Automatic Fixing of Programs with Contracts

Yu Pei

Chair of Software Engineering, ETH Zurich

December 2, 2015

To Err Is Human

A French rail company has ordered 2000 new trains
that are too big for 1300 stations they are due to serve.

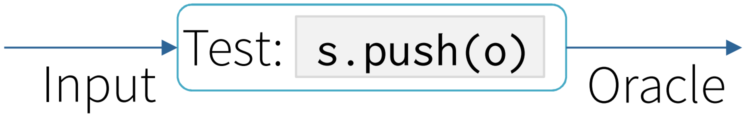
Programs Have Faults

- Specification vs. Implementation
  - What a program should do vs. what a program really does
  - When they conform, the program is correct.

- Program faults are discrepancies between the two
  - Unpleasant, unacceptable, or even fatal
  - Expensive
  - Overwhelming to fix manually

Automatic Unit Testing

- Unit testing



- AutoTest
  - Automatic test case generation
    - Precondition of the routine as the input filter
    - Postcondition of the routine as the oracle
  - Test case categorization

```
push(element: E)
  require
    element /= Void
  ensure
    count = old count + 1
    top() = element
```

Precondition	Postcondition	Test Case
×	--	Invalid
✓	✓	Valid, Passing
✓	×	Valid, Failing

Outline

AutoFix

Program with Contracts

178 citations in total

An Example Fault

```
class CIRCULAR [G]
  duplicate (m: INTEGER): CIRCULAR [G]
      -- A duplicate with at most ‘m’
      -- elements copied from ‘Current’.
    require m >= 0
    do
      create Result.make (count)
      ...
    end

  make (n: INTEGER)
      -- Initialize ‘Current’ for
      -- ‘n’ elements.
    require n >= 1
    do
      create list.make_list (n)
    end

  list:  ARRAYED_LIST [G]     -- Storage
  count: INTEGER   -- Length of circular
  ...
```

An Example Fault

```
class CIRCULAR [G]
  duplicate (m: INTEGER): CIRCULAR [G]
      -- A duplicate with at most ‘m’
      -- elements copied from ‘Current’.
    require m >= 0
    do
      create Result.make (count)
      ...
    end

  make (n: INTEGER)
      -- Initialize ‘Current’ for
      -- ‘n’ elements.
    require n >= 1
    do
      create list.make_list (n)
    end

  list:  ARRAYED_LIST [G]     -- Storage
  count: INTEGER   -- Length of circular
  ...
```

①

```
duplicate (m: INTEGER): ...
  do      -- fix implementation
    if count = 0 then
      create Result.make (1)
    else
      create Result.make (count)
    end
    ...
  end
```

