Università
della
Svizzera
italiana

Dynamic analysis

Software Analysis
Topic 8

Carlo A. Furia
USI – Università della Svizzera Italiana

Today's menu

Test case generation

Input simplification

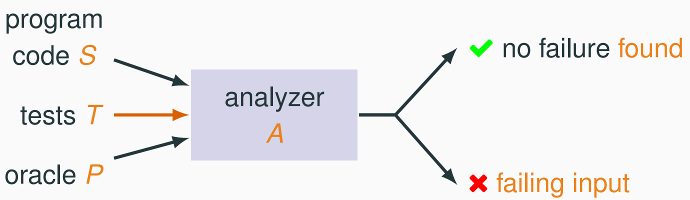
Program simplification

Fault localization

Dynamic assertion checking

Case studies: putting it all together

Dynamic analysis: the very idea



Dynamic analysis:

- analyzes real program code
- is fully automatic, as it is based on executing concrete code
- properties are encoded as executable oracles, such as expected outputs, assertions, and reference implementations
- is unsound because it only analyzes a finite set of inputs T
- is complete because every failure comes with a concrete input that triggers it

Analysis, dynamically

Static:

- without executing the software
- on generic/abstract inputs
- based on symbolic constraints
- typically sound and incomplete

Dynamic:

- while executing the software
- on specific/concrete inputs
- based on concrete states
- typically unsound and complete

Analysis, dynamically

Static:

- without executing the software
- on generic/abstract inputs
- based on symbolic constraints
- typically sound and incomplete

Dynamic:

- while executing the software
- on specific/concrete inputs
- based on concrete states
- typically unsound and complete

"Dynamic analysis" is any software analysis technique that targets program behavior on a finite set of concrete inputs.

Analysis, dynamically

Static:

- without executing the software
- on generic/abstract inputs
- based on symbolic constraints
- typically sound and incomplete

Dynamic:

- while executing the software
- on specific/concrete inputs
- based on concrete states
- typically unsound and complete

"Dynamic analysis" is any software analysis technique that targets program behavior on a finite set of concrete inputs.

In other words, it summarizes a program's behavior on sample inputs.

Analysis, dynamically

Static:

- without executing the software
- on generic/abstract inputs
- based on symbolic constraints
- typically sound and incomplete

Dynamic:

- while executing the software
- on specific/concrete inputs
- based on concrete states
- typically unsound and complete

"Dynamic analysis" is any software analysis technique that targets program behavior on a finite set of concrete inputs.

In other words, it summarizes a program's behavior on sample inputs.

STATIC → DYNAMIC

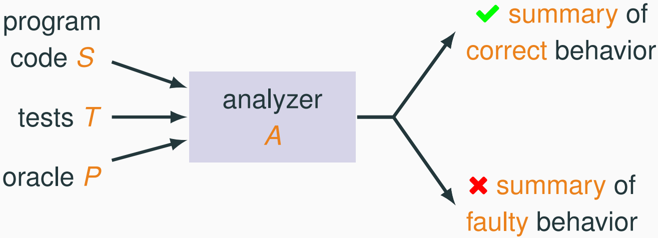
static analysis | deductive verification | software model checking | model checking | symbolic execution | dynamic analysis

Debugging

"Dynamic analysis" is any software analysis technique that summarizes a program's behavior on sample inputs.

Debugging

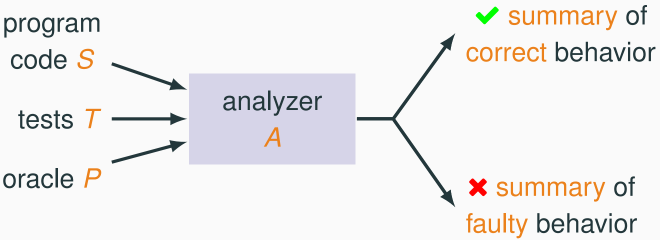
"Dynamic analysis" is any software analysis technique that summarizes a program's behavior on sample inputs.



Dynamic analysis's output is often used to support debugging or further analysis of the software.

Test case generation

Where do tests come from?

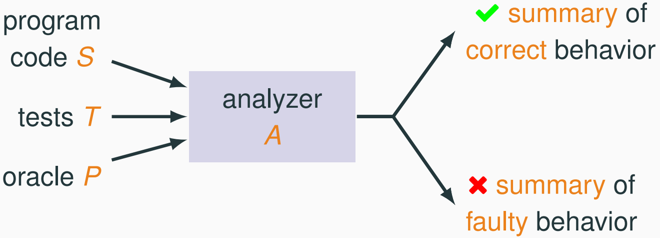


Dynamic analysis needs tests (test cases) to run the program on.

Tests may be written manually and provided as input to the analysis.

An alternative is developing test-case generation techniques, which then can be used to bootstrap dynamic analysis.

Where do tests come from?



Dynamic analysis needs tests (test cases) to run the program on.

Tests may be written manually and provided as input to the analysis.

An alternative is developing test-case generation techniques, which then can be used to bootstrap dynamic analysis.

A detailed description of strategies to generate tests is outside the scope of this course. However, we give a brief overview of the main automatic test-generation approaches.

What’s the goal of testing?

The goal of testing is

What's the goal of testing?

The goal of testing is making programs fail.

What's the goal of testing?

The main goal of testing is making programs fail.

What's the goal of testing?

The main goal of testing is making programs fail.

Other goals of testing:

- exercise different parts of each program (demonstrating correct and faulty behavior)
- validate program changes
- ensure that bugs introduced in the past do not happen again (regression testing)

What's the goal of testing?

The main goal of testing is making programs fail.

Other goals of testing:

- exercise different parts of each program (demonstrating correct and faulty behavior)
- validate program changes
- ensure that bugs introduced in the past do not happen again (regression testing)

For debugging, we typically need:

- succinct tests that trigger a failure
- other tests that do not trigger a failure but are similar to the failure-inducing ones

What's the goal of testing?

The main goal of testing is making programs fail.

Other goals of testing:

- exercise different parts of each program (demonstrating correct and faulty behavior)
- validate program changes
- ensure that bugs introduced in the past do not happen again (regression testing)

For debugging, we typically need:

- succinct tests that trigger a failure
- other tests that do not trigger a failure but are similar to the failure-inducing ones

We focus on unit testing (testing of units of software in isolation), even though other kinds of testing are also relevant to debugging.

Generating tests automatically

Main families of dynamic techniques to generate tests:

combinatorial testing systematically enumerates all possible inputs following some conventional order (for example, from smaller to larger)

random testing picks inputs at random among all possible valid ones

search-based testing explores the space of valid input looking for those that improve some metrics (for example, coverage, diversity, failure inducing capabilities, . . .)

What a test is made of

A test case is essentially an input that we can run our program on.

What a test is made of

A test case is essentially an input that we can run our program on.

In imperative programs, the input consists of:

1. the input proper – the actual arguments of a method or procedure, or the user input
2. the program state where execution starts – the object state, or the value of global variables

What a test is made of

A test case is essentially an input that we can run our program on.

In imperative programs, the input consists of:

1. the input proper – the actual arguments of a method or procedure, or the user input
2. the program state where execution starts – the object state, or the value of global variables

Therefore a test case (for imperative programs) normally consists of $2 + 1$ phases:

1. setup: bring the program to the intended initial state
2. execution: run the program on the actual input
3. teardown: record the output, the final state, and any failure that may have occurred

Testing examples

We demonstrate the various test generation techniques on two simple examples:

A Java procedure that sums all elements in an array:

```java
// sum of all values in 'a'
static int sum(int[] a)
```

A Java class with the following interface:

```java
class List<T>
{
   // create empty list
   List();
   // append 'e' to end
   void add(T e);
   // remove kth element
   void remove(int k);
}
```

Combinatorial testing: procedures

Combinatorial testing generates all possible inputs – normally up to a certain bound since exhaustive enumeration is not feasible.

```
# generate all inputs up to size 'max_size'
def combinatorial_tests(max_size):
   tests = ["null"]
   for n in 0 |to| max_size:
      tests += generate(n)
   return tests

## generate all arrays with 'size' int elements
def generate(size):
  tests = []
  if size == 0:
    tests += [[]]  # empty array
  else:
    for t in generate(size - 1):
      for v in MIN_INT |to| MAX_INT:  # MIN_INT .. MAX_INT
         tests += [t + [v]]  # add one element to each t
  return tests
```

Generating tests for:

```
// sum of all values in 'a'
static int sum(int[] a)
```

Combinatorial testing: classes

When testing a class, combinatorial testing can generates all possible sequences of method calls.

```
# generate all call sequences up to 'max_len'
def combinatorial_objects(max_len):
   tests = []
   for n in 0 |to| max_len:
      tests += calls(n)
   return tests

## generate all sequences of 'n' calls
def calls(n):
  tests = []
  if size == 0:
    tests += ["new List<T>()"]  # new object
  else:
    for t in calls(n - 1):
      for m in {"add", "remove"}:
         for a in generate(domain(m)):  # all possible arguments
            tests += [t.m(a)]           # add one call to sequence t
  return tests
```

Generating tests for:

```
class List<T>
{
   // create empty list
   List();
   // append 'e' to end
   void add(T e);
   // remove kth element
   void remove(int k);
}
```

Random testing: procedures

Random testing just picks some inputs at random.

```
# generate a random array of size up to ‘max_size’
def random_array(max_size):
   # random number in 0 .. max_size
   size = random_int(0 |to| max_size)
   t = []
   for k in 0 |to| size:
              # random element
      e = random_int(MIN_INT |to| MAX_INT)
      t += [e]   # add to array
   return t

# generate ‘num’ random arrays of size up to ‘max_size’
def random_tests(num, max_size):
   tests = []
   for n in 1 |to| num:
      tests += [random_array(max_size)]
   return tests
```

Generating tests for:

```
// sum of all values in ‘a’
static int sum(int[] a)
```

Random testing: classes

When testing a class, random testing maintains a pool of random objects, which mutates to get new ones.

```
# generate a set of ‘num’ random objects
def random_objects(num):
   pool = {"new List<T>()"}
   while |pool| < num:
      # random object from pool (cloned)
      o = pick_random(pool).clone()
      # random method to run
      m = pick_random({"add", "remove"})
      # random argument of method
      a = random_argument(m)
      # add new object to pool (if not already there)
      pool += { o.m(a) }
   return pool
```

Generating tests for:

```
class List<T>
{
   // create empty list
   List();
   // append ‘e’ to end
   void add(T e);
   // remove kth element
   void remove(int k);
}
```

Search-based testing

Search-based testing may generate inputs that try to maximize the branch coverage of a procedure's implementation.

