Past Expression: Encapsulating Pre-states at Post-conditions by Means of AOP

Jooyong Yi[†] Robby[‡] Xianghua Deng[‡,*] Abhik Roychoudhury[†]

[†]School of Computing, National University of Singapore [‡]Kansas State University
[†]{jooyong,abhik}@comp.nus.edu.sg, [‡]{robby,deng}@ksu.edu

ABSTRACT

Providing a pair of pre and post-condition for a method or a procedure is a typical way of program specification. When specifying a post-condition, it is often necessary to compare the post-state value of a variable with its pre-state value. To access a pre-sate value at a post-condition, most contract languages such as Eiffel and JML provide an old expression; old(x) returns a pre-state value of variable x. However, old expressions pose several problems, most notably the lack of encapsulation; old(x) does not encapsulate an object graph rooted from the pre-state value of x. Thus, method-call expressions like x.equals(old(x)) should generally not be used, and instead each field of x should be compared individually as in x.f1==old(x.f1) && x.f2==old(x.f2). In this paper, we first describe this lack of encapsulation and other problems of old expressions in more detail. Then, to address those problems, we propose our novel past expression along with its formal semantics. We also describe how our past expression can be supported during runtime assertion checking. We explain the involved problems, and show how we solve them. We implement our solution by means of AOP where we exploit various primitive pointcuts including our custom branch pointcut.

Categories and Subject Descriptors

D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*Semantics, Syntax*; D.2.4 [**Software Engineering**]: Software/Program Verification—*Programming by contract*

Keywords

Contract; Encapsulation; Old Expression; Past Expression; Branch Pointcut; Runtime Assertion Checking (RAC)

1 INTRODUCTION

Modern programming languages and tools are not only able to construct and run a program, they can also specify and check what programmers really intend to do with a program. One promising way of doing such program-level specification is to use general assertions such as pre-conditions and post-conditions. Programming languages supporting such general assertions include Eiffel [14], JML [5], Spec# [3] and SPARK [2], to name but a few. Those languages are often called design-by-contract languages (contract languages in short). Those contract languages extend base languages, such as Java, C# and a subset of Ada, with specification-related features such as pre-conditions, post-conditions, and specification-purpose expressions.

The specification-purpose expression that is most commonly used is an old expression. An old expression is used at a post-condition to compare a post-state value (i.e., the value at the end of a method or a procedure) with a pre-state value (i.e., the value at the entry of a method or a procedure). More specifically, an old expression, old(E), returns a pre-state value of expression E. In fact, an old expression is the only means of retrieving pre-state values in most contract languages.

However, old expressions of contract languages pose several problems, most notably the lack of encapsulation; old(x) does not encapsulate an object graph rooted from the pre-state value of x. For this reason, non-destructive method-call expressions such as x.equals(old(x)) in general should not be used, and instead each field of x should be compared one by one as in x.f1==old(x.f1) && x.f2==old(x.f2). In the next section, we describe this and other problems in more detail before proposing our solution.

Our major contributions of this paper are as follows:

1. We identify three problems of old expressions that have been largely neglected, most notably the lack of encapsulation.
2. To address the identified problems, we suggest a past expression as an alternative to an old expression, and provide its formal semantics.
3. We identify the obstacles in supporting past expressions for runtime assertion checking, and show how they can be overcome through various aspect-oriented techniques.

2 OVERVIEW

In this section, we identify the three problems of existing old expressions of contract languages before proposing our solution by a past expression. We also compare our solution with more traditional formal specification languages such as Z [19] and VDM [9]. Lastly, we provide an overview of the obstacles in supporting our past expression in a runtime

[*] This author moved to Google Inc.

AOSD'13, March 24–29, 2013, Fukuoka, Japan.
Copyright 2013 ACM 978-1-4503-1766-5/13/03 ...$15.00.

```
public class PatientSet {
  private Patient[] patients;  private int size;
  //@ invariant (\exists int i; 0 <= i && i < size; patients[i] != null);

  //@ requires contains(p);
  //@ ensures size == \old(size) && (\forall int i; 0 <= i && i < size; (\exists int j; 0 <= j && j < size;
  //@   patients[i].name.equals(\old(patients[j].name)) && patients[i].height == \old(patients[j].height) &&
  //@   patients[i].weight == \old(patients[j].weight) && patients[i].birthDate.year == \old(patients[j].birthDate.year) &&
  //@   patients[i].birthDate.day == \old(patients[j].birthDate.day) && patients[i].birthDate.year == \old(patients[j].birthDate.year)));
  //@ also
  //@ requires !contains(p);
  //@ ensures size == \old(size)+1 && contains(p) && (\forall int i; 0 <= i && i < \old(size); (\exists int j; 0 <= j && j < \old(size);
  //@   patients[i].name.equals(\old(patients[j].name)) && patients[i].height == \old(patients[j].height) &&
  //@   patients[i].weight == \old(patients[j].weight) && patients[i].birthDate.year == \old(patients[j].birthDate.year) &&
  //@   patients[i].birthDate.day == \old(patients[j].birthDate.day) && patients[i].birthDate.year == \old(patients[j].birthDate.year)));
  public void add(/*@ non_null @*/ Patient p) { /* omitted */ }

  public /*@ pure @*/ boolean contains(Patient p) { for (int i = 0; i < size; i++) {if (patients[i].equals(p)) return true;} return false; }
  public /*@ pure @*/ boolean containsAll(/*@ non_null @*/ PatientSet set)
  { for (int i = 0; i < set.size(); i++) {if (!contain(set.get(i))) return false;} return true; }
  public /*@ pure @*/ int size() { return size; }   public /*@ pure @*/ Patient get(int i) { return patients[i]; }
}
```

(a) A PatientSet Java-class stub annotated with JML specifications

```
public class Patient {
  /*@ non_null @*/ String name; float height, weight;
  /*@ non_null @*/ Date birthDate;

  public boolean equals(Object o) {
    return (o instanceof Patient) && ((Patient) o).name.equals(name)
      && ((Patient) o).height == height && ((Patient) o).weight==weight
      && ((Patient) o).birthDate.equals(birthDate); }
}
```

```
public class Date {
  short year, month, day;

  public boolean equals(Object o) {
    return (o instanceof Date) &&
    ((Date) o).year == year && ((Date) o).month == month
    && ((Date) o).day == day; }
}
```

(b) Java classes Patient and Date

```
public class Set<T> {
  private T[] arr;   private int size;

  public void add(T p) { /* omitted */ }
  public boolean contains(T p) { /* omitted */ }
  public boolean containsAll(Set<T> set) { /* omitted */ }
}
```

(c) A Set stub with a type variable T

```
public class Set {
  private ISetElem[] arr;   private int size;

  public void add(ISetElem p) { /* omitted */ }
  public boolean contains(ISetElem p) { /* omitted */ }
  public boolean containsAll(Set set) { /* omitted */ }
}
```

(d) A Set stub with an interface ISetElem

Figure 1: PatientSet example (above the line) and generalized set examples (below the line)

assertion checker. Later sections explain how those obstacles are addressed (§ 4) and implemented through AOP (§ 5) after providing formal semantics of a past expression (§ 3). In § 6, we provide related work.

2.1 Three problems of an old expression

Problem I. To see the first problem, consider a Java class PatientSet of Figure 1(a). An instance of class PatientSet stores patient information in its array, patients. To add new patient information, PatientSet has a method add, and its specification is provided above the method. The specification says in a nutshell: (i) if array patients already contains a Patient instance equivalent to the one given through parameter p, then after exiting add, the patients array should only contain all the Patient information that existed before the add method is called, and (ii) otherwise, not only that but also the new patient information should be added to patients.

The first problem of an old expression is that it tends to cause a specification to be lengthy. Notice the long ensures clauses of method add. To compare i-th patient information, patients[i], to its pre-state value, each of the final fields reachable from patients[i] is compared to its corresponding pre-state value obtained using an old expression. These long ensures clauses are in contrast to the short requires clauses of the same method that use method contains for the containment check.

While it is tempting to use method containsAll in the ensures clauses, using containsAll(\old(this)) will result in a misleading specification due to the semantics of an old expression that can be described informally as follows. Given an old expression, old(x), the value of x is stored in a fresh variable y before executing a method. When old(x) appears in a post-condition, say through old(x).f, old(x) is replaced by y resulting in y.f. It is important to notice that the resulting y.f is executed at the post-state because this expression is inside a post-condition. Thus, going back to the aforementioned tempting solution, i.e., containsAll(\old(this)), even if some fields of a Patient instance accessible through this are modified by method add, the call to containsAll returns, to the surprise of those who are not familiar with contract languages, true. In general, it is risky to pass to a method a reference value returned from an old expression unless the object pointed to by that reference is immutable as is the case of Java's String.