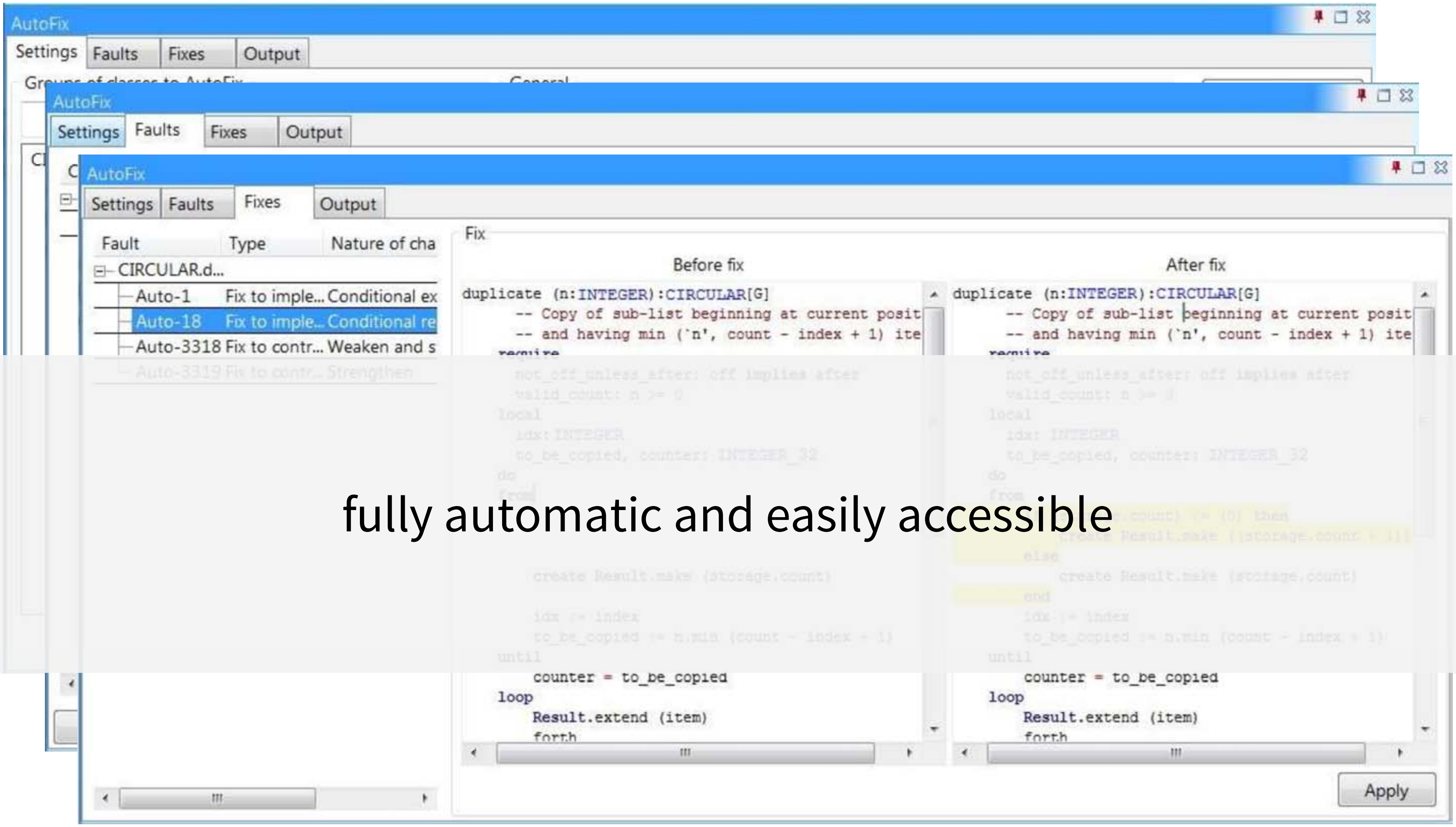
②

```
duplicate (m: INTEGER): ...
  require      -- strengthen
    count > 0  -- precondition
    m >= 0
```

③

```
make (n: INTEGER)
  require      -- weaken
    n >= 0     -- precondition
```

The AutoFix Tool



Fault	Type	Nature of cha
CIRCULAR.d...		
Auto-1	Fix to imple...	Conditional ex
Auto-18	Fix to imple...	Conditional re
Auto-3318	Fix to contr...	Weaken and s
Auto-3319	Fix to contr...	Strengthen

Before fix

```
duplicate (n:INTEGER):CIRCULAR[G]
    -- Copy of sub-list beginning at current posit
    -- and having min (`n', count - index + 1) ite
  require
    not_off_unless_after: off implies after
    valid_count: n >= 0
  local
    idx: INTEGER
    to_be_copied, counter: INTEGER_32
  do
  from

      create Result.make (storage.count)

      idx := index
      to_be_copied := n.min (count - index + 1)
  until
      counter = to_be_copied
  loop
      Result.extend (item)
      forth
```

After fix

```
duplicate (n:INTEGER):CIRCULAR[G]
    -- Copy of sub-list beginning at current posit
    -- and having min (`n', count - index + 1) ite
  require
    not_off_unless_after: off implies after
    valid_count: n >= 0
  local
    idx: INTEGER
    to_be_copied, counter: INTEGER_32
  do
  from
      if (storage.count) <= (0) then
        create Result.make ((storage.count + 1))
      else
        create Result.make (storage.count)
      end
      idx := index
      to_be_copied := n.min (count - index + 1)
  until
      counter = to_be_copied
  loop
      Result.extend (item)
      forth
```