For example, genetic algorithms are a kind of search-based algorithm that build new tests by mutating and combining existing ones.

```
# generate arrays that achieve given branch 'coverage'
def search_tests(coverage):
  tests = []
  while coverage(tests) < coverage:
    new_tests = []
    for t in tests:
      m = mutate(t)  # mutation of t
      # keep the mutation if it improves coverage
      if coverage([tests, m]) > coverage (tests):
        new_tests += [m]
    # add new tests that have been retained
    tests += new_tests
  return tests
```

Generating tests for:

```
// sum of all values in 'a'
static int sum(int[] a)
```

When testing a class, search-based testing maintains a pool of objects, which it extends with mutants that improve the metric.

Search-based testing

Search-based testing may generate inputs that try to maximize the branch coverage of a procedure's implementation.

For example, genetic algorithms are a kind of search-based algorithm that build new tests by mutating and combining existing ones.

```
# generate arrays that achieve given branch 'coverage'
def search_tests(coverage):
    tests = []
    while coverage(tests) < coverage:
        new_tests = []
        for t in tests:
            m = mutate(t)  # mutation of t
            # keep the mutation if it improves coverage
            if coverage([tests, m]) > coverage (tests):
                new_tests += [m]
        # add new tests that have been retained
        tests += new_tests
    return tests
```

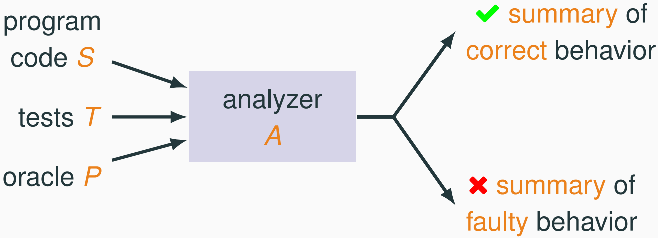
Generating tests for:

```
// sum of all values in 'a'
static int sum(int[] a)
```

called "fitness function" in genetic algorithms

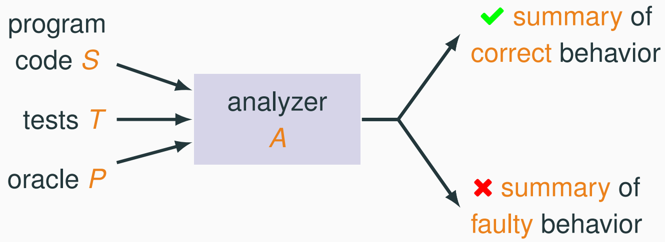
When testing a class, search-based testing maintains a pool of objects, which it extends with mutants that improve the metric.

Oracles



Oracles are a form of specification and hence they have to encode the intended program behavior.

Oracles



Oracles are a form of specification and hence they have to encode the intended program behavior.

Oracle generation may be automated in some simpler cases:

crashing oracle: check that a program runs without crashing (including throwing uncaught exceptions)

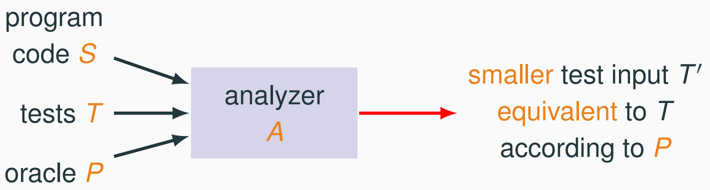
overflow oracle: when executing an arithmetic operation, fail if an overflow occurs

equivalence oracle: check that the program under test returns the same output as a reference implementation

Input simplification

Input simplification for debugging

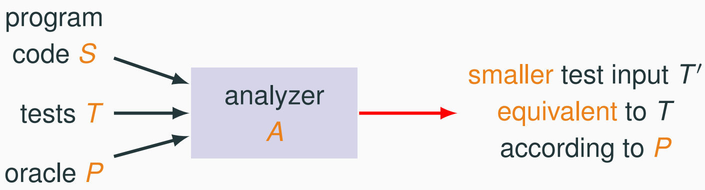
Finding the source of a bug is easier if we only have to analyze a small input that still triggers the fault.



We describe delta debugging: a technique to shrink some input (in the form of tests) in a way that preserves its behavior (according to the oracle).

Input simplification for debugging

Finding the source of a bug is easier if we only have to analyze a small input that still triggers the fault.



We describe delta debugging: a technique to shrink some input (in the form of tests) in a way that preserves its behavior (according to the oracle).

The delta debugging algorithm is often referred to as "minimizing" even though it is not guaranteed to minimize the input: it is a greedy search that may only find a local minimum.

Delta debugging: high-level overview

The basic idea of delta debugging is to perform a binary search on partitions of the input.

How the input can be split depends on what kind of data it represents:

- if the input is a string (for example, an HTML page), we can split it at every character
- if the input is a list, we can split out a slice of consecutive elements
- if the input is a tree, we can split it into subtrees
- ...

Delta debugging is applicable as long as there is some meaningful way of splitting the input into chunks.

Delta debugging: algorithm

```
# shrink 'input' without changing behavior with respect to 'oracle'
def shrink(input, oracle, n):
    if size(input) == 1:
        # 'input' cannot be split into smaller chunks
        return input
    else:
        # split into n chunks
        chunks = split(input, n)
        for chunk in chunks:
            # consider 'input' without 'chunk'
            shrunk_input = input - chunk
            # if behavior of 'shrunk_input' same as 'input'
            if oracle(shrunk_input) == oracle(input):
                # try to further shrink 'shrunk_input'
                return shrink(shrunk_input, oracle, max(n - 1, 2))
        # none of the shrunk inputs is equivalent to 'input'
        if n < size(input):
            # try shrinking into smaller chunks
            return shrink(input, oracle, min(2*n, size(input)))
        else:
            # smallest chunk size reached: stop
            return input
```

Delta debugging: algorithm

The main algorithm `dd(input, oracle)` is a a shorthand for `shrink(input, oracle, 2)`

```
# shrink 'input' without changing behavior with respect to 'oracle'
def shrink(input, oracle, n):
   if size(input) == 1:
      # 'input' cannot be split into smaller chunks
      return input
   else:
      # split into n chunks
      chunks = split(input, n)
      for chunk in chunks:
         # consider 'input' without 'chunk'
         shrunk_input = input - chunk
         # if behavior of 'shrunk_input' same as 'input'
         if oracle(shrunk_input) == oracle(input):
            # try to further shrink 'shrunk_input'
            return shrink(shrunk_input, oracle, max(n - 1, 2))
      # none of the shrunk inputs is equivalent to 'input'
      if n < size(input):
         # try shrinking into smaller chunks
         return shrink(input, oracle, min(2*n, size(input)))
      else:
         # smallest chunk size reached: stop
         return input
```

Delta debugging: algorithm

```
# shrink 'input' without changing behavior with respect to 'oracle'
def shrink(input, oracle, n):
  if size(input) == 1:
    # 'input' cannot be split into smaller chunks
    return input
  else:
    # split into n chunks
    chunks = split(input, n)
    for chunk in chunks:
      # consider 'input' without 'chunk'
      shrunk_input = input - chunk
      # if behavior of 'shrunk_input' same as 'input'
      if oracle(shrunk_input) == oracle(input):
        # try to further shrink 'shrunk_input'
        return shrink(shrunk_input, oracle, max(n - 1, 2))
    # none of the shrunk inputs is equivalent to 'input'
    if n < size(input):
      # try shrinking into smaller chunks
      return shrink(input, oracle, min(2*n, size(input)))
    else:
      # smallest chunk size reached: stop
      return input
```

The main algorithm `dd(input, oracle)` is a a shorthand for `shrink(input, oracle, 2)`

they both fail in the same way (triggering the same failure)

halve chunk size (as in binary search)

Using delta debugging for debugging

The Python following implementation of insertion sort is buggy.

```python
def insertion_sort(lst):
    """Return a sorted copy of LST"""
    if len(lst) <= 1:
        return lst
    head = lst[0]
    tail = lst[1:]
    return insert(head, insertion_sort(tail))

def insert(elem, lst):
    """Return a copy of LST with ELEM sorted in"""
    if len(lst) == 0:
        return [elem]
    head = lst[0]
    tail = lst[1:]
    if elem <= head:
        return lst + [elem]
    return [head] + insert(elem, tail)
```

Initial failure:

`insertion_sort(t)`, where `t = [5, 4, 2, 3, 1]`, returns `[1, 3, 2, 4, 5]`.

Using delta debugging for debugging

The Python following implementation of insertion sort is buggy.

```python
def insertion_sort(lst):
    """Return a sorted copy of LST"""
    if len(lst) <= 1:
        return lst
    head = lst[0]
    tail = lst[1:]
    return insert(head, insertion_sort(tail))

def insert(elem, lst):
    """Return a copy of LST with ELEM sorted in"""
    if len(lst) == 0:
        return [elem]
    head = lst[0]
    tail = lst[1:]
    if elem <= head:
        return lst + [elem]
    return [head] + insert(elem, tail)
```

Initial failure:
`insertion_sort(t)`, where `t = [5, 4, 2, 3, 1]`, returns `[1, 3, 2, 4, 5]`.

