Extracting Runtime Monitors from Tests: An Overview and a Way Forward

Abigail Cauchi, Luke Chircop, Christian Colombo, Adrian Francalanza, Mark Micallef, and Gordon Pace

Project GOMTA financed by the Malta Council for Science & Technology through the National Research & Innovation Programme 2013
Why generate monitors from tests?

• Monitors can provide extra assurance at runtime

• Industry already invests a lot in testing
  (but little in runtime verification)

• Creating monitors after creating tests feels repetitive/waste
Verification – A language problem

All behaviours

Good behaviours
Testing

All behaviours

Test some behaviours
Runtime verification

All behaviours

Good behaviours

Runtime behaviours
Testing

All behaviours

Test some behaviours

An assertion per checked behaviour
Runtime verification

All behaviours

Good behaviours

Runtime behaviours

One “assertion” for all behaviours!
Generating runtime verifiers from tests

All behaviours

Test some behaviours

Approximate a decision procedure for all behaviours from individual ones

All behaviours

Good behaviours
Why is it difficult?

All behaviours

Typical language inference challenges:
• Few examples
• Usually no negative examples

Test some behaviours

All behaviours

Good behaviours
Why not use test assertions directly?

Test

Sequence of Method Invocations

Assertions

VS

Runtime Verifier

Pattern Matching

Assertions
Test assertions are typically very specific

```java
@Test
public void testWithdraw(){
    Account a = new Account();
a.setBalance(100);
a.withdraw(60);
assertEquals(a.getBalance(),40);
}
```
Idealistic test assertions

```java
@Test
public void testWithdraw() {
    int initialBalance = 100;
    int withdrawAmount = 60;
    Account a = new Account();
    a.setBalance(initialBalance);
    a.withdraw(withdrawAmount);
    assertEquals(a.getBalance(), initialBalance - withdrawAmount);
}
```
What if you insist on using assertions?

• There might be other hidden assumptions:
  • Assumptions on the global state (shared data structures, files, etc)
  • Assumptions on the control/data flow leading up to the assertion (test setup, method call sequence in test, etc)
A look at related approaches

Testing to more “generalised” testing
1. EUnit $\rightarrow$ QuickCheck (Thomas Arts et al.)
2. Gherkin $\rightarrow$ QuickCheck (Christian Colombo et al.)

Testing to RV
3. QuickCheck $\rightarrow$ Larva (Gordon Pace and Kevin Falzon)

Testing to Regression testing/Debugging
4. Invariant detection with Daikon (Pastore et al.)
1. EUnit ➔ QuickCheck

- Generates QuickCheck automaton from sequences of method calls
- Uses algorithm to learn automata
- Uses learned automaton to improve testsuite
Points to consider

• Assumes the availability of negative traces
  • Not usually present in testsuites
• Suitable for testing, probably also for RV if negative traces are available
2. Gherkin → QuickCheck

• Similar to previous but state identification is easier as more explicit in Gherkin tests
Standard Business Specifications

Scenario: Model definition for myHealth - Doctors Section

Given I am on the "start state"
When I "login as a doctor"
Then I should go to the "doctors landing page"

Given I am on the "doctors landing page"
When I "click on Appointments"
Then I should go to the "appointments page"

Given I am on the "doctors landing page"
When I "click on Case Summaries"
Then I should go to the "case summaries page"

Given I am on the "doctors landing page"
When I "click on Laboratory Results"
Then I should go to the "lab results page"

Given I am on the "doctors landing page"
When I "click on Medical Image Reports"
Then I should go to the "medical image reports page"
Scenario: Model definition for myHealth – Doctors Section

Given I am on the "start state"
When I "login as a doctor"
Then I should go to the "doctors landing page"

Given I am on the "doctors landing page"
When I "click on Appointments"
Then I should go to the "appointments page"

Given I am on the "doctors landing page"
When I "click on Case Summaries"
Then I should go to the "case summaries page"

