On the Implementation of @pre

Piotr Kosiuczenko

Institute of Information Systems, WAT, Warsaw

Abstract. The paradigm of design by contract provides a transparent way of specifying object-oriented systems. There exist a number of languages for contractual specification including OCL, JML and Spec#. Nevertheless, there are still a number of research problems concerning this approach. One of them is the implementation of primitive @pre in OCL or equivalently old in JML. Those primitives are used in post-conditions to refer to attribute values in states preceding an operation execution. There are a number of implementations of this operators, but all suffer logical and computational problems. In this paper, existing approaches to the implementation of @pre are discussed and a new solution is proposed.

1 Introduction

The main goal of software engineering is the construction of correct software. There exist various methods for ensuring that software meets requirements ranging from testing to automatic theorem proving. They have been developed in the late 1960s, most prominently by Floyd [6] and Hoare [9]. In the following years several approaches to system specification have been developed. Contracts are the prevailing way of specifying object-oriented systems from the user point of view (see [13]). A contract consists of three basic constraint types: invariants, pre- and post-conditions. The system consistency is ensured by invariants. A pre-condition specifies in which states a method can be called. A post-condition specifies the system state after the method execution.

Eiffel [14] is an object-oriented language including in a pioneering way contractual specification. It turns contracts into run-time checks of program correctness. The implementation of post-conditions is not an easy thing since it is necessary to compare attribute values in method's pre- and post-state. Of course copying the whole pre-state before a method is executed is out of question due to time- and memory-cost. Moreover if a method is called several times in a recursive manner, then it is necessary to copy the whole state again and again. Eiffel avoids this problem by saving before a method execution values of those attributes whose old values are referred to in the post-condition. However as we explain below, this approach has serious drawbacks in case of high level specification languages such as Object Constraint Language (OCL, see [15]).

Old attribute's values are accessed with the help of `@pre` in case of OCL and `old` in case of JML and Spec#. These primitives can be used in post-conditions and allow one for the comparison of attribute values in states before and after

M. Chechik and M. Wirsing (Eds.): FASE 2009, LNCS 5503, pp. 246–261, 2009.
© Springer-Verlag Berlin Heidelberg 2009

operation execution; e.g. `salary = salary@pre + amount`. In case of OCL, one can use `@pre` with an attribute, an association-end and a query (OCL names the first two 'property'). Although the idea is simple, those primitives are not easy to implement.

The Eiffel approach requires restrictions of post-conditions' syntax. It is necessary to restrict post-conditions to formulas of the form: $t_0[t_1@pre/x_1, \ldots, t_n@pre/x_n]$ (we use here the OCL notation), where term t_0 does not include `@pre` and where t_i`@pre` is obtained from t_i by replacing every attribute, association-end and query `a` by `a@pre`, for i = 1,..., n; for example, term `(self.a.b)@pre` is an abbreviation for OCL term `self.a@pre.b@pre`. $[t_1@pre/x_1, \ldots, t_n@pre/x_n]$ denotes simultaneous substitution of terms t_i for variables x_i, for i = 1,..., n. Values of terms t_i are computed before the underlying method is executed, saved and then used after the method execution to compute the value of the post-condition. It should be noted that term `self.a@pre.b@pre.c.d->size()=self.a->size()` is of the above mentioned kind, since it can be presented in the form: `(x.c.d->size() = self.a->size())[(self.a.b)@pre/x]`. However, it is not possible to present terms such as `self.a.b@pre` in this form. The value of `self.a` is not known in the pre-state and consequently cannot be pre-computed. It is possible to formulate in OCL more natural constraints of the latter kind; e.g. one can specify an operation which creates a new department of a company and makes sure that for all employees assigned to this department their salaries are increased.

It is disputable how severe the above mentioned syntax restriction is. It is true that most post-conditions can be written in that form. However, there are more serious problems with this approach. For example there are subtle problems with quantification (cf. [1]). Even more serious is the problem of cloning and more generally copying of large parts of the state. Cloning disallows dealing directly with object identity. Deep clones are problematic in the presence of circular references. There are also problems in case when more then one object references the same object; in this case one needs to avoid multiple clones of the referenced object. If objects are cloned then reference identity cannot be used for comparison.

