# Runtime Verification of Hash Code in Mutable Classes

Davide Ancona
Angelo Ferrando
Viviana Mascardi
davide.ancona@unige.it
angelo.ferrando@unige.it
viviana.mascardi@unige.it
DIBRIS, Università di Genova
Italy

## ABSTRACT

Most mainstream object-oriented languages provide a notion of equality between objects which can be customized to be weaker than reference equality, and which is coupled with the customizable notion of object hash code. This feature is so pervasive in object-oriented code that incorrect redefinition or use of equality and hash code may have a serious impact on software reliability and safety.

Despite redefinition of equality and hash code in mutable classes is unsafe, many widely used API libraries do that in Java and other similar languages. When objects of such classes are used as keys in hash tables, programs may exhibit unexpected and unpredictable behavior. In this paper we propose a runtime verification solution to avoid or at least mitigate this issue.

Our proposal uses RML, a rewriting-based domain specific language for runtime verification which is independent from code instrumentation and the programming language used to develop the software to be verified.

## CCS CONCEPTS

• **Software and its engineering → Formal software verification**; *Software testing and debugging*.

## KEYWORDS

object-oriented languages, hash code, mutable classes, runtime verification

**ACM Reference Format:**
Davide Ancona, Angelo Ferrando, and Viviana Mascardi. 2023. Runtime Verification of Hash Code in Mutable Classes. In *Proceedings of the 25th ACM International Workshop on Formal Techniques for Java-like Programs (FTfJP '23), July 18, 2023, Seattle, WA, USA*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3605156.3606452

## 1 INTRODUCTION

Most mainstream object-oriented languages provide a notion of equality between objects which can be customized to be weaker than reference equality, and which is coupled with the customizable notion of object hash code [6]. Such two notions are provided through two corresponding methods defined in the predefined class `Object` which is at the root of the inheritance hierarchy; hence, they are inherited or can be redefined in any class, and are callable on any type of object.

For this reason, they are pervasive in object-oriented code and the correct functioning of some features in many libraries rely on them; hence, their incorrect redefinition or use may have a serious impact on software reliability and safety.

A classical example of useful redefinition of equality is for value classes, where typically a notion of logical equality is needed which differs from reference equality. Obeying the general contract for equality is challenging, and equality redefinition invalidates the general contract for computing object hash codes [6].

Indeed, implementations of hash tables typically use equality and object hash codes, therefore a general contract has to be satisfied: if two objects are equal, then the same hash code must be computed for them.

If this requirement is not satisfied, then hash tables fail to behave correctly. Indeed, to find an element in a hash table, its hash code is computed to identify its bucket, then equality is used to test whether the element is contained in such a bucket. If an equal element is already contained in the hash table, but in a different bucket, because the computed hash code is different, then the element cannot be found.

While this problem is well known and there have been some attempts to detect it with verification techniques [6, 23], hash code redefinition for mutable classes has been overlooked. When objects of such classes are used as keys in hash tables, programs may exhibit unexpected and unpredictable behavior. Indeed, if an object is modified while contained in a hash table, then most likely the same object can no longer be found in the table even though no operations have been performed on the hash table.

Redefinition of equality and hash code in mutable classes is unsafe, as pointed out in the documentation for `java.util.Set` [22] and, similarly, `java.util.Map`: "*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 manner 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.*"

Despite this note, many widely used API libraries do that in Java and other similar languages. Verifying that mutable objects with redefined hash code are used correctly in hash tables is not


*FTfJP '23, July 18, 2023, Seattle, WA, USA*

ACM ISBN 979-8-4007-0246-4/23/07…$15.00
https://doi.org/10.1145/3605156.3606452

an easy task, because state modification needs to be tracked with a certain precision and rather complex control-oriented properties [1, 2] have to be ensured.

In this paper we present a solution based on Runtime Verification (RV), a dynamic verification technique where a single execution of the system under scrutiny (SUS) is abstracted by an event trace which is checked by a monitor compiled from the formal specification defining the correct behavior of the SUS.

Events are usually generated by instrumented code of the SUS, and logged or directly sent to the monitor. Although specification of properties and code instrumentation can be mixed together, decoupling the two activities favors abstraction, reuse and interoperability of the generated monitors.

Monitors can be *offline* or *online*; in offline RV a trace is typically generated by the instrumented SUS and stored into a log file and then is analyzed by the monitor. In online RV traces are analyzed real-time to allow error detection to trigger specific actions on the SUS. Offline RV [10] is a useful solution to integrate other approaches as debugging and testing; online RV can be employed to allow error recovery in critical scenarios, providing that such a choice is compatible with the overhead of code instrumentation and of the monitor execution.