Delta debugging shrinks the input to `t_min = [2, 3]`, which is incorrectly sorted to `[3, 2]`.

Using delta debugging for debugging

The Python following implementation of insertion sort is buggy.

```python
def insertion_sort(lst):
    """Return a sorted copy of LST"""
    if len(lst) <= 1:
        return lst
    head = lst[0]
    tail = lst[1:]
    return insert(head, insertion_sort(tail))

def insert(elem, lst):
    """Return a copy of LST with ELEM sorted in"""
    if len(lst) == 0:
        return [elem]
    head = lst[0]
    tail = lst[1:]
    if elem <= head:
        return lst + [elem]
    return [head] + insert(elem, tail)
```

Initial failure: `insertion_sort(t)`, where `t = [5, 4, 2, 3, 1]`, returns `[1, 3, 2, 4, 5]`.

Delta debugging shrinks the input to `t_min = [2, 3]`, which is incorrectly sorted to `[3, 2]`.

Tracing the execution of `insertion_sort` on the smaller `t_min` it is easier to see that the bug is due to the incorrect insertion of `elem` in the back instead of in front.

Delta debugging: example

Let's see how the delta debugging algorithm shrinks the input `[5, 4, 2, 3, 1]` into a smaller one that still makes `insertion_sort` fail.

INPUT	n	REMOVED CHUNKS
`[5, 4, 2, 3, 1]`	2	

Delta debugging: example

Let's see how the delta debugging algorithm shrinks the input `[5, 4, 2, 3, 1]` into a smaller one that still makes `insertion_sort` fail.

INPUT	n	REMOVED CHUNKS	
`[5, 4, 2, 3, 1]`	2	`[5, 4, 2]` ✔	`[3, 1]` ✔

Delta debugging: example

Let's see how the delta debugging algorithm shrinks the input `[5, 4, 2, 3, 1]` into a smaller one that still makes `insertion_sort` fail.

input without chunk `[5, 4, 2]`
passes (is sorted correctly)

INPUT	n	REMOVED CHUNKS	
`[5, 4, 2, 3, 1]`	2	`[5, 4, 2]` ✔	`[3, 1]` ✔

Delta debugging: example

Let's see how the delta debugging algorithm shrinks the input `[5, 4, 2, 3, 1]` into a smaller one that still makes `insertion_sort` fail.

input without chunk `[5, 4, 2]` passes (is sorted correctly)

INPUT	n	REMOVED CHUNKS	
`[5, 4, 2, 3, 1]`	2	`[5, 4, 2]` ✔	`[3, 1]` ✔
`[5, 4, 2, 3, 1]`	4		

Delta debugging: example

Let's see how the delta debugging algorithm shrinks the input `[5, 4, 2, 3, 1]` into a smaller one that still makes `insertion_sort` fail.

input without chunk `[5, 4, 2]` passes (is sorted correctly)

input without chunk `[4]` fails (is sorted incorrectly)

INPUT	n	REMOVED CHUNKS			
`[5, 4, 2, 3, 1]`	2	`[5, 4, 2]` ✔	`[3, 1]` ✔		
`[5, 4, 2, 3, 1]`	4	`[5]` ✖	`[4]` ✖	`[2, 3]` ✔	`[1]` ✖

Delta debugging: example

Let's see how the delta debugging algorithm shrinks the input `[5, 4, 2, 3, 1]` into a smaller one that still makes `insertion_sort` fail.

input without chunk `[5, 4, 2]` passes (is sorted correctly)

input without chunk `[4]` fails (is sorted incorrectly)

INPUT	n	REMOVED CHUNKS			
`[5, 4, 2, 3, 1]`	2	`[5, 4, 2]` ✔	`[3, 1]` ✔		
`[5, 4, 2, 3, 1]`	4	`[5]` ✘	`[4]` ✘	`[2, 3]` ✔	`[1]` ✘
`[4, 2, 3, 1]`	3				

Delta debugging: example

Let's see how the delta debugging algorithm shrinks the input `[5, 4, 2, 3, 1]` into a smaller one that still makes `insertion_sort` fail.

input without chunk `[5, 4, 2]` passes (is sorted correctly)

input without chunk `[4]` fails (is sorted incorrectly)

INPUT	n	REMOVED CHUNKS			
`[5, 4, 2, 3, 1]`	2	`[5, 4, 2]` ✓	`[3, 1]` ✓		
`[5, 4, 2, 3, 1]`	4	`[5]` ✗	`[4]` ✗	`[2, 3]` ✓	`[1]` ✗
`[4, 2, 3, 1]`	3	`[4]` ✗	`[2, 3]` ✓	`[1]` ✗	

Delta debugging: example

Let's see how the delta debugging algorithm shrinks the input `[5, 4, 2, 3, 1]` into a smaller one that still makes `insertion_sort` fail.

input without chunk `[5, 4, 2]` passes (is sorted correctly)

input without chunk `[4]` fails (is sorted incorrectly)

INPUT	n	REMOVED CHUNKS			
`[5, 4, 2, 3, 1]`	2	`[5, 4, 2]` ✔	`[3, 1]` ✔		
`[5, 4, 2, 3, 1]`	4	`[5]` ✘	`[4]` ✘	`[2, 3]` ✔	`[1]` ✘
`[4, 2, 3, 1]`	3	`[4]` ✘	`[2, 3]` ✔	`[1]` ✘	
`[2, 3, 1]`	2				

Delta debugging: example

Let's see how the delta debugging algorithm shrinks the input `[5, 4, 2, 3, 1]` into a smaller one that still makes `insertion_sort` fail.

input without chunk `[5, 4, 2]` passes (is sorted correctly)

input without chunk `[4]` fails (is sorted incorrectly)

INPUT	n	REMOVED CHUNKS			
`[5, 4, 2, 3, 1]`	2	`[5, 4, 2]` ✓	`[3, 1]` ✓		
`[5, 4, 2, 3, 1]`	4	`[5]` ✗	`[4]` ✗	`[2, 3]` ✓	`[1]` ✗
`[4, 2, 3, 1]`	3	`[4]` ✗	`[2, 3]` ✓	`[1]` ✗	
`[2, 3, 1]`	2	`[2, 3]` ✓	`[1]` ✗		

Delta debugging: example

Let's see how the delta debugging algorithm shrinks the input `[5, 4, 2, 3, 1]` into a smaller one that still makes `insertion_sort` fail.

input without chunk `[5, 4, 2]` passes (is sorted correctly)

input without chunk `[4]` fails (is sorted incorrectly)

INPUT	n	REMOVED CHUNKS			
`[5, 4, 2, 3, 1]`	2	`[5, 4, 2]` ✓	`[3, 1]` ✓		
`[5, 4, 2, 3, 1]`	4	`[5]` ✗	`[4]` ✗	`[2, 3]` ✓	`[1]` ✗
`[4, 2, 3, 1]`	3	`[4]` ✗	`[2, 3]` ✓	`[1]` ✗	
`[2, 3, 1]`	2	`[2, 3]` ✓	`[1]` ✗		
`[2, 3]`	2				

Delta debugging: example

Let's see how the delta debugging algorithm shrinks the input `[5, 4, 2, 3, 1]` into a smaller one that still makes `insertion_sort` fail.

input without chunk `[5, 4, 2]` passes (is sorted correctly)

input without chunk `[4]` fails (is sorted incorrectly)

INPUT	n	REMOVED CHUNKS			
`[5, 4, 2, 3, 1]`	2	`[5, 4, 2]` ✓	`[3, 1]` ✓		
`[5, 4, 2, 3, 1]`	4	`[5]` ✗	`[4]` ✗	`[2, 3]` ✓	`[1]` ✗
`[4, 2, 3, 1]`	3	`[4]` ✗	`[2, 3]` ✓	`[1]` ✗	
`[2, 3, 1]`	2	`[2, 3]` ✓	`[1]` ✗		
`[2, 3]`	2	`[2]` ✓	`[3]` ✓		

Delta debugging: example

Let's see how the delta debugging algorithm shrinks the input `[5, 4, 2, 3, 1]` into a smaller one that still makes `insertion_sort` fail.

input without chunk `[5, 4, 2]` passes (is sorted correctly)

input without chunk `[4]` fails (is sorted incorrectly)

INPUT	n	REMOVED CHUNKS			
`[5, 4, 2, 3, 1]`	2	`[5, 4, 2]` ✔	`[3, 1]` ✔		
`[5, 4, 2, 3, 1]`	4	`[5]` ✘	`[4]` ✘	`[2, 3]` ✔	`[1]` ✘
`[4, 2, 3, 1]`	3	`[4]` ✘	`[2, 3]` ✔	`[1]` ✘	
`[2, 3, 1]`	2	`[2, 3]` ✔	`[1]` ✘		
`[2, 3]`	2	`[2]` ✔	`[3]` ✔		
`[2, 3]`		done: $2 =$ `size([2, 3])`			

Delta debugging on realistic examples

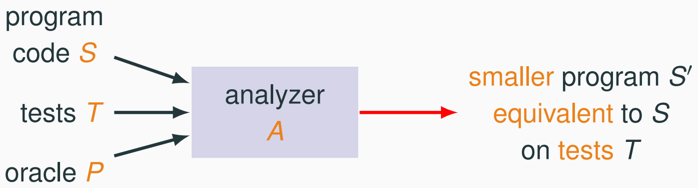
Thanks to its fast search through possible ways of splitting the input, delta debugging is applicable to large inputs with complex oracles.