fully automatic and easily accessible

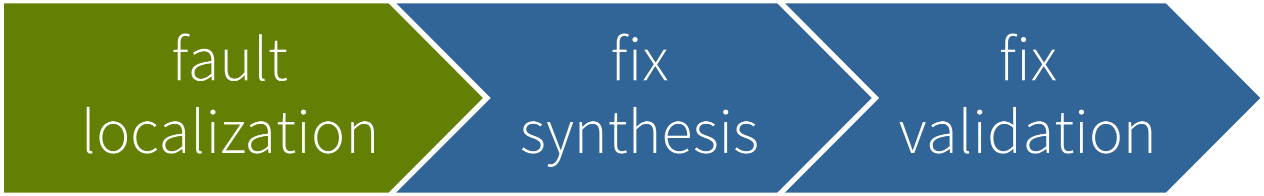
Fixing the Implementation with ImpleFix

- Assumption
  - Contracts are correct

- Target faults
  - Incorrect source states of object transitions as causes
  - Simple changes as fixes

- Three steps
  - Fault localization
  - Fix synthesis
  - Fix validation

Fixing the Implementation with ImpleFix

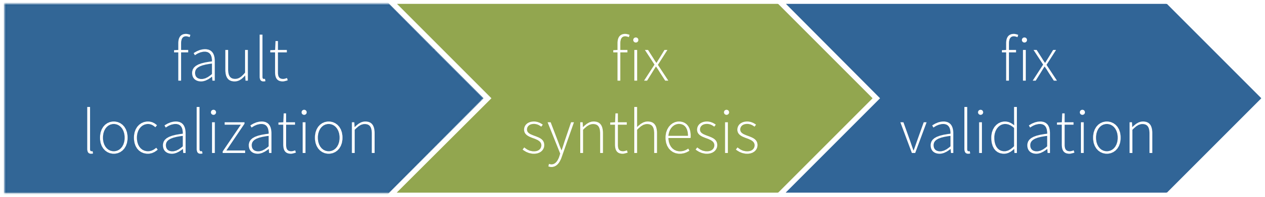


- Abstract execution traces using state snapshots:
  [e, l, v]
- Compute suspiciousness scores of the snapshots using multiple metrics
- Consider the most suspicious snapshots as potential fault causes

```
L5. create Result.make (count)
```

```
[m >= 0,   L5, True]           0.2
[count=0, L5, True]            1.3
…
```

Fixing the Implementation with ImpleFix



- Construct fix actions to change the snapshot states
  - call **remove**
  - replace **count** with **count + 1**
- Instantiate candidate fixes from schemas using fix actions and suspicious snapshots

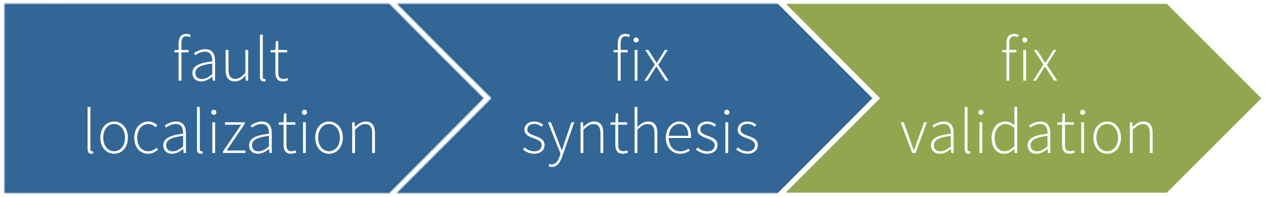
```
L5. create Result.make (count)
```

```
[count=0, L5, True]                1.3
…
```

```
if count = 0 then
  remove
end
create Result.make (count)
```

```
if count = 0 then
  create Result.make (count + 1)
else
  create Result.make (count)
end
```

Fixing the Implementation with ImpleFix



- Apply each fix to the program and re-execute all the tests
- Mark as valid the fixes that, when applied, make all the tests pass
- Report the first **n** valid fixes to the user

```
if count = 0 then
  remove
end
create Result.make (count)
```

✗

```
if count = 0 then
  create Result.make (1)
else
  create Result.make (count)
end
```

