp\_ins}_N(\sigma \mid C)(i) = \Sigma$$

Definition: Under-approximation – II

Symbolic execution *under-approximates* the concrete execution, if

for any

- symbolic result state with behaviour, and $(\sigma' \mid C')_\beta \in \Sigma$
- instance of the result state $\Gamma' \in \mathit{Inst}(\sigma' \mid C')$

there exists

- an instance of the initial state, such that $\Gamma \in \mathit{Inst}(\sigma \mid C)$
- $i/_\Gamma \Downarrow^N \beta/_{\Gamma'}$.

Symbolic execution properties: Under-approximation

Given a symbolic execution

$$\texttt{sym\_interp\_ins}_N(\sigma \mid C)(i) = \Sigma$$

Definition: Under-approximation – III

Symbolic execution *under-approximates* the concrete execution, if

$$\forall\ \sigma', C', \beta, \rho'.\ (\sigma' \mid C')_\beta \in \Sigma \to \rho' \models C' \to$$
$$\exists\ \rho.\ \rho \models C \wedge i/_{\rho\circ\sigma} \Downarrow^N \beta/_{\rho'\circ\sigma'}$$

Problem: existential quantification

- existential quantifications are hard for automatic (SMT) provers
- two (problematic) sources of existential quantifications
  - interpretations in conclusions
  - witness for solution predicate, $\rho \models \exists v. C$

Ghost annotations in Why3

Before

let $f\,(x)$ **returns** y
  ensures { $\forall z.\, P(x,z) \rightarrow \exists t.\, Q(x,y,z,t)$ }

After

let $f\,(x,$ **ghost** $z)$ **returns** $(y,$ **ghost** $t)$
  requires { $P(x,z)$ }
  ensures { $Q(x,y,z,t)$ }

- use program code to construct ghost values required in the proof
- ghost code and ghost values cannot influence the program and are removed by the Why3 extraction

Ghost-extended symbolic states

Idea

- make the interpretation ρ a ghost field of the symbolic state
- use ghost code in symbolic interpreter to create witnesses for existential quantifications

Ghost-extended symbolic states

Idea

- make the interpretation ρ a ghost field of the symbolic state
- use ghost code in symbolic interpreter to create witnesses for existential quantifications

New signature of symbolic interpreter:

$$\texttt{\textbf{val}}\ \texttt{sym-interp}_N(\sigma \mid C;\ \texttt{\textbf{ghost}}\ \rho)(i)\ \texttt{:}\ \Sigma$$

Symbolic state with ghost interpretation

Set of symbolic states and behaviour, and a **ghost** interpretation $(\sigma \mid C;\ \texttt{\textbf{ghost}}\ \rho)_\beta \in \Sigma$

Reformulated correctness properties of symbolic execution

Definition: Over-approximation – IV, final

Symbolic execution *over-approximates* the concrete execution:

$$\forall\ \beta, \Gamma'.\ \rho \models C \rightarrow i/_{\rho\circ\sigma} \Downarrow^{N} \beta/_{\Gamma'} \rightarrow$$
$$\exists\ \sigma', C', \rho'.\ (\sigma' \mid C';\ \rho')_{\beta} \in \Sigma \wedge \rho' \models C' \wedge \Gamma' = \rho' \circ \sigma'$$

Definition: Under-approximation – IV, final

Symbolic execution *under-approximates* the concrete execution, if

$$\forall\ \sigma', C', \rho', \beta.\ (\sigma' \mid C';\ \rho')_{\beta} \in \Sigma \rightarrow \rho' \models C' \rightarrow$$
$$\rho \models C \wedge i/_{\rho\circ\sigma} \Downarrow^{N} \beta/_{\rho'\circ\sigma'}$$

