

Automatically Detecting Inconsistencies in Program Specifications

Aditi Tagore and Bruce W. Weide

The Resolve/Reusable Software Research Group (RSRG)

Department of Computer Science and Engineering

The Ohio State University

Supported by the National Science Foundation under

<http://www.cse.ohio-state.edu/rsrg/>

Grants No. CCF-0811737, ECCS-0931669, CCF-1162331

Introduction

- The robustness of an automated formal verification system depends on
 - having an intended **specification**
 - a theorem prover's ability to **prove the verification conditions** generated from a proposed implementation
- **Inconsistencies** in the specification may be introduced due to human errors

Contributions

- Lightweight techniques that
 - formulate the conditions for admissibility of **program specifications**
 - check the logical consistency of **loop annotations** (invariants and variants)
 - determine whether a programmer has supplied the **appropriate modes for the parameters** of an operation

Background

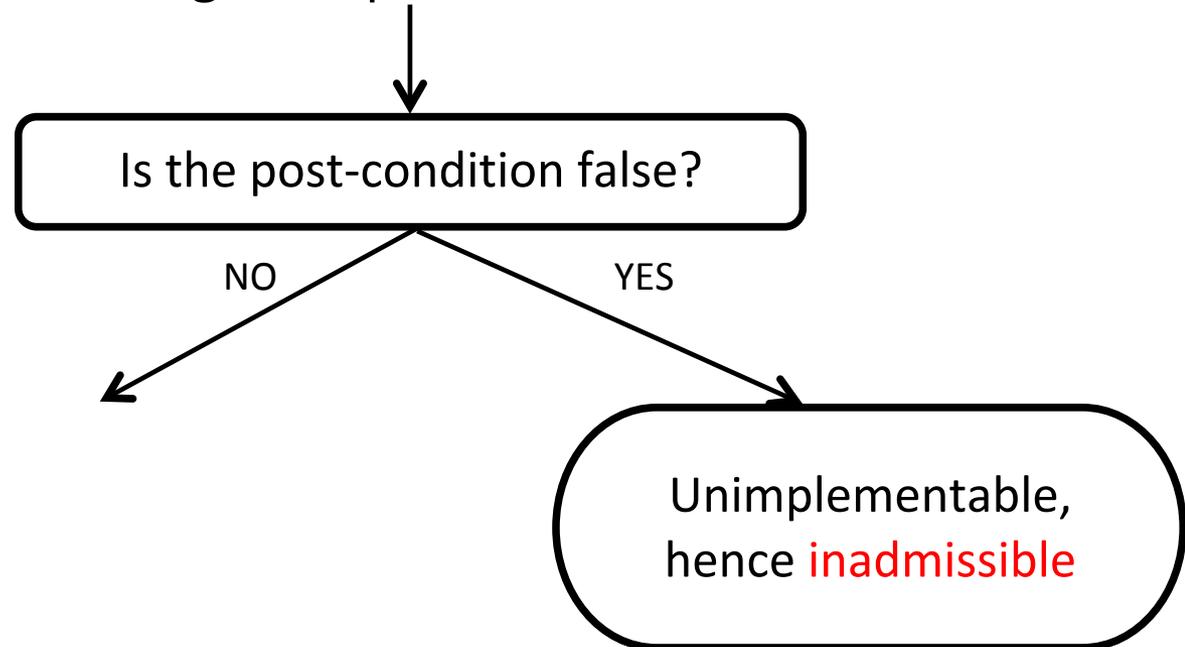
- Specifications and their implementations are written in **Resolve**
- Back-end prover to detect logical inconsistencies is the SMT solver **Z3**
- Main ideas apply to specifications and formal verification / theorem-proving technology in general

Contributions

- Lightweight techniques that
 - formulate the conditions for admissibility of **program specifications**
 - check the logical consistency of **loop annotations** (invariants and variants)
 - determine whether a programmer has supplied the **appropriate modes for the parameters** of an operation

