Automated Fixing of Programs with Contracts

Yi Wei, Yu Pei, Carlo A. Furia, Lucas S. Silva,
Stefan Buchholz, Bertrand Meyer and Andreas Zeller

Chair of Software Engineering, ETH Zürich

Software Engineering Chair, Saarland University

Context

Testing finds *faults;*

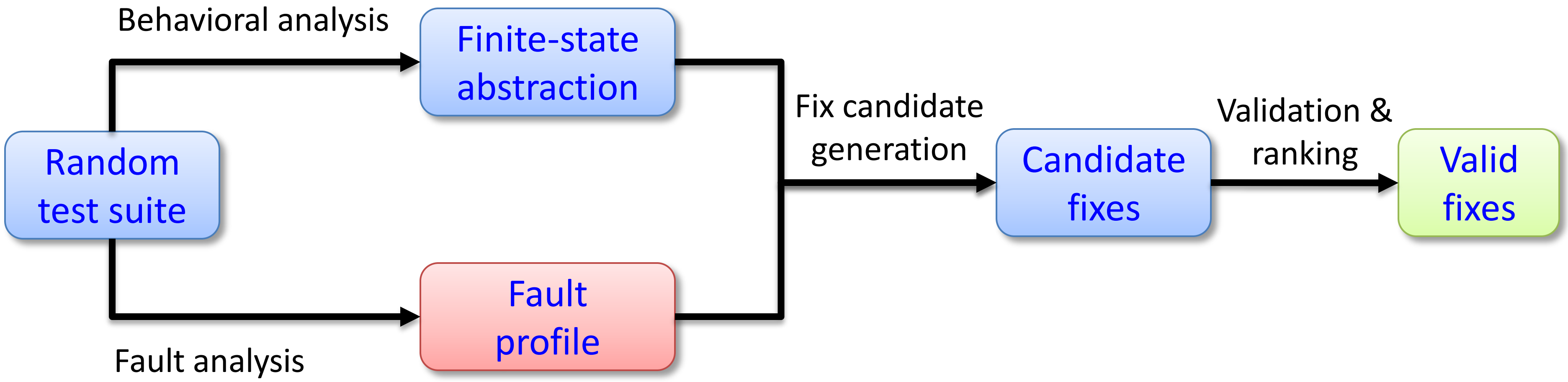
Automated debugging finds *locations;*

Automated fixing finds *corrections.*

Automatic fixing in production software

- 16 out of 42 (38%) faults are fixed.
- Capable for fixing faults due to missing method calls.
- Average fixing time is 2.6 minutes per fault.
- It takes 3 to 5 minutes to understand a fix.
- In a small user study, 4 out of 6 of the selected fixes are the same as those from programmers.