RV is complementary to formal verification and testing: as formal verification, RV is based on a specification formalism; as happens for software testing, it scales well to real systems and complex properties, but cannot guarantee exhaustiveness. Differently from testing, it is particularly useful to ensure control-oriented properties [1, 2] and detect errors due to non-deterministic behavior [16, 26]. Furthermore, online monitoring allows runtime contract enforcement, fault protection and automatic program repair. Finally, several RV tools are based on abstract and intuitive specification languages that can be easily mastered by the user and favor system agnosticism, portability, reuse, and interoperability.

Our proposed solution is based on offline RV and uses RML[1], a rewriting-based Domain Specific Language (DSL) for RV which allows definition of formal specifications independently of code instrumentation and of the programming language used to develop the software to be verified. The choice of RML makes our solution easily portable to different Java-like languages. Offline RV has been preferred over online RV because the main aim is to detect unsafe use of hash tables; this allows also a simpler solution which minimize overhead.

The paper is structured as follows. Section 2 introduces the problem in detail and analyzes it in the context of several mainstream object-oriented languages, Section 3 provides an introduction to RML, Section 4 presents the proposed solution and discussed possible generalization, Section 5 is devoted to the related work and conclusions.

## 2 HASH CODE AND MUTABLE CLASSES

Correctness issues concerned with the relationship between methods `equals` and `hashCode` are well-known in Java [6, 23] and other object-oriented languages as C#, Kotlin, Scala, and Python supporting redefinition of object equality and hash code; however, less attention has been devoted to the potentially dangerous effects of the redefinition of `hashCode` in mutable classes when their instances are used in container objects implemented with hash tables.

In Java (and Kotlin and Scala as well) such a problem is more serious because the widely used mutable classes of `java.util` implementing interfaces as `Collection` and `Map`[2] redefine method `hashCode` as their instances where immutable (i.e. value objects).

Let us consider an example, where for simplicity a unique class is used both for containers (which use a hash table) and container elements since `HashTable` is a mutable class redefining `hashCode()` and implementing `Collection`. Similar examples can be built with other types of contained elements, for instance, linked lists.

```
1 var sset = new HashSet<Set<Integer>>();
2 var s = new HashSet<>(asList(1,2,3));
3 sset.add(s); // sset is {{1,2,3}}
4 assert sset.contains(s); // success
5 s.remove(1);
6 assert sset.contains(s); // failure
7 s.add(1);
8 assert sset.contains(s); // success
```

Two sets are created with class `HashSet`; in such a class, method `hashCode` of the elements is used to identify the bucket where they are stored in the hash table, and method `equals` to search them in the bucket. After execution of the first three lines `sset` contains `s` as stated by the successful assertion at line 4; in turn, `s` contains the three elements of type `Integer` corresponding to 1, 2 and 3.

At line 5 element 1 is removed from `s` and at the next line the same assertion is checked again; this time the assertion fails, although no method has been invoked on `sset` and, hence, its state should be the same as in the previous assertion.

This does not come to surprise once one looks at the documentation and discovers that methods `equals` and `hashCode` are overridden in `HashSet`[3] to depend on all the elements contained in the set. As a consequence, the integer returned by `s.hashCode()` changes after removing element 1 from `s` and, hence, the assertion at line 6 fails because `s` is searched in the wrong bucket of the hash table of `sset`. As a matter of fact, the assertions at line 4, 6, and 8 depend on the state of both `sset` and `s`.

What is worst is that the outcome of the assertion at line 6 is unpredictable; indeed, it is still possible, although unlikely, that the searched bucket is the right one after removing the element from `s`. In this case the assertion succeeds. Finally, considering also that the general contract states that the hash code needs not remain consistent from one execution of an application to another execution of the same application, we can state that the behavior of assertion at line 6 can be non-deterministic.

Once element 1 is inserted back in `s`, the computed hash code of the object is again that at line 4, hence assertion at line 8 succeeds.

Putting it all together, the main source of the problem consists in the fact that in the mutable classes implementing `Collection` the receiver in the redefined methods `equals` and `hashCode` is considered as an immutable object. This should be avoided for all mutable classes whose objects may be used as keys in hash tables, because the consequence is that the object should be "frozen" until is no longer in the table to avoid misbehavior as described above. In case

[1] https://rmlatdibris.github.io/.

[2] For brevity we refer to types in `java.util` with their simple names.

[3] Actually, in its direct abstract superclass `AbstractSet`.

an application does not follow this good practice, code should be verified to detect issues that leads to inconsistencies in hash tables.

