RML

Runtime Verification of Hash Code in Mutable Classes

Davide Ancona, Angelo Ferrando and Viviana Mascardi

DIBRIS, Università di Genova, Italy

UNIVERSITÀ DEGLI STUDI DI GENOVA

Outline

1. **Object equality and hash code**
2. **Runtime Verification and RML**
3. **Specification of safe hash sets**

UNIVERSITÀ DEGLI STUDI DI GENOVA

Contract for equality and hash code

Features of most object-oriented languages

- two different notions of equality
  - by reference, predefined (`==`)
  - weaker equality, user-defined (`equals`)

General contract in `java.lang.Object`

If two objects are equal, then the same hash code must be computed for them

Reason

Classes as `HashSet` or `HashMap` rely on `equals` and `hashCode`:

- `hashCode` is used to retrieve a specific bucket
- `equals` is used to find an element in such a specific bucket

Contract for equality and hash code

A stricter contract in `java.util.Set`

Great care must be exercised if mutable objects are used as set elements.

The behavior of a set is not specified if the value of an object is changed in a way that affects equals comparisons while the object is an element in the set.

A special case of this prohibition is that it is not permissible for a set to contain itself as an element.

Contract for equality and hash code

A simple example

```
var sset = new HashSet<Set<Integer>>();
var s = new HashSet<>(asList(1)); // s is {1}
sset.add(s); // sset is {{1}}
assert sset.contains(s);
s.remove(1);
assert sset.contains(s);
s.add(1);
assert sset.contains(s);
```

Contract for equality and hash code

A simple example

```
var sset = new HashSet<Set<Integer>>();
var s = new HashSet<>(asList(1)); // s is {1}
sset.add(s); // sset is {{1}}
assert sset.contains(s); // success
s.remove(1);
assert sset.contains(s); // failure
s.add(1);
assert sset.contains(s); // success
```

UNIVERSITÀ DEGLI STUDI DI GENOVA

Contract for equality and hash code

Another example

```
var sset = new HashSet<Set<Integer>>();
var s = new HashSet<>(asList(0)); // s is {0}
sset.add(s); // sset is {{0}}
assert sset.contains(s);
s.remove(0);
assert sset.contains(s);
s.add(0);
assert sset.contains(s);
```

UNIVERSITÀ DEGLI STUDI DI GENOVA

Contract for equality and hash code

Another example

```
var sset = new HashSet<Set<Integer>>();
var s = new HashSet<>(asList(0)); // s is {0}
sset.add(s); // sset is {{0}}
assert sset.contains(s); // success
s.remove(0);
assert sset.contains(s); // success!
s.add(0);
assert sset.contains(s); // success
```

Issues

- almost unpredictable code behavior
- non-deterministic behavior if `hashCode` depends on object references
  - object references may change from one execution to another
  - hash code needs not remain consistent from one execution to another

UNIVERSITÀ DEGLI STUDI DI GENOVA

Theory versus practice

Theory

- mutable classes should not redefine `equals`
- weaker contract: `hashCode` should not depend on “mutable” fields

Practice

- mutable classes of `java.util.Collection` do not satisfy such a contract
- similar problems in Kotlin and Scala, but not in C#

Aims

- verify that `Collection` objects are not modified while in a hash table
- proposed solution: Runtime Verification (RV)
- related work: study of `equals` and `hashCode` contract in Java collections [NelsonPearceNoble@TOOLS2010]

UNIVERSITÀ DEGLI STUDI DI GENOVA

Runtime verification

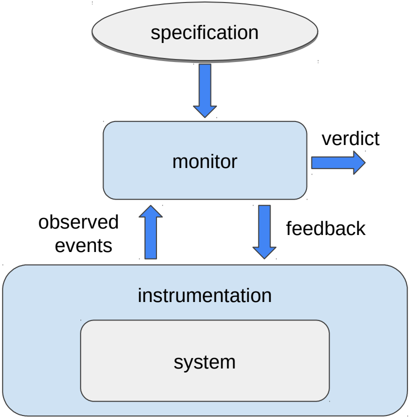
Definition

Runtime Verification (RV) is a verification technique that allows for checking whether a run of a system under scrutiny (SUS) satisfies or violates a given correctness property.

Main ingredients

- run = possibly infinite event trace
- instrumentation = generates the relevant events
- formal specification = a set of event traces
- monitor = generated from a specification, dynamically checks finite prefixes of a run

Runtime verification

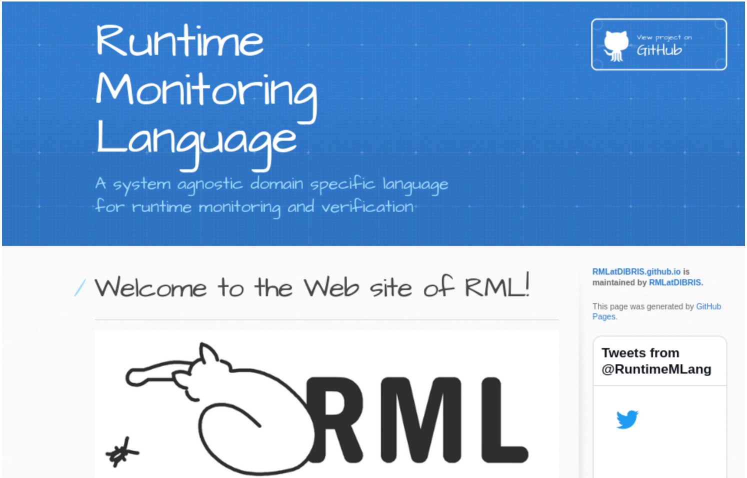


Why RV?