- Delta debugging shrinks an 896-line HTML page that crashes a version of Mozilla's rendering engine down to the single line: `<SELECT NAME="priority" MULTIPLE SIZE=7>`, and then further down to the single tag `<SELECT>`
- Delta debugging can shrink long sequences of recorded interactive user input (for example in a GUI) down to a small combination of events that triggers a failure
- Delta debugging can shrink randomly generated tests (for example, input files to Unix command line utilities) down to small ones that go right to the point of failure

Program simplification

Program simplification for debugging

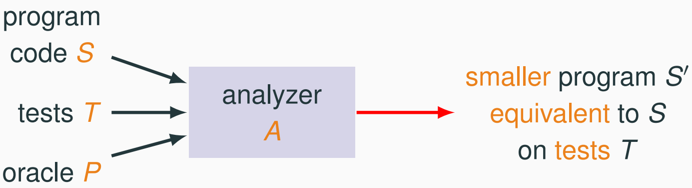
Finding the source of a bug is easier if we only have to analyze a small program that still is faulty.



We describe dynamic slicing: a technique to shrink some program in a way that preserves its behavior on a given set T of tests (often a single failing test).

Program simplification for debugging

Finding the source of a bug is easier if we only have to analyze a small program that still is faulty.



We describe dynamic slicing: a technique to shrink some program in a way that preserves its behavior on a given set T of tests (often a single failing test).

Dynamic slicing follows the same general approach as static slicing, but is only concerned with preserving the behavior on concrete inputs T – as opposed to a generic input. As a result, dynamic slices are often smaller, and hence better support debugging.

Slicing: static vs. dynamic

The program slice of program P according to slicing criterion ℓ (where ℓ is a location in P) is a subset of all statements in P that:

static slicing: may affect values of variables at ℓ

dynamic slicing: affects values of variables at ℓ when P runs on T

Slicing: static vs. dynamic

The program slice of program P according to slicing criterion ℓ (where ℓ is a location in P) is a subset of all statements in P that:

static slicing: may affect values of variables at ℓ

dynamic slicing: affects values of variables at ℓ when P runs on T

Procedure in Lithium:

```
procedure proc(n, a: Integer):
  var s, x, b, k: Integer
  x := 1
  b := a + x
  a := a + 1
  k := 1
  s := 0
  while k ≤ n
    if b > 0
      if a > 1
        x := 2
    s := s + x
    k := k + 1
  print(s)
```

Slicing: static vs. dynamic

The program slice of program P according to slicing criterion ℓ (where ℓ is a location in P) is a subset of all statements in P that:

static slicing: may affect values of variables at ℓ

dynamic slicing: affects values of variables at ℓ when P runs on T

Procedure in Lithium:

```
1  procedure proc(n, a: Integer):
2    var s, x, b, k: Integer
3    x := 1
4    b := a + x
5    a := a + 1
6    k := 1
7    s := 0
8    while k ≤ n
9      if b > 0
10       if a > 1
11         x := 2
12     s := s + x
13     k := k + 1
14   print(s)
```

Static slice of `proc` according to criterion 14: the whole `proc`.

Slicing: static vs. dynamic

The program slice of program P according to slicing criterion ℓ (where ℓ is a location in P) is a subset of all statements in P that:

static slicing: may affect values of variables at ℓ

dynamic slicing: affects values of variables at ℓ when P runs on T

Procedure in Lithium:

```
1  procedure proc(n, a: Integer):
2    var s, x, b, k: Integer
3    x := 1
4    b := a + x
5    a := a + 1
6    k := 1
7    s := 0
8    while k ≤ n
9      if b > 0
10       if a > 1
11         x := 2
12     s := s + x
13     k := k + 1
14   print(s)
```

Static slice of `proc` according to criterion 14: the whole `proc`.

Dynamic slice of `proc` according to criterion 14 and input T $n = 2, a = 0$:

```
procedure slice_proc(n: Integer):
  var s, x, k: Integer
  x := 1
  k := 1
  s := 0
  while k ≤ n
    s := s + x
    k := k + 1
  print(s)
```

Data and control dependencies

Just like static slicing, dynamic slicing follows the data and control dependencies – but only in a specific run.

Data and control dependencies

Just like static slicing, dynamic slicing follows the data and control dependencies – but only in a specific run.

First, we collect the data and control dependencies of each statement:

- $WD(\ell)$**:** set of variables written by the statement at line ℓ
- $RD(\ell)$**:** set of variables read (used) by the statement at line ℓ
- $WC(\ell)$**:** ℓ if ℓ is a branch, nothing otherwise
- $RC(\ell)$**:** closest location (on the CFG) of a branch whose outcome determines whether ℓ executes

Data and control dependencies

Just like static slicing, dynamic slicing follows the data and control dependencies – but only in a specific run.

First, we collect the data and control dependencies of each statement:

data dependencies

$WD(\ell)$: set of variables written by the statement at line ℓ

$RD(\ell)$: set of variables read (used) by the statement at line ℓ

$WC(\ell)$: ℓ if ℓ is a branch, nothing otherwise

$RC(\ell)$: closest location (on the CFG) of a branch whose outcome determines whether ℓ executes

control dependencies

Dependencies: example

ℓ	STATEMENT	*WD*	*RD*	*WC*	*RC*
1	`proc(n, a)`	n, a			
3	`x := 1`	x			
4	`b := a + x`	b	a, x		
5	`a := a + 1`	a	a		
6	`k := 1`	k			
7	`s := 0`	s			
8	**while** `k` $\leq$ `n`		k, n	8	
9	**if** `b` $>$ `0`		b	9	8
10	**if** `a` $>$ `1`		a	10	9
11	`x := 2`	x			10
12	`s := s + x`	s	s, x		8
13	`k := k + 1`	k	k		8
14	`print(s)`		s		8

Traces

Since a dynamic slice replicates program behavior only for a specific input, we log the trace of statements executed when running tests T.

In the running example, the only test we consider is $\mathtt{n} = 2, \mathtt{a} = 0$.

Traces: example

n	STATEMENT	n	a	x	b	k	s
1	`proc(n, a)`	2	0				
2	`x := 1`	2	0	1			
3	`b := a + x`	2	0	1	1		
4	`a := a + 1`	2	1	1	1		
5	`k := 1`	2	1	1	1	1	
6	`s := 0`	2	1	1	1	1	0
7	**while** k $\leq$ n	2	1	1	1	1	0
8	**if** b $>$ 0	2	1	1	1	1	0
9	**if** a $>$ 1	2	1	1	1	1	0
10	`s := s + x`	2	1	1	1	1	1
11	`k := k + 1`	2	1	1	1	2	1
12	**while** k $\leq$ n	2	1	1	1	2	1
13	**if** b $>$ 0	2	1	1	1	2	1
14	**if** a $>$ 1	2	1	1	1	2	1
15	`s := s + x`	2	1	1	1	2	2
16	`k := k + 1`	2	1	1	1	3	2
17	**while** k $\leq$ n	2	1	1	1	3	2
18	`print(s)`	2	1	1	1	3	2

Dynamic slicing: algorithm

We build the dynamic slice by working forward on the recorded trace.

Dynamic slicing: algorithm

We build the dynamic slice by working forward on the recorded trace.

$\mathcal{DS}(n)$ denotes the dynamic slice at step n in a trace:

- for every variable v read at n (that is, in $RD(n)$), $\mathcal{DS}(n)$ includes:
  1. the location $\ell(p_d)$ of the statement executed at step p_d
  2. transitively, the dynamic slice $\mathcal{DS}(p_d)$ at step p_d

  where p_d is the step when v was written most recently before n in the trace
- for every branching statement c controlling n (that is, in $RC(n)$), $\mathcal{DS}(n)$ includes:
  1. the location $\ell(p_c)$ of the statement executed at step p_c
  2. transitively, the dynamic slice $\mathcal{DS}(p_c)$ at step p_c

  where p_c is the most recent step before n in the trace with a branch statement whose outcome led to c executing

$$\mathcal{DS}(n) = \bigcup_{v \in RD(n)} latest(n, v, WD) \cup \bigcup_{c \in RC(n)} latest(n, c, WC)$$

$$latest(n, x, S) = \mathcal{DS}(p) \cup \{\ell(p)\} \quad \text{where } p = \max\{m \mid m < n \text{ and } x \in S(m)\}$$

Dynamic slicing algorithm: example

n	ℓ	STATEMENT	*WD*	*RD*	*WC*	*RC*	$\mathcal{DS}(n)$
1	1	`proc (n, a)`	`n, a`				

Dynamic slicing algorithm: example

n	ℓ	STATEMENT	WD	RD	WC	RC	$\mathcal{DS}(n)$
1	1	`proc (n, a)`	n, a				
2	3	`x := 1`	x				

Dynamic slicing algorithm: example

n	ℓ	STATEMENT	WD	RD	WC	RC	$\mathcal{DS}(n)$
1	1	`proc (n, a)`	n, a				
2	3	`x := 1`	x				
3	4	`b := a + x`	b	a, x			

Dynamic slicing algorithm: example

n	ℓ	STATEMENT	*WD*	*RD*	*WC*	*RC*	$\mathcal{DS}(n)$
1	1	`proc (n, a)`	n, a				
2	3	`x := 1`	x				
3	4	`b := a + x`	b	a, x			1, 3
4	5	`a := a + 1`	a	a			

Dynamic slicing algorithm: example

n	ℓ	STATEMENT	WD	RD	WC	RC	$\mathcal{DS}(n)$
1	1	`proc (n, a)`	n, a				
2	3	`x := 1`	x				
3	4	`b := a + x`	b	a, x			1, 3
4	5	`a := a + 1`	a	a			1
5	6	`k := 1`	k				

Dynamic slicing algorithm: example

n	ℓ	STATEMENT	WD	RD	WC	RC	$\mathcal{DS}(n)$
1	1	`proc (n, a)`	n, a				
2	3	`x := 1`	x				
3	4	`b := a + x`	b	a, x			1, 3
4	5	`a := a + 1`	a	a			1
5	6	`k := 1`	k				
6	7	`s := 0`	s				