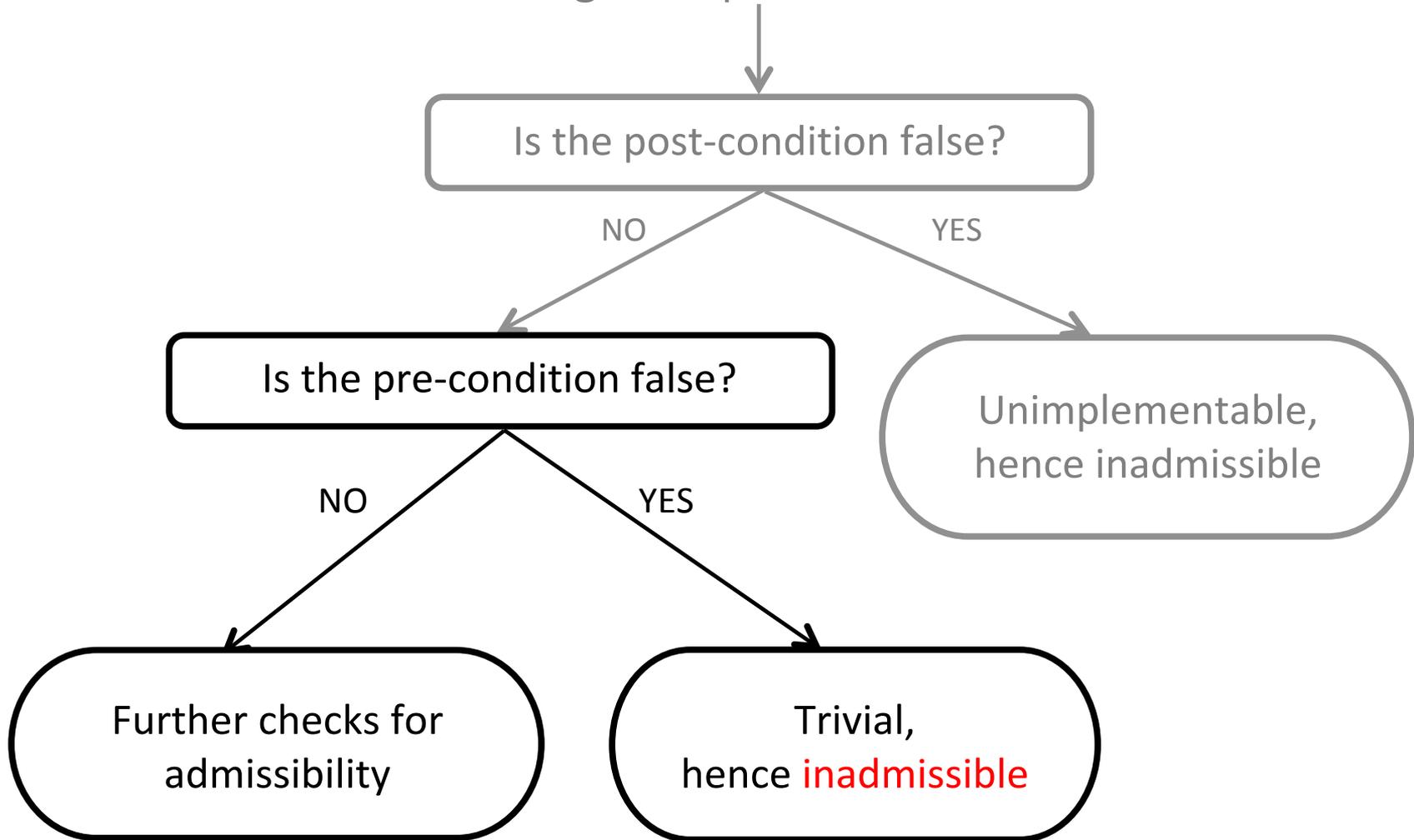
Specification Filter

Program Specification



Specification Filter

Program Specification



Example 1: Proposed Specification

procedure DecrementBy3

(updates i : Integer)

requires

$i \geq 3$

Pre-condition
(pre)

ensures

$i = \#i - 3$ and $i > 0$

Post-condition
(post)

Formula to Check Admissibility

- Check validity of

$$\forall x_1, \dots, x_n (\textit{pre} \Rightarrow \exists y_1, \dots, y_m (\textit{post}))$$

where:

(x_1, \dots, x_n) are the incoming values of the variables in *pre* and

(y_1, \dots, y_m) are the outgoing values of the variables in *post*

Invalid Formula

Pre-condition
(pre)

$$\forall \#i (\#i \geq 3 \Rightarrow \exists i (i = \#i - 3 \wedge i > 0))$$

Post-condition
(post)