Fixing process overview



Behavioral analysis

Finite-state abstraction

Random test suite

Fault analysis

Fault profile

Fix candidate generation

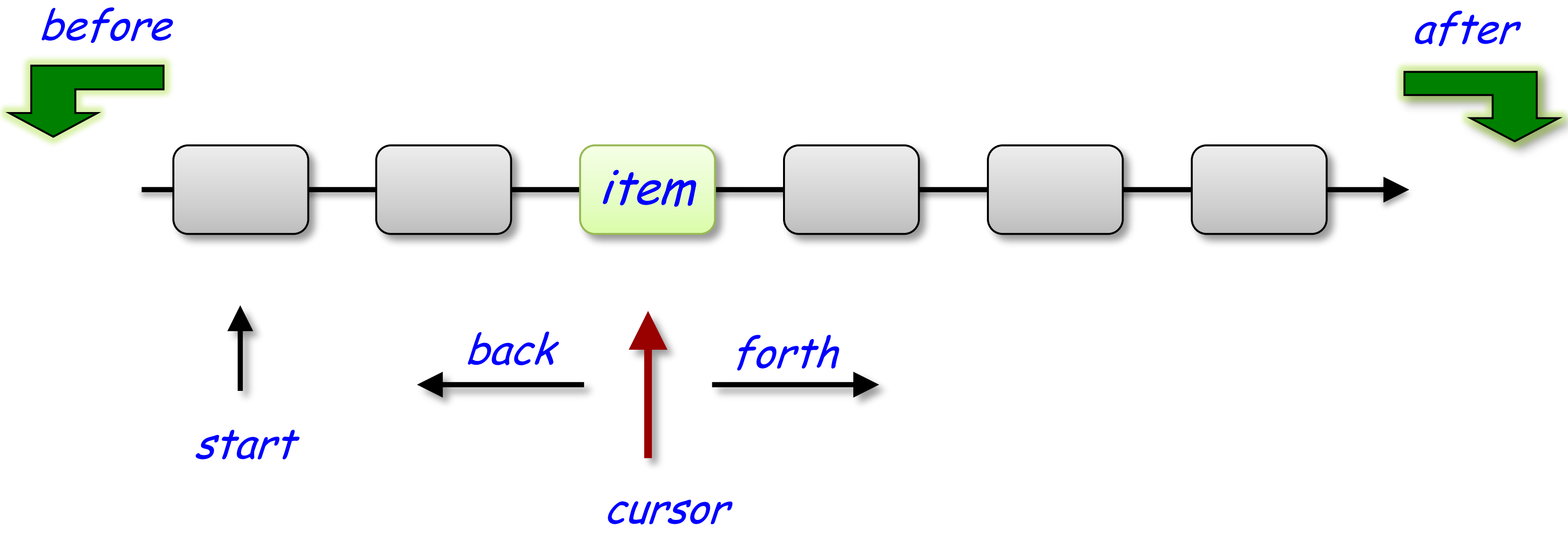
Candidate fixes

Validation & ranking

Valid fixes

Fault in *TWO_WAY_SORTED_SET.duplicate*

```
duplicate (n: INTEGER): TWO_WAY_SORTED_SET
        -- Copy of sub-set beginning at cursor position,
        -- containing at most n element.
        -- Class implemented using a LINKED_LIST.
```



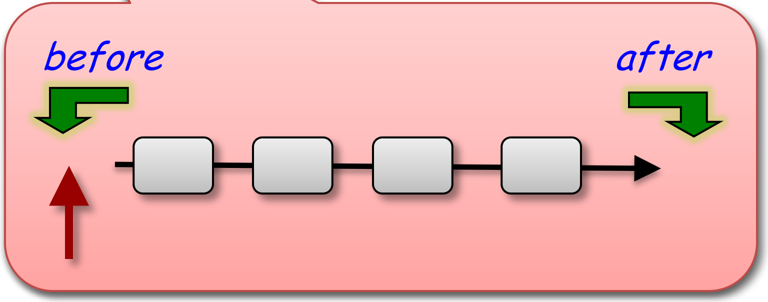
Failure in implementation

```
duplicate (n: INTEGER): TWO_WAY_SORTED_SET
    do
        pos := cursor
        Result := new_chain
        Result.forth

        from until (counter = n) or after loop
            Result.put_left (item)
            forth
            counter := counter + 1
        end

        go_to (pos)
    end
```

```
item: ANY
        -- Element under cursor
    require
        (not before) and (not after)
```



Proposed fix

```
duplicate (n: INTEGER ): ...
  do
     pos   := cursor
     Result := new_chain
     Result.forth
     from until  (counter = n) or after  loop
          Result.put_left (item )
          forth
          counter  := counter + 1
     end
     go_to   (pos )
  end
```

Faulty version

```
duplicate (n: INTEGER ): ...
  do
     pos   := cursor
     Result := new_chain
     Result.forth

     from until  (counter = n ) or after  loop
        if before then
            forth
        else
            Result.put_left (item )
            forth
            counter  := counter + 1
        end
     end
     go_to   (pos )
  end
```

Fixed version

Steps to generate fixes

1. Abstract program state.
2. Compare passing and failing state invariant.
3. Synthesize candidates from fix schema and behavioral model.
4. Validate and then rank candidates.