Dynamic slicing algorithm: example

n	ℓ	STATEMENT	*WD*	*RD*	*WC*	*RC*	$\mathcal{DS}(n)$
1	1	`proc (n, a)`	n, a				
2	3	`x := 1`	x				
3	4	`b := a + x`	b	a, x			1, 3
4	5	`a := a + 1`	a	a			1
5	6	`k := 1`	k				
6	7	`s := 0`	s				
7	8	**while** k $\leq$ n		k, n	8		

Dynamic slicing algorithm: example

n	ℓ	STATEMENT	WD	RD	WC	RC	$\mathcal{DS}(n)$
1	1	`proc (n, a)`	n, a				
2	3	`x := 1`	x				
3	4	`b := a + x`	b	a, x			1, 3
4	5	`a := a + 1`	a	a			1
5	6	`k := 1`	k				
6	7	`s := 0`	s				
7	8	**while** k $\leq$ n		k, n	8		6, 1
8	9	**if** b $>$ 0		b	9	8	

Dynamic slicing algorithm: example

n	ℓ	STATEMENT	WD	RD	WC	RC	$\mathcal{DS}(n)$
1	1	`proc (n, a)`	n, a				
2	3	`x := 1`	x				
3	4	`b := a + x`	b	a, x			1, 3
4	5	`a := a + 1`	a	a			1
5	6	`k := 1`	k				
6	7	`s := 0`	s				
7	8	**while** k $\leq$ n		k, n	8		6, 1
8	9	**if** b $>$ 0		b	9	8	4, 1, 3, 8, 6
9	10	**if** a $>$ 1		a	10	9	

Dynamic slicing algorithm: example

n	ℓ	STATEMENT	WD	RD	WC	RC	$\mathcal{DS}(n)$
1	1	`proc (n, a)`	n, a				
2	3	`x := 1`	x				
3	4	`b := a + x`	b	a, x			1, 3
4	5	`a := a + 1`	a	a			1
5	6	`k := 1`	k				
6	7	`s := 0`	s				
7	8	**while** k $\leq$ n		k, n	8		6, 1
8	9	**if** b $>$ 0		b	9	8	4, 1, 3, 8, 6
9	10	**if** a $>$ 1		a	10	9	5, 1, 9, 4, 3, 8, 6
10	12	`s := s + x`	s	s, x		8	

Dynamic slicing algorithm: example

n	ℓ	STATEMENT	WD	RD	WC	RC	$\mathcal{DS}(n)$
1	1	`proc (n, a)`	n, a				
2	3	`x := 1`	x				
3	4	`b := a + x`	b	a, x			1, 3
4	5	`a := a + 1`	a	a			1
5	6	`k := 1`	k				
6	7	`s := 0`	s				
7	8	**while** `k` $\leq$ `n`		k, n	8		6, 1
8	9	**if** `b` $>$ `0`		b	9	8	4, 1, 3, 8, 6
9	10	**if** `a` $>$ `1`		a	10	9	5, 1, 9, 4, 3, 8, 6
10	12	`s := s + x`	s	s, x		8	7, 3, 8, 6, 1
11	13	`k := k + 1`	k	k		8	

Dynamic slicing algorithm: example

n	ℓ	STATEMENT	*WD*	*RD*	*WC*	*RC*	$\mathcal{DS}(n)$
1	1	`proc (n, a)`	n, a				
2	3	`x := 1`	x				
3	4	`b := a + x`	b	a, x			1, 3
4	5	`a := a + 1`	a	a			1
5	6	`k := 1`	k				
6	7	`s := 0`	s				
7	8	**while** k $\leq$ n		k, n	8		6, 1
8	9	**if** b $>$ 0		b	9	8	4, 1, 3, 8, 6
9	10	**if** a $>$ 1		a	10	9	5, 1, 9, 4, 3, 8, 6
10	12	`s := s + x`	s	s, x		8	7, 3, 8, 6, 1
11	13	`k := k + 1`	k	k		8	6, 8, 1
12	8	**while** k $\leq$ n		k, n	8		

Dynamic slicing algorithm: example

n	ℓ	STATEMENT	WD	RD	WC	RC	$\mathcal{DS}(n)$
1	1	`proc (n, a)`	n, a				
2	3	`x := 1`	x				
3	4	`b := a + x`	b	a, x			1, 3
4	5	`a := a + 1`	a	a			1
5	6	`k := 1`	k				
6	7	`s := 0`	s				
7	8	**while** k $\leq$ n		k, n	8		6, 1
8	9	**if** b $>$ 0		b	9	8	4, 1, 3, 8, 6
9	10	**if** a $>$ 1		a	10	9	5, 1, 9, 4, 3, 8, 6
10	12	`s := s + x`	s	s, x		8	7, 3, 8, 6, 1
11	13	`k := k + 1`	k	k		8	6, 8, 1
12	8	**while** k $\leq$ n		k, n	8		13, 6, 8, 1
13	9	**if** b $>$ 0		b	9	8	

Dynamic slicing algorithm: example

n	ℓ	STATEMENT	*WD*	*RD*	*WC*	*RC*	$\mathcal{DS}(n)$
1	1	`proc (n, a)`	n, a				
2	3	`x := 1`	x				
3	4	`b := a + x`	b	a, x			1, 3
4	5	`a := a + 1`	a	a			1
5	6	`k := 1`	k				
6	7	`s := 0`	s				
7	8	**while** k $\leq$ n		k, n	8		6, 1
8	9	**if** b $>$ 0		b	9	8	4, 1, 3, 8, 6
9	10	**if** a $>$ 1		a	10	9	5, 1, 9, 4, 3, 8, 6
10	12	`s := s + x`	s	s, x		8	7, 3, 8, 6, 1
11	13	`k := k + 1`	k	k		8	6, 8, 1
12	8	**while** k $\leq$ n		k, n	8		13, 6, 8, 1
13	9	**if** b $>$ 0		b	9	8	4, 1, 3, 8, 13, 6
14	10	**if** a $>$ 1		a	10	9	

Dynamic slicing algorithm: example

n	ℓ	STATEMENT	*WD*	*RD*	*WC*	*RC*	$\mathcal{DS}(n)$
1	1	`proc (n, a)`	n, a				
2	3	`x := 1`	x				
3	4	`b := a + x`	b	a, x			1, 3
4	5	`a := a + 1`	a	a			1
5	6	`k := 1`	k				
6	7	`s := 0`	s				
7	8	**while** k $\leq$ n		k, n	8		6, 1
8	9	**if** b $>$ 0		b	9	8	4, 1, 3, 8, 6
9	10	**if** a $>$ 1		a	10	9	5, 1, 9, 4, 3, 8, 6
10	12	`s := s + x`	s	s, x		8	7, 3, 8, 6, 1
11	13	`k := k + 1`	k	k		8	6, 8, 1
12	8	**while** k $\leq$ n		k, n	8		13, 6, 8, 1
13	9	**if** b $>$ 0		b	9	8	4, 1, 3, 8, 13, 6
14	10	**if** a $>$ 1		a	10	9	5, 1, 9, 4, 3, 8, 13, 6
15	12	`s := s + x`	s	s, x		8	

Dynamic slicing algorithm: example

n	ℓ	STATEMENT	WD	RD	WC	RC	$\mathcal{DS}(n)$
1	1	`proc (n, a)`	n, a				
2	3	`x := 1`	x				
3	4	`b := a + x`	b	a, x			1, 3
4	5	`a := a + 1`	a	a			1
5	6	`k := 1`	k				
6	7	`s := 0`	s				
7	8	**while** k $\leq$ n		k, n	8		6, 1
8	9	**if** b $>$ 0		b	9	8	4, 1, 3, 8, 6
9	10	**if** a $>$ 1		a	10	9	5, 1, 9, 4, 3, 8, 6
10	12	`s := s + x`	s	s, x		8	7, 3, 8, 6, 1
11	13	`k := k + 1`	k	k		8	6, 8, 1
12	8	**while** k $\leq$ n		k, n	8		13, 6, 8, 1
13	9	**if** b $>$ 0		b	9	8	4, 1, 3, 8, 13, 6
14	10	**if** a $>$ 1		a	10	9	5, 1, 9, 4, 3, 8, 13, 6
15	12	`s := s + x`	s	s, x		8	12, 7, 3, 8, 6, 1, 13
16	13	`k := k + 1`	k	k		8	

Dynamic slicing algorithm: example

n	ℓ	STATEMENT	WD	RD	WC	RC	$\mathcal{DS}(n)$
1	1	`proc (n, a)`	n, a				
2	3	`x := 1`	x				
3	4	`b := a + x`	b	a, x			1, 3
4	5	`a := a + 1`	a	a			1
5	6	`k := 1`	k				
6	7	`s := 0`	s				
7	8	**while** k $\leq$ n		k, n	8		6, 1
8	9	**if** b $>$ 0		b	9	8	4, 1, 3, 8, 6
9	10	**if** a $>$ 1		a	10	9	5, 1, 9, 4, 3, 8, 6
10	12	`s := s + x`	s	s, x		8	7, 3, 8, 6, 1
11	13	`k := k + 1`	k	k		8	6, 8, 1
12	8	**while** k $\leq$ n		k, n	8		13, 6, 8, 1
13	9	**if** b $>$ 0		b	9	8	4, 1, 3, 8, 13, 6
14	10	**if** a $>$ 1		a	10	9	5, 1, 9, 4, 3, 8, 13, 6
15	12	`s := s + x`	s	s, x		8	12, 7, 3, 8, 6, 1, 13
16	13	`k := k + 1`	k	k		8	13, 6, 8, 1
17	8	**while** k $\leq$ n		k, n	8		