While in C# and Python it is still possible for the programmers to define mutable classes where the corresponding methods for equality and hash code are not well-behaved w.r.t. hash tables, predefined mutable collections do not exhibit the problems of Java, Kotlin and Scala.

```
var sset = new HashSet<ISet<int>>();
var s = new HashSet<int>(new int[] { 1, 2, 3 });
sset.Add(s);
Debug.Assert(sset.Contains(s)); // success
s.Remove(1);
Debug.Assert(sset.Contains(s)); // success
s.Add(1);
Debug.Assert(sset.Contains(s)); // success
```

In the C# code snippet above all assertions succeed simply because methods `Equals` and `GetHashCode` are not redefined in mutable classes implementing collections, but inherited from `Object`.

Interestingly, in Python for the predefined types `set`, `list` and `dict` another strategy has been adopted: the objects are compared[4] as immutable objects, but computing their hash code throws an exception:

```
sset=set()
s1=set([1,2,3])
s2=set([1,2,3])
assert s1==s2 // success
sset.add(s1) # TypeError: unhashable type: 'set'
```

In this way it is not possible to use sets, lists and dictionaries as hash table keys; this a drastic solution which prevents, for instance, to easily manage sets of sets or lists.

Finally, JavaScript does not support redefinition of object equality and hash code, hence does not exhibit the issue shown above.

## 3 RML

RML [4] is a rewriting-based DSL for RV which allows developers to define formal specifications independently of code instrumentation.

It is based on the notion of *event type* (denoting a set of events) and *trace expression* (denoting a set of event traces), and it is implemented by a compiler, which generates monitors able to run independently of the SUS and of its instrumentation.

In RML an *event* is any observation relevant for monitoring the SUS. Events are represented in a general way with object literals and consist of properties which identify the type of event and the data associated with it. For instance,

```
{event:"func_post",targetId:9,name:"add",
    res:true,args:[1]}
```

represents the event 'call to method `add` on target object with id 9 and with argument 1 has returned value **`true`**'. Another type of events which are often useful to monitor is `'func_pre'`, that is, entering a constructor or method call; of course, in this case, no information on the returned value can be provided. Depending on the features of the instrumentation tool, other finer grained types, as reading or updating a field, can be used, but at the cost of making specifications more coupled with the specific application that needs to be verified, and, hence, less reusable and portable.

[4]In Python object equality can be redefined through method `__eq__` to change the behavior of the == operator.

An RML specification defines the set of event traces expected from correct runs of the SUS; the monitor automatically generated from such a specification checks that the trace generated by a single run of the SUS belongs to such a set.

The basic blocks which constitute an RML specification are *patterns* built from *event types* defining sets of events.

Event types are defined with clauses:

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

In this example `add` matches events parametric in the ids `hash_id` and `elem_id` of the target and argument of the call. While property `args` is useful when arguments are primitive values, `argIds` is used when arguments are objects, denoted by their unique id; similarly, for the returned value the two properties `res` and `resultId` are available.

RML allows also the definition of event types derived from others:

```
not_add(hash_id) not matches add(hash_id,_);
op(hash_id,elem_id) matches {targetId:hash_id}|{targetId:elem_id};
```

The event pattern `not_add(hash_id)` matches all events which do not correspond to the return from method `add` called on target `hash_id`; the wildcard `_` is used when a value is not relevant for the definition of the event type.

The event pattern `op(hash_id,elem_id)` matches all events matching either `{targetId:hash_id}` or `{targetId:elem_id}`, that is, all calls on target `hash_id` or `elem_id`.

The basic layer of RML are expressions that define sets of event traces and built by combining together event patterns with primitive and derived operators. The former kind of operators includes, among others, the following binary operators on sets of event traces: *concatenation* (denoted by juxtaposition), *intersection* `/\`, *union* `\/` and *shuffle* `|`. Other useful derivable operators are available, including the standard postfix operators `?`, `+` and `*`, borrowed from regular expressions, the constant `all`, which denotes the universe of all traces, and the conditional filter operator `_ >> _ : _`.

The formal semantics of trace expressions is defined in terms of a labeled transition system [4].