The described problem reflects the lack of encapsulation of an old expression; \old(this) does not represent the PatientSet object that existed at the time when method add is entered; it only represents the reference to that object. This is the

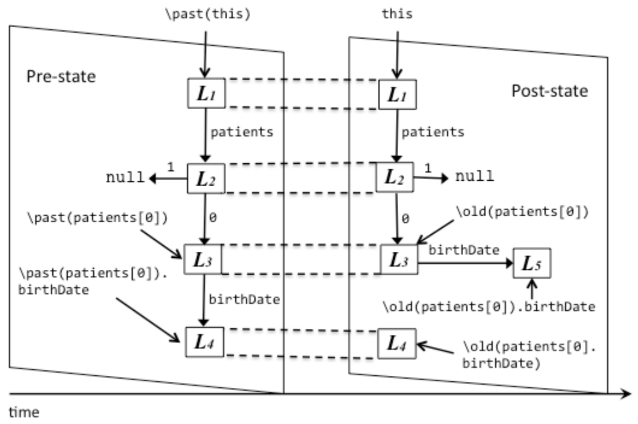

Figure 2: A possible pair of pre/post-states of the add method of class PatientSet

high-level reason why expressions like containsAll(\old(this)) cannot be used in the way one would wish. As a result, one has to end up writing a long specification such as the one in Figure 1(a) where all the fields are revealed. It is unfortunate that the built-in encapsulation capability of the underlying OOP language is lost when using an old expression.

Problem II. The second problem of an old expression is even more serious in terms of expressibility. To see it, consider the two styles of generalized class Set shown in the bottom part of Figure 1. Notice how different add methods are from before. The first one takes as its parameter a generic-type value, and the second one an interface-type value. For these kinds of modern programs, the previous verbose approach is not even applicable because the data structure of the passed parameter cannot be known a priori.

One possible workaround is to confine the type of a set element to a specific one. For example, method add of Figure 1(c) can be specified as follows assuming the Patient type for set elements. In the below, \typeof(p) returns the runtime type of p.

```
//@ requires \typeof(p) <: Patient && contains(p);
//@ ensures size==\old(size) && (\forall int i; 0 <= i && i < size;
//@  (\exists int j; 0 <= j && j < size;
//@    ((Patient) arr[i]).name.equals(\old(((Patient) arr[j]).name))
//@    && ((Patient) arr[i]).height==\old(arr[j].height)
//@ /* rest of them omitted */
public void add(T p) { /* omitted */ }
```

The above approach, however, not only makes the specification even lengthier (because similar specification should be given to each potential type of set elements), but it also does not match the nature of generic programs. Similarly, programs using interfaces (e.g., Figure 1(d)) suffer from the same problem. Overall, conventional old expressions are not expressive enough to handle modern programming languages.

Problem III. Lastly, we also point out that the old expression's copy-based semantics is not memory efficient. Each instance of an old expression takes up an additional variable. Note that one old expression can cause many instances of it if it appears in a quantified expression. Figure 1(a)'s old-expression-based specification in effect deep-clones the patients array. Such semantics of old expressions causes exponential increase in memory usage as the size of PatientSet increases as will be shown in § 5.2.

```
public class PatientSet {
 private Patient[] patients;  private int size;
 //@ invariant (\exists int i; 0 <= i && i < size; patients[i] != null);

 //@ requires contains(p);
 //@ ensures size == \past(size) && containsAll(\past(this));
 //@ also
 //@ requires !contains(p);
 //@ ensures size==\past(size)+1 && contains(p) && containsAll(\past(this));
 public void add(Patient p) { /* omitted */ }

 public /*@ pure @*/ boolean contains(Patient p) { /* omitted */ }
 public /*@ pure @*/ boolean containsAll(PatientSet set) { /* omitted */ }
}
```

Figure 3: A PatientSet stub specified with past expressions

2.2 Our solution with a past expression

The root cause of the identified problems is that only a single object of the pre-state is accessible from one old expression instance. In order to use an expression like patient.equals(old(patient)) with proper meaning, an object graph of the pre-state rooted from patient should be available to equals. To achieve this, we suggest in this paper an alternative to an existing old expression, i.e., a past expression. The central goal of our past expression is to provide a user means to make an access to not only a single object but also an object graph so that an expression such as patient.equals(past(patient)) can be used with proper meaning. In more general terms, we want to make encapsulated accesses to pre-state objects; one clear benefit of it is the ability to apply a non-destructive (i.e., pure [13]) method [1] to pre-state objects at post-conditions.

Before providing the formal semantics of a past expression in the next section, let us first explain the overall idea of a past expression with Figure 2. The two panes of the figure show one possible pair of pre-state and post-state of method add shown earlier. The left pane shows that before entering add, the current instance of PatientSet points to location L_1; its patients field points to an array-type location L_2; its first two array elements are location L_3 of type Patient and null, respectively; lastly, L_3 points to L_4 through field birthDate. Similarly, the right pane shows the post-state of method add. Most of the locations of the pre-state remain the same as depicted through a set of two parallel dashed lines. For expositional purposes, however, we assume that patients[0].birthDate is modified to a fresh location L_5.

Notice in the figure that \old(patients[0]).birthDate refers to L_5 while \past(patients[0]).birthDate refers to L_4. Below, we explain the reason for that difference. Since the location of patients[0] is not modified over the method execution, \old(patients[0]) returns the value of patients[0], i.e., L_3. The same is true for \past(patients[0]). Meanwhile, \old(patients[0]).birthDate accesses field birthDate at the post-state as explained earlier, and returns the modified field value L_5. In contrast, \past(patients[0]).birthDate accesses the same field at the pre-state, and returns the pre-state value L_4. This is because \past(patients[0]) represents not only the pre-state value of patients[0] but also the object graph rooted from patients[0].

Being equipped with a past expression, it becomes easier to compare values of the post-state with their pre-state counterparts. Compare the specification shown in Figure 3 to the original specification using old expressions; the new specifica-

[1] I.e., a method that always terminates and whose execution does not change the program state of its caller.

tion is shorter and more comprehensible; the previously used long quantified expression is now replaced with a simpler and more understandable expression, containsAll(\past(this)).

The same benefit of a past expression also goes to generic programs and interface-oriented programs. For example, the add methods of Figure 1(c) and Figure 1(d) can be specified identically to Figure 3. Without having to reveal the details of the Set class which may be unknown a priori, a specification can be written only with method calls and abstract data including those provided by past expressions.

Our past expressions can be supported in a memory efficient way. However, before moving into tool issues, let us first finish the issue of expressibility by comparing our past expression to old expressions of traditional formal specification languages.

2.3 Traditional formal specification languages

Contract languages such as Eiffel and JML were greatly influenced by more traditional formal specification languages such as Z [19] and VDM [9]. A number of specification expressions of contract languages, including an old expression, were originated from those formal specification languages.

For example, Figure 4 shows a PatientSet expressed in VDM notations. Special attention needs to be paid to the post clause where two implies clauses are listed; those two clauses correspond to the two pre/post-condition pairs of the previous PatientSet examples.[2] Notations size˜ and patients˜ refer to the pre-state values of size and patients, respectively. Notice that the patients variable is defined as a mathematical sequence (i.e., patients: seq of Patient). Thus, patients˜ means the sequence of patients that existed at the pre-state, not the memory address that pointed to that sequence at the pre-state. Thus, patients˜(n) refers to the n-th element of the pre-state patients sequence. In the same regard, patients = patients˜ means the mathematical equality of two sequences, patients and patients˜. Similarly, patients = patients˜^[p] means the equality of the patients sequence and the pre-state patients sequence extended with a Patient p at its tail.

As can be seen in this example, traditional formal specification languages in general do not suffer from the lack of encapsulation seen in modern contract languages. This difference between the old expressions of modern contract languages and traditional formal specification languages stems from the the fact that the latter treats mathematical data types such as a sequence atomically as encapsulated data types. Meanwhile, programming languages like Java provide such mathematical data types in a form of classes. While those classes themselves can be nicely encapsulated, the accesses to their pre-state instances through old expressions are not encapsulated. Our past expression fills this missing gap. But of course, there is no reason to restrict the use of a past expression to only the classes for mathematical data types. Our past expression can be used with instances of any classes.

2.4 Tool support

Contract languages are usually shipped with tools to check specifications that are performed through either static checking or runtime assertion checking (RAC). According to the Chalin's survey of over 200 programmers in industry [6], 97% of respondents use specifications for RAC. Meanwhile, only 20% of respondents use specifications for static checking. Thus, it seems logical to conclude from that survey result that support for RAC is practically more urgent than support for static checking. In this paper, we show how past expressions can be supported for RAC of Java programs. Our approaches should be applicable to other similar languages.

```
class PatientSet
 types
   Date:: year: nat1  month: nat1  day: nat1;
   Patient:: name: seq of char  height: real  weight: real  birthDate: Date
 instance variables
   private patients: seq of Patient;   private size: nat
 operations
   add : Patient ==> ()
   add(p) == /* body omitted */
   post (exists n : nat1 & n <= size˜ and patients˜(n) = p)
          => (size = size˜ and patients = patients˜)
        and
        (not (exists n : nat1 & n <= size˜ and patients˜(n) = p))
          => (size = size˜ + 1 and patients = patients˜ ^ [p])
end PatientSet
```

Figure 4: PatientSet in VDM notations

In fact, it is quite challenging to support past expressions for RAC. This is because the language's standard execution semantics should be respected during RAC unless a custom execution environment is available. RAC is usually performed by instrumenting the original source code with assertion code and then running that instrumented code on a standard execution environment such as the HotSpot JVM (Java Virtual Machine) in the case of Java.