Computing all potentially needed values in the pre-state could be very time- and space-consuming. In the case of `if ex then ex1 else ex2 endif`, we have to compute value of an expression `e` in the case when `e@pre` occurs in `ex1` or `ex2`. For example, let us consider expression `if 1 + 1 = 2 then 1 else q@pre endif` where `q` is a computationally complex, integer-valued query. Obviously, in general there is no need to evaluate `q` in the pre-state to compute the value of the whole expression. Nevertheless when the Eiffel approach is used, it is necessary to evaluate `q` in the pre-state. This unnecessarily increases the time and space complexity. Moreover if `q` does not terminate, then the evaluation of the whole expression does not terminate. Thus evaluation of constraints may not only slow down program execution, but also prohibit termination.

The problems with the Eiffel approach to `@pre`-values can be classified as follows:

- the support only for the restricted form of constraints
- the need of extensive cloning

– the lack of transparency in respect to object identity
– a potential increase of computational complexity

In this paper we present an algorithm addressing those problems. This algorithm is implemented in AspectJ. AspectJ is the most popular aspect-oriented language. (We refer the interested reader to [11] for a good introduction to aspect-oriented programming). The algorithm can be implemented in other aspect-oriented languages and also using reflective features present in languages such as Java and C#.

This paper is organized as follows. In section 2, we comment on the related work. In section 3, we present an example and use it to describe problems with existing approaches and to explain our algorithm. In section 4, we present the algorithm for computing `@pre`-values; we show how to deal with collection types and inheritance; we argue that this algorithm does not increase the time-complexity of constrained methods and that checking old values has constant time-complexity. Section 5 concludes this paper.

2 Related Work

There exist various languages for contractual system specification. In the realm of Java, there exist Java Modeling Language (JML, see [3,12]). It allows one for a modular system specification based on a type system grouping objects in master-slave hierarchies, the master being fully in control of its slaves. There is the concept of visible states being the moments when a public or a non-helper method is called or terminates. In the realm of .Net, and in particular C#, there is Spec# [1]. This language is closely related to JML, the main dissimilarity being a different approach to modularity and the relativization of visibility notion to objects' states. An efficient implementation was the primary goal of those specification languages instead of expressivity. Moreover, both specification languages are intimately related to the corresponding programming languages and therefore they are rather low level.

There exist high level, programming independent languages for contractual system specification, the most prominent being OCL [15]. It focuses on expressivity rather than an efficient implementation. There exist numerous tools for monitoring the satisfaction of OCL constraints. We mention here only Dresden OCL Toolkit (DOT) [4], jContractor [10] and OCL2J [5,2] (see [17] for an overview of other tools). Those tools implement OCL partially due to the complexity of the language. Due to high abstraction level of this language, contract monitoring often causes a significant slowdown in program execution.

All above mentioned languages follow the Eiffel approach. Interestingly also the current research focuses on the implementation problems concerned with this approach (cf. e.g. [5,2]). Archiving attributes before method execution is simple in the case of basic OCL types and classes. In the first case a single value is stored in a variable before method execution. In the case of terms of a class type, only an object reference is stored; the object itself is not cloned. If a term t_i (see section 1) is of a collection type, then the situation is more complicated. A collection

is determined by elements it contains. Thus a clone of a collection has to contain every element of the original collection. As pointed out in [5], using `@pre` on a collection requires in most cases a duplication of the collection. In some cases, if operations like `size()` are used on the collection, then only the corresponding value has to be stored. If a post-condition relates attribute's values before and after operation execution and those values are of a collection type, then the collection before the operation execution has to be cloned. As pointed out in [5,2], cloning of collections is the major slow down factor in the automatic constraint evaluation and the authors classify this problem as a research challenge.

Another sort of problems emerges when objects are cloned in the pre-state, as it is done in OCL2J. As pointed out in [2], in this approach objects are cloned before being returned by a method, as required by the so called 'strong encapsulation principle'. In this case, one has to deal with user defined objects equality instead of identity (being identity of references or addresses). Some programming styles encourage exclusive use of user defined equality instead of identity; however a specification language should be implementation style independent. There are a number of logical problems when in programs identity comparison, denoted in Java by `==`, have to be replaced by user defined equality relation based on values of object attributes. Redirecting of links is needed when one wants to deal with true object identity and not a kind of equality based on attributes identification (cf. [2]).