RV bridges the gap between formal verification and testing

- the notion of event trace abstracts over system runs
- RV offers error recovery, self-adaptation, and issues that go beyond software reliability
- some information available only at runtime
- the behavior of an application may depend heavily on the environment of the target system
- if security is an issue, RV provides cast-iron guarantees for properties that have been statically proved or tested

RML



RML Web page: https://rmlatdibris.github.io/

UNIVERSITÀ DEGLI STUDI DI GENOVA

RML

Main features of RML

- inspired by global session types
- based on formal languages: extension of deterministic CF grammars
- usability: developers are familiar with regular expressions and grammars
- expressive power: more expressive than deterministic CF grammars
- interoperability: clear separation between specification and instrumentation

UNIVERSITÀ DEGLI STUDI DI GENOVA

Structure of RML specifications

Four layers

- event types: relevant events
- trace expressions: primitive and derived operators for defining sets of event traces
- parametricity: existential quantification w.r.t. data carried by events
- genericity: specification with parameters to enhance modularity, reuse and expressive power

Events and event types in RML

An event trace

```
...
{"event":"func_pre","name":"add","targetId":5,"argIds":[13]}
{"event":"func_post","name":"add","res":true,"targetId":5,"argIds":[13]}
{"event":"func_pre","name":"remove","targetId":5,"argIds":[9]}
{"event":"func_post","name":"remove","res":true,"targetId":5,"argIds":[9]}
...
```

Event type declarations

```
add(hash_id,elem_id) matches {event:'func_post', targetId:hash_id, name:'add',
     argIds:[elem_id], res:true};
remove(hash_id,elem_id) matches {event:'func_post', targetId:hash_id,
     name:'remove', argIds:[elem_id], res:true};

modify(targ_id) matches add(targ_id,_) | remove(targ_id,_);
```

UNIVERSITÀ DEGLI STUDI DI GENOVA

RML specifications

Kleene star

recursion

```
Main = not_new_hash*
      {let hash_id;new_hash(hash_id)(SafeHashSet<hash_id> ∧ Main)}?;
```

existential quantification

concatenation

intersection

optionality

```
SafeHashTable<hash_id> = ...
```

parametric specification

UNIVERSITÀ DEGLI STUDI DI GENOVA

RML semantics

In a nutshell

- based on the notion of Brzozowski derivative
- defined by a labeled transition systems with rewriting rules
- labels are the monitored events
- the initial state is the specification of the property

RML semantics

$$\text{(par-t)}\ \frac{t \xrightarrow{e} t'; \sigma}{\{\text{let } x;\ t\} \xrightarrow{e} \sigma_{|x} t'; \sigma_{\setminus x}}\ x \in dom(\sigma)$$

$$\text{(par-f)}\ \frac{t \xrightarrow{e} t'; \sigma}{\{\text{let } x;\ t\} \xrightarrow{e} \{\text{let } x;\ t'\}; \sigma}\ x \notin dom(\sigma)$$

$$\text{(app)}\ \frac{\sigma t \xrightarrow{e} t'; \sigma'}{(\langle x_1, \ldots, x_n\rangle.t)\langle d_1, \ldots, d_n\rangle \xrightarrow{e} t'; \sigma'}\ \sigma = \{x_1 \mapsto ev(d_1), \ldots, x_n \mapsto ev(d_n)\}$$

$$\text{(cond-t)}\ \frac{t_1 \xrightarrow{e} t; \sigma}{\text{if } (d)\ t_1 \text{ else } t_2 \xrightarrow{e} t; \sigma}\ ev(d) = true$$

$$\text{(cond-f)}\ \frac{t_2 \xrightarrow{e} t; \sigma}{\text{if } (d)\ t_1 \text{ else } t_2 \xrightarrow{e} t; \sigma}\ ev(d) = false \qquad \text{(n-par)}\ \frac{t \not\xrightarrow{e}}{\{\text{let } x;\ t\} \not\xrightarrow{e}}$$

$$\text{(n-app)}\ \frac{\sigma t \not\xrightarrow{e}}{(\langle x_1, \ldots, x_n\rangle.t)\langle d_1, \ldots, d_n\rangle \not\xrightarrow{e}}\ \sigma = \{x_1 \mapsto ev(d_1), \ldots, x_n \mapsto ev(d_n)\}$$

$$\text{(n-cond-t)}\ \frac{t_1 \not\xrightarrow{e}}{\text{if } (d)\ t_1 \text{ else } t_2 \not\xrightarrow{e}}\ ev(d) = true$$