Abstracting state through boolean queries

Boolean queries are argument-less functions returning a boolean value:

- Define object states absolutely.
- Usually don't have preconditions.
- Widely used in contracts, capturing important object properties.

For *TWO_WAY_SORTED_SET*, the abstract state consists of: *after*, *before*, *is_empty*, …

State invariant difference as fault profile

- Apply random testing.
- Retrieve states represented as boolean queries.
- Derive state invariant at each program location.
- Compare state invariant difference between passing and failing runs.

Deriving state invariant

```
duplicate (n: INTEGER ): TWO_WAY_SORTED_SET
        -- Copy of sub-set beginning at cursor position,
         containing at most n element
           := cursor
          lt := new_chain
          lt.forth

    fr  m until (counter = n) or a   er loo
          Result.put_left (item)
          forth
          counter := counter + 1
    end
    go_to (pos)
```

Passing test 1
not is_empty

Passing state invariant:
not is_empty
not before
not after

Passing test 2
not is_empty
not before
not after
not isfirst

Passing test 3
not is_empty
not before
not after
sorted

Failing test 1

Failing state invariant:
not is_empty
before
not after

Failing test 2
not is_empty
before
not after
sorted

Use daikon

Benefits of state invariant

- Pinpoint the essential difference between passing and failing runs.
- Avoid generating fixes specific to a particular test.

Empirically, non-invariant properties tend to be filtered out easily.

In our experiment, the per-fault average number of passing and failing test cases is 9 and 6.5.

Synthesizing fixes

Assumptions:

1. State invariant difference is the cause of the failure.
2. Minimizing the difference before system fails should bring the system back to a normal configuration.

Synthesis steps:

1. generate method calls to minimize state invariant difference using object behavioral model.
2. Arrange generated method calls in fix schema.

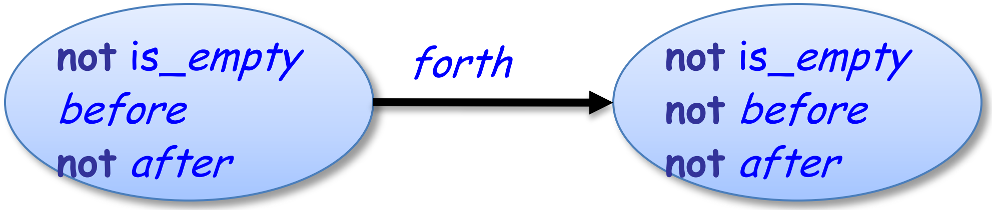
Object behavioral model

The model suggests ways to change a state property:

calling *forth* can change *before* from true to false.

Object behavioral model is a set of transitions:

the starting and ending points are abstract states;
the label is a method.



not *is_empty*
before
not *after*

forth

not *is_empty*
not *before*
not *after*

All the transitions are observed in random tests.

Fix schema

Fix schema capture common fixing styles. For a fault, different schema are tried.

The schema used in the running example:

```
if failing_condition then
        snippet
else
        original statements
end
```

If the failure is going to happen, snippet brings the system back to normal.

Otherwise, invoke original statements to preserve normal behavior.

Instantiating an actual fix from schema

```
if  failing_condition  then
        snippet
else
        original statements
end
```

Fix schema

```
if  before  then
        forth
else
        Result.put_left (item)
        forth
        counter := counter + 1
end
```

Actual fix

Validating candidate fixes

Run the patched program against both passing and failing tests, requiring:

- Passing tests still pass.
- Failing tests now pass.

Ranking valid fixes statically and dynamically

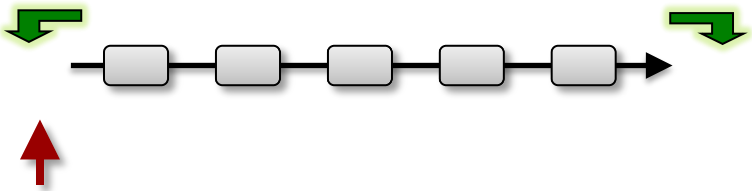
- Static metrics favors:
  - simple textual changes
  - changes close to the failing location
  - changes involving less original statements
- Dynamic metric favors behavioral preservation:

  Passing tests should end with similar resulting abstract states.