Symbolic interpreter in Why3

Signature with reformulated correctness properties as post-conditions

```
type sym_state = (σ | C; ghost ρ)

let rec sym-interp_N(σ | C; ρ)(i)
  ensures { (* Over-approximation *)
    ∀ β, Γ'. ρ ⊨ C → i/_{ρ∘σ} ⇓^N β/_{Γ'} →
    ∃ σ', C', ρ'. (σ' | C'; ρ')_β ∈ Σ ∧ ρ' ⊨ C' ∧ Γ' = ρ' ∘ σ'  }
  ensures { (* Under-approximation *)
    ∀ σ', C', ρ', β. (σ' | C'; ρ')_β ∈ Σ → ρ' ⊨ C' →
    ρ ⊨ C ∧ i/_{ρ∘σ} ⇓^N β/_{ρ'∘σ'}  }
= match i with ...
```

In math notation:

$\texttt{type sym\_state} = (\sigma \mid C;\ \textbf{ghost}\ \rho)$

$\texttt{let rec sym-interp}_N(\sigma \mid C;\ \rho)(i)$

$\quad\texttt{ensures}\ \{\ \text{(* Over-approximation *)}\ \forall\, \beta, \Gamma'.\ \rho \models C \rightarrow i/_{\rho\circ\sigma} \Downarrow^N \beta/_{\Gamma'} \rightarrow \exists\, \sigma', C', \rho'.\ (\sigma' \mid C';\ \rho')_\beta \in \Sigma \wedge \rho' \models C' \wedge \Gamma' = \rho' \circ \sigma'\ \}$

$\quad\texttt{ensures}\ \{\ \text{(* Under-approximation *)}\ \forall\, \sigma', C', \rho', \beta.\ (\sigma' \mid C';\ \rho')_\beta \in \Sigma \rightarrow \rho' \models C' \rightarrow \rho \models C \wedge i/_{\rho\circ\sigma} \Downarrow^N \beta/_{\rho'\circ\sigma'}\ \}$

$= \texttt{match i with} \ldots$

Symbolic interpreter in Why3

Symbolic execution of assignment – IV, final

```
let rec sym-interp_N(σ | C; ρ)(i) =
  match i with ...
  | Assign x e →
    try
      let se = σ(e) in
      let v = fresh_var () in
      let σ′ = σ[x ← v] in
      let C′ = match σ(x) with
        | None → C ∧ (v = se) end in
        | Some v′ → quantify-existentially v′ C ∧ (v = se) in
      let ghost ρ′ = ρ[v ← ρ(se)] in
      {(σ′ | C′; ρ′)_Normal}
    with Unbound_var → {(σ | C; ρ)_UnboundVar}
  end
```

Update ghost interpretation to keep it a solution.
Values become witnesses when variables are existentially quantified!

Solutions for existential constraints

- witnesses of existentials in interpretation allow for simplifying the solution predicate

$$
\rho \models C \quad \text{iff.} \quad \mathrm{vars}(C) \subseteq \mathrm{dom}(\rho) \\
\wedge \quad \begin{cases} \top & \text{when } C = \top \\ \rho(se_1) = \rho(se_2) & \text{when } C = (se_1 = se_2) \\ \rho(se_1) \neq \rho(se_2) & \text{when } C = (se_1 \neq se_2) \\ \rho \models C_1 \wedge \rho \models C_2 & \text{when } C = C_1 \wedge C_2 \\ \cancel{\exists\, n.\, \rho[v \leftarrow n] \models C_1} & \text{when } C = \exists\, v.C_1 \\ \rho \models C_1 & \end{cases}
$$

ρ carries witness to v!

Implication of (not) modelling existential constraints

Problem

Solution is not invariant to α-renaming!

Implication of (not) modelling existential constraints

Solution

type sym_state = $(\sigma \mid C;\, \rho)$
  invariant { $\mathrm{codom}(\sigma) \cup \mathrm{vars}(C) \subseteq \mathrm{dom}(\rho)$ }

val fresh_var (ρ) : SVar
  ensures { **result** $\notin \mathrm{dom}(\rho)$ }

val quantify-existentially v C : $\mathit{Constraint}$
  ensures { $\mathit{vars}($**result**$) \subseteq \mathrm{vars}(\exists v.\, C)$ }
  ensures { $\forall \rho.\, \rho \models$ **result** $\leftrightarrow \rho \models \exists v.\, C$ }

predicate $\rho \sqsubseteq \rho'$ =
  $\mathrm{dom}(\rho) \subseteq \mathrm{dom}(\rho') \wedge \forall v \in \mathrm{dom}(\rho).\, \rho(v) = \rho'(v)$

let rec sym-interp$_N$ $(\sigma \mid C;\, \rho)$ (i)
  (* Result interpretations extend initial interpretation *)
  ensures { $\forall (\sigma' \mid C';\, \rho')_\beta \in$ **result** $\rightarrow \rho \sqsubseteq \rho'$ }
  ensures { (* Over-/underapproximation *) … }