Dynamic slicing algorithm: example

n	ℓ	STATEMENT	*WD*	*RD*	*WC*	*RC*	$\mathcal{DS}(n)$
1	1	`proc (n, a)`	n, a				
2	3	`x := 1`	x				
3	4	`b := a + x`	b	a, x			1, 3
4	5	`a := a + 1`	a	a			1
5	6	`k := 1`	k				
6	7	`s := 0`	s				
7	8	**while** $k \leq n$		k, n	8		6, 1
8	9	**if** $b > 0$		b	9	8	4, 1, 3, 8, 6
9	10	**if** $a > 1$		a	10	9	5, 1, 9, 4, 3, 8, 6
10	12	`s := s + x`	s	s, x		8	7, 3, 8, 6, 1
11	13	`k := k + 1`	k	k		8	6, 8, 1
12	8	**while** $k \leq n$		k, n	8		13, 6, 8, 1
13	9	**if** $b > 0$		b	9	8	4, 1, 3, 8, 13, 6
14	10	**if** $a > 1$		a	10	9	5, 1, 9, 4, 3, 8, 13, 6
15	12	`s := s + x`	s	s, x		8	12, 7, 3, 8, 6, 1, 13
16	13	`k := k + 1`	k	k		8	13, 6, 8, 1
17	8	**while** $k \leq n$		k, n	8		13, 6, 8, 1
18	14	`print(s)`		s		8	

Dynamic slicing algorithm: example

n	ℓ	STATEMENT	WD	RD	WC	RC	$\mathcal{DS}(n)$
1	1	`proc (n, a)`	n, a				
2	3	`x := 1`	x				
3	4	`b := a + x`	b	a, x			1, 3
4	5	`a := a + 1`	a	a			1
5	6	`k := 1`	k				
6	7	`s := 0`	s				
7	8	**while** k $\leq$ n		k, n	8		6, 1
8	9	**if** b $>$ 0		b	9	8	4, 1, 3, 8, 6
9	10	**if** a $>$ 1		a	10	9	5, 1, 9, 4, 3, 8, 6
10	12	`s := s + x`	s	s, x		8	7, 3, 8, 6, 1
11	13	`k := k + 1`	k	k		8	6, 8, 1
12	8	**while** k $\leq$ n		k, n	8		13, 6, 8, 1
13	9	**if** b $>$ 0		b	9	8	4, 1, 3, 8, 13, 6
14	10	**if** a $>$ 1		a	10	9	5, 1, 9, 4, 3, 8, 13, 6
15	12	`s := s + x`	s	s, x		8	12, 7, 3, 8, 6, 1, 13
16	13	`k := k + 1`	k	k		8	13, 6, 8, 1
17	8	**while** k $\leq$ n		k, n	8		13, 6, 8, 1
18	14	`print(s)`		s		8	12, 7, 3, 8, 6, 1, 13

The dynamic slice

The dynamic slice of program P according to slicing criterion ℓ (where ℓ is a location in P) is a subset of all statements in P that affect the values of variables at ℓ when P runs on T

The dynamic slice

The dynamic slice of program P according to slicing criterion ℓ (where ℓ is a location in P) is a subset of all statements in P that affect the values of variables at ℓ when P runs on T

The dynamic slice according to slicing criterion ℓ when P runs on a test $t \in T$ is $\mathcal{DS}(t_\ell) \cup \{\ell\}$, where t_ℓ is the last step in the trace executing t where ℓ appears.

The dynamic slice

The dynamic slice of program P according to slicing criterion ℓ (where ℓ is a location in P) is a subset of all statements in P that affect the values of variables at ℓ when P runs on T

The dynamic slice according to slicing criterion ℓ when P runs on a test $t \in T$ is $\mathcal{DS}(t_\ell) \cup \{\ell\}$, where t_ℓ is the last step in the trace executing t where ℓ appears.

The overall slice according to slicing criterion ℓ when P runs on T is the union $\bigcup_{t \in T} \mathcal{DS}(t_\ell)$.

Dynamic slicing: example

The last step s where `print(s)` is executed has a dynamic slice $\mathcal{DS}(s) = \{1, 3, 6, 7, 8, 12, 13\}$. We also add the local variable declaration on line 2, which is needed to make the procedure consistent.

```
1  procedure proc(n, a: Integer):
2    var s, x, b, k: Integer
3    x := 1
4    b := a + x
5    a := a + 1
6    k := 1
7    s := 0
8    while k ≤ n
9      if b > 0
10       if a > 1
11         x := 2
12     s := s + x
13     k := k + 1
14   print(s)
```

```
1  procedure proc(n, a: Integer):
2    var s, x, b, k: Integer
3    x := 1
6    k := 1
7    s := 0
8    while k ≤ n
12     s := s + x
13     k := k + 1
```

Dynamic slicing: example

The last step s where `print(s)` is executed has a dynamic slice $\mathcal{DS}(s) = \{1, 3, 6, 7, 8, 12, 13\}$. We also add the local variable declaration on line 2, which is needed to make the procedure consistent.

```
1  procedure proc(n, a: Integer):
2    var s, x, b, k: Integer
3    x := 1
4    b := a + x
5    a := a + 1
6    k := 1
7    s := 0
8    while k ≤ n
9      if b > 0
10       if a > 1
11         x := 2
12     s := s + x
13     k := k + 1
14   print(s)
```

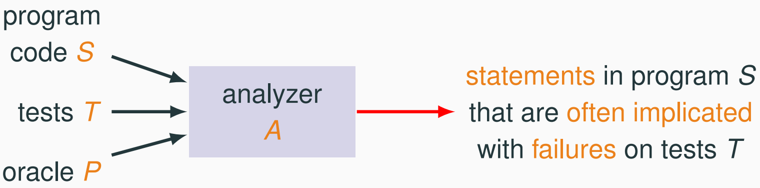
```
1  procedure proc(n, a: Integer):
2    var s, x, b, k: Integer
3    x := 1
6    k := 1
7    s := 0
8    while k ≤ n
12     s := s + x
13     k := k + 1
```

Since argument `a` and local variable `b` are no longer used in the slice, we can remove those as well.

Fault localization

Fault localization for debugging

Finding the source of a bug is easier if we know which statements are more likely to be implicated with faulty behavior.



Fault localization is the process of ranking statements of a program according to how frequently they are implicated with faults in a given set of tests.

Families of fault localization techniques

spectrum-based: using the information about which statements or states are reached by passing vs. failing runs

mutation-based: using the information about which statements of the original program turn a passing test into a failing one (or vice versa) when they are randomly mutated

slice-based: using slices of the original program that only include statements that determined the final incorrect results

statistical: using the distribution of program predicates sampled in passing vs. failing runs; predicates whose distributions differ significantly are indicative of faulty behavior

Families of fault localization techniques

spectrum-based: using the information about which statements or states are reached by passing vs. failing runs

mutation-based: using the information about which statements of the original program turn a passing test into a failing one (or vice versa) when they are randomly mutated

slice-based: using slices of the original program that only include statements that determined the final incorrect results

statistical: using the distribution of program predicates sampled in passing vs. failing runs; predicates whose distributions differ significantly are indicative of faulty behavior

We present the basic ideas behind spectrum-based fault localization techniques, which are the simplest, and hence most widely applicable using dynamic analysis.

Spectrum-based fault localization

The spectrum of a program S is a summary of its runs in terms of information such as covered statements, reached states, and whether the runs were passing or failing.

Spectrum-based fault localization

The spectrum of a program S is a summary of its runs in terms of information such as covered statements, reached states, and whether the runs were passing or failing.

The basic ideas of spectrum-based fault localization:

1. trace runs of S on all tests T, recording the information about which tests execute (cover) which statements
2. define a metric for each statement, which reflects whether the statement was executed more often in passing <u>or</u> in failing tests
3. rank statements according to the metric

Fault localization: example

This Python program should compute the middle of three integers, but it has a bug:

```python
def middle(x, y, z):
    m = z
    if y < z:
        if x < y:
            m = y
        else:
            if x < z:
                m = y
    else:
        if x > y:
            m = y
        else:
            if x > z:
                m = x
    return m
```

Five tests are passing, one is failing:

- ✔ t_1 : `middle(3, 3, 5)` returns 3
- ✔ t_2 : `middle(1, 2, 3)` returns 2
- ✔ t_3 : `middle(3, 2, 1)` returns 2
- ✔ t_4 : `middle(5, 5, 5)` returns 5
- ✔ t_5 : `middle(5, 3, 4)` returns 4
- ✘ t_6 : `middle(2, 1, 3)` returns 1

Let's use the information about these 6 runs to help us find where the bug originates.

Tracing executions: example

We log which statements are executed ("covered") by each test.

STATEMENT	TESTS t_1 ✓	t_2 ✓	t_3 ✓	t_4 ✓	t_5 ✓	t_6 ✗
1 `def middle(x, y, z):`	1	1	1	1	1	1
2 `m = z`	1	1	1	1	1	1
3 `if y < z:`	1	1	1	1	1	1
4 `if x < y:`	1	1	0	0	1	1
5 `m = y`	0	1	0	0	0	0
6 `else:`	0	0	0	0	1	1
7 `if x < z:`	1	0	0	0	1	1
8 `m = y`	1	0	0	0	0	1
9 `else:`	0	0	1	1	0	0
10 `if x > y:`	0	0	1	1	0	0
11 `m = y`	0	0	1	0	0	0
12 `else:`	0	0	0	1	0	0
13 `if x > z:`	0	0	0	1	0	0
14 `m = x`	0	0	0	0	0	0
15 `return m`	1	1	1	1	1	1

line 13 executed by t_4

line 13 not executed by t_5

Suspiciousness score

Each statement ℓ gets a suspiciousness score *susp*(ℓ) that reflects the chance it is implicated with the failure.

Many different heuristics to compute suspiciousness scores exist. The basic criteria used by spectrum-based fault localization are:

- the more failing tests execute ℓ, the higher *susp*(ℓ)
- the more passing tests execute ℓ, the lower *susp*(ℓ)

Suspiciousness score

Each statement ℓ gets a suspiciousness score *susp*(ℓ) that reflects the chance it is implicated with the failure.