Example 2: Proposed Specification

**procedure Divide (updates i: Integer,
restores j: Integer,
replaces r: Integer)**

ensures

**#i = i * j + r and 0 < r
and r < |j|**

Formula to Check Admissibility

$\forall \#i, \#j, \#r$ (*true* \Rightarrow

$\exists i, j, r ((\#i = i * j + r) \wedge (0 < r)$
 $\wedge (r < |j|) \wedge (j = \#j))$)

Invalid Formula

- The formula is automatically translated into Z3's SMT2 input format
 - Asked to prove it
 - Concludes that **the formula is invalid**
 - Produces a counter-example in which $\#j = 0$

Invalid Formula

$\forall \#i, \#j, \#r \text{ (true} \Rightarrow$

$$\begin{aligned} & \exists i, j, r \text{ ((}\#i = i * j + r) \wedge (0 < r) \\ & \wedge (r < |j|) \wedge (j = \#j)) \end{aligned}$$

- Problem when $\#j = 0$
- Programmer sees that a value of 0 cannot be allowed for the divisor j

New Proposed Pre- and Post-condition

**procedure Divide (updates i : Integer,
restores j : Integer,
replaces r : Integer)**

requires

$$j \neq 0$$

ensures

$$\#i = i * j + r \text{ and } 0 < r \\ \text{and } r < |j|$$

Formula to Check Admissibility

$\forall \#i, \#j, \#r ((\#j \neq 0) \Rightarrow$

$\exists i, j, r ((\#i = i * j + r) \wedge (0 < r)$
 $\wedge (r < |j|) \wedge (j = \#j)))$

- Formula declared invalid by Z3
- Produces a counter-example in which $\#j = 1$

Formula to Check Admissibility

$\forall \#i, \#j, \#r ((\#j \neq 0) \Rightarrow$

$$\begin{aligned} & \exists i, j, r ((\#i = i * j + r) \wedge (0 < r) \\ & \wedge (r < |j|) \wedge (j = \#j))) \end{aligned}$$

- Problem when $\#j = 1$
- Programmer sees that the remainder from the Divide operation may be equal to 0

Final Pre- and Post-condition

**procedure Divide (updates i :Integer,
restores j :Integer,
replaces r :Integer)**

requires

$$j \neq 0$$

ensures

$$\#i = i * j + r \text{ and } 0 \leq r$$

$$\text{and } r < |j|$$

Example 3

- Resolve supports user-defined mathematical functions and predicates

```
definition IS_ODD(i: integer): boolean is  
  i mod 2 /= 0
```

Proposed Specification

```
procedure Halve(updates i: Integer)
```

```
  ensures
```

```
    if IS_ODD(#i) then
```

```
      #i = i + i
```

```
    else
```

```
      #i = i + i + 1
```

Formula to Check Admissibility

$\forall \#i$ (*true* \Rightarrow

$\exists i$ ($(IS_ODD(\#i) \Rightarrow \#i = i + i) \wedge$

$(\neg IS_ODD(\#i) \Rightarrow \#i = i + i + 1)))$

Invalid Formula

$\forall \#i \text{ (true} \Rightarrow$

$\exists i \text{ ((IS_ODD}(\#i) \Rightarrow \#i = i + i) \wedge$

$(\neg \text{IS_ODD}(\#i) \Rightarrow \#i = i + i + 1)))$

- Problem when $\#i = 0 \wedge i = 0$

Final Pre- and Post-condition

procedure Halve(updates i: Integer)

ensures

if IS_ODD(#i) then

#i = i + i + 1

else

#i = i + i

Other Datatypes

- Techniques mentioned are also applicable to specifications involving **other data-structures**
- **Not** restricted to integers

**procedure Dequeue (updates q: Queue,
replaces x: Item)**

ensures

#q = <x> * q

Contributions

- Lightweight techniques that
 - formulate the conditions for admissibility of **program specifications**
 - **check the logical consistency of loop annotations (invariants and variants)**
 - determine whether a programmer has supplied the **appropriate modes for the parameters** of an operation

Consistency of Loop Annotations

- Beneficial to detect defects in loop annotations at an **early stage**, even before VCs are generated
- Checks that can be carried out by examining only the **loop annotations** and the boolean condition
- Will **not** examine the **loop body**