As a very simple example, the specification `Main` in Figure 1 defines the set of event traces starting with a call to a constructor of class `HashSet` returning the object id 42, followed by zero or more calls to method `add` on target id 42 with returned value **`true`** and ending with a call to method `remove` on the same target id with returned value **`true`**.

```
new_hash(hash_id) matches {event:'func_post', name:'HashSet',
      resultId:hash_id};
remove(hash_id) matches {event:'func_post', targetId:hash_id, name
      :'remove', res:true};
add(hash_id) matches {event:'func_post', targetId:hash_id, name:'
      add', res:true};

Main = new_hash(42) add(42)* remove(42);
```

**Figure 1: Example of specification.**

The expression `new_hash(42) add(42)* remove(42)` is of very limited use because it refers to a specific object id; however, RML provides a **`let`** construct [3] for declaring existentially quantified variables. With such an abstraction and the shuffle operator, it is possible to write a *parametric* specification working for any instance of class `HashSet`:

```
Main = {let hash_id; new_hash(hash_id) (add(hash_id)* remove(
      hash_id) | Main)};
```

When the first event matches `new_hash(hash_id)`, `hash_id` is bound to the specific id returned by the constructor in the two occurences on the left-hand-side of the shuffle. The binding does not affect the recursive use of `Main` because its nested **let** declaration masks the outer one, thus allowing to properly verify the specified property for any new instance of the class.

It is worth noting that such a specification pattern where recursion occurs on one side of shuffle (or intersection, as shown in the next section) is quite useful for specifying several kinds of properties [4] which cannot be specified with regular expressions (and hence with LTL which is less expressive [27]). Indeed, while regular expressions are closed w.r.t. shuffle, they are not w.r.t. iterated shuffle [15]; intersection allows even more expressive power since context-free languages are not closed w.r.t. such an operation [4].

RML provides a further abstract layer with *generic* specifications, to enhance modularity and reuse and increase its expressive power [4]. With the generic `Spec<hash_id>`, the specification above can be generalized as follows to make it more readable and possibly reusable:

```
Spec<hash_id> =  add(hash_id)* remove(hash_id);
Main = {let hash_id; new_hash(hash_id) ( Spec<hash_id> | Main)};
```

## 4 A SPECIFICATION OF SAFE USE OF COLLECTIONS IN HASH SETS

In this section we show how it is possible to define a specification in RML for dynamically verifying that hash sets and their elements of type `Collection` are managed correctly to avoid the issue highlighted by the examples in Section 2.

The only methods of `Collection<E>` that can modify the state of a collection are `add(E)` and `remove(E)`; other methods, as `addAll` and `removeAll`, are defined in terms of the primitive ones `add` and `remove`, hence the specification we consider here covers also them. However, there are additional methods contained in subtypes of collection, consider for instance method `add(int,E)` and `remove(int)` of `List`, which cannot be monitored through `add(E)` and `remove(E)`. Possible generalization of the solution presented here are discussed in the last part of this section.

### 4.1 Events and Event Types

Since the specification has to verify hash sets, creation of instances of `HashTable` is a relevant event to be monitored.

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

Interestingly enough, creation of the elements inserted in the sets need not to be monitored, unless they are hash sets themselves; as seen in the examples in Section 2, the main point is to trace addition to new elements in sets.

An interesting feature of `add` and `remove` of `Collection` is that they both return true if and only if the operation modifies the collection, hence modifications can be easily monitored at runtime, and it is possible to write a specification based only on events of type `'func_post'`.

```
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};
```

After an event matches `add(hash_id,elem_id)`, the specification needs to verify that element `elem_id`, which has just been inserted in the hash set `hash_id`, is not modified until the element is removed from the set, that is, an event matching `remove(hash_id,elem_id)` occurs. The fact that event type `add(hash_id,elem_id)` requires the returned value to be **true** is important to avoid useless checks on elements that are already contained in the set. The same constraint for `remove(hash_id,elem_id)` is less important here because, by construction (see the specification below), the first event matching `remove(hash_id,elem_id)` must necessarily be for a call returning value true, assuming correct the implementation of `remove`.

The returned value true in the definition of `add(hash_id,elem_id)` and `remove(hash_id,elem_id)` is important to avoid false positives when checking that elements in a hash set are not modified. Indeed, the only harmful calls to `add` and `remove` are those that effectively change the state of elements and, hence, their hash codes.

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

There still might be some false positive in (the quite unlikely) case a modification of the element does not change the bucket of the hash table where it should be contained, as already observed in Section 2. However, this would be hard to be checked and the policy to ban any attempt at modifying elements in a hash set is safer. One might also adopt the stricter policy of prohibiting any call to `add` and `remove` by omitting the requirement `res:true` in the definition of `add(hash_id,elem_id)` and `remove(hash_id,elem_id)`.

### 4.2 Specification

The whole specification of safe use of collections in hash sets can be found in Figure 2.

The first part of the specification contains the definitions of all needed event types. The main types have been already introduced, but there are also some auxiliary types, most of them derived.