Many different heuristics to compute suspiciousness scores exist. The basic criteria used by spectrum-based fault localization are:

- the more failing tests execute ℓ, the higher *susp*(ℓ)
- the more passing tests execute ℓ, the lower *susp*(ℓ)

Tarantula, one of the first tools implementing spectrum-based fault localization, uses the score:

$$susp(\ell) \quad = \quad \frac{F(\ell)/F}{F(\ell)/F + P(\ell)/P}$$

$F(\ell) =$ # failing tests that execute ℓ $\qquad F =$ total # failing tests

$P(\ell) =$ # passing tests that execute ℓ $\qquad F =$ total # passing tests

Suspiciousness score: example

Each statement's Tarantula suspiciousness score based on the number of passing and failing tests.

	STATEMENT	# TESTS $P(\ell)$	$F(\ell)$	$susp(\ell)$
1	`def middle(x, y, z):`	5	1	0.50
2	`    m = z`	5	1	0.50
3	`    if y < z:`	5	1	0.50
4	`        if x < y:`	3	1	0.63
5	`            m = y`	1	0	0.00
6	`        else:`	2	1	0.71
7	`            if x < z:`	2	1	0.71
8	`                m = y`	1	1	0.83
9	`    else:`	2	0	0.00
10	`        if x > y:`	2	0	0.00
11	`            m = y`	1	0	0.00
12	`        else:`	1	0	0.00
13	`            if x > z:`	1	0	0.00
14	`                m = x`	0	0	–
15	`    return m`	5	1	0.50

Indeed, changing the most suspicious line 8 to `m = x` fixes the bug.

Using fault localization in practice

There has been over a decade of research in fault localization, trying to improve the accuracy of the heuristic suspiciousness ranking.

Using fault localization in practice

There has been over a decade of research in fault localization, trying to improve the accuracy of the heuristic suspiciousness ranking.

Are Automated Debugging Techniques Actually Helping Programmers?

Chris Parnin and Alessandro Orso
Georgia Institute of Technology
College of Computing
{chris.parnin|orso}@gatech.edu

*ISSTA'*11, July 17–21, 2011, Toronto, ON, Canada

Fault localization is not so useful for human debugging:

- ranking heuristics remain fairly imprecise
- defining heuristics for different kinds of programs is challenging
- information about statements may not be the most critical one

Using fault localization in practice

There has been over a decade of research in fault localization, trying to improve the accuracy of the heuristic suspiciousness ranking.

Are Automated Debugging Techniques Actually Helping Programmers?

Chris Parnin and Alessandro Orso
Georgia Institute of Technology
College of Computing
{chris.parnin|orso}@gatech.edu

*ISSTA'*11, July 17–21, 2011, Toronto, ON, Canada

Fault localization is not so useful for human debugging:

- ranking heuristics remain fairly imprecise
- defining heuristics for different kinds of programs is challenging
- information about statements may not be the most critical one

Fault localization heuristics are mainly used as part of fully automated tools based on dynamic analysis (such as for automated program repair), rather than to directly help programmers.

Dynamic assertion checking

Checking assertions dynamically

Dynamic assertion checking means evaluating assertions at run time:

- assertion evaluates to `true`: continue execution
- assertion evaluates to `false`: abort execution, report assertion failure

Checking assertions dynamically

Dynamic assertion checking means evaluating assertions at run time:

- assertion evaluates to `true`: continue execution
- assertion evaluates to `false`: abort execution, report assertion failure

Assertions are useful with dynamic analysis as well:

specification: rigorously documenting programmer's intent

debugging: an assertion failure indicates an invalid program state before it "infects" the output

design: design by contract supports dynamic analysis of partial and abstract implementations

Run-time semantics of assertions

Assertions may be impractical or impossible to check at run time.

Run-time semantics of assertions

Assertions may be impractical or impossible to check at run time.

A linked list's nodes do not have loops: `assert ¬has_loops(list)`

Checkable at run time but may be a performance hog.

Run-time semantics of assertions

Assertions may be impractical or impossible to check at run time.

A newly allocated object is fresh (uses a new memory location):

`list := new LinkedList ; assert` $\forall r \in$ `ref` $\bullet$ $allocated(r) \Longrightarrow r \neq$ `list`

Requires to check all allocated memory, which may be very large and not entirely accessible by the program's runtime; also, implementation details may make this assertion fail spuriously – for example because the runtime allows reusing memory location if they are not modified.

Run-time semantics of assertions

Assertions may be impractical or impossible to check at run time.

There are infinitely many prime numbers:

$\forall$`n:` **`Integer`** $\bullet$ $\exists$`p:` **`Integer`** $\bullet$ `p` $>$ `n` $\wedge$ `prime(p)`

In principle, it requires to check all infinitely many integers, or at least all valid machine integers.

Run-time semantics of assertions

Assertions may be impractical or impossible to check at run time.

Method `m` is pure – has no side effects: `@Pure int m(int x)`

Some side effects may be technically detectable at run time, but may still be expensive (for example, changing the state of an object). Others may be outside the direct control of the runtime (for example, those involving input/output or concurrency schedules).

Run-time semantics of assertions

Assertions may be impractical or impossible to check at run time.

Most of these examples are complex to check statically too. However, with static reasoning we can always abstract the program state in a way that makes our assumptions explicit.

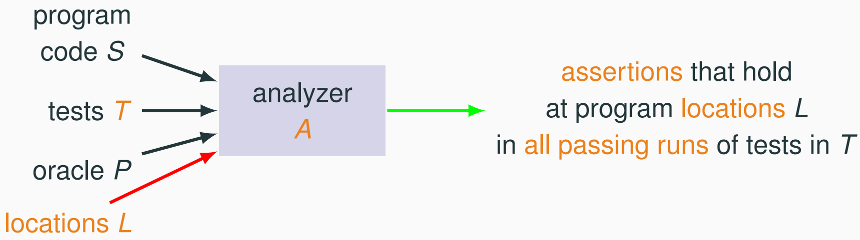
Assertion checking in practice

Tips to use dynamic assertion checking on realistic programs:

- Write assertions with run-time checking in mind
- Instrument assertion checking carefully to avoid performance bottlenecks
- Select which assertions to check, according to what's the main target of the analysis:
    - library clients: preconditions
    - suppliers: postconditions
    - object consistency: class invariants
- Disable assertion checking completely when releasing software to final users

Dynamic assertion mining

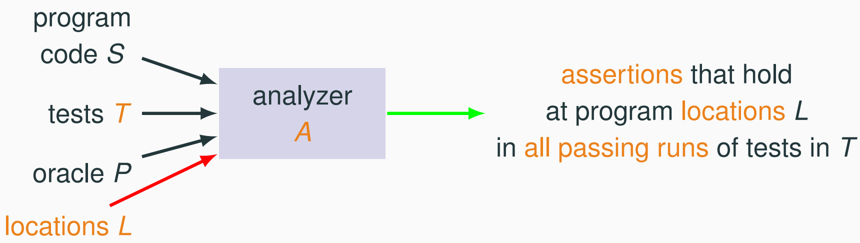
Assertions can also be used to summarize how a program behaves on the given tests.



We describe dynamic assertion mining: a technique to report which assertions – among those that can be built following predefined templates – hold in all test runs.

Dynamic assertion mining

Assertions can also be used to summarize how a program behaves on the given tests.



We describe dynamic assertion mining: a technique to report which assertions – among those that can be built following predefined templates – hold in all test runs.

Dynamic assertion mining is also called: assertion inference (even though dynamic inference is unsound) or invariant inference (since the assertions are invariant in the observed runs).

Dynamic assertion mining: outline

We monitor predefined assertions and discard all those that fail:

1. Instantiate assertions according to predefined templates:
   - $\mathsf{v} < n$ for all integer variables v and all constants $-100 \leq n \leq 100$
   - $\mathsf{v} > n$ for all integer variables v and all constants $-100 \leq n \leq 100$
   - $\mathsf{u} = \mathsf{v}$ for all distinct program variables u and v
   - $\mathsf{u} \neq \mathsf{v}$ for all distinct program variables u and v
   - $\mathsf{u} < \mathsf{v}$ for all distinct integer program variables u and v
   - ...
2. Run program S on the tests T, monitoring the assertions at program locations $\ell \in L$
3. If an assertion fails, discard it (a counterexample)
4. All assertions that survived all passing tests are outputted

Dynamic assertion mining: outline

We monitor predefined assertions and discard all those that fail:

1. Instantiate assertions according to predefined templates:
   - $\mathsf{v} < n$ for all integer variables v and all constants $-100 \leq n \leq 100$
   - $\mathsf{v} > n$ for all integer variables v and all constants $-100 \leq n \leq 100$
   - $\mathsf{u} = \mathsf{v}$ for all distinct program variables u and v
   - $\mathsf{u} \neq \mathsf{v}$ for all distinct program variables u and v
   - $\mathsf{u} < \mathsf{v}$ for all distinct integer program variables u and v
   - ...
2. Run program S on the tests T, monitoring the assertions at program locations $\ell \in L$
3. If an assertion fails, discard it (a counterexample)
4. All assertions that survived all passing tests are outputted

This mining technique is unsound (because an assertion that survives all tests may fail with other inputs) and complete (because every discarded assertion failed on a concrete counterexample).