As in the case of OCL2J, jContractor [10] allows post-conditions to refer to the state of an object at method entry, however the method is a bit different. A class includes an instance variable named `OLD` of the same type as the class. Post-conditions access old values of properties by referencing `OLD`. During compilation, resulting byte-code is instrumented in such a way that the reference to `OLD` is routed to a clone of the object created at method entry using Java method `clone()`. When a method is executed and its post-condition includes `@pre`, a clone of the object is created and stored in `OLD` [10]. As pointed out by the authors, there is a problem with object cloning if the method calls are nested, since in such a case the older clone is overwritten by a newer one. Therefore jContractor adopts a stack based approach: when execution enters a method that needs to save `OLD`, an object clone is created and pushed onto a stack; when the method terminates, the object is removed from the stack and used to check the post-condition. Pushing an object onto and removing it from the stack when the method is called or terminates does not differ from defining local variables which are then pushed onto and removed from the program stack. In fact, it is equivalent to wrapping a constrained method in another one which checks its pre-condition, saves old values, executes the original method and checks its post-condition. However in the case of post-conditions requiring navigation via several objects, as for example the OCL constraint presented in the introduction, deep cloning is needed. It should be noted that there is also the so called Memo pattern aimed at storing and handling past object states [8]. It should also be noted that there exist various methods for implementing OCL constraints in aspect-oriented languages, in particular AspectJ (cf. e.g. [18,16]).

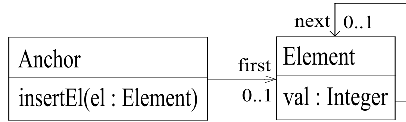

Fig. 1. List

3 Example

In this section we present an example in order to describe the problems and explain the proposed solution. The class diagram in Fig. 1 models a list composed of an anchor object of class `Anchor` and a number of elements instantiating class `Element`. Anchor objects have attribute `first` storing the first element of the list. There is also method `insertEl(Element el)` for inserting elements into the list. `Element` objects have attribute `val` storing integers and attribute `next` pointing to the next element. Insertion of an element into a list may be specified by a formula saying that the set of list elements is enlarged by the new element.

```
context Anchor::insertEl(el : Element) post insertPC :
self.elements = self.elements@pre->union(Set{el})
```

where

```
context Anchor def :
elements : Set(Element) =  if self.first = null then Set{}
else self.first.successorsOf(Set{self.first}) endif
context Element def :
successorsOf(Acc : Set(Element)) :  Set(Element) =
if self.next = null or Acc->includes(self.next) then Acc
 else self.next.successorsOf(Acc->union(Set{self.next}) endif
```

One can use in this post-condition `including`, but this would not help us to avoid cloning. Eiffel approach requires copying of the list before operation execution and then comparison of the new form of the list with the copied one. Using datatypes such as lists would require deep cloning causing problems listed in the introduction.

We implement `insertEl` in a very inefficient way. An element `el` is inserted into the list if the list is empty or `el.val` is smaller than the value of the first element. If not, then the list elements are removed as long as the first element does not exist or its value is larger than the value of the inserted element, then `el` is inserted and afterwards all removed elements are inserted too. Execution of this method causes a cascade of recursive calls. If we insert into an empty list n+1 elements with increasing values, then we need to make n copies of the list; i.e. 1+2+...+n = n(n+1)/2 copies of object references.

We present implementation of classes `Anchor` and `Element`. It should be noted that during execution of `insertEl` several other calls to this method can be made if the inserted element is not smaller than the first element saved in the list. In particular the call stack can contain several calls to this method. On the

other hand, during execution of this method some calls may be terminated. This shows that a proper stamping policy is needed to guarantee that the method calls are numbered uniquely and that one needs a proper logic to figure out which snapshots are valid.

```
public class Anchor {
  public Element first;
  public void insertEl(Element el) {
    if(first == null) {
      first = el; el.next = null;
    }
    else if(el.val <= first.val)
      el.next = first; first = el;
    else {
      Element oldFirst = first;
      first = first.next; oldFirst.next = null;
      insertEl(el); insertEl(oldFirst);
    }
  }
}

public class Element {
  public Integer val = 0; public Element next = null;
  public Element(Element n, Integer v) {
    next = n; val = v;
  }
}
```

4 Implementation of @pre

In this section we present an implementation of the primitive `@pre` in AspectJ. We explain how to deal with collection types other than lists (i.e. vectors and arrays), queries and inheritance. We present a complexity estimation of the proposed algorithm. This algorithm has the advantage that we neither need to restrict syntax of post-conditions nor redirect references. In our approach we avoid the problems of deep cloning. However we need to reassemble the states of objects. Therefore we have to treat cloned object parts carefully and we need a logic for reassembling objects and for navigating via archived and not archived properties.