The definition of the main specification `Main` is recursive, similarly as shown in the example of parametric specification in Section 2, but the intersection operation is used instead of the shuffle. This is necessary because several hash sets may coexist and modification of a collection has to be checked for all of them, since such a collection could be contained in any of them.

Before a new hash set is created (`new_hash(hash_id)`), several other events relevant for `SafeHashTable` or `SafeHashElem` may occur (trace expression `not_new_hash*`). After a new hash table is created with id `hash_id`, the specification `SafeHashTable` checks the correct behavior of the newly created set and `Main` manages creation of new hash sets (trace expression `SafeHashTable<hash_id> /\ Main`).

For instance, after creation of two hash sets with id 5 and 9, the specification defined by `Main` is rewritten into

```
(SafeHashTable<5> /\ (SafeHashTable<9> /\ Main)?)?;
```

Such a specification represents the current state of the monitor generated from `Main`.

```
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};

Main = not_new_hash*
  {let hash_id;new_hash(hash_id)
   (SafeHashTable<hash_id> /\ Main)
  }?;
SafeHashTable<hash_id> = not_add(hash_id)*
  {let elem_id;add(hash_id,elem_id)
   (SafeHashElem<hash_id,elem_id> /\ SafeHashTable<hash_id>)
  }?;
SafeHashElem<hash_id,elem_id> =
  not_modify_remove(hash_id,elem_id)* (remove(hash_id,elem_id) all
     )?;
```

**Figure 2: Specification of safe hash sets.**

The regular expression operator `?` (optionality) is used to cover cases where a specific run of the SUS does not create any hash table.

The definition of `SafeHashTable` follows the same pattern as `Main`. Before a new element is added to the hash set `hash_id` (event pattern `add(hash_id,elem_id)`), several other events relevant for the generic specification `SafeHashTable` may occur (`not_add(hash_id)*`).

After a new element with id `elem_id` is added, the specification `SafeHashElem` checks that `elem_id` is not modified until it is removed from `hash_id` and `SafeHashTable` manages addition of new elements to the hash sets (trace expression `SafeHashElem<hash_id,elem_id> /\ SafeHashTable<hash_id>`).

`SafeHashElem<hash_id,elem_id>` is defined by the trace expression

```
not_modify_remove(hash_id,elem_id)* (remove(hash_id,
     elem_id) all)?
```

It defines the set of traces where modifications on `elem_id` are not allowed before an event matching `remove(hash_id,elem_id)` occurs. The pattern `not_modify_remove(hash_id,elem_id)` matches all events that do not match `modify(elem_id)` and `remove(hash_id,elem_id)`. The latter constraint is needed to ensure that removal of `elem_id` is checked only once, that is, the corresponding event matches pattern `remove(hash_id,elem_id)`. The predefined constant `all` (the universe of all traces) specifies that no further checks are needed once `elem_id` has been removed from `hash_id`. The use of the optional operator in `(remove(hash_id,elem_id) all)?` reflects the fact that the specification is used to monitor a safety property: removing `elem_id` from `hash_id` is a necessary condition for considering safe all those operations that modify `elem_id`, but execution can safely terminate even though `elem_id` has not been removed, if no modification of `elem_id` occurred after its insertion in `hash_id`.

Reconsidering the rewriting example above, after creation of two hash sets with id 5 and 9, and insertion of the set with id 9 into the set with id 5, the specification defined by `Main` is rewritten into

```
((SafeHashElem<5,9> /\ SafeHashTable<5>)? /\ (SafeHashTable<9> /\
     Main)?)?;
```

## 4.3 Possible Generalization of the Specification

The specification above deals exclusively with objects of type `HashSet` for what concerns classes based on hash tables, and objects of type `Collection` for what concerns objects of mutable classes that redefine `hashCode`.

### *4.3.1 Classes Based on Hash Tables.*

`HashMap` is a widely used class of the Java API. To consider also this class, some event types, as `new_hash` or `add`, need to be generalized.

```
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,_], res:null};
```

Differently from `HashSet`, for some methods it is more challenging to keep exact track of the keys contained in a map, because **`null`** values are allowed. For instance, `put(key,value)` returns the previous value associated with `key` (which may include **`null`**) or **`null`** if there was no mapping for `key`. Hence, if we require the result to be **`null`** as done above, then in some cases checking that a key is not modified can be duplicated. This can be more problematic when hash maps are elements of a hash set (see below).

### *4.3.2 Mutable Classes Redefining* `hashCode`.

Package `java.util` provides a number of mutable classes where `hashCode` is redefined.

The specification above does not take into account that several classes implement subinterfaces of `Collection`, such as `List`. For instance, a stack can be modified with methods `pop` and `push` and the target object is always modified when the method is called.

