Automatic Verification of Computer Programs

these slides contain advanced material and are optional
What is verification

• Check correctness of the implementation given the specification

• **Static verification**
  – Check correctness without executing the program
  – E.g. static type systems, theorem provers

• **Dynamic verification**
  – Check correctness by executing the program
  – E.g. unit tests, automatic testing

• **Automatic verification**
  – Push-button verification
• Verification is just one part of the process
• All parts can (in theory) be automated
How to get the specification

• Need **machine-readable** specification for automatic verification (not just comments)

• Different variants:
  – Eiffel‘s „**Design by Contract**“
    • Built-in contracts
  – .Net 4.0 „**Code Contracts**“
    • Contracts implemented as a library
  – JML „**Java Modeling Language**“
    • Dialect of Java featuring contracts as special comments
  – D „**Contracts**“
    • Evolved from C++, built-in contracts

**Specification** – Verification – Correction
Contracts in different languages

**Eiffel**

```eiffel
deposit (amount: INTEGER)
  require
    amount >= 0
  do
    balance := balance + amount
  ensure
    balance = old balance + amount
end
```

**CodeContracts**

```java
public void deposit(int amount)
{
  Contract.Requires(amount >= 0);
  Contract.Ensures(balance == Contract.OldValue<int>(balance) + amount);
  balance += amount;
}
```

**JML**

```jml
/*@ 
requires amount >= 0; 
ensures balance == \old(balance)+amount 
@*/
public void deposit(int amount) {
  balance += amount
}
```

**D**

```d
function deposit(int amount) __in { assert(amount >= 0); 
  int oldb = balance; } __out {
  assert(bal == oldb + amount); } __body {
  balance += amount
}
```
Writing full specifications

- Writing expressive specification is difficult
- Specifying full effect of routines

- Describing what changes
- Describing what does not change (frame condition)

**Specification** – Verification – Correction
MML and EiffelBase2

- Model-based contracts use mathematical notions for expressing full specifications

```eiffel
map : MML_MAP [INTEGER, G]
-- Map of keys to values.

note
  status : specification
do
  create Result
  across Current as it loop
    Result := Result.updated (it.key, it.item)
  end
end

put (v : G; i : INTEGER)
-- Replace value at `i'.

note
  modify : map
require
  has_index (i)
do
  at (i).put (v)
ensure
  map ||= old map.updated (i, v)
end
```

Specification – Verification – Correction
Contract inference

• Generate contracts based on implementation

• **Dynamic** contract inference
  – Infer contracts based on program runs

• **Static** contract inference
  – Infer contracts without running the program

*Specification – Verification – Correction*
Dynamic contract inference

- Location **invariant** – a property that always holds at a given point in the program

  \[
  \ldots \quad x \leftarrow 0 \quad \ldots \quad x = 0
  \]

- Dynamic **invariant inference** – detecting location invariants from values observed during execution

- For pre- and postcondition inference, select routine entry and exit as program points

**Specification** – Verification – Correction
DAIKON example

• Uses templates for inferred contracts, e.g.
  \[ x = \text{const} \quad x \geq \text{const} \quad x = y \]

• Program point: ACCOUNT.deposit::ENTER

• Variables of interest: balance, amount

• Invariants:
  - balance = 0
  - balance \geq 0
  - amount = 10
  - amount \geq 1
  - balance = amount

• Samples

<table>
<thead>
<tr>
<th>balance</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>30</td>
<td>1</td>
</tr>
</tbody>
</table>

Specification – Verification – Correction
Static contract inference

- Infer precondition from postcondition/body
  - Weakest precondition calculus
- Infer loop invariants from postcondition
  - Generate mutations from postcondition

bubble_sort (a: ARRAY [T])
  require
    a.count > 0
  ensure
    sorted (a)
    permutation (a, old a)

from i := n until i = 1
invariant
  1 <= i <= n
  sorted (a[i+1..n])
  permutation (a, old a)
loop
  -- move the largest element
  -- in 1..i to position i
end

Specification – Verification – Correction
Dynamic verification

• Check that program satisfies its specification by executing the program

• Manual
  – Write unit tests (xUnit framework)
  – Execute program and click around

• Automatic
  – Random testing

Specification – Verification – Correction
Automatic testing with contracts

• Select routine under test
  • **Precondition** used for input validation
    – Test is valid if it passes precondition
  • **Postcondition** used as test oracle
    – Test is successful if it passes postcondition

Specification – **Verification** – Correction
Automatic testing with contracts

Test Input → Test Execution → Test Oracle

```
deposit (v: INTEGER)

require
  v > 0

do
  balance := balance + v

ensure
  balance = old balance + v
end
```