✓

Experimental Evaluation of ImpleFix

- To understand the behavior of ImpleFix and the quality of generated fixes

- Experimental setup
  - AutoTest for fault detection and test preparation
  - 204 faults from 4 different code bases
  - 9 different settings of testing time for each fault
  - 30 repetitions for each fault and setting

Evaluation Results of ImpleFix

- How many faults can ImpleFix fix?
  - Valid fixes to **86** faults **(42%)**
- What is the quality of the fixes produced by ImpleFix?
  - Proper fixes to **51** faults **(25%)**
- What is the cost of fixing faults with ImpleFix?
  - On average **≤20 minutes** per valid fix, including the time required for test generation
- How robust is ImpleFix's performance?
  - **48 (56%)** of the faults that ImpleFix managed to fix at least once were fixed in **over 95%** of the sessions

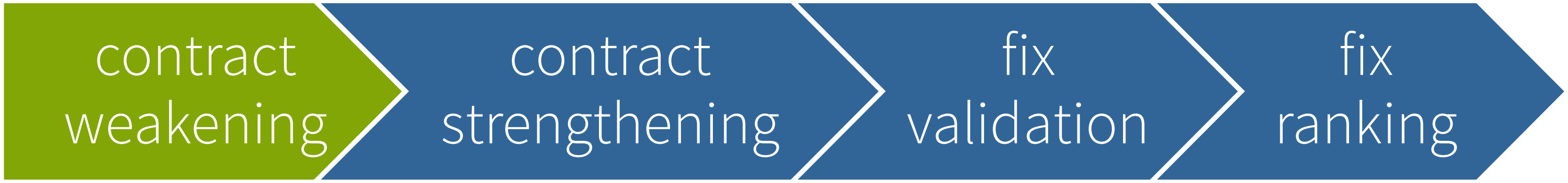
Correcting the Specification with SpeciFix

- Assumption
  - The implementation is correct

- Goal of fixing
  - Successful executions should be allowed
  - Unsuccessful executions should be forbidden

- Four steps
  - Contract weakening
  - Contract strengthening
  - Fix validation
  - Fix ranking

Correcting the Specification with SpeciFix



contract weakening

contract strengthening

fix validation

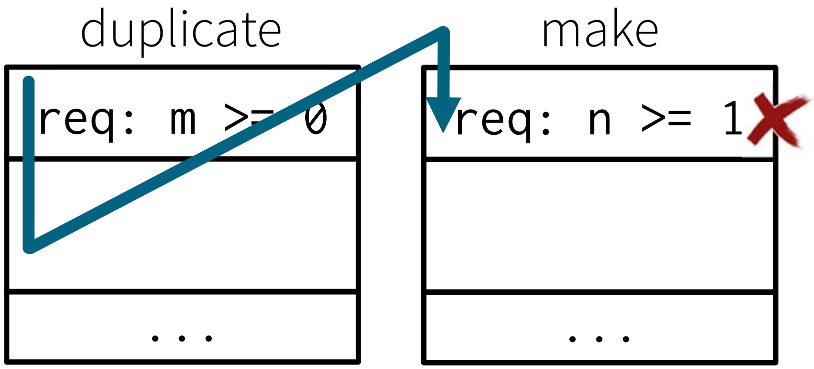
fix ranking

- Infer the set $\boldsymbol{\Omega}$ of the weakest preconditions for `make`
- Weaken the precondition of `make`:

$$P_{\texttt{make}} \textbf{ or } \omega$$

$\boldsymbol{\Omega}$ = {n >= 0, ...}

```
make
  require n >= 1 or n >= 0
  require ...
```



duplicate

req: m >= 0

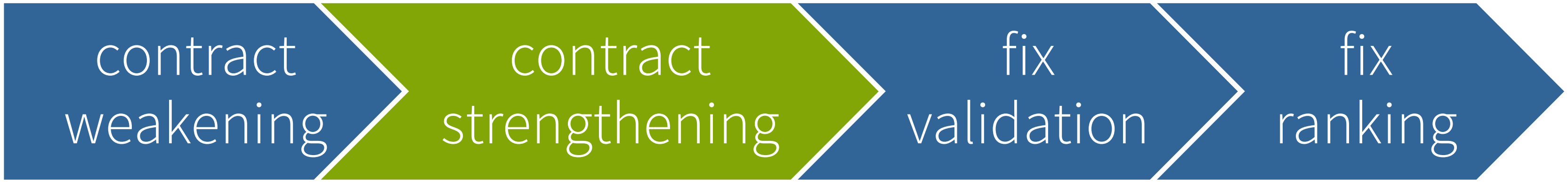
...

make

req: n >= 1

...