In most cases the required extension to the specification does not pose any challenge. However, there are some methods, as `set (index,elem)` of `List`, for which it is not easy to test whether the target object has been really modified. For instance, the method may replace an element in a list with the same object, or an equal object. In that case, the verification detects a false positive.

In `java.util` there are also mutable classes redefining `hashCode` and implementing interfaces different from `Collection`. We have already considered class `HashMap` which implements `Map`. In a scenario where a hash set contains a hash map, modifications on the hash map should be avoided while it is contained in the set. The `put` method above has similar problems as method `set`, hence the verification may be less accurate in this case.

### *4.3.3 Preliminary Experiments.*

We have conducted preliminary experiments to test the correctness of the specification with the offline monitor generated from it by the RML compiler. The monitor has been run on event traces that simulate the execution of simple Java programs, as shown below, and that have been stored in log files.

```
1 var sset = new HashSet<Set<Integer>>();
2 var s1 = new HashSet<Integer>();
3 var s2 = new HashSet<Integer>();
4 s1.add(1);
5 s2.add(2);
6 sset.add(s1);
7 s1.contains(1);
8 s1.add(1);
9 sset.add(s2);
10 sset.remove(s1);
11 //s2.remove(2);
12 s1.remove(1);
13 s2.remove(1);
14 sset.remove(s2);
15 s1.add(1);
16 s2.add(2);
```

The only critical instruction has been commented. Indeed, at line 11 an element of `s2` is removed, while `s2` is in the hash set `sset`.

With line 11 commented, the corresponding trace is accepted by the monitor, as expected. In particular, the monitor recognizes that the two methods `contains` and `add` called on `s1` while contained in `sset` (lines 7 and 8) are safe because do not change the state of `s1`. Similarly, line 13 for `s2`.

Line 12 is safe, although `s1.remove(1)` changes the state of the object, because `s1` no longer belongs to `sset`; the same consideration for `s1` applies to line 15 and line 16 for `s2`.

If the comment at line 13 is removed, then the corresponding trace is rejected.

## 5 RELATED WORK AND CONCLUSIONS

Although in previous work [22] experiments have been conducted on real Java programs to understand to what extent mutable objects with redefined hash code are used correctly in hash tables, we are not aware of papers where such a property has been formalized so that it can be dynamically verified on programs for all those object-oriented languages for which the issue may manifest.

In the preliminary conducted experiments traces have been generated by a simple script and not through Java code instrumentation. A simple solution to analyze real Java code is exploiting the Java Logging API offered by module `java.logging`.

Other more sophisticated tools have been proposed in literature.

Early attempts to visualize Java programs date back to the end of the millennium. They were initially motivated by teaching reasons [14], and became soon a fundamental engineering step for developing correct and safe Java applications [25, 28]. The Java visualization research strand is still active [18, 24] but since – in order to visualize a program behavior – it is necessary to trace it [21], most efforts are currently oriented towards the more general problem of Java tracing.

JavaMop [9] is a tool based on AspectJ which allows users to specify and monitor properties in Java programs.

Different approaches to tracing exist, mainly depending on which part of the Java architecture, the bytecode, the source code, the JVM, is modified or instrumented to make the tracing possible.

In a work dating back 2001 [5], Bechini and Prete present a solution for tracing and replaying Java concurrent applications based on the automatic instrumentation of the original source code.

A less invasive approach is MuTT [19] that works on top of JPDA (the Java Platform Debugger Architecture, available for old JDKs) and exploits JPDA features to collect the run-time information of multi-threaded Java programs without source code or JVM instrumentation.

More recently, JBInsTrace [8] computes complex dynamic metrics used to categorize programs according to dynamic metrics related to program size and structure, use of data structures, use of polymorphism, memory footprint and concurrency. To this aim, JBInsTrace instruments and traces Java bytecode. It does not alter the JVM and does not statically modify class files.

When tracing takes place while the program is running, the effect of tracing is indeed a runtime monitoring of the program's behavior or, using the terminology adopted in this paper, its runtime verification.

Indeed, runtime verification of Java programs started to be addressed in 2001, when the Java PathExplorer was developed [17]. Java PathExplorer tested the execution traces of the Java program against high level specifications expressed as temporal logic formulae. An initial prototype of the tool was applied to the executive module of the planetary Rover K9, developed at NASA Ames.

JASSDA [7] was developed one year after Java PathExplorer. It is a RV framework for Java programs based on CSP-like specifications and implemented in Java. JASSDA is very simple and does not support concatenation; parametricity is obtained through slicing.