Given I am on the "doctors landing page"
When I "click on Laboratory Results"
Then I should go to the "lab results page"

Given I am on the "doctors landing page"
When I "click on Medical Image Reports"
Then I should go to the "medical image reports page"
Scenario: Model definition for myHealth - Doctors Section

Given I am on the "start state"
When I "login as doctor"
Then I should go to the "doctors landing page"

Given I am on the "doctors landing page"
When I "click on Appointments"
Then I should go to the "appointments page"

Given I am on the "doctors landing page"
When I "click on Case Summaries"
Then I should go to the "case summaries page"

Given I am on the "doctors landing page"
When I "click on Laboratory Results"
Then I should go to the "lab results page"

Given I am on the "doctors landing page"
When I "click on Medical Image Reports"
Then I should go to the "medical image reports page"
Standard Business Specifications

Given I am on the "doctors landing page"
When I "click on Laboratory Results"
Then I should go to the "lab results page"

Given I am on the "lab results page"
When I "search patient data"
Then I should go to the "lab results search results page" and the result should be "true" when "no data found" and "false" when "ok"

Given I am on the "lab results search results page"
When I "click on the myHealth logo"
Then I should go to the "doctors landing page"

Given I am on the "lab results search results page"
When I "search results"
Then I should go to the "view lab results page"

Given I am on the "view lab results page"
When I "release lab result"
Then I should remain on the "view lab results page"

Given I am on the "view lab results page"
When I "click on go back"
Then I should go to the "lab results search results page"
Automatically Generated QC Model
Points to consider

• The higher the testing level, the more useful for RV
3. QuickCheck → Larva

• Translates QC automata into Larva script
• Main challenge is to make sure you match corresponding entry and exit points
  • recursiveMethod() -entry
    • recursiveMethod() -entry
    • recursiveMethod() -exit
  • recursiveMethod() -exit
Points to consider

• It is easy to go from Model-Based Testing to RV
• Model-Based Testing not very commonplace
4. Invariant detection with Daikon

- Detect invariants from running testsuite
- Filter out invariants which no longer hold in modified testsuite
- Use model checking to detect invariants which are violated in update
Points to consider

• How can we adapt it to RV?
Approach 1: Gherkin → QC → Larva

- We know how to go from Gherkin to QC
- We know how to go from QC to Larva
- Go from Gherkin to Larva
Approach 2: Infer invariants

- Daikon – an invariant generation tool
Approach 2: Infer invariants

• Daikon – an invariant generation tool

```java
transactionsystem.UserAccount.deposit(double):::ENTER
this.opened == true
amount one of { 100.0, 500.0, 1000.0 }

transactionsystem.UserAccount.deposit(double):::EXIT
this.opened == orig(this.opened)
this.account_number == orig(this.account_number)
this.owner == orig(this.owner)
this.opened == true
this.account_number.toString == orig(this.account_number.toString)
this.balance > orig(this.balance)
this.balance >= orig(amount)
this.balance - orig(this.balance) - orig(amount) == 0
```
Approach 2: Infer invariants

1. Original program
2. Tests
3. Instrument and run
4. Data traces
5. Infer invariants
6. Invariants
7. Generate and instrument monitors
8. Program with runtime monitors
Approach 2: Infer invariants

Pattern match on deposit + Check postconditions if preconditions hold
Two main challenges

• Make monitors useful
  • Weaken preconditions
  • Tighten postconditions

• Avoid false negatives
Challenge – Weaken preconditions

Is this deliberate?

```java
transactionsystem.UserService.deposit(double):
  if (this.opened == true
  amount one of { 100.0, 500.0, 1000.0 }

  transactionsystem.UserService.deposit(double)::EXIT
  this.opened == orig(this.opened)
  this.account_number == orig(this.account_number)
  this.owner == orig(this.owner)
  this.opened == true
  this.account_number.toString == orig(this.account_number.toString)
  this.balance > orig(this.balance)
  this.balance >= orig(amount)
  this.balance - orig(this.balance) - orig(amount) == 0
```
Challenge – Weaken preconditions

```java
transactionsystem.UserAccount.deposit(double);  
this.opened == true  
amount one of { 100.0, 500.0, 1000.0 }
```