Consistency of Loop Annotations

```
procedure Add(updates n: Natural,  
             restores m:Natural)
```

```
...
```

```
loop
```

```
maintains  $n + m = \#n + \#m$  and  $k + m = \#k + \#m$   
and  $z = 0$ 
```

```
decreases m
```

```
while not AreEqual(m, z) do
```

```
...
```

```
end loop
```

```
...
```

```
end Add
```

Invariant (inv)

Variant (var)

Boolean condition
(b)

Loop Annotations - Check 1

- The invariant and the boolean condition are **contradiction-free**
- If two or more conjuncts in the loop invariant contradict each other, then the invariant evaluates to *false* and **the invariant is inadmissible**
- Same with the boolean condition

Loop Annotations - Check 2

- The **variant is positive** every time the loop iterates

$$\forall x_1, \dots, x_n (b \wedge inv \Rightarrow var > 0)$$

Check 2 - Formula to Check Consistency

- Check validity of :

$\forall n, \#n, m, \#m, k, \#k, z, \#z ($

$$m \neq z$$

$$\wedge n + m = \#n + \#m$$

$$\wedge k + m = \#k + \#m$$

$$\wedge z = 0$$

$$\Rightarrow m > 0)$$

Loop Annotations - Check 3

- The **loop invariant is valid** before the loop iterates for the first time

$$\exists x_1, \dots, x_n (inv_{init})$$

where inv_{init} is the invariant with $\#$ symbols removed

Check 3 - Formula to Check Consistency

- Check validity of :

$$\begin{aligned} \exists n, m, k, z & (n + m = n + m \\ & \wedge k + m = k + m \\ & \wedge z = 0) \end{aligned}$$

Contributions

- Lightweight techniques that
 - formulate the conditions for admissibility of **program specifications**
 - check the logical consistency of **loop annotations** (invariants and variants)
 - determine that a programmer has supplied the **appropriate modes for the parameters** of an operation

Detecting Incorrect Parameter Modes

- The ways in which a parameter to an operation are used in the specification may not be **consistent** with the mode of the parameter
- **Syntactic checks** are employed to give suggestions to the programmer about the appropriate mode

Parameter Mode	Description
restores	The incoming and the outgoing values of the parameter are the same
updates	The incoming and outgoing values of the parameter are potentially different
replaces	The operation's behavior does not depend on the incoming value (a special case of updates)
clears	The outgoing value of the parameter is an initial value of its type (a special case of updates)

Benefits of Parameter Modes

- Simplify specifications by **reducing redundancy** — restores mode
- Help programmers write, and clients read, specifications — replaces mode, clears mode

Parameter Modes – Check 1

- A **replaces**-mode parameter should not appear in the **requires** clause

```
procedure Divide (updates i: Integer,  
                 replaces j: Integer,  
                 replaces r: Integer)
```

```
requires
```

```
  j /= 0
```

Parameter Modes – Check 2

- **Incoming value** of a parameter should occur in the post-condition if and only if the parameter mode is one of
 - `updates`
 - `clears`

The Header of Add

procedure Add

(updates n : Natural,
restores m : Natural)

Post-conditions

ensures

$$n = \#n + \#m$$

INCONSISTENT

ensures

$$n = n + m$$

INCONSISTENT

ensures

$$n = \#n + m$$

CONSISTENT

Parameter Modes – Check 3

- When the parameter mode is `clears`, the outgoing parameter value should **not** appear in the post-condition
- Example: Adding conjunct $i = 0$ to the post-condition is unnecessary

Parameter Modes – Check 4

- If the mode of the variable is restores, then a conjunct $i = \mathit{constant}$ should not appear in the post-condition
- Example: A conjunct $i = 5$ should not be added to the post-condition

Summary

- Lightweight techniques that
 - formulate the conditions for admissibility of **program specifications**
 - check the logical consistency of **loop annotations** (invariants and variants)
 - determine whether a programmer has supplied the **appropriate modes for the parameters** of an operation

Questions?

Aditi Tagore and Bruce W. Weide

The Resolve/Reusable Software Research Group (RSRG)

Department of Computer Science and Engineering

The Ohio State University

<http://www.cse.ohio-state.edu/rsrg>