The first problem encountered when supporting RAC of Java programs is that a standard JVM does not use two separate heaps while one of the key elements of the formal semantics of a past expression is to maintain and manipulate two separate heaps independent of each other. A naive workaround would be to use cloning. However, not only is cloning resource-consuming but also its naive uses cause a serious semantic flaw. The reference to a cloned object is always different from the reference to the original object. Thus, if cloning was used, expressions such as \past(x) == x would wrongly return false even if x points to the same object before and after a method execution. Although such a flaw can be avoided by relating cloned objects with their original objects at the expense of more memory, problems still remain because not every Java class is cloneable.

There is a more challenging obstacle caused by Java's call-by-value semantics. When using an expression like x.equals (\past(x)), we need to inform the body of equals that its parameter is passed as a past expression. This is because, depending on whether the parameter of equals is past-ed at a call site, a field of that parameter should be accessed either in the pre-state or the post-state. Java's call-by-value semantics, however, makes it difficult to distinguish whether or not a parameter value is returned from a past expression.

In the latter part of this paper, we show how the above obstacles in supporting RAC can be tackled. We will also show how our solution is implemented through AOP.

3 FORMAL SEMANTICS

In this section, we provide formal semantics of our past expression. For efficiency of discussion, we define formal semantics on a minimal language. A past expression can be added to any contract languages that can accommodate our minimal language.

[2] VDM does not allow multiple pre/post-condition pairs.

3.1 Programming language

We use a typical imperative procedural language shown in Figure 5(a) that can manipulate integers, booleans, and records. To manipulate records, our language has field-access expressions $E.f$ and field-update commands $E.f{:=}\ E$. We omit arrays because they can be dealt with similarly to records by treating array indices as if they are record fields.

A past expression, like an old expression, can be used only in a post-condition. In our minimal language, a post-condition appears as an assert command at the end of a procedure. We assume that every procedure ends with a single assert command. We do not consider a pre-condition because a past expression cannot be used there.

An assert command takes as its argument a boolean expression E^b such as $E{=}{=}E$ and $E{>}E$. Note that $E{>}E$ can be used only with integer-type expressions. Record-type expressions can be compared only with ==. Although our minimal language does not allow a call to a boolean function such as equals, extension towards it is straightforward.

Only the boolean expressions used in an assert command can have past expressions $\backslash\mathsf{past}(E)$ as their sub-expressions. The commands of a procedure preceding its only assert command cannot use a past expression.

A past expression cannot be nested inside another past expression. For example, we disallow $\backslash\mathsf{past}(\backslash\mathsf{past}(x))$ and $\backslash\mathsf{past}(\backslash\mathsf{past}(x).f)$. This coincides with the fact that the conventional specification does not consider the pre-state of a pre-state at a post-condition.

For simplicity, procedures of our minimal language do not take a parameter. We assume that every variable of a program is declared global. We also assume that every variable is initialized to a certain value. An extension to a more sophisticated language supporting multiple scopes is straightforward.

3.2 Semantic rules

Figure 6 shows the operational semantic rules for the critical expressions in a big-step style. As mentioned, we restrict past expressions to be used only in the single assert command located at the end of a procedure. Such restriction is denoted with the assertion-context notation "assert $\vdash$" in the rules. Semantic reductions described behind this notation should occur in an assertion context, i.e., inside an assert command. We only describe the semantics under an assertion context while assuming the standard semantics under a non-assertion context.

An assertion context is introduced when a boolean expression of an assert command starts to be evaluated. And it exits when that boolean expression is finished to be evaluated. Figure 6(a) shows the rules that introduce an assertion context. Note that a past expression, $\backslash\mathsf{past}(E)$, is as a whole considered a base expression in an assertion context. Therefore, when encountered with assert $\backslash\mathsf{past}(x) > 0$, an assertion context should be introduced by $\backslash\mathsf{past}(x)$, not by x.[3]

In an assertion context, an expression is evaluated in an extended state $(\sigma_1, h_1, \sigma_2, h_2)$. Its first two components, a store σ_1 and a heap h_1, constitute the pre-state, and similarly, σ_2 and h_2 the post-state. Such meanings of those symbols are assigned through $\langle C, (\sigma_1, h_1)\rangle \Downarrow_c (\sigma_2, h_2)$ in the premises of the rules. This command-reduction relation $\Downarrow_c$ between configuration $\langle C, (\sigma_1, h_1)\rangle$ and state (σ_2, h_2) captures the fact that the program state changes from (σ_1, h_1) to (σ_2, h_2) by executing command C. Since C represents the whole command preceding the sole assertion at the end of a procedure, (σ_1, h_1) and (σ_2, h_2) are interpreted as the pre-state and the post-state of a procedure, respectively.

$z \in \mathbb{Z}$ $\quad x \in$ **Variable** $\quad f \in$ **Field** $\quad E \in$ **Expression**
$C \in$ **Command** $\quad P \in$ **Procedure** $\quad D \in$ **Declaration**
$T \in$ **Type** $\quad Prg \in$ **Program**

$$E ::= \backslash\mathsf{past}(E) \mid z \mid \mathsf{nil} \mid x \mid E.f \mid E^b$$
$$E^b ::= \mathsf{true} \mid \mathsf{false} \mid E{=}{=}E \mid E{>}E \mid !E^b \mid E^b \mathbin{\&\&} E^b \mid E^b \mathbin{||} E^b$$
$$C ::= x{:=}E \mid E.f{:=}E \mid \mathsf{new}(x : T) \mid C;C \mid \mathsf{if}\ E^b\ \mathsf{then}\ C\ \mathsf{else}\ C$$
$$\mid \mathsf{while}\ E^b\ \mathsf{do}\ C \mid \mathsf{call}\ P$$
$$P ::= \mathsf{begin}\ C; \mathsf{assert}\ E^b\ \mathsf{end}$$
$$Prg ::= \mathsf{begin}\ D\ \mathsf{in}\ C\ \mathsf{end}$$

(a) Our minimal programming language

$v \in \mathbf{Value} \overset{\text{def}}{=} \mathbb{Z} \cup \mathbb{B} \cup \mathbf{Loc} \cup \{\mathsf{nil}\}$ $\qquad \mathbb{Z} \overset{\text{def}}{=} \{\ldots, -1, 0, 1, \ldots\}$

$\mathbb{B} \overset{\text{def}}{=} \{\mathsf{true}, \mathsf{false}\}$ $\qquad \sigma \in \mathbf{Store} \overset{\text{def}}{=} \mathbf{Variable} \rightarrow \mathbf{Value}$

$h \in \mathbf{Heap} \overset{\text{def}}{=} \mathbf{Loc} \rightarrow (\mathbf{Field} \rightarrow \mathbf{Value})$

(b) Semantic domains

Figure 5: Programming language and its semantic domains

Our past expressions, $\backslash\mathsf{past}(E)$, are evaluated using the two rules in the upper row of Figure 6(a). Its sub-expression E should be evaluated with the pre-state, (σ_1, h_1), for an obvious reason. Thus, those rules have in common an expression-reduction relation, $\langle E, (\sigma_1, h_1)\rangle \Downarrow_e v$, in their premises.

The uniqueness of a past expression is revealed in the consequent part of the left-hand-side rule of the upper row; $\backslash\mathsf{past}(E)$ reduces to not a value v but a pair (v, h_1). As will be described in detail shortly, the second component h_1 indicates the heap in which fields of the record represented by v are accessed. Since h_1 is the heap of the pre-state, field accesses such as $\backslash\mathsf{past}(E).f$ will be made in the pre-state as desired. On the contrary, when reducing non-past base expressions x, the value of x is paired with h_2, i.e., the heap of the post-state. Such semantics is described in the left-hand-side rule of the lower row of Figure 6(a).

However, not every base expression should be reduced to a pair. If a base-expression value v represents not a record but an integer or a boolean value, there is no need to pair that value v with a heap because no further field access from v is possible. Such a semantic difference is captured in the two right-hand-side rules of Figure 6(a). In case where v is nil, we still pair v with a heap despite that a further field access from nil is impossible too. As will be shown shortly, this slight compromise reduces the number of necessary semantic rules for equality expressions.

As mentioned, such paired heaps are looked up when a field is accessed subsequently. The two upper-row rules of Figure 6(b) show such a usage of a paired heap. Given an expression $E.f$, the owner expression E is first reduced to a pair (v, h). The subsequent access to the field f is made

[3] Although the rules allow to enter an assertion context through x, there is no rule to interpret the remaining \past afterwards.