Correcting the Specification with SpeciFix



contract weakening

contract strengthening

fix validation

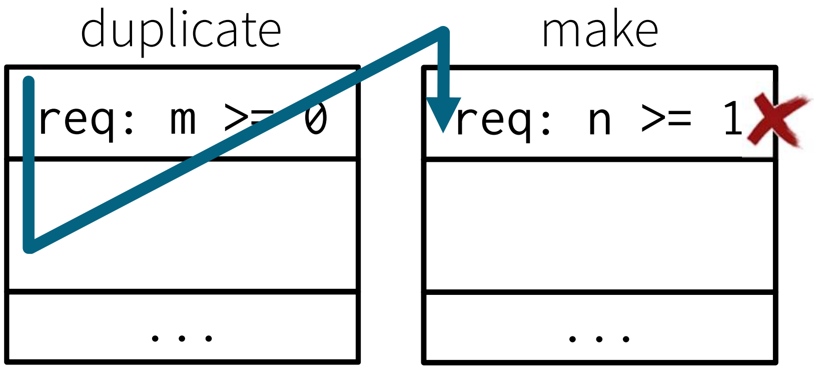
fix ranking

- Infer the set $\boldsymbol{\Sigma}$ of preconditions for `duplicate`
- Strengthen the precondition of `duplicate`:

  $P_{\texttt{duplicate}}$ **and** $\boldsymbol{\sigma}$

```
Σ = {count /= 0, ...}
```

```
duplicate
  require m >= 0 and count /= 0
  require ...
```



duplicate

req: m >= 0

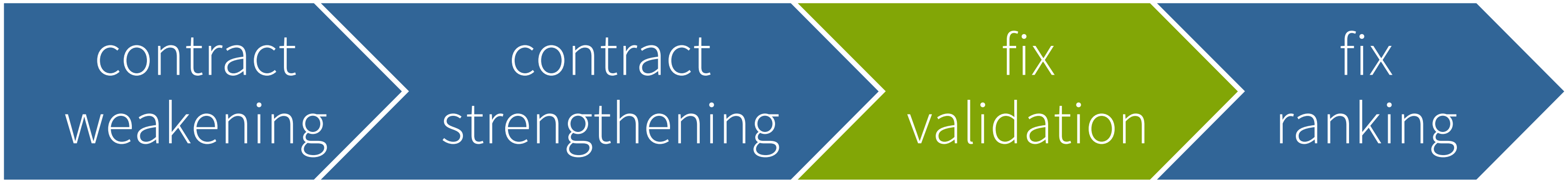
...

make

req: n >= 1

...

Correcting the Specification with SpeciFix



contract weakening

contract strengthening

fix validation

fix ranking

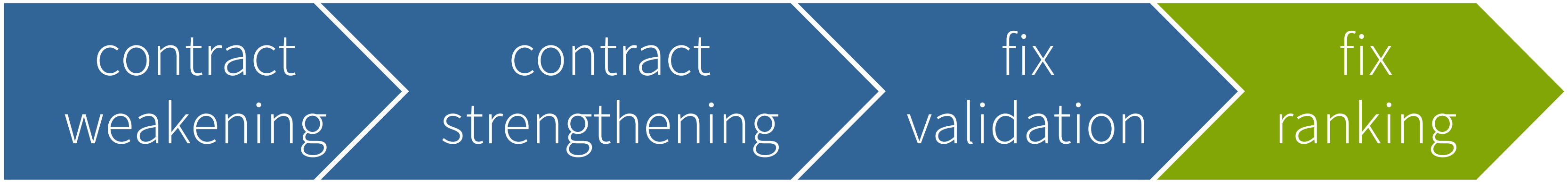
- Apply each fix to the program and re-execute the tests
- Mark as valid the fixes that, when applied, make the tests either passing or invalid

```
make
✓ require n >= 1 or n >= 0
  require ...
```

```
duplicate
✓ require m >= 0 and count /= 0
  require ...
```