Dynamic assertion mining: example

Running Daikon (a widely used dynamic assertion mining tool) on `sum_array` running with random inputs finds several assertions that hold at the procedure's output.

```
procedure sum_array (a: Array<int>): (s: Integer)
  var k: Integer
  k, s := 0, 0
  while k < a.size
    k, s := k + 1, s + a[k]
  // sum_array: exit
```

Dynamic assertion mining: example

Running Daikon (a widely used dynamic assertion mining tool) on `sum_array` running with random inputs finds several assertions that hold at the procedure's output.

```
1 procedure sum_array (a: Array<int>): (s: Integer)
2   var k: Integer
3   k, s := 0, 0
4   while k < a.size
5     k, s := k + 1, s + a[k]
6   // sum_array: exit
```

ASSERTIONS AT 6	CORRECT?
k = a.size	

Dynamic assertion mining: example

Running Daikon (a widely used dynamic assertion mining tool) on `sum_array` running with random inputs finds several assertions that hold at the procedure's output.

```
procedure sum_array (a: Array<int>): (s: Integer)
  var k: Integer
  k, s := 0, 0
  while k < a.size
    k, s := k + 1, s + a[k]
  // sum_array: exit
```

ASSERTIONS AT 6	CORRECT?
`k = a.size`	✔
`a = old(a)`	

Dynamic assertion mining: example

Running Daikon (a widely used dynamic assertion mining tool) on `sum_array` running with random inputs finds several assertions that hold at the procedure's output.

```
1 procedure sum_array (a: Array<int>): (s: Integer)
2   var k: Integer
3   k, s := 0, 0
4   while k < a.size
5     k, s := k + 1, s + a[k]
6   // sum_array: exit
```

ASSERTIONS AT 6	CORRECT?
`k = a.size`	✓
`a = old(a)`	✓
`s = sum(a)`	

Dynamic assertion mining: example

Running Daikon (a widely used dynamic assertion mining tool) on `sum_array` running with random inputs finds several assertions that hold at the procedure's output.

```
1 procedure sum_array (a: Array<int>): (s: Integer)
2   var k: Integer
3   k, s := 0, 0
4   while k < a.size
5     k, s := k + 1, s + a[k]
6   // sum_array: exit
```

ASSERTIONS AT 6	CORRECT?
k = a.size	✓
a = **old**(a)	✓
s = sum(a)	✓
7 ≤ a.size ≤ 13	

Dynamic assertion mining: example

Running Daikon (a widely used dynamic assertion mining tool) on `sum_array` running with random inputs finds several assertions that hold at the procedure's output.

```
1 procedure sum_array (a: Array<int>): (s: Integer)
2  var k: Integer
3  k, s := 0, 0
4  while k < a.size
5    k, s := k + 1, s + a[k]
6  // sum_array: exit
```

ASSERTIONS AT 6	CORRECT?
`k = a.size`	✓
`a = old(a)`	✓
`s = sum(a)`	✓
`7 ≤ a.size ≤ 13`	✗
`∀ 0 ≤ i < a.size (a[i] ≥ -100)`	

Dynamic assertion mining: example

Running Daikon (a widely used dynamic assertion mining tool) on `sum_array` running with random inputs finds several assertions that hold at the procedure's output.

```
1 procedure sum_array (a: Array<int>): (s: Integer)
2   var k: Integer
3   k, s := 0, 0
4   while k < a.size
5     k, s := k + 1, s + a[k]
6   // sum_array: exit
```

ASSERTIONS AT 6	CORRECT?
`k = a.size`	✓
`a = old(a)`	✓
`s = sum(a)`	✓
`7 ≤ a.size ≤ 13`	✗
`∀ 0 ≤ i < a.size (a[i] ≥ -100)`	✗

Dynamic assertion mining: example

Running Daikon (a widely used dynamic assertion mining tool) on `sum_array` running with random inputs finds several assertions that hold at the procedure's output.

```
1 procedure sum_array (a: Array<int>): (s: Integer)
2   var k: Integer
3   k, s := 0, 0
4   while k < a.size
5     k, s := k + 1, s + a[k]
6   // sum_array: exit
```

ASSERTIONS AT 6	CORRECT?
k = a.size	✓
a = **old**(a)	✓
s = sum(a)	✓
7 ≤ a.size ≤ 13	✗
∀ 0 ≤ i < a.size (a[i] ≥ -100)	✗

The spurious assertions reflect the range of values used to generate the inputs, not the actual expected behavior of the program.

Dynamic assertion mining in practice

While the basic idea of dynamic assertion mining is quite simple, numerous implementation details ensure that the technique is really scalable and useful in practice.

Dynamic assertion mining in practice

While the basic idea of dynamic assertion mining is quite simple, numerous implementation details ensure that the technique is really scalable and useful in practice.

Redundancy elimination:

- compute which assertions imply other assertions
- if $A_1 \Longrightarrow A_2$ and both A_1 and A_2 are mined, report only A_1
- if $A_1 \Longrightarrow A_2$ and A_2 fails, discard A_1 as well

For example $A_1 = \texttt{x > 1}$ and $A_2 = \texttt{x > 0}$

Dynamic assertion mining in practice

While the basic idea of dynamic assertion mining is quite simple, numerous implementation details ensure that the technique is really scalable and useful in practice.

Negative assertions:

- if a variable v can take R possible values, the probability that v is randomly never k over n runs is $(1 - 1/r)^n$
- if the assertion $v \neq k$ passes all tests, it is reported only if $(1 - 1/r)^n < \epsilon$ for some user-defined confidence level ϵ

Dynamic assertion mining in practice

While the basic idea of dynamic assertion mining is quite simple, numerous implementation details ensure that the technique is really scalable and useful in practice.

Abstract types: variables of the same type that represent unrelated information cannot be meaningfully compared in an assertion.

For example `temperature` $<$ `population` is likely true but is not a meaningful assertion.

Using dynamic information about whether variables are combined in any runs, we can infer these “abstract types” and avoid reporting spurious assertions.

Assertion mining tools

Daikon is the first implementation of the idea of dynamic assertion mining, and is still widely used

Assertion mining tools

Daikon is the first implementation of the idea of dynamic assertion mining, and is still widely used

Assertion mining can also be done statically in a way that is sound. Tools that can do this are often limited in the program features that they support – so they may lack practicality of scalability.

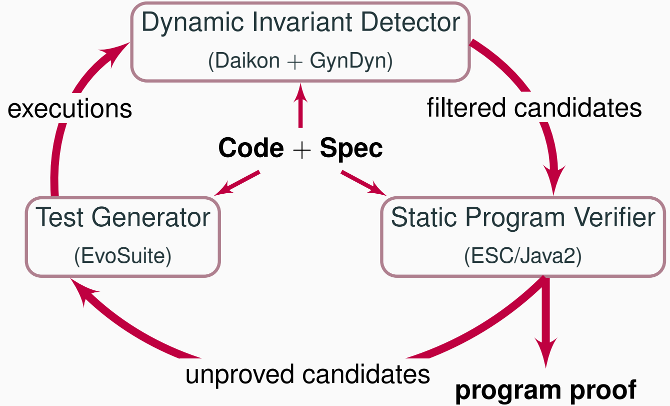
InvGen supports a very restricted subset of the C language (similar to Helium).

Valigator and other tools based on the Vampire first-order theorem prover also tackle subsets of the C language (including certain kinds of loops on arrays).

Case studies: putting it all together

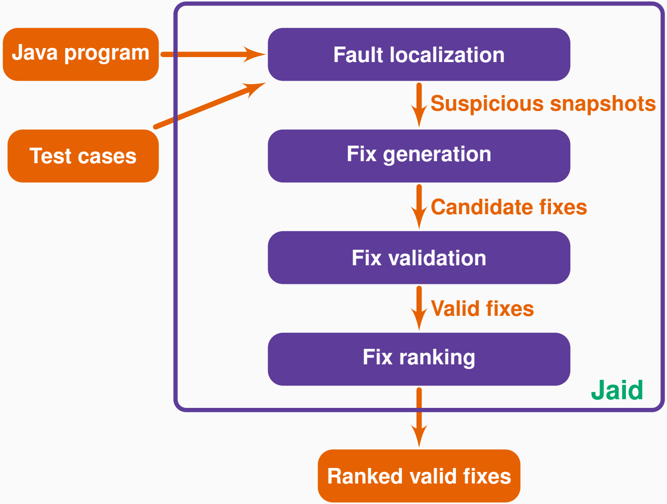
DynaMate: combining static and dynamic

DynaMate combines dynamic assertion mining with deductive verification to automatically infer loop invariants and use them to prove methods correct against a pre/post specification.



Jaid: automated program repair

Jaid combines several dynamic analysis techniques to automatically build fix suggestions for bugs in Java programs.



Summary

Dynamic analysis: techniques

Dynamic analysis is a large family of techniques based on summarizing a program's behavior on concrete inputs.

Dynamic analysis techniques are best effort and often used to support debugging with simplifications and abstractions.

soundness/completeness: unsound and complete – dynamic analysis is based on a finite set of test inputs (and hence under-approximations of general program behavior)

complexity: generally more time consuming than static analysis for the same task

automation: fully automated given the tests (which can also be generated automatically) and oracles

expressiveness: assertions that can be effectively monitored at run time

Dynamic analysis: tools and practice

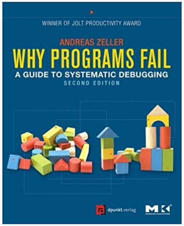
Dynamic analysis tools support various tasks such as test-case generation, input simplification, program simplification, slicing, and assertion mining.

Dynamic analysis is routinely used in practical case studies, often in combination with other analysis techniques.

Main outstanding challenges:

- scalability to complex assertions and large programs
- accuracy (soundness) of the best-effort analysis in practice on average
- combining it effectively with static techniques to leverage their complementary features

Further reading



This class's presentations of delta debugging and dynamic slicing is based on the book Why Programs Fail.

Some papers originally presenting, or providing more details, about some of the techniques we have seen:

delta debugging: Zeller and Hildebrandt: Simplifying and isolating failure-inducing input, 2002

dynamic slicing: Tip: A survey of program slicing techniques, 1995

dynamic assertion mining: Ernst et al.: Dynamically discovering likely program invariants to support program evolution, 2001

fault localization: Wong et al.: A survey on software fault localization, 2016

These slides' license