When $P ::= \mathsf{begin}\ C; \mathsf{assert}\ E^b\ \mathsf{end}$,

$$\frac{\langle C, (\sigma_1, h_1)\rangle \Downarrow_c (\sigma_2, h_2) \qquad \langle E, (\sigma_1, h_1)\rangle \Downarrow_e v \qquad v \in \mathbf{Loc} \cup \{\mathsf{nil}\}}{\mathsf{assert} \vdash \langle \backslash\mathsf{past}(E), (\sigma_1, h_1, \sigma_2, h_2)\rangle \Downarrow_e (v, h_1)} \qquad \frac{\langle C, (\sigma_1, h_1)\rangle \Downarrow_c (\sigma_2, h_2) \qquad \langle E, (\sigma_1, h_1)\rangle \Downarrow_e v \qquad v \in \mathbb{Z} \cup \mathbb{B}}{\mathsf{assert} \vdash \langle \backslash\mathsf{past}(E), (\sigma_1, h_1, \sigma_2, h_2)\rangle \Downarrow_e v}$$

$$\frac{\langle C, (\sigma_1, h_1)\rangle \Downarrow_c (\sigma_2, h_2) \qquad \langle x, (\sigma_2, h_2)\rangle \Downarrow_e v \qquad v \in \mathbf{Loc} \cup \{\mathsf{nil}\}}{\mathsf{assert} \vdash \langle x, (\sigma_1, h_1, \sigma_2, h_2)\rangle \Downarrow_e (v, h_2)} \qquad \frac{\langle C, (\sigma_1, h_1)\rangle \Downarrow_c (\sigma_2, h_2) \qquad \langle x, (\sigma_2, h_2)\rangle \Downarrow_e v \qquad v \in \mathbb{Z} \cup \mathbb{B}}{\mathsf{assert} \vdash \langle x, (\sigma_1, h_1, \sigma_2, h_2)\rangle \Downarrow_e v}$$

(a) Semantic rules for base expressions (i.e., $\backslash\mathsf{past}(E)$ and x) under an assertion context

When $S = (\sigma_1, h_1, \sigma_2, h_2)$,

$$\frac{\mathsf{assert} \vdash \langle E, S\rangle \Downarrow_e (v, h) \qquad h(v)(f) = v' \qquad v' \in \mathbf{Loc} \cup \{\mathsf{nil}\}}{\mathsf{assert} \vdash \langle E.f, S\rangle \Downarrow_e (v', h)} \qquad \frac{\mathsf{assert} \vdash \langle E, S\rangle \Downarrow_e (v, h) \qquad h(v)(f) = v' \qquad v' \in \mathbb{Z} \cup \mathbb{B}}{\mathsf{assert} \vdash \langle E.f, S\rangle \Downarrow_e v'}$$

$$\frac{\mathsf{assert} \vdash \langle E_1, S\rangle \Downarrow_e (v, h) \qquad \mathsf{assert} \vdash \langle E_2, S\rangle \Downarrow_e (v, h')}{\mathsf{assert} \vdash \langle E_1 {=}{=} E_2, S\rangle \Downarrow_e \mathsf{true}} \qquad \frac{\mathsf{assert} \vdash \langle E_1, S\rangle \Downarrow_e (v_1, h) \qquad \mathsf{assert} \vdash \langle E_2, S\rangle \Downarrow_e (v_2, h') \qquad v_1 \neq v_2}{\mathsf{assert} \vdash \langle E_1 {=}{=} E_2, S\rangle \Downarrow_e \mathsf{false}}$$

(b) Semantic rules for field access expressions and equality expressions under an assertion context

When $P ::= \mathsf{begin}\ C; \mathsf{assert}\ E^b\ \mathsf{end}$,

$$\frac{\langle C, (\sigma_1, h_1)\rangle \Downarrow_c (\sigma_2, h_2) \qquad \langle E, (\sigma_1, h_1)\rangle \Downarrow_e v \qquad v \in \mathbf{Loc} \cup \{\mathsf{nil}\}}{\mathsf{assert} \vdash \langle \backslash\mathsf{old}(E), (\sigma_1, h_1, \sigma_2, h_2)\rangle \Downarrow_e (v, h_2)} \qquad \frac{\langle C, (\sigma_1, h_1)\rangle \Downarrow_c (\sigma_2, h_2) \qquad \langle E, (\sigma_1, h_1)\rangle \Downarrow_e v \qquad v \in \mathbb{Z} \cup \mathbb{B}}{\mathsf{assert} \vdash \langle \backslash\mathsf{old}(E), (\sigma_1, h_1, \sigma_2, h_2)\rangle \Downarrow_e v}$$

(c) Semantic rules for \old expressions under an assertion context (shown for comparison with \past expressions)

Figure 6: Semantic rules for past and other critical expressions under an assertion context

using this pair. That is, the value of the field f is obtained by $h(v)(f)$. Recall that the previous rules of Figure 6(a) pass the pre-state heap when evaluating a past expression, and the post-state heap when evaluating a non-past expression. By the combination of the rules in Figure 6(a) and Figure 6(b), only a field access followed by a past expression is looked up in the pre-state as desired.

Once the value of $E.f$ is obtained using a passed heap h, the rules for $E.f$ continuously pass the same heap h by pairing it with the obtained field value. This way, subsequent field accesses can be made in the same heap. Of course, there is no need to continue to pass the heap when the obtained field value is an integer or a boolean.

Although our semantic rules reduce a field access expression $E.f$ to a pair (v, h), only v should be considered a value. Thus, when comparing (v, h) with (v', h'), the actual comparison is made only between v and v' while ignoring h and h'. Such semantics of comparison is described in the two lower-row rules of Figure 6(b). Recall that we earlier chose to pair nil with a heap. Thank to that choice, we do not need additional rules to handle the comparison involving nil.

Our semantic rules are sound in the sense of the following. The proof is trivial and omitted.

Theorem 1. *Given a procedure "* $\mathsf{begin}\ C; \mathsf{assert}\ E^b\ \mathsf{end}$ *" and a legitimate reduction relation* $\langle C, (\sigma_1, h_1)\rangle \Downarrow_c (\sigma_2, h_2)$*, let us define an extended state* S *as* $(\sigma_1, h_1, \sigma_2, h_2)$*. Then the following holds true: If* $\mathsf{assert} \vdash \langle E^b, S\rangle \Downarrow_e \mathsf{true}$*, then it holds that* $S \models E^b$*, which means that* E^b *is true at extended state* S*. Also conversely, if* $\mathsf{assert} \vdash \langle E^b, S\rangle \Downarrow_e \mathsf{false}$*, then* $S \models \neg E^b$*.*

Comparison to old expressions. While the conventional old expression, \old(E), is not part of our language, we describe in Figure 6(c) its semantics for the comparison with our past expression. While the sub-expression E is evaluated in the pre-state as in a past expression, its reduction result is paired with h_2, i.e., the heap of the post-state, unlike in a past expression, if E represents a record.

4 SUPPORT FOR RAC

As been alluded to earlier, the semantic rules of Figure 6 cannot be naturally extended for runtime assertion checking (RAC) for Java programs. The problems are twofold. First, the conventional Java virtual machine does not provide pre-states. Second, Java's call-by-value semantics does not pass heap information used extensively in the semantic rules. That is, only v instead of (v, h) is passed around. Recall that the second component, h, indicates the heap to consult when accessing the fields of v. We address each of the problems by the techniques we call difference heap and proxy, respectively. We describe our solutions with the same minimal language we used before, and refine the original semantics to the one in Figure 9. It is straightforward to extend the given solutions to Java language as empirically evidenced by our prototype. We describe Java-specific issues we found interesting in the last part of this section.

4.1 Difference heap

Given a procedure, "$\mathsf{begin}\ C; \mathsf{assert}\ E^b\ \mathsf{end}$", only a small portion of the program state is likely to be modified while executing C. Recall that we consider the state before and after executing C as the pre-state and the post-state of a given procedure, respectively. If those differences between the pre-state and the post-state are known, the pre-state can be retrieved from the post-state and can be used to evaluate past expressions. While a conventional execution environment does not make pre-states readily available, it is relatively easy to maintain difference heaps at run time by instrumenting the original code.

To maintain such state differences, we use a difference heap for each procedure call. As its name indicates, a difference heap maintains differences occurring to the heap. Note that there is no need to do the same for the store σ because for an arbitrary variable, its pre-state value can be stored in an extra fresh variable without too much cost. Meanwhile, it is impractical to use the same technique for the heap given the complexity and the size of the heap.

Whenever a procedure is called, a fresh empty difference heap is associated with that procedure call. Being equipped with difference heaps, we reformulate a program state to a tuple of a store σ, a heap h, and a stack of difference heaps δ. The top of the difference-heap stack is the difference heap of the current procedure. In the semantic rules, we denote a state with triple (σ, h, δ) for store σ, heap h, and the difference heap δ of the current procedure. Unlike before, we do not distinguish an assertion context, and use this triple throughout the all semantic rules. Recall that in our first formal semantics, we use a state (σ_1, h_1) under a non-assertion context, and an extended state $(\sigma_1, h_1, \sigma_2, h_2)$ under an assertion context.