4.1 Implementation in AspectJ

Our algorithm is implemented in AspectJ. This language allows us to instrument classes with additional attributes and methods, to add so called pointcuts, for registering relevant events, and advices, for handling those events. As it is common in distributed systems, we use a kind of time-stamp to be able to reassemble objects states which existed at different times, but in our case those stamps refer to method calls.

Instrumenting classes looks as follows. If a property (i.e. attribute or association-end) `a` of type `T` occurs in a post-condition in the form `a@pre`, then we instrument the class corresponding to `a` by superimposing a history attribute `aHIST` of type `Stack<SnapshotVal<T>>` to store the history of `a`, a pointcut which listens to changes of `a` and an advice which is responsible for storing values of this attribute. For every class possessing such attributes we add a new aspect. The value of `a@pre` is returned by a superimposed method `aATpre()`. It should be noted that attributes which do not occur in post-conditions in the above mentioned form do not need to be archived. Objects of parametric class `Stack<SnapshotVal<T>>` store histories of attributes in the form of a stack. Class `SnapshotVal<T>` defines attribute snapshots, i.e. a temporary value of an attribute plus the corresponding time-stamp, or as we call it later meter-reading. Class `Meter` stores information about the number of relevant method calls on the program stack and the lists of objects modified during execution of those methods; a method is relevant if it has a post-condition with attributes which require archiving. `MeterAspect` handles calls to relevant methods. Class `Archive` includes the core logic for value archiving and implements method `aATpre()`.

We describe our algorithm more precisely using the list example. In this example there are two attributes which must be archived: `first` and `next`. They do not occur in the post-condition directly but are used to define set `elements` storing list elements. It should be noted that values of attribute `val` are not archived, since `val@pre` does not occur in the post-condition.

Class `Meter` handles time-stamps. They correspond to the number of calls of `insertEl` which are still on the program stack. This number is stored in attribute `meterValue`. `Meter` handles also lists of objects modified during those method calls. Those lists are stored as attributes of objects of class `SnapshotModified`. Those objects are stored in stack `meter`. The goal is to minimize the memory use. After a method termination the corresponding list of modified objects is scanned and outdated snapshots are removed.

```
public abstract class Meter {
  static int meterValue = 0;
  static final Stack<SnapshotModified> meter =
                                 new Stack<SnapshotModified>();
  public static int getReading() {
    if(meter.size() == 0) return 0;
    return meter.top().meterReading;
  }
  public static void increaseMeter() {
    ++meterValue;
    SnapshotModified n = new SnapshotModified(meterValue);
    meter.push(n);
  }
  public static void decreaseMeter() {
    meter.pop();
  }
}
```

Note that part of `Meter` functionality can also be implemented by an aspect with aspect-object creation per control-flow; in that case `meter` attribute of class `Stack<SnapshotModified>` could be replaced by an attribute of snapshot class.

`MeterAspect`is an aspect implementing the logic concerning calls of methods with a post-condition, in this case `insertEl`.When the method is called, the meter is increased, a new object of class `SnapshotModified`storing lists of modified objects is created, stamped with the current meter-reading and pushed onto the meter-stack by executing `increaseMeter()`.After `insertEl`terminates, if the stack is not empty, then the top-most object storing lists of modified object is removed from the stack and objects stored in the lists are checked for inclusion of outdated snapshots. If the stack is empty, then all history attributes are emptied and `meterValue`is set to 0, since there is no active call of a relevant method on the program stack. Then the lists of modified objects are emptied. Note that pointcut `ins1`catches only calls to method `insertEl`;consequently all other method calls do not have influence on the `meterValue`and are in a sense irrelevant. It should also be noted that aspect `MeterAspect`has the highest priority. This is necessary since during a relevant method call advices for archiving and retrieving attribute values must be executed after increasing `meterValue`and adding the new modified snapshot, and before `meterValue`is decreased and the modified stack is popped.

```
public aspect MeterAspect {
  declare precedence: MeterAspect, *;
  public pointcut ins1() : call(void insertEl(Element));
  before() : ins1() {
    Meter.increaseMeter();
  }
  after() : ins1() {
    SnapshotModified sm = Meter.meter.top();
    Meter.decreaseMeter();
    if(Meter.meter.size() > 0) {
      for(Anchor o : sm.modifiedFirst) o.firstHIST.clean();
      for(Element o : sm.modifiedNext) o.nextHIST.clean();
    }
    else /* Meter.meter.size() == 0 */ {
      for(Anchor an : sm.modifiedFirst) an.firstHIST.empty();
      for(Element e : sm.modifiedNext) e.nextHIST.empty();
      sm.modifiedFirst.removeAllElements();
      sm.modifiedNext.removeAllElements();
      Meter.meterValue = 0;
    }
  }
}
```