Human solutions vs. tool solutions

- Sent 3 faults to 2 professional Eiffel programmers.
- In 4 out of 6 cases, the reported fixes are the same as automated proposed ones.

What's the difference then?



Has better run-time Performance.

start seems more related to the concept *before*.

```
duplicate (n: INTEGER ):…
      new_chain
    Res  .forth
    from
        if before then start end
    until  (counter = n) or after  loop
        Result.put_left (item )
        forth
        counter := counter + 1
     end
    go_to  (pos )
  end
```

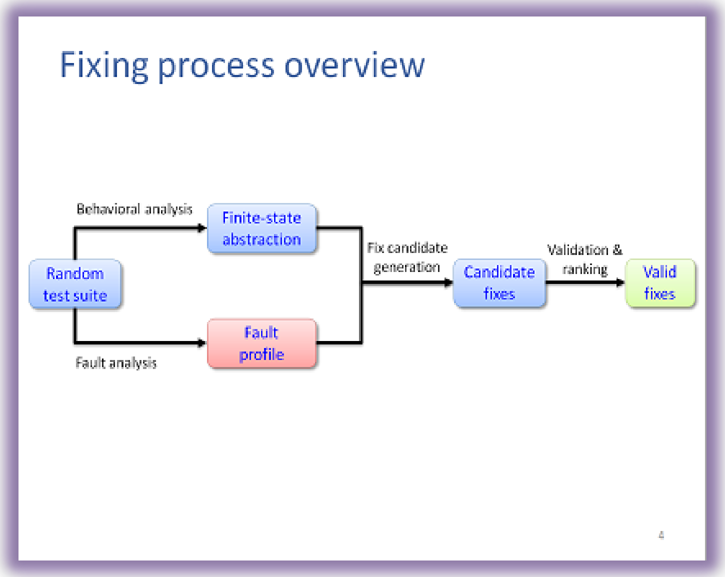
```
duplicate (n: INTEGER ): …
  do
    pos  := cursor
    Result := new_chain
    Result.forth

    from until  (counter = n ) or after  loop
        if before then
            forth
        else
            Result.put_left (item )
            forth
            counter := counter + 1
        end
    end
    go_to  (pos )
  end
```