Is this deliberate?

```java
transactionsystem.UserAccount.deposit(double);  
this.opened == orig(this.opened)  
this.account_number == orig(this.account_number)  
this.owner == orig(this.owner)  
this.opened == true  
this.account_number.toString == orig(this.account_number.toString)  
this.balance > orig(this.balance)  
this.balance >= orig(amount)  
this.balance - orig(this.balance) - orig(amount) == 0
```

Missing test cases?
Challenge – Weaken preconditions

...
Challenge – Weaken preconditions

```java
transactionsystem.UserAccount.deposit(double):::ENTER
this.opened == true
amount one of { 100.0, 500.0, 1000.0 }

transactionsystem.UserAccount.deposit(double):::EXIT
this.opened == orig(this.opened)
this.account_number == orig(this.account_number)
this.owner == orig(this.owner)
this.opened == true
this.account_number.toString == orig(this.account_number.toString)
this.balance > orig(this.balance)
this.balance >= orig(amount)
this.balance - orig(this.balance) - orig(amount) == 0
```

Remove such invariants
Set appropriate threshold
A test case improvement problem

Refined invariants

Generate invariants

Insight on test suite

Intelligent test suggestion (boundary value analysis, etc)

Improve test suite
A test case improvement problem

Refined invariants

Generate invariants

Improve testsuite

When satisfied

Insight on testsuite

Use as monitors
A test case improvement problem

- Generate invariants
- Improve testsuite
- Use as monitors
- Refined invariants

Mutation testing

When satisfied

Insight on testsuite
Challenge – Avoiding false negatives

All behaviours

Good behaviours
Challenge – Avoiding false negatives

```java
transactionsystem.UserAccount.deposit(double):::ENTER
this.opened == true
amount one of { 100.0, 500.0, 1000.0 }

transactionsystem.UserAccount.deposit(double):::EXIT
this.opened == orig(this.opened)
this.account_number == orig(this.account_number)
this.owner == orig(this.owner)
this.opened == true
this.account_number.toString == orig(this.account_number.toString)
this.balance > orig(this.balance)
this.balance >= orig(amount)
this.balance - orig(this.balance) - orig(amount) == 0
```

amount > orig(this.balance)
Challenge – Avoiding false negatives

Generate test cases which purposefully try to violate the postcondition

```java
transactionsystem.UserAccount.deposit(double):::ENTER
this.opened == true
amount one of { 100.0, 500.0, 1000.0 }

transactionsystem.UserAccount.deposit(double):::EXIT
this.opened == orig(this.opened)
this.account_number == orig(this.account_number)
this.owner == orig(this.owner)
this.opened == true
this.account_number.toString == orig(this.account_number.toString)
this.balance > orig(this.balance)
this.balance >= orig(amount)
this.balance - orig(this.balance) - orig(amount) == 0
```

amount > orig(this.balance)
Approach 3: Combine testing and RV by design

- Specification of tests and monitors in a single language (like property-based testing but allowing some properties to be specified by examples)
  - If a precise specification is available, generate test cases automatically
  - If not, have test cases and specifications specified separately
Approach 3: Combine testing and design

• Specification of tests and monitors in a single language
  (like property-based testing but allowing some properties to be specified by examples)
  • If a precise specification is available, generate test cases automatically
  • If not, have test cases and specifications specified separately

E.g., balance' = balance + deposit
Automatically generates
200 = 150 + 50
350 = 290 + 60

E.g., balance' >= 0
Conclusion

• Generating monitors from tests is hard!
• 3 approaches being explored
• Still a lot of questions!