Class `Archive` stores the logic for handling old attribute values. This class contains generic methods for the storing and the retrieval of attribute values. `getValueATpre` returns the old value of an attribute. It contains two parameters: `val` corresponding to the current value of an underlying attribute and `st`

corresponding to the attribute's history-stack. If `st` is empty, then the attribute was not modified and `val` is returned. If not, then outdated snapshots are removed from the attribute's history-stack. If the topmost element in the stack has stamp smaller than the current meter-reading, then it means that during the recent method call the value was not modified and `val` is returned. If not, then the attribute was modified during this or a later, but already terminated, method call and the stored topmost value is returned. Method `getLastUpdateTime` returns last update time of an attribute. This value is taken from the topmost snapshot in the attribute's history-stack, if it is not empty; in the other case 0 is returned. `doArchiving` is meant for storing attribute's snapshots on an underlying attribute's history stack. This method has four parameters: `st` corresponds to the history stack of the underlying attribute, `modTop` corresponds to the topmost list of objects for which the underlying attribute was modified, `modBottom` corresponds to the bottommost list, `target` corresponds to the object for which the relevant attribute is above being modified, `cur` corresponds the current attribute's value. If stack `st` is empty, then a new attribute snapshot is created with the current meter-reading and put onto the stack. At the same time the target object is saved in `modBottom`. The object is saved at the bottom, since it is its first modification and therefore the saved value is the `@pre`-value for all previous method calls. If `st` is not empty, then it is checked if the attribute's value was already saved. If it was not saved, then a new attribute snapshot with the current meter-reading is created and pushed on stack `st`; at the same time the target object is saved in `modTop` which is the top list of objects for which the underlying attribute was modified. If it was saved, then only the meter-reading in the corresponding snapshot is updated.

```
  public class Archive {
  static <T> T getValueATpre(Stack<SnapshotVal<T>> st, T val) {
  if(st.size() == 0) return val;
  st.clean();
  if(st.topReading() < Meter.getReading()) return val;
  else return st.top().value;
}
static <T> Integer getLastUpdateTime(Stack<SnapshotVal<T>> st) {
  if(st.size() == 0) return 0;
  st.clean();
  return st.top().meterReading;
}
static <T, S> void doArchiving(Stack<SnapshotVal<S>> st,
      Vector<T> modTop, Vector<T> modBottom, T target, S cur) {
  if(st.size() == 0) {
    st.push(new SnapshotVal<S>(cur, Meter.getReading()));
    modBottom.add(target);
  }
  else if(Meter.getReading() > st.top().meterReading) {
    if(st.top().value != cur) {
      st.push(new SnapshotVal<S>(cur, Meter.getReading()));
      modTop.add(target);
    }
    else st.top().meterReading = Meter.getReading();
```

```
    }
  }
}
```

It should be noted that in the case when the archived value coincides with the attribute value before it is set for the first time during a method execution we need to update the meter-reading of the actual snapshot. This is because the attribute can be changed several times during the method execution. If the meter-reading was not updated before the first modification, then during the next modification the stored value would be treated as out of date and a value different from the stored, and in fact correct one, would be saved as the value in the pre-state.

Abstract class `Snapshot` has attribute `meterReading` for saving the current meter-reading of a snapshot, i.e. the number of relevant method calls on the program stack.

```
public abstract class Snapshot {
  int meterReading;
}
```

`SnapshotModified` extends `Snapshot` and is meant for storing the lists of objects for which attributes requiring archiving were modified during a method execution. In our case, these are attributes `first` and `next`. Inherited attribute `meterReading` stores the current meter-reading.

```
public class SnapshotModified extends Snapshot {
  public Vector<Anchor> modifiedFirst = new Vector<Anchor>();
  public Vector<Element> modifiedNext = new Vector<Element>();
  public SnapshotModified(Integer i) {
    meterReading = i;
  }
}
```