We mentioned that a fresh empty difference heap is associated with each procedure call. An empty difference heap satisfies the following property: for every location $l \in \mathbf{Loc}$ and field $f \in \mathbf{Field}$, $\delta(l)(f) = \bot$. The difference heap may be updated during the execution of C given a procedure, "begin C; assert E^b end". More specifically, whenever a field f of a location l that was already in use in the pre-state is modified for the first time during the execution of C, the original field value is stored in the difference heap of the procedure.

In our minimal language, such a field update can be made only through field update commands, $E_1.f := E_2$. The corresponding semantic rules are shown in Figure 7(a). Note that in the above semantic rules, $\langle E, (\sigma, h, \delta)\rangle$ reduces to its value v regardless of difference heap δ because a past expression cannot appear in E. Notice in the first rule that the regular heap h and the difference heap δ are updated differently. Given an owner location v_1 and a field f, $h(v_1)(f)$ is updated to an assigning value v_2; i.e., $h[(v_1, f) \mapsto v_2]$. Meanwhile, $\delta(v_1)(f)$ is updated to the original value of the field, v_f; i.e., $\delta[(v_1, f) \mapsto v_f]$.

Such an update of the difference heap can take place only if the three conditions listed in the second line of the first rule hold. These three conditions are that (i) the owner location v_1 of the field f is already in use in the pre-state – in other words, v_1 is not freshly allocated during the current procedure, i.e., $v_1 \notin Fresh$ where $Fresh$ represents a set of locations allocated during the current procedure, (ii) the assigning field value v_2 is different from the original field value v_f, and (iii) the field f of v_1 has not been updated before at this instance of procedure call. Such a first-time-update condition can be checked by looking at the value of $\delta(v_1)(f)$ before updating the difference heap. Only if $\delta(v_1)(f)$ remains undefined (i.e., $\delta(v_1)(f) = \bot$), the field update is considered the first-time update. The third condition is necessary because we are interested only in the pre-state value, and mid-state values are unnecessary for our purposes. If one of those three conditions is not met, the difference heap δ is not modified as shown in the second rule of Figure 7(a).

$$\frac{\begin{array}{c}\langle E_1,(\sigma,h,\delta)\rangle \Downarrow_e v_1 \quad \langle E_2,(\sigma,h,\delta)\rangle \Downarrow_e v_2 \quad \langle E_1.f,(\sigma,h,\delta)\rangle \Downarrow_e v_f \\ (v_1 \notin Fresh) \wedge (v_2 \neq v_f) \wedge (\delta(v_1)(f) = \bot)\end{array}}{\langle E_1.f{:=}E_2,(\sigma,h,\delta)\rangle \Downarrow_c \langle \sigma, h[(v_1,f)\mapsto v_2], \delta[(v_1,f)\mapsto v_f]\rangle}$$

$$\frac{\begin{array}{c}\langle E_1,(\sigma,h,\delta)\rangle \Downarrow_e v_1 \quad \langle E_2,(\sigma,h,\delta)\rangle \Downarrow_e v_2 \quad \langle E_1.f,(\sigma,h,\delta)\rangle \Downarrow_e v_f \\ (v_1 \in Fresh) \vee (v_2 = v_f) \vee (\delta(v_1)(f) \neq \bot)\end{array}}{\langle E_1.f{:=}E_2,(\sigma,h,\delta)\rangle \Downarrow_c \langle \sigma, h[(v_1,f)\mapsto v_2], \delta\rangle}$$

(a) A semantic rule for field update commands; $Fresh$ represents a set of locations allocated during the current procedure.

For every location $l \in \mathbf{Loc}$, and every field $f \in \mathbf{Field}$,

$$(h \lhd \delta)(l)(f) = \begin{cases} \delta(l)(f) & \text{if } \delta(l)(f) \neq \bot \\ h(l)(f) & \text{if } \delta(l)(f) = \bot \end{cases}$$

(b) Override operator $\lhd$; essentially, the difference heap δ overrides the regular heap h when possible.

Figure 7: Update and use of a difference heap δ

When a procedure is about to exit, items in the difference heap of the exiting procedure are transferred into the difference heap of the caller. When this happens, it is unnecessary to move all items; we move only the items (l, f, v)[4] whose location elements l are in use in the pre-state of the caller and $\delta(l)(f)$ is not $\bot$. Recall that the first condition can be checked by looking up l in the $Fresh$ set of the caller. In addition to migrating difference heaps, we also move all locations contained in the $Fresh$ set of the exiting procedure into the $Fresh$ set of the caller to ignore subsequent updates made on the fields of those locations during the remaining execution of the caller.

The semantic rules of Figure 7(a) guarantee the following property. The proof is trivial and omitted.

Property 1. *Given a procedure, "*begin C; assert E^b end*", suppose that for an arbitrary location l and field f, it holds that $h_1(l)(f) = v$ before executing C for the then heap h_1. Then, the following holds true after executing C. In the below, δ and h_2 represent, respectively, the difference heap and the regular heap existing after executing C.*

$$h_1(l)(f) = v \Rightarrow (\delta(l)(f) = v \vee (\delta(l)(f) = \bot \wedge h_2(l)(f) = v))$$

Now it is possible to retrieve pre-state values from the difference heap and the post-state heap. The difference heap can be used to retrieve the pre-state values when those values are overwritten while executing a procedure. If the difference heap returns $\bot$ for a location l and a field f, that means that that value was not overwritten with a different value while executing the procedure. Thus, the post-state heap can be used to retrieve the original value. Formally, we define an override operator $\lhd$ as in Figure 7(b), and use it in the new semantic rules that will be shown shortly.

[4] Recall that $\mathbf{Heap} \stackrel{\text{def}}{=} \mathbf{Loc} \to (\mathbf{Field} \to \mathbf{Value})$.

4.2 Proxy record

While pre-state values can be retrieved using the difference heap, the difference heap alone is not enough to support past expressions at run time. It is in addition necessary to be able to decide when a pre-state value should be retrieved. For example, given a field access expression $E.f$, it is necessary to be able to distinguish the case where E is \past(x) from the case where E is x.

To make such a distinction, we in our first formal semantics paired a value with a heap that should be used subsequently. However, call-by-value semantics used in languages like Java generally cannot accommodate such pairing unless the language itself is modified to allow such pairs of a location and a heap as part of program values. For this reason, the previous pairing-based semantics is not suitable for runtime assertion checking that uses the execution environment of a language as it is.

To see the problem caused by call-by-value semantics, consider x.equals(\past(x)). Following the call-by-value semantics, \past(x) is first evaluated before method equals receives that evaluated value as its parameter. Thus, from the perspective of equals, it cannot distinguish whether its parameter was evaluated from a past expression or a non-past expression. This causes a problem when accessing a field through that parameter. There is no clue about in which heap a field should be accessed.

To address the above problem, we use a proxy record (a proxy in the sequel), i.e., a proxy for a non-scalar pre-state value such as a pre-state location. As the name indicates, every non-scalar pre-state value is accessed through a proxy. In the above example of x.equals(\past(x)), equals receives as its parameter a proxy for the pre-state value of x. Thus, when encountered with $E.f$, we can decide the heap in which the field should be accessed depending on whether or not the evaluation result of an owner expression E is a proxy. Only if an owner expression returns a proxy, we look up a subsequent field in the pre-state heap restored using the difference heap and the post-state heap.

Let us take a concrete example. Figure 8 shows a post-state (σ, h, δ) for a store σ, a heap h, and a difference heap δ of the current method. This post-state satisfies the following three conditions: $\sigma(\mathsf{x\_old}) = l_2$, $h(l_2)(\mathsf{f}) = l_4$, and $\delta(l_2)(\mathsf{f}) = l_3$. A special variable x_old has location l_2 as its value equivalent to the pre-state value of the original program variable x. A field f of l_2 points to location l_4 at the given post-state (i.e., $h(l_2)(\mathsf{f}) = l_4$). However, at the pre-state, the same field pointed to another location l_3 as shown with the difference heap δ (i.e., $\delta(l_2)(\mathsf{f}) = l_3$). To distinguish those two distinct field accesses between h and δ, notations $l_2 \xrightarrow{\mathsf{f}}_h l_4$ and $l_2 \xrightarrow{\mathsf{f}}_\delta l_3$ are used in the figure.

Notice in the figure that \past(x) does not directly refer to x's pre-state value l_2. Instead, it refers to l_1, the proxy for l_2. Every proxy record has a special field, actual, through which the "actual" value of a proxy can be accessed.

Now that \past(x) returns a proxy, we can choose the pre-state heap when accessing a subsequent field. To get the value of \past(x).f, the actual value of \past(x) is first retrieved and then the difference heap is consulted to obtain the pre-state field value l_3. However, l_3 is not directly returned. Its proxy l_5 is created at runtime while being linked to l_3 through the actual field, and returned. This way, subsequent field accesses occurring after \past(x).f can still be looked up in the pre-state heap.