PQL [20] is an expressive language supporting RV of open-source Java applications that allows specifications of properties covering the closure of context-free languages combined with intersection; however, it does not support shuffle, and parametricity. Its implementation is based on Java, Python and DataLog.

LARVA [12] is a RV tool expressly designed for checking real-time properties of Java programs. Properties are specified in DATEs [11] based on an extension of timed automata; in particular, it supports symbolic states to guard transitions, replication of automata, and CCS-like communication between automata. LARVA is implemented in Java and code instrumentation is based on AspectJ.

SAGA [13] is another framework for RV of Java programs based on attribute grammars. With attribute grammars it is possible to support parametricity and to mix specifications with code instrumentation by exploring the full computational power of Java. Its implementation exploits Java, ANTLR and Rascal.

Differently to the other tools, RML allows generic specifications fully independent from Java. The specification provided in Section 4 can be reused for other Java-like languages, except for some renaming and adjustment in the definition of event types, needed because of the different method signatures used in the libraries.

For what concerns future work, once traces can be generated from real Java programs with a specific instrumentation tool, two challenges should be investigated: benchmarks with traces generated from real programs have to be considered, to understand whether the approach scales; experiments with Java programs extensively using hash tables should be conducted to understand how many true and false positives can be detected to assess the effectiveness of the approach.

## ACKNOWLEDGMENTS

This work was partially funded by the MUR project "T-LADIES" (PRIN 2020TL3X8X).

## REFERENCES


[1] Wolfgang Ahrendt, Jesús Mauricio Chimento, Gordon J. Pace, and Gerardo Schneider. 2017. Verifying data- and control-oriented properties combining static and runtime verification: theory and tools. *Formal Methods in System Design* 51, 1 (2017), 200–265.

[2] Davide Ancona, Francesco Dagnino, and Luca Franceschini. 2018. A formalism for specification of Java API interfaces. In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops, ISSTA 2018, Amsterdam, Netherlands, July 16-21, 2018*. 24–26. https://doi.org/10.1145/3236454.3236476

[3] Davide Ancona, Angelo Ferrando, and Viviana Mascardi. 2017. Parametric Runtime Verification of Multiagent Systems. In *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems, AAMAS 2017, São Paulo, Brazil, May 8-12, 2017*. 1457–1459.

[4] Davide Ancona, Luca Franceschini, Angelo Ferrando, and Viviana Mascardi. 2021. RML: Theory and practice of a domain specific language for runtime verification. *Science of Computer Programming* 205 (2021), 102610. https://doi.org/10.1016/j.scico.2021.102610

[5] Alessio Bechini and Cosimo Antonio Prete. 2001. Behavior investigation of concurrent Java programs: an approach based on source-code instrumentation. *Future Gener. Comput. Syst.* 18, 2 (2001), 307–316. https://doi.org/10.1016/S0167-739X(00)00095-9

[6] Joshua Bloch. 2018. *Effective Java* (3 ed.). Addison-Wesley.

[7] Mark Brörkens and Michael Möller. 2002. Dynamic Event Generation for Runtime Checking using the JDI. *Electr. Notes Theor. Comput. Sci.* 70, 4 (2002), 21–35.

[8] Pierre Caserta and Olivier Zendra. 2014. JBInsTrace: A tracer of Java and JRE classes at basic-block granularity by dynamically instrumenting bytecode. *Sci. Comput. Program.* 79 (2014), 116–125. https://doi.org/10.1016/j.scico.2012.02.004

[9] Feng Chen, Marcelo d'Amorim, and Grigore Roşu. 2006. Checking and Correcting Behaviors of Java Programs at Runtime with Java-MOP. *Electronic Notes in Theoretical Computer Science* 144, 4 (2006), 3–20. https://doi.org/10.1016/j.entcs.2006.02.002 Proceedings of the Fifth Workshop on Runtime Verification (RV 2005).

[10] Christian Colombo and Gordon J. Pace. 2022. *Offline Runtime Verification*. Springer International Publishing, 155–163. https://doi.org/10.1007/978-3-031-09268-8_12

[11] Christian Colombo, Gordon J. Pace, and Gerardo Schneider. 2008. Dynamic Event-Based Runtime Monitoring of Real-Time and Contextual Properties. In *Formal Methods for Industrial Critical Systems, 13th International Workshop, FMICS 2008, L'Aquila, Italy, September 15-16, 2008, Revised Selected Papers*. 135–149.

[12] Christian Colombo, Gordon J. Pace, and Gerardo Schneider. 2009. LARVA – Safer Monitoring of Real-Time Java Programs. In *SEFM 2009*. 33–37.