- Use more tests for validation than for fix generation

Correcting the Specification with SpeciFix



- Prefer fixes resulting in weaker contracts, or more passing tests

```
make                                      ①
  require n >= 1 or n >= 0
```

```
duplicate                                 ②
  require m >= 0 and count /= 0
```

```
...
```

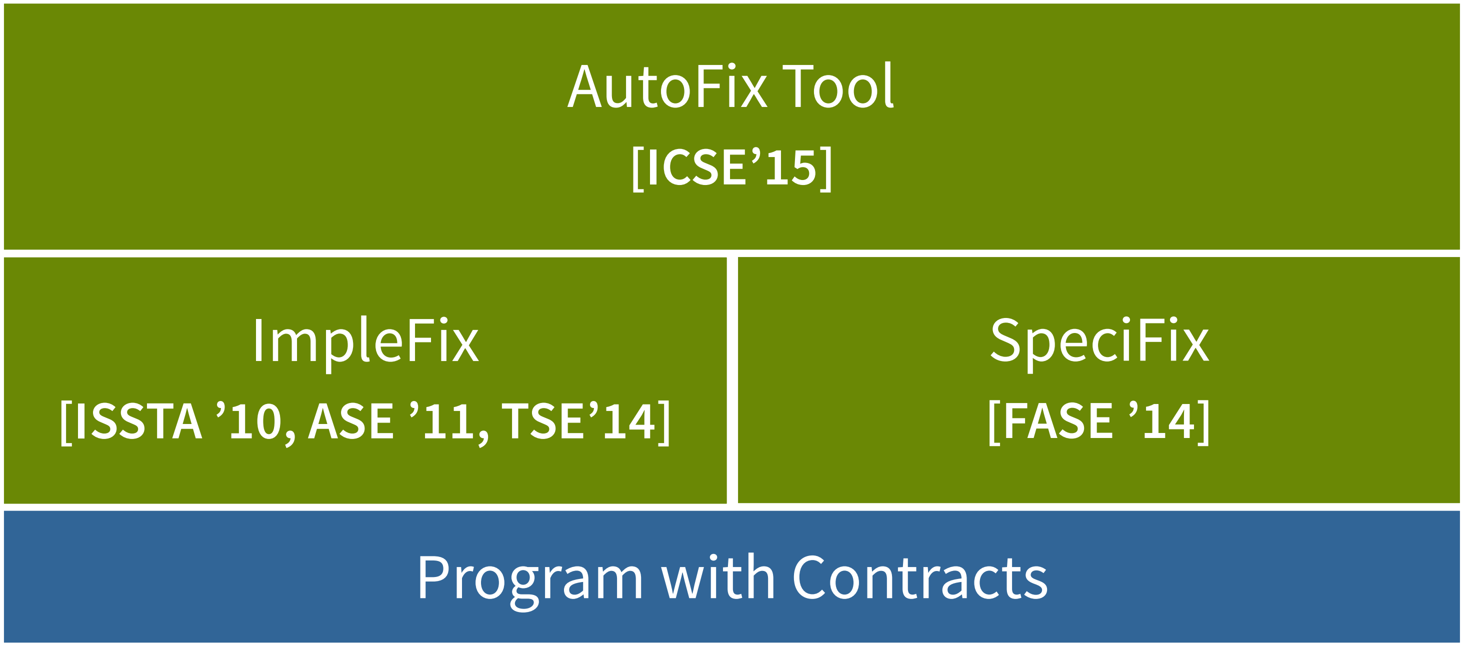
Experimental Evaluation of SpeciFix

- Experimental subjects
  - **44** faults from **10** standard library classes

- Result
  - Valid fixes to **42** faults, and proper fixes to **11**
  - On average, **3** minutes for fixing and **31** minutes for testing per fix

  - When both available, proper fixes to contracts are often preferable to proper fixes that change the implementation

Summary



AutoFix Tool

[ICSE’15]

ImpleFix

[ISSTA ’10, ASE ’11, TSE’14]

SpeciFix

[FASE ’14]

Program with Contracts

Thank you!