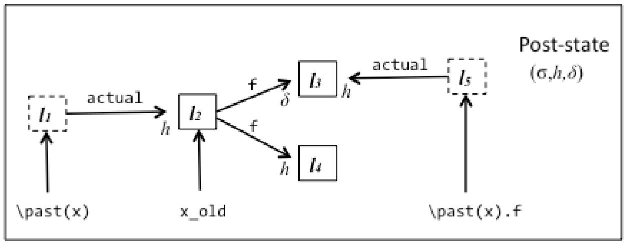

Figure 8: A post-state where the locations for proxy records are in dashed boxes; at this post state, it holds that $\sigma(\mathsf{x\_old}) = l_2$, $h(l_2)(\mathsf{f}) = l_4$, and $\delta(l_2)(\mathsf{f}) = l_3$.

Non past-expressions are evaluated ordinarily without using a proxy. For example, consider the case where it holds that $\sigma(\mathsf{x}) = l_2$ in the above figure. The evaluation result of x is directly l_2. Similarly, the evaluation result of x.f is l_4.

Note that using proxies barely increases memory usage in general. Although multiple proxies can be generated while evaluating expressions such as $\backslash\mathsf{past}(x).f_1.f_2.f_3$, most of them soon become garbages. Also, the fact that proxies are generated on-demand basis has a positive impact on memory maintenance.

4.3 Refined semantics

Now that we explained a difference heap and a proxy record, we can show in Figure 9 the refined semantics of the original one. Unlike in the original semantics, we do not distinguish an assertion context. Notice that notation assert $\vdash$ is not used in the refined semantics. This is because we uniformly use the same state configuration (σ, h, δ) across a whole program.

As shown in the left-hand-side rule of Figure 9(a), $\backslash\mathsf{past}(E)$ now reduces to a fresh proxy v_p for the pre-state value v of E if v is non-scalar, i.e., $v \in \mathbf{Loc} \cup \{\mathsf{nil}\}$. We use a predicate notation "$v_p : \mathsf{Proxy}$" to denote that v_p is a proxy. Notice that after applying the rule, the current heap h_2 is updated to reflect that v_p's actual field points to a pre-state value v. The store and the difference heap are not modified. Meanwhile, if the pre-state value of E is a scalar value v (i.e., $v \in \mathbb{Z} \cup \mathbb{B}$), then v is directly returned as the right-hand-side rule shows.

Figure 9(b) shows how to evaluate a field access expression $E.f$ when its owner expression E reduces to a proxy. As informally explained earlier, in such cases, the actual value v_a of the proxy v_p is first retrieved from the current heap h'. See $h'(v_p)(\mathsf{actual}) = v_a$. Then, the field value v_f is obtained from the restored pre-state heap. See $(h' \lhd \delta)(v_a)(f) = v_f$. If v_f is non-scalar, a fresh proxy v_p' that points to v_f through the actual field is returned (the left-hand-side rule). Otherwise, v_f is directly returned (the right-hand-side rule). We omit to show the rules for the rest of the cases where an owner expression reduces to a non-proxy value. They are handled in a standard way.

Lastly, Figure 9(c) shows some of the rules for equality expressions $E_1 {==} E_2$ while the rest of the cases can be handled similarly. As usual, we assume that E_1 is evaluated first before E_2. The rules show that if a sub-expression reduces to a proxy, then its actual value should be used for comparison.

The semantics presented in this section is a refinement of the original semantics of § 3 in the following sense. The proofs for them are not difficult and omitted for the lack of space.

When $P ::= \mathsf{begin}\ C; \mathsf{assert}\ E^b\ \mathsf{end}$,

$$\frac{\begin{array}{c}\langle C, (\sigma_1, h_1, \delta_1)\rangle \Downarrow_c (\sigma_2, h_2, \delta_2) \qquad \langle E, (\sigma_1, h_1, \delta_1)\rangle \Downarrow_e v \\ v \in \mathbf{Loc} \cup \{\mathsf{nil}\} \qquad v_p : \mathsf{Proxy} \qquad v_p \notin dom\ h_2\end{array}}{\langle \backslash\mathsf{past}(E), (\sigma_2, h_2, \delta_2)\rangle \Downarrow_e \langle v_p, (\sigma_2, h_2[(v_p, \mathsf{actual}) \mapsto v], \delta_2)\rangle} \qquad \frac{\begin{array}{c}\langle C, (\sigma_1, h_1, \delta_1)\rangle \Downarrow_c (\sigma_2, h_2, \delta_2) \qquad \langle E, (\sigma_1, h_1, \delta_1)\rangle \Downarrow_e v \\ v \in \mathbb{Z} \cup \mathbb{B}\end{array}}{\langle \backslash\mathsf{past}(E), (\sigma_2, h_2, \delta_2)\rangle \Downarrow_e \langle v, (\sigma_2, h_2, \delta_2)\rangle}$$

(a) Semantic rules for $\backslash\mathsf{past}(E)$

$$\frac{\begin{array}{c}\langle E, (\sigma, h, \delta)\rangle \Downarrow_e \langle v_p, (\sigma, h', \delta)\rangle \qquad v_p : \mathsf{Proxy} \\ h'(v_p)(\mathsf{actual}) = v_a \qquad (h' \lhd \delta)(v_a)(f) = v_f \\ v_f \in \mathbf{Loc} \cup \{\mathsf{nil}\} \qquad v'_p : \mathsf{Proxy} \qquad v'_p \notin dom\ h'\end{array}}{\langle E.f, (\sigma, h, \delta)\rangle \Downarrow_e \langle v'_p, (\sigma, h'[(v'_p, \mathsf{actual}) \mapsto v_f], \delta)\rangle} \qquad \frac{\begin{array}{c}\langle E, (\sigma, h, \delta)\rangle \Downarrow_e \langle v_p, (\sigma, h', \delta)\rangle \qquad v_p : \mathsf{Proxy} \\ h'(v_p)(\mathsf{actual}) = v_a \qquad (h' \lhd \delta)(v_a)(f) = v_f \qquad v_f \in \mathbb{Z} \cup \mathbb{B}\end{array}}{\langle E.f, (\sigma, h, \delta)\rangle \Downarrow_e \langle v_f, (\sigma, h', \delta)\rangle}$$

(b) Semantic rules for field access expressions (partial)

$$\frac{\begin{array}{c}\langle E_1, (\sigma, h, \delta)\rangle \Downarrow_e \langle v_p, (\sigma, h', \delta)\rangle \qquad v_p : \mathsf{Proxy} \qquad h'(v_p)(\mathsf{actual}) = v_a \\ \langle E_2, (\sigma, h', \delta)\rangle \Downarrow_e \langle v'_p, (\sigma, h'', \delta)\rangle \qquad v'_p : \mathsf{Proxy} \qquad h''(v'_p)(\mathsf{actual}) = v_a\end{array}}{\langle E_1 == E_2, (\sigma, h, \delta)\rangle \Downarrow_e \mathsf{true}} \qquad \frac{\begin{array}{c}\langle E_1, (\sigma, h, \delta)\rangle \Downarrow_e \langle v, (\sigma, h, \delta)\rangle \qquad \neg(v : \mathsf{Proxy}) \\ \langle E_2, (\sigma, h, \delta)\rangle \Downarrow_e \langle v_p, (\sigma, h', \delta)\rangle \\ v_p : \mathsf{Proxy} \qquad h'(v_p)(\mathsf{actual}) = v\end{array}}{\langle E_1 == E_2, (\sigma, h, \delta)\rangle \Downarrow_e \mathsf{true}}$$

(c) Semantic rules for equality expressions (partial)

Figure 9: Refined semantic rules friendly to runtime assertion checking (RAC)

THEOREM 2. *Given a procedure "*begin C; assert E^b end*", if one can obtain* $\langle C, (\sigma_1, h_1, \delta_1)\rangle \Downarrow_c (\sigma_2, h_2, \delta_2)$ *using the refined semantics, then* $\langle C, (\sigma_1, h_1)\rangle \Downarrow_c (\sigma_2, h_2)$ *can be obtained in the original semantics. And also subsequently, if one can obtain* $\langle E^b, (\sigma_2, h_2, \delta_2)\rangle \Downarrow_e$ true *using the refined semantics, then* assert $\vdash \langle E^b, (\sigma_1, h_1, \sigma_2, h_2)\rangle \Downarrow_e$ true *can be obtained in the original semantics. A similar implication also holds for the* false *case.*

4.4 Java-specific issues

Arrays. While our minimal language does not have arrays, our prototype tool supports arrays. We treat arrays similarly to records. For example, \past(a[0]) returns a proxy for the pre-state value of a[0]. Similarly, \past(a) refers to a proxy array for a pre-state array a. The length of such a proxy array is set to zero to minimize memory usage. When the length of a pre-state array is queried through a proxy array, the actual array of the given proxy array is first retrieved and its length is returned.

However, a proxy array cannot have an actual field unlike a proxy record. To address this issue, we maintain a global map that associates proxy arrays with their actual arrays.

Equality with this. Semantic rules for equality expressions shown in Figure 9(c) enforce equality between values regardless of whether they belong to the pre- or the post-state. Thus, an expression such as \past(o) == o for a reference variable o returns true if o points to the same object in the pre and the post-state. Such equality across the time is usually desirable because comparison methods such as equals often compare between two references of some fields.