1. All variables in the symbolic state are in the domain of the interpretation

2. Fresh variables not in domain of the interpretation

3. Existential quantification does not introduce new variables

4. Extension of an interpretation: all values are retained

5. All result interpretations are extensions of the initial interpretation

Proofs of the symbolic interpreter

- three main functions for symbolic execution: `sym-interp`, `sym-interp-list`, `sym-interp-loop`
- post-conditions covering over-approximation, under-approximation, extension of interpretations
- 31 verification goals, 86 lightweight interactive transformations, 186 leaf verification conditions

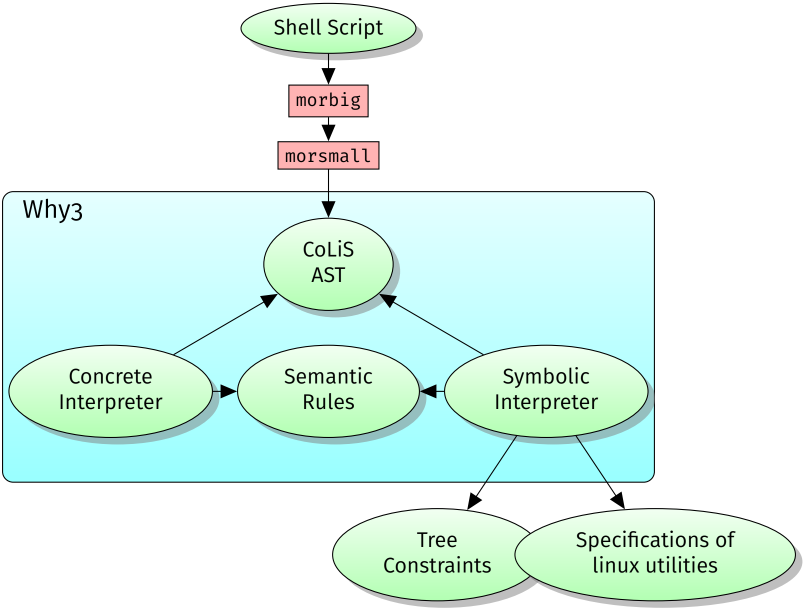
Prover	Verification conditions	Fastest	Slowest	Average
CVC4 1.6	162	0.03	2.57	0.26
Alt-Ergo 2.2.0	20	0.03	3.59	0.42
Eprover 2.2	4	0.09	0.31	0.20

Extraction to OCaml

- Why3 code is extracted to OCaml and can be executed
  - test unsatisfiability of constraints using Alt-Ergo library
  - symbolic variables substituted by abstract OCaml type that ensure post-condition of `fresh_var`

```
$ symbolic-imp p0
Initial state: (x → v1, y → v2 | ⊤)
Normal states: 2
  state 0:
    (x → v1, y → v3 | ∃v4. ∃v2. ⊤ ∧ v4 = v1 − v2 − 1 ∧ v4 = 0 ∧ v3 = v1 − 3)
  state 1:
    (x → v5, y → v4 | ∃v1. ∃v2. ⊤ ∧ v4 = v1 − v2 − 1 ∧ v4 ≠ 0 ∧ v5 = v4 − 3)
```