<table>
<thead>
<tr>
<th></th>
<th>Successful</th>
<th>Failed</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Precondition</strong></td>
<td><strong>Test valid</strong></td>
<td>Test invalid</td>
</tr>
<tr>
<td><strong>Body</strong></td>
<td>(see postcondition)</td>
<td><strong>Error in program</strong></td>
</tr>
<tr>
<td><strong>Postcondition</strong></td>
<td><strong>Test successful</strong></td>
<td><strong>Error in program</strong></td>
</tr>
</tbody>
</table>
Random testing

• Create random objects
  – Call random creation procedure
  – Call random commands
  – For arguments, generate random input

• Basic types
  – Random numbers
  – Interesting values: max_value, 1, 0, -1, ...
• Basic operation:
  – Record sequence of calls made to create objects
  – Call routine under test with different objects
  – If execution is ok, this is a **successful test case**
  – If a postcondition is violated, this is a **failing test case**

• Improve test case generation
  – Smarter input selection
    (e.g. use static analysis to select objects)
  – Test case minimization (removing unnecessary calls)
  – Build object pool
  – ...
Static verification

• Need a model of the programming language
  – What is the effect of an instruction
• Translate program to a mathematical representation
• Use an automatic or interactive theorem prover to check that specification is satisfied in every possible execution
AutoProof process

- Translates AST from EiffelStudio to Boogie
- Uses Boogie verifier to check Boogie files
- Traces verification errors back to Eiffel source

Specification – Verification – Correction
make
  local
    a: ACCOUNT
  do
    create a.make
    check a.balance = 0 end
end

implementation APPLICATION.make { 
  var a;
  entry:
    havoc a;
    assume (a! = Void) && (!Heap[a, $allocated]);
    Heap[a, $allocated] := true;
    Heap[a, $type] := ACCOUNT;
    call create.ACCOUNT.make(a);
    assert Heap[a, ACCOUNT.balance] = 0;
}
Automatic Fault Correction

• Build a test suite
  – Manual or automatic
• Find and localize faults
  – Failing test cases
  – Static analysis
• Try fixes
  – Apply fix templates with random code changes
• Validate fixes
  – Run test suite again, now all tests have to pass

Specification – Verification – **Correction**
AutoFix: model-based localization

- Abstract state as boolean queries
- Find differences between passing and failing tests

move_item (v: G)

-- from TWO_WAY_SET
-- Move `v' to the left
require v /= Void ; has (v)
local idx: INTEGER ; found: BOOLEAN
do
  idx := index
  from start until found or after loop
    found := (v = item)
    if not found then forth end
  end
  remove
  go_i_th (idx)
  put_left (v)
end

Invariant from passing
not is_empty
not before
not after
...

Invariant from failing
not is_empty
before
not after
...

0 1 count-1 count count+1

Specification – Verification – Correction
AutoFix: instantiating fixes

• Fix schema for common fixes

```plaintext
if fail_condition then
  fixing_action
else
  original_instruction
end
```

```plaintext
move_item (v: G)
  require v /= Void ; has (v)
  local idx: INTEGER ; found: BOOLEAN
  do
    idx := index
    from start until found or after loop
      found := (v = item)
      if not found then forth end
    end
  remove
  go_i_th (idx)
  put_left (v)
end
```
Demo

• AutoTest
• AutoProof
• AutoFix
Eiffel Verification Environment (EVE)

• Research branch of EiffelStudio
• Integrates most tools developed by us
  – AutoTest (dynamic verification)
  – AutoProof (static verification)
  – AutoFix (fault correction)
  – AutoInfer (dynamic contract inference)
  – MultiStar (static verification)
  – AliasAnalysis (static analysis)
• Other tools currently not integrated
  – CITADEL (dynamic contract inference)
  – gin-pink (static loop invariant inference)
Putting It All Together

EVE

AliasAnalysis
gin-pink

static inference

AutoProof MultiStar

fix found

proof found

AutoFix

tests failed

Manual Fixes

no fix found

tests ok

Element Dynamically Verified

dynamically verified

CITADEL AutoInfer

proof ok

no new contracts

Element Statically Verified

proof ok

no new contracts

Manual Proof

proof ok

Element Statically Verified

proof failed

new contracts

AutoTest

tests ok

no fix found

Manual Fixes

EVE

AutoInfer

Manual

Proof
References

- EVE: Eiffel Verification Environment
  [http://se.inf.ethz.ch/research/eve/](http://se.inf.ethz.ch/research/eve/)
- AutoTest, AutoProof, AutoFix, CITADEL, ...
  [http://se.inf.ethz.ch/research/](http://se.inf.ethz.ch/research/)
- CodeContracts
- Java Modeling Language (JML)
- D Programming Language
  [http://dlang.org/](http://dlang.org/)
- Daikon
- Boogie Verifier