`SnapshotVal` is a parametric class extending `Snapshot`; its objects are used to store attributes' snapshots. Attribute `value` stores the attribute's value.

```
public class SnapshotVal<T> extends Snapshot {
  T value;
  public SnapshotVal(T el, Integer t){
    value = el;
    meterReading = t;
  }
  //... other SnapshotVal constructors
}
```

Parametric class `Stack` implements a stack with methods for popping and pushing snapshots. We implement this class using `Vector` but it can be implemented using class `Stack` from the Java API standard library. Apart of above mentioned methods, it contains method `subTop` for returning the element below

the topmost position. Method `clean()` is used for removing outdated snapshots; i.e. snapshots located at the top of the stack whose meter-reading is larger then the current meter reading and who have a direct follower, returned by `subTop()`, with meter-reading larger than or equal to the current meter reading. It should be noted that the cleaning must be done in a while loop, since in general it is not enough to remove only the topmost outdated snapshot. small`bottom()` returns the bottommost element and `empty()` removes all elements from the stack.

```
public class Stack<S extends Snapshot> {
  Vector<S> stack = new Vector<S>();
  public void push(S el) {
    stack.add(el);
  }
  public S top() {
    return stack.lastElement();
  }
  public int topReading() {
    if(stack.isEmpty()) return 0;
    return top().meterReading;
  }
  public S pop() {
    S s = top();
    stack.remove(stack.size()-1);
    return s;
  }
  public S subTop() {
    return stack.get(stack.size()-2);
  }
  public int size() {
    return stack.size();
  }
  public void clean() {
    while(size() >= 2 &&
                  subTop().meterReading >= Meter.getReading())
      stack.remove(stack.size()-1);
  }
  public S bottom() {
    return stack.firstElement();
  }
  public void empty() {
    stack.removeAllElements();
  }
}
```

For every class `C` with attributes requiring archiving we introduce aspect `ArchiveC` which superimposes corresponding history attributes and methods returning old values of attributes. This aspect also manages setting of attributes. In our example, for classes `Anchor` and `Element` we introduce aspects `ArchiveAnchor` and `ArchiveElement`. `first` is the only attribute of class `Anchor` which has to be

archived. For this attribute method `getFirstATpre()` is superimposed on `Anchor`. This method is implemented with the help of `getValueATpre` and `getLastUpdateTime`. Every manipulation of `first` is detected by pointcut `modFirst`. If the current meter-reading is larger than 0, meaning that there is a relevant method on the stack, then the archiving is performed by `doArchiving`. It should be pointed out, that the archiving is performed only if there is a relevant method on the programm stack, or equivalently the `meterValue` is larger than 0. This is due to the fact that if no method with a post-condition is executed, then there is no need for archiving the pre-state. The archiving is needed first when a relevant method, in this case `insertEl`, starts to execute.

```
public aspect ArchiveAnchor {
  public Stack<SnapshotVal<Element>> Anchor.firstHIST =
                              new Stack<SnapshotVal<Element>>();
  Element Anchor.getFirstATpre() {
    return Archive.getValueATpre(firstHIST, first);
  }
  Integer Anchor.getFirstLastUpdateTime() {
    return Archive.getLastUpdateTime(firstHIST);
  }
  pointcut modFirst(Anchor target) :
    target(target) && set(Element Anchor.first);
  before(Anchor target) : modFirst(target) {
    if(Meter.getReading() > 0) {
        Archive.doArchiving(target.firstHIST,
          Meter.meter.top().modifiedFirst,
            Meter.meter.bottom().modifiedFirst,
              target, target.first);
    }
  }
}
```

`ArchiveElement` is an aspect analogous to `ArchiveAnchor`. It instruments class `Element` by superimposing history attribute `nextHIST` and method `getNextATpre()` on class `Element`. It defines also pointcut `modNext` which detects changes of attribute `next` and the corresponding advice which does the archiving of old `next`-values.

```
public aspect ArchiveElement{
  public Stack<SnapshotVal<Element>> Element.nextHIST =
                              new Stack<SnapshotVal<Element>>();
  Element Element.getNextATpre() {
    return Archive.getValueATpre(nextHIST, next);
  }
  Integer Element.getNextLastUpdateTime() {
    return Archive.getLastUpdateTime(nextHIST);
  }
  pointcut modNext(Element target) :
    target(target) && set(Element Element.next);
  before(Element target) : modNext(target) {
    if(Meter.getReading() > 0) {
      Archive.doArchiving(target.nextHIST,
```

```
        Meter.meter.top().modified Next,
          Meter.meter.bottom().modifiedNext,
            target, target.next);
    }
  }
}
```