However, there is an important exception to consider. A number of Java classes have equals methods that in common start with the following if-statement for an efficiency reason:

```
public boolean equals(Object o) {if (this == o) return true;
```

Thus, an expression such as o.equals(\past(o)) would return true even if some fields of o are modified during method execution and the equals method of o is defined to compare those modified fields between the receiver and the parameter.

A solution for this problem is debatable. Currently, our prototype, when faced with expressions such as this == o or o == this, returns false if o represents a pre-state value. If this and o point to the same object, a warning is issued. Disequality expressions are handled similarly; true is returned from this != o if o represents a pre-state value, and a warning is issued if this and o point to the same object.

Type of proxies. Consider x.equals(\past(x)) again. Typical equals method returns false if its parameter is not a subtype of the enclosing class C. Substituting a proxy for the parameter of equals should not deceive equals into returning false despite that the type of the actual value the proxy represents is a subtype of C. More generally speaking, substituting a proxy for its actual value should not fool the type system. To achieve that, we assign a proxy the type of its actual value. This can be done by dynamically generating a proxy class as a subtype of the runtime type of its actual value. Simple set of bytecode engineering of non-proxy classes, which includes inserting absent default constructors and removing final modifiers, is necessary to instantiate proxy classes. Such a proxy class also implements the IProxy interface to manifest itself as a proxy.

Dynamic dispatch. Not only x.equals(\past(x)) but also \past(x).equals(x) is valid while resulting in the same result. This is because \past(x) is assigned the type as specific as the dynamic type of x.

5 IMPLEMENTATION THROUGH AOP

In implementing our prototype of an OpenJML [16]-based RAC system for Java programs that supports past expressions, we exploited AOP. Figure 10 shows the part of the aspect that captures our RAC solution for past expressions. In the figure, we list the critical pointcuts and advices used to implement our prototype.[5] Auxiliary parts and the part to handle the migration of difference heaps and fresh sets at method boundaries are omitted.

[5] Slight simplification is done for the presentation.

```
pointcut fieldWrite(Object obj): set(* *.*) && target(obj) && ... ;

pointcut fieldRead(Object obj): get(* *.*) && target(obj) && ... ;

pointcut arrElemWrite(Object[] arr, int idx):
  arrayset() && target(arr) && args(idx) && ... ;

pointcut arrElemRead(Object[] arr, int idx):
  arrayget() && target(arr) && args(idx) && ... ;

pointcut eq(Object thiz, Object o1, Object o2):
  this(thiz) && branch(Object == Object) && args(o1,o2) && ... ;

pointcut diseq(Object thiz,Object o1, Object o2):
  this(thiz) && branch(Object != Object) && args(o1,o2) && ... ;
```

(a) Pointcuts (partial); omissions are indicated by ellipses

```
// 1.updates of difference heap due to field writes
before(Object obj) : fieldWrite(obj) {
  updateDiffHeap(obj,(FieldSignature) thisJoinPoint.getSignature());
}

// 2.updates of difference heap due to array element writes
before(Object[] arr, int idx) : arrElemWrite(arr, idx)
{ updateDiffHeap(arr, idx); }

// 3.field accesses
Object around(Object obj) : fieldRead(obj) {
  if (obj instanceof IProxy) {
    try { return fieldVal(actual(obj),
            (FieldSignature) thisJoinPoint.getSignature());
    } catch (NoFieldCase e) { return proxy(proceed(actual(obj))); }
  } else { return proceed(obj); }
}

//4. array element accesses
Object around(Object[] arr,int idx): arrElemRead(arr,idx) {
  if (isProxyArray(arr)) {
    try { return arrElem(actual(arr), idx);
    } catch (NoArrayElemCase e) {
      return proxy(proceed(actual(arr), idx));
    }
  } else { return proceed(arr,idx); }
}

//5. equality evaluation
boolean around(Object thiz,Object o1,Object o2):eq(thiz,o1,o2) {
  if ((o1 == thiz && o2 instanceof IProxy) ||
      (o2 == thiz && o1 instanceof IProxy)) {
    println("Warning"); return false;
  }
  return proceed(thiz,actual(o1),actual(o2));
}

//6. disequality evaluation
boolean around(Object thiz,Object o1,Object o2):diseq(thiz,o1,o2) {
  if ((o1 == thiz && o2 instanceof IProxy) ||
      (o2 == thiz && o1 instanceof IProxy)) {
    println("Warning"); return true;
  }
  return proceed(thiz, actual(o1), actual(o2));
}
```

(b) Advices (partial)

Figure 10: An aspect to support RAC of past expressions

In defining pointcuts of Figure 10(a), various primitive pointcuts are used. In addition to some standard primitive pointcuts of AspectJ [10] such as set and get, we also used non-AspectJ pointcuts provided by *abc* [1] such as arrayset and arrayget. In addition, we extended *abc* with our custom primitive pointcut branch to accommodate our special needs.

In the sequel, we explain how each of advices of Figure 10(b) matches a specific need of RAC. The first advice updates the difference heap when a field-write takes place in the target program. The body of this advice calls updateDiffHeap with two parameters, i.e., (i) the target object of the current field update and (ii) the field information.

The second advice similarly updates the difference heap when an array-element-write takes place. To make necessary interventions, we use *abc*'s custom pointcut, arrayset, in defining the pointcut of the advice.

The third advice looks up the proper value of the field that is currently being read in either the pre-state or the post-state of the currently running method. If the target object obj is a proxy, the pre-state field value is first looked up in the difference heap by calling fieldVal along with the actual object of the proxy (obtained by actual(obj)) and the field information. This difference-heap-lookup-method fieldVal returns a proxy for the pre-state field value of the given object at its normal termination. If the field under consideration was not modified during method execution, fieldVal raises a NoFieldCase exception because the pre-state field value is not in the difference heap; instead, it is in the current heap. In this case, proceed is called to obtain the unmodified pre-state field value from the current heap, and subsequently method proxy is called to return a fresh proxy for the obtained pre-state value.

The fourth advice is a synonym of the previous one that deals with array element accesses. Instead of an object and field information, an array and an index are used. We use *abc*'s custom pointcut, arrayget, for this advice.

The last two advices handle (dis)equality expressions. To handle those expressions, we need to intervene when two values are compared to each other with operator == or !=. We, however, could not find an appropriate pointcut for our need neither in AspectJ nor *abc*, and extended *abc* with our custom branch pointcut.

5.1 Branch pointcut

Our branch pointcut reveals as join points the binary comparison expressions satisfying the comparison pattern given as the parameter of this pointcut. The grammar of the branch pointcut is "branch($Type_1$ *op* $Type_2$)" where *op* represents a binary comparison operator such as == and !=. A comparison pattern is deemed met if the comparison expression of a program consists of the matching comparison operator in the middle, the left-hand-side sub-expression that is an instance of $Type_1$, and the right-hand-side sub-expression that is an instance of $Type_2$. As usual, the compared values can be exposed to an advice through an additional accompanying args pointcut.

The last two advices of Figure 10(b) use branch pointcuts so that actual values of proxies can be first retrieved before performing comparison by calling proceed. Notice that the call to proceed located at the last line of each advice takes as its parameters actual(o1) and actual(o2) to pass actual values of proxies. We define actual(o) to return o itself if o is not a proxy.

We explained in §4.4 that we treat (dis)equalities with this conservatively; e.g., for the equality case, we consider this and \past(this) to be different from each other. Such conservatism is programmed in the last two advices as an if-statement before the proceed call. The conditional expression checks if one of comparison operands is the same as this and the other operand is a proxy. If that is a case, the same conservative boolean value is returned regardless of the actual value of the proxy.

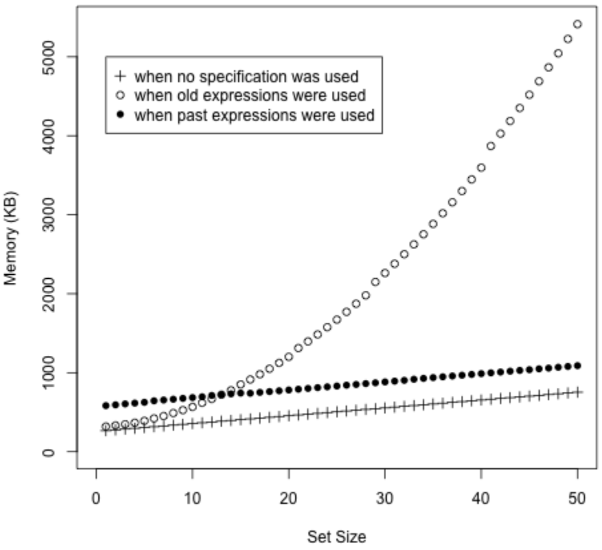

Figure 11: Comparison of memory usage

We originally developed the branch pointcut to support symbolic execution [11]. The path condition of symbolic execution is determined depending on the execution result of the comparison expression of an encountered branch statement. Our branch pointcut can be used to construct the path condition on the fly.