Human solution

Tool solution

Summary



Fixing process overview

Random test suite

Behavioral analysis

Finite-state abstraction

Fault analysis

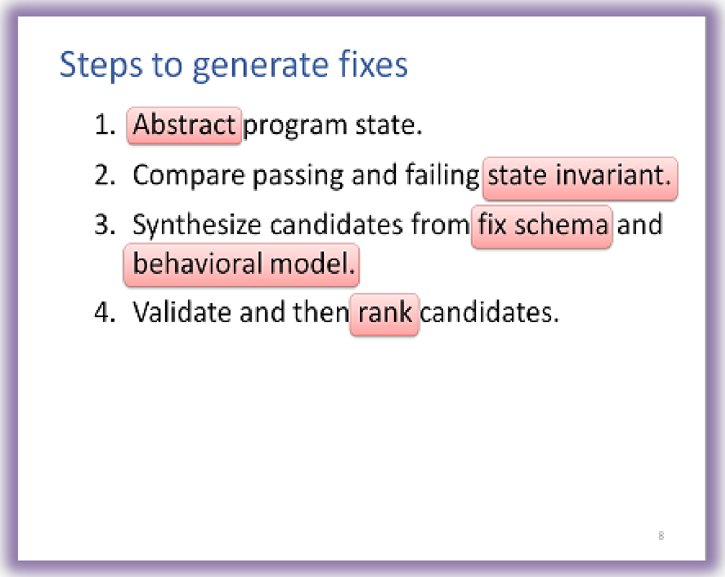
Fault profile

Fix candidate generation

Candidate fixes

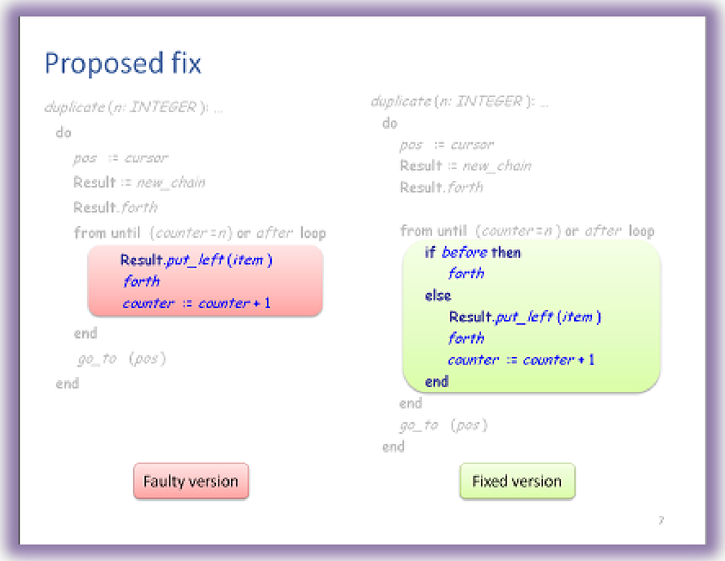
Validation & ranking

Valid fixes



Steps to generate fixes

1. Abstract program state.
2. Compare passing and failing state invariant.
3. Synthesize candidates from fix schema and behavioral model.
4. Validate and then rank candidates.



Proposed fix

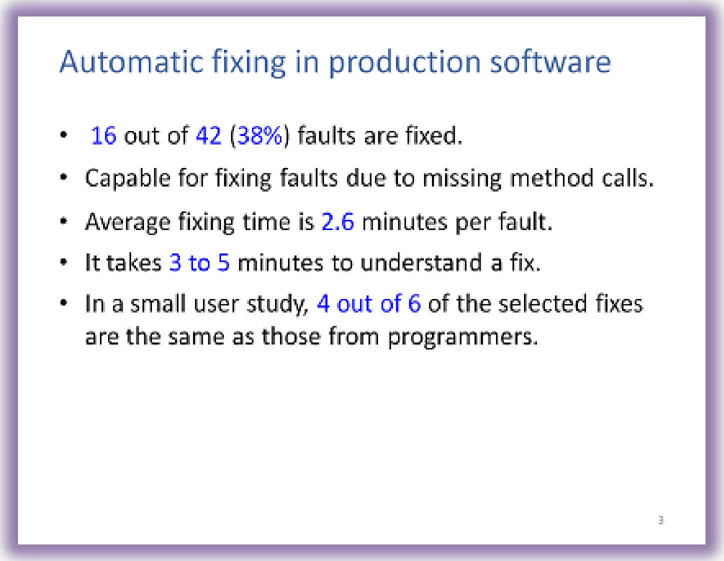
```
duplicate (n: INTEGER ): ...
  do
    pos  := cursor
    Result := new_chain
    Result.forth
    from until (counter=n) or after loop
        Result.put_left (item )
        forth
        counter := counter + 1
    end
    go_to (pos )
  end
```

Faulty version

```
duplicate (n: INTEGER ): ...
  do
    pos  := cursor
    Result := new_chain
    Result.forth

    from until (counter=n ) or after loop
      if before then
        forth
      else
        Result.put_left (item )
        forth
        counter := counter + 1
      end
    end
    go_to (pos )
  end
```

Fixed version



Automatic fixing in production software

- 16 out of 42 (38%) faults are fixed.
- Capable for fixing faults due to missing method calls.
- Average fixing time is 2.6 minutes per fault.
- It takes 3 to 5 minutes to understand a fix.
- In a small user study, 4 out of 6 of the selected fixes are the same as those from programmers.