Post condition `insertPC` (see section 3) can be checked with an aspect of the following form. It should be noted that in contrast to aspect-oriented implementations of the Eiffel approach we do not need an around-advice for passing saved values; in such an advice first the values are saved then the underlying method is executed with `proceed` command and then the post-condition is checked (cf. e.g. [18,16]). We do not store any values before method execution. Therefore it suffices to use an after-advice. This makes the constraint implementation more elegant and efficient, since the around-advice causes a significant method slowdown.

```
public aspect ConstraintsAspect {
  public pointcut ins1(Anchor a, Element el) :
    target(a) && args(el) && call(void insertEl(Element));
  after(Anchor a, Element el) : ins1(a, el) {
    //... check the post-condition
  }
}
```

4.2 Collections, Queries and Inheritance

As mentioned in subsection 2 (see also [5]), in the case of collections, the Eiffel approach requires deep cloning. In subsection 4.1, we have shown how to deal with lists. We can deal in a similar way with arrays and vectors. The problem with those collections is that they cannot be directly instrumented by AspectJ, since it is not possible to superimpose attributes and methods on classes from the standard Java API library and other predefined types.

In case of arrays, there is no array class as such. In this case we need to replace arrays with classes in order to instrument them. An array of the form `C[]` can be replaced by class `ArrayC` with an attribute `array` of type `C[]` and history attribute `arrayHist` of type `Stack<SnapshotVal<C>>[]`. The old values of the array can be dealt with as in the case of object valued attributes; the difference is that we have to access certain positions in the array; i.e. we have to define method `getElementATpre(int i)`, which returns the correct value. The method can be defined as `aATpre()` the only difference is that it accesses values at position `i` in `array` and history stacks occurring at position `i` in `arrayHist`.

In case of vectors, we need to extend class `Vector` to be able to instrument it with AspectJ. Thus we can define class `VectorI<C>` which extends `Vector<C>` and has two additional attributes: `vHist` of type `Vector<Stack<SnapshotVal<C>>>` for storing elements' snapshots and `sizeHist` for storing old vector sizes, or more

precisely the history of size(). We need also to define new query methods for accessing values corresponding to the pre-state. We have implemented in a natural way methods sizeATpre(), getATpre(int i), lastElementATpre() corresponding to methods size(), get(int i), lastElement(), respectively, of class Vector. The implementation has been based on method Archive.getValueATpre, but takes into consideration the change of length. We skip it here due to the lack of space.

Dealing with queries is very simple in our approach. Every occurrence of q@pre can be replaced by query qATpre, where the body of qATpre is obtained from the body of query q by replacing every attribute b by getBATpre() and every invocation of a query r by the corresponding query rATpre.

The design by contract approach assumes that constraints are inherited by subclasses and that subclasses may be additionally constrained. If class Anchor was extended by class AnchorB and the method insertEl possessed in AnchorB an additional post-condition, then we would have to make sure that also the additional post-condition is checked. This can be achieved by defining another aspect for monitoring constraints which differs from ConstraintsAspect in that parameter a in pointcut ins1(Anchor a, Element el) is restricted to objects of class AnchorB, i.e. by defining an additional pointcut with signature ins2(AnchorB a, Element el). In this case only methods executed on objects of class AnchorB, and of its subclasses, will be constrained by the additional post-condition corresponding to ins2.

Overloading attribute names differs from method inheritance. Attributes are bound at compilation time. Whereas methods are bound at execution time depending on the class of the actual implicit parameter. In our approach we introduce method aATpre for every attribute a which requires archiving; thus we have to make sure that different attributes correspond to different methods. This can be achieved either by renaming attributes or by defining different methods for defacto different attributes. In the second case, it is necessary to replace every occurrence of a@pre by the corresponding method. We can distinguish between defacto different attributes by checking the type of every attribute occurrence in a post-condition.