[13] Frank S. de Boer and Stijn de Gouw. 2014. Combining Monitoring with Run-Time Assertion Checking. In *SFM 2014*. 217–262.

[14] Herbert L Dershem, Daisy Erin Parker, and Rebecca Weinhold. 1999. A Java function visualizer. *Journal of Computing in Small Colleges* 15, 1 (1999), 220–230.

[15] Nils Erik Flick and Manfred Kudlek. 2012. On a Hierarchy of Languages with Catenation and Shuffle. In *Developments in Language Theory - 16th International Conference, DLT 2012, Taipei, Taiwan, August 14-17, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7410)*, Hsu-Chun Yen and Oscar H. Ibarra (Eds.). Springer, 452–458. https://doi.org/10.1007/978-3-642-31653-1_40

[16] Klaus Havelund and Grigore Roşu. 2004. An overview of the runtime verification tool Java PathExplorer. *Formal methods in system design* 24, 2 (2004), 189–215.

[17] Klaus Havelund, Grigore Rosu, and Daniel Clancy. 2001. Java pathexplorer: A runtime verification tool. In *International Space Conference*.

[18] Swaminathan Jayaraman, Bharat Jayaraman, and Demian Lessa. 2017. Compact visualization of Java program execution. *Softw. Pract. Exp.* 47, 2 (2017), 163–191. https://doi.org/10.1002/spe.2411

[19] Dapeng Liu and Shaochun Xu. 2009. MuTT: A Multi-Threaded Tracer for Java Programs. In *8th IEEE/ACIS International Conference on Computer and Information Science, IEEE/ACIS ICIS 2009, June 1-3, 2009, Shanghai, China*, Huaikou Miao and Gongzhu Hu (Eds.). IEEE Computer Society, 949–954. https://doi.org/10.1109/ICIS.2009.159

[20] Michael Martin, Benjamin Livshits, and Monica S. Lam. 2005. Finding application errors and security flaws using PQL: a program query language. In *OOPSLA 2005*. 365–383.

[21] Katharina Mehner. 2001. JaVis: A UML-Based Visualization and Debugging Environment for Concurrent Java Programs. In *Software Visualization, International Seminar Dagstuhl Castle, Germany, May 20-25, 2001, Revised Lectures (Lecture Notes in Computer Science, Vol. 2269)*, Stephan Diehl (Ed.). Springer, 163–175. https://doi.org/10.1007/3-540-45875-1_13

[22] Stephen Nelson, David J. Pearce, and James Noble. 2010. Understanding the Impact of Collection Contracts on Design. In *Objects, Models, Components, Patterns*, Jan Vitek (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 61–78.

[23] Kozo Okano, Satoshi Harauchi, Toshifusa Sekizawa, Shinpei Ogata, and Shin Nakajima. 2019. Consistency Checking between Java Equals and hashCode Methods Using Software Analysis Workbench. *IEICE Trans. Inf. Syst.* 102-D, 8 (2019), 1498–1505. https://doi.org/10.1587/transinf.2018EDP7254

[24] Jevitha K. P., Swaminathan Jayaraman, Bharat Jayaraman, and M. Sethumadhavan. 2021. Finite-state model extraction and visualization from Java program execution. *Softw. Pract. Exp.* 51, 2 (2021), 409–437. https://doi.org/10.1002/spe.2910

[25] Wim De Pauw, Erik Jensen, Nick Mitchell, Gary Sevitsky, John M. Vlissides, and Jeaha Yang. 2001. Visualizing the Execution of Java Programs. In *Software Visualization, International Seminar Dagstuhl Castle, Germany, May 20-25, 2001, Revised Lectures (Lecture Notes in Computer Science, Vol. 2269)*, Stephan Diehl (Ed.). Springer, 151–162. https://doi.org/10.1007/3-540-45875-1_12

[26] Subodh Sharma, Ganesh Gopalakrishnan, Eric Mercer, and Jim Holt. 2009. MCC: A runtime verification tool for MCAPI user applications. In *2009 Formal Methods in Computer-Aided Design*. IEEE, 41–44.

[27] Jan Strejček. 2004. *Linear Temporal Logic: Expressiveness and Model Checking*. Ph. D. Dissertation. Masaryk University Brno.

[28] Tarja Systä. 2000. Understanding the Behavior of Java Programs. In *Proceedings of the Seventh Working Conference on Reverse Engineering, WCRE'00, Brisbane, Australia, November 23-25, 2000*. IEEE Computer Society, 214–223. https://doi.org/10.1109/WCRE.2000.891472


Received 2023-05-26; accepted 2023-06-23