A similar pointcut to our branch pointcut was used in [17] to define the criteria for measuring code coverage such as branch coverage of C# programs. We point out three subtle differences from our branch pointcut, which are mainly caused by different needs. First, not comparison expressions but branch statements are revealed as join points. Second, types of comparison operands are not exposed as if the usage of the pointcut is "branch()". Third, various kinds of branch statements such as if-statements, switch-statements and while-statements are distinguished from one another.

A more recent AOP language such as LogicAJ2 [18] can flexibly define a number of pointcuts including our branch pointcut only with a few primitive pointcuts. However, it does not yet support dynamic information available in AspectJ through this and target pointcuts, and cannot be used for our needs.

5.2 Evaluation

Use of past expressions improves not only comprehensibility and expressiveness of specifications, but also memory usage during RAC as compared to when using old expressions. To compare the memory usage, we measured the memory used when executing the add method of class PatientSet introduced in § 2. Recall that an instance of PatientSet represents a set of Patient instances. We observed how memory usage increased as we added Patient instances one by one to increase the set size. To make the changes of memory usage more easily visible, we padded out Patient with an extra array of bytes. We observed memory usage increase in three cases where method add is either (i) specified using old expressions similarly to Figure 1(a), (ii) specified using past expressions similarly to Figure 3, or (iii) not specified at all.

Figure 11 shows the result of our experiment. The graph shows that memory usage increases in different rates depending on whether past or old expressions are used. While memory usage increased exponentially when old expressions were used for the reason we explained in § 2, only linear growth was observed when past expressions were used. When no specification was used, a similar linear growth was observed as expected.

```
//@ model import org.jmlspecs.models.JML_Elem_Set;
public class PatientSet {
 private Patient[] patients;   private int size;
 //@ public model JML_Elem_Set set;

 //@ private represents set <- abs();
 /*@ private model pure JML_Elem_Set abs() {
   @   JML_Elem_Set ret = new JML_Elem_Set();
   @   for (Patient p : patients) ret = ret.insert(p.clone());
   @   return ret; }
   @*/

 //@ requires set.has(p);
 //@ ensures size == \old(size) && \old(set).isSubset(set);
 //@ also
 //@ requires !set.has(p);
 //@ ensures size==\old(size)+1 && set.has(p) && \old(set).isSubset(set);
 public void add(Patient p) { /* omitted */ }
}
```

Figure 12: A PatientSet stub described using a modeling type

The reason why past expressions do not cause exponential memory usage growth should be evident by now; unlike an old expression, a past expression does not make a copy of the pre-state value.

In the figure, observe the two parallel lines representing respectively the memory usage when past expressions were used and when no specification was used. The gap between these two parallel lines corresponds to the memory overhead of using past expressions. For our prototype, we did not employ optimized collection data structures and simply used Java's standard collections to implement the difference heap. Replacing Java's standard collections with more memory efficient collections will shrink the gap between the two lines.

6 RELATED WORK

We mentioned in § 2 that benefits of our past expressions include encapsulated accesses to pre-state values representing mathematical data types. There is another approach that provides similar benefit by using a library of modeling types [7]. Each modeling type of this library corresponds to a mathematical data type such as a sequence and a set.

For example, Figure 12 shows a PatientSet that uses a modeling type JML_Elem_Set for its specification. The set field of this modeling type is used to specify method add unlike in Figure 1(a) where the patients field was directly used. The fact that set is a specification-only field is indicated with modifier model. The relation between the two fields, set and patients, needs to be made, and this is done through a model method abs and a represents clause in the example; the notation "represents set <- abs()" of the figure means that the value of set is assigned by calling abs. Notice in the definition of abs that a fresh set of a modeling type JML_Elem_Set is returned as a result whose set elements are the clones of the patients-array elements. [6] Note that set and \old(set) of add's ensures clause point to two different instances of JML_Elem_Set because their values are assigned through two different calls of

[6] We assume that class Patient implements interface Cloneable with an appropriate clone method being provided.

abs at the pre-state and the post-state of add, respectively. Also, those two sets contain the elements that are not shared between the sets because of the aforementioned cloning. In this special case, the subset relation, \old(set) $\subseteq$ set, can be validly checked through a method call \old(set).isSubset(set).

The main potential benefit of using the modeling types is that program data such as an array (e.g., patients) can be treated in a specification as a mathematical data such as a set (e.g., set). However, the use of modeling types often entails extra effort to relate modeling-type data to programming-type data. Our past expressions can be used as an alternative that is simpler and arguably more programmer-friendly; programmers of an object-oriented language are already familiar with the concept of encapsulation.

While formal semantics of past expressions is the novelty of this paper, informal descriptions of similar concepts, that are all based on cloning unlike in past expressions, can be found in the literature. Jass [4], another contract language for Java, requires its old expression Old(E) to be used only with expressions E of interface Cloneable. This is because the value of Old(E) is obtained by calling the clone method of E. We pointed out in § 2 the problems of cloning such as performance penalty and semantic flaws. In addition, it seems too intrusive to require a class to be a Cloneable one whenever a variable of that class needs to be used in an old expression.

In our experience in building a prototype, AOP proved to be handy for supporting past expressions during RAC. Similarly, AspectJ was used to support OCL (Object Constraint Language) [15]'s pre expression by Kosiuczenko [12]. While OCL is originally a specification language for a modeling language (i.e., UML), and thus independent of a specific programming language, tools such as ocl2j [8] use OCL to monitor behaviors of Java programs. The semantics of OCL's pre expressions is generally identical to the one of old expressions. Kosiuczenko used AspectJ to store pre-state field values at their modification sites. While being similar to our difference heap, his approach is distributed unlike our centralized difference heap; for each field f of class C whose pre-state value needs to be stored, a fresh field fHIST is inserted into C to keep track of the history of f. Also, a fresh method fATPre() is added to C to be used when getting the pre-state value of f. Our implementation of the difference heap is less intrusive; we do not add extra fields and methods to the original program. In addition, we used AOP not only to implement the difference heap but also to access fields in desired time contexts and compare objects that may reside in different time contexts.

7 CONCLUSION AND DISCUSSION

In this paper, we have (i) pointed out the problems of existing old expressions of contract languages, most notably the lack of encapsulation, (ii) suggested a past expression as an alternative, and (iii) showed how past expressions can be supported during RAC through AOP in a memory-efficient way.

While we focused on RAC in this paper in terms of tool support, this does not mean that static checking with past expressions is impossible. In fact, it seems, as compared to RAC, straightforward to implement static checking following the formal semantics we provide in this paper. Indeed, our prototype tool can also support static checking at the rudimentary level. However, static checking poses its own challenges. For example, it is not yet clear to us how the add method of a generic class Set<T> can be statically checked against a specification where past expressions are used. We leave static-checking support for past expressions as future work.

Acknowledgements

We thank Bruno Oliveira for his valuable comments. This work was partially supported by a Ministry of Education research grant MOE2010-T2-2-073 (R-252- 000-456-112 and R-252-100-456-112) from Singapore, a US National Science Foundation grant (# 0709169 and # 0644288), and a US Air Force Office of Scientific Research grant (contract FA9550-09-1-0138).

References

[1] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. *abc*: an extensible AspectJ compiler. In *AOSD*, pages 87–98, 2005.

[2] J. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley, 2003.

[3] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *CASSIS*, pages 49–69, 2004.

[4] D. Bartetzko, C. Fischer, M. Möller, and H. Wehrheim. Jass - Java with assertions. *ENTCS*, 55(2):103–117, October 2001.

[5] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *STTT*, 7(3):212–232, 2005.

[6] P. Chalin. Logical foundations of program assertions: what do practitioners want? In *SEFM*, pages 383–392, 2005.

[7] Y. Cheon, G. T. Leavens, M. Sitaraman, and S. Edwards. Model variables: cleanly supporting abstraction in design by contract. *SPE*, 35(6):583–599, May 2005.

[8] W. J. Dzidek, L. C. Briand, and Y. Labiche. Lessons learned from developing a dynamic OCL constraint enforcement tool for Java. In *MoDELS*, pages 10–19, 2006.

[9] C. B. Jones. *Systematic Software Development using VDM*. Prentice-Hall, 1990.

[10] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. In J. Knudsen, editor, *ECOOP*, pages 327–354, 2001.

[11] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[12] P. Kosiuczenko. On the implementation of @pre. In *FASE*, pages 246–261, 2009.

[13] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: a behavioral interface specification language for Java. *ACM SIGSOFT Software Engineering Notes*, 31(3):1–38, May 2006.

[14] B. Meyer. *EIFFEL: The language and environment.* Prentice Hall, 1992.

[15] OMG. *Object Constraint Language - version 2.2*, 2010.

[16] OpenJML. http://jmlspecs.sourceforge.net/.

[17] H. Rajan and K. Sullivan. Aspect language features for concern coverage profiling. In *AOSD*, pages 181–191, 2005.

[18] T. Rho, G. Kniesel, and M. Appeltauer. Fine-grained generic aspects. In *FOAL*, pages 29–35, 2006.

[19] J. M. Spivey. An introduction to Z and formal specifications. *Software Engineering Journal*, 4(1):40–50, January 1989.