$$\text{(n-cond-f)}\ \frac{t_2 \not\xrightarrow{e}}{\text{if } (d)\ t_1 \text{ else } t_2 \not\xrightarrow{e}}\ ev(d) = false \qquad \text{(e-par)}\ \frac{E(t)}{E(\{\text{let } x;\ t\})}$$

$$\text{(e-app)}\ \frac{E(\sigma t)}{E((\langle x_1, \ldots, x_n\rangle.t)\langle d_1, \ldots, d_n\rangle)}\ \sigma = \{x_1 \mapsto ev(d_1), \ldots, x_n \mapsto ev(d_n)\}$$

$$\text{(e-cond-t)}\ \frac{E(t_1)}{E(\text{if } (d)\ t_1 \text{ else } t_2)}\ ev(d) = true$$

$$\text{(e-cond-f)}\ \frac{E(t_2)}{E(\text{if } (d)\ t_1 \text{ else } t_2)}\ ev(d) = false \qquad \text{(ne-par)}\ \frac{NE(t)}{NE(\{\text{let } x;\ t\})}$$

$$\text{(ne-app)}\ \frac{NE(\sigma t)}{NE((\langle x_1, \ldots, x_n\rangle.t)\langle d_1, \ldots, d_n\rangle)}\ \sigma = \{x_1 \mapsto ev(d_1), \ldots, x_n \mapsto ev(d_n)\}$$

$$\text{(ne-cond-t)}\ \frac{NE(t_1)}{NE(\text{if } (d)\ t_1 \text{ else } t_2)}\ ev(d) = true$$

$$\text{(ne-cond-f)}\ \frac{NE(t_2)}{NE(\text{if } (d)\ t_1 \text{ else } t_2)}\ ev(d) = false$$

Specification of safe hash sets

Declaration of event types

```
new_hash(hash_id) matches
  {event:'func_post', name:'HashSet', resultId:hash_id};
not_new_hash not matches new_hash(_);

add(hash_id,elem_id) matches
  {event:'func_post', targetId:hash_id, name:'add',
   argIds:[elem_id], res:true};
not_add(hash_id) not matches add(hash_id,_);

remove(hash_id,elem_id) matches
  {event:'func_post', targetId:hash_id, name:'remove',
   argIds:[elem_id], res:true};

modify(targ_id) matches add(targ_id,_) | remove(targ_id,_);

not_modify_remove(hash_id,elem_id) not matches
  modify(elem_id) | remove(hash_id,elem_id);
op(hash_id,elem_id) matches
  {targetId:hash_id} | {targetId:elem_id};
```

UNIVERSITÀ DEGLI STUDI DI GENOVA

Specification of safe hash sets

Whole specification

```
Main = not_new_hash*
  {let hash_id;new_hash(hash_id)(SafeHashSet<hash_id> ∧ Main)}?;

SafeHashSet<hash_id> = not_add(hash_id)*
  {let elem_id;add(hash_id,elem_id)(SafeHashElem<hash_id,elem_id> ∧
     SafeHashSet<hash_id>)}?;

SafeHashElem<hash_id,elem_id> = not_modify_remove(hash_id,elem_id)*
     (remove(hash_id,elem_id) all)?;
```

UNIVERSITÀ DEGLI STUDI DI GENOVA

Monitor at work

Events

- new hash set with id 5
- new hash set with id 9
- insertion of set with id 9 into set with id 5

Reached state

```
(SafeHashElem<5,9> ∧ SafeHashSet<5>) ∧ (SafeHashSet<9> ∧ Main);
```

Monitor at work

Preliminary experiments

- aim: validation of the specification
- on event traces that simulate the execution of simple Java programs

Monitor at work

Preliminary experiments: example

```
var sset = new HashSet<Set<Integer>>();
var s1 = new HashSet<Integer>();
var s2 = new HashSet<Integer>();
s1.add(1);
s2.add(2);
sset.add(s1);
s1.contains(1);
s1.add(1);
sset.add(s2);
sset.remove(s1);
//s2.remove(2);
s1.remove(1);
s2.remove(1);
sset.remove(s2);
s1.add(1);
s2.add(2);
```

UNIVERSITÀ DEGLI STUDI DI GENOVA

Generalization of the specification

Extension to other methods and classes

```
new_hash(hash_id) matches {event:'func_post', name:'HashSet' |
    'HashMap', resultId:hash_id};

add(hash_id,elem_id) matches // addition to a set
  {event:'func_post', targetId:hash_id, name:'add',
   argIds:[elem_id], res:true}
 |   // addition to a map
  {event:'func_post', targetId:hash_id, name:'put',
   argIds:[elem_id,_]};

clear(hash_id) matches  {event:'func_pre', targetId:hash_id,
    name:'clear'};
```

Remark:

- with `put` and `clear` is not possible to monitor modification accurately
- false positives are possible
- the same specification can be used also for Kotlin and Scala

UNIVERSITÀ DEGLI STUDI DI GENOVA

Assessment of the approach

Experiments with real Java applications

- scalability
- ability to detect bugs

Q&A time

Thank you!