CoLiS Workflow for script analysis



Identifying bugs in Debian maintainer scripts using symbolic execution

Example

```
$ colis --run-symbolic --add-symbolic-fs simple.fs \\
    sgml-base.preinst install
- id: error-11
  root: r3879
  clause: ∃ lib4, var5, sbin8, lib11, local12, lib14...
    r1[bin]bin35 ∧ r1[etc]etc3889 ∧ r1[run]run21 ∧...
    r1[usr]usr17 ∧ r1[var]var5 ∧ var5[lib]lib4 ∧...
    etc3889[sgml]sgml3873 ∧ file(sgml3873)...
    etc3889[sgml]sgml3873 ∧ file(sgml)...

  stdout: |
    [UTL] test 'install' = 'install': strings equal
    [UTL] test 'install' = 'upgrade': strings not equal
    [UTL] test -d /var/lib/sgml-base: no resolve
    [UTL] mkdir /var/lib/sgml-base: create directory
    [UTL] test -d /etc/sgml: path resolves to file
    [UTL] mkdir /etc/sgml: target already exists
 ...  20 normal states and 74 other error states
```

- CoLiS interpreters available at https://github.com/colis-anr/colis-language

Identifying bugs in Debian maintainer scripts using symbolic execution

Example

```
$ colis --run-symbolic --add-symbolic-fs simple.fs \\
    sgml-base.preinst install
- id: error-11
  root: r3879
  clause: ∃ lib4, var5, sbin8, lib11, local12, lib14...
    r1[bin]bin35 ∧ r1[etc]etc3889 ∧ r1[run]run21 ∧...
    r1[usr]usr17 ∧ r1[var]var5 ∧ var5[lib]lib4 ∧...
    etc3889[sgml]sgml3873 ∧ file(sgml3873)...
    etc3889[sgml]sgml3873 ∧ file(sgml)...

  stdout: |
    [UTL] test 'install' = 'install': strings equal
    [UTL] test 'install' = 'upgrade': strings not equal
    [UTL] test -d /var/lib/sgml-base: no resolve
    [UTL] mkdir /var/lib/sgml-base: create directory
    [UTL] test -d /etc/sgml: path resolves to file
    [UTL] mkdir /etc/sgml: target already exists
 ...  20 normal states and 74 other error states
```

Concrete test

```
$  touch /etc/sgml
$  apt install sgml-base
dpkg: error processing archive /var/cache/apt/
  archives/sgml-base_1.29_all.deb (--unpack):
new sgml-base package pre-installation script subprocess
  returned error exit status 1
Errors were encountered while processing:
 /var/cache/apt/archives/sgml-base_1.29_all.deb
E: Sub-process /usr/bin/dpkg returned an error code (1)
```

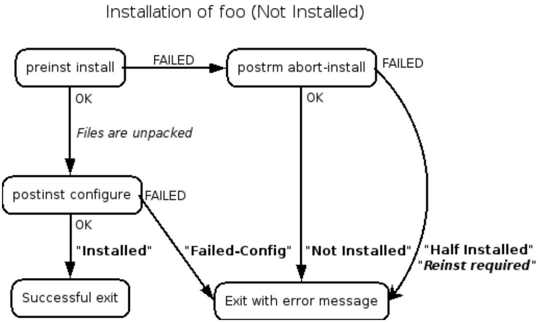
sgml-base.preinst

```
...
if [ ! -d /var/lib/sgml ]; then
  mkdir /var/lib/sgml 2>/dev/null
fi
...
```

- CoLiS interpreters available at https://github.com/colis-anr/colis-language
- statistics for Debian maintainer scripts at http://ginette.informatique.univ-paris-diderot.fr/~niols/colis-covering-report/

What to verify on an installation script?

- no runtime error (i.e. return code of script should be 0)
- composition properties
  - install ; purge = identity
  - failed install ; successful install = successful install
  - proper combinations of `preinst`/`postinst`/`prerm`/`postrm` with respect to the *Debian Policy*



https://www.debian.org/doc/debian-policy/ap-flowcharts.html

Conclusions

- formalisation of correctness properties of a symbolic interpreter
- formalised and verified symbolic interpreter for IMP
- ghost annotations useful to reformulate correctness properties
- transfer of correctness properties to the symbolic interpreter for CoLiS language

Thanks for your attention!

Questions?