For example let us assume that class AnchorB extends class Anchor and that both classes define attribute first. We can either rename the attribute in class AnchorB to avoid name clash or define two different methods for accessing old values of those two different attributes. In the second case we can superimpose methods getAnchorFirstATpre and getAnchorBFirstATpre returning the old attributes' values as well as the corresponding history attributes Anchor.firstHIST and AnchorB.firstHIST. Observe that in the case of history attributes we do not need to use different names since they are superimposed on different classes. Finally for every subterm of the form t.first occurring in the post-condition we have to replace first by getAnchorBFirstATpre if t defines objects of class AnchorB, or by getAnchorFirstATpre in the other case.

4.3 Complexity

In this section we informally show that the algorithm presented in subsection 4.1 does not increase the time-complexity class of the constraint validation nor the time complexity class of constrained methods.

More precisely, let `m` be a method with post-condition `postCond` including the primitive `@pre`. The execution of book-keeping activities performed by the algorithm to archive old values of attributes does not increase the time-complexity class of `m`. Let post-condition `postCond'` be obtained form `postCond` by replacing every occurrence of an attribute `a@pre` by query `aATpree`. The evaluation of `postCond'` has the same time complexity as it would have when evaluation of `a@pre` required one time unit. This is due to the fact that the execution of `aATpree` apart of cleaning requires a constantly bound number of steps. Since the `@pre`-values are computed when needed, there is no problem with unnecessary pre-computation of term values, in particular with unnecessary execution of nonterminating queries (see the introduction).

Our method does not increase time complexity class of constrained methods since setting an attribute is accompanied by at most one snapshot archiving and removal and there is only a bound number of steps needed for those two operations. A call of a constrained method results in increasing of `meterValue`, creation of a new `ModifiedSnapshot` object and pushing it onto the stack. When the method terminates, the corresponding lists of modified objects are scanned, irrelevant snapshots are removed from the history stacks and the `ModifiedSnapshot` object is removed from the stack. The creation and removal of a `ModifiedSnapshot` object requires bound time; scanning of modified object lists and removal of outdated snapshots from an attribute's history can be accounted for when counting the steps associated with the corresponding attribute modification.

The space complexity may be increased and in the worst case equal to the time complexity of `m`. If in constraint $t_0[t_1@pre/x_1, ..., t_n@pre/x_n]$ (see the introduction) term t_0 does not contain `if then else endif`-statements and if terms t_i are of basic OCL type or return single objects as values, then it is not necessary to archive attributes' values. It is enough to save values of t_i. In this and other cases it may be advantageous to use the Eiffel approach.

In some cases a combination of the Eiffel approach and the algorithm described in subsection 4.1 may be the optimal solution. Of course pre-computing values of some terms and then archiving all values of attributes occurring in those terms is not reasonable. However it makes sense in the case when the set of attributes which require archiving is disjoint from the set of attributes occurring in terms t_i.

5 Conclusion and Future Work

In this paper we discussed currently existing approaches to the implementation of the primitive `@pre`and pointed out that they are all based on the Eiffel approach to `old`-implementation. We listed the corresponding problems and proposed a new algorithm which avoids those problems. This algorithm does not require restriction of the post-condition syntax; no collection cloning is needed and the identity of objects is preserved. Moreover, a post-condition can be implemented

with an after-advice, instead of an around-advice, what makes the constraint implementation more elegant and efficient. We investigated also the complexity of this algorithm and showed that it does not increase time-complexity of method execution and constraint evaluation.

In the future we are going to implement this algorithm using Java reflectivity features. We are going to investigate its defacto time and space overhead, and to figure out when the Eiffel approach and our algorithm can be most efficiently combined. We are also going to provide a formal proof of algorithm's correctness.

References

1. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# Programming System: An Overview. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 49–69. Springer, Heidelberg (2005)
2. Briand, L., Dzidek, W., Labiche, Y.: Using Aspect-Oriented Programming to Instrument OCL Contracts in Java, Tech. Rep. SCE-04-03, Carleton Univ. (2004)
3. Darvas, A., Müller, P.: Reasoning About Method Calls in JML Specifications. In: Proceedings of the 7th Workshop on Formal Techniques for Java-like Programs (FTfJP 2005), Glasgow, Scotland (July 2005)
4. DOT, Dresdener OCL Toolkit, `http://dresden-ocl.sourceforge.net/`
5. Dzidek, W., Briand, L., Labiche, Y.: Lessons Learned from Developing a Dynamic OCL Constraint Enforcement Tool for Java. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 10–19. Springer, Heidelberg (2006)
6. Floyd, R.W.: Assigning meanings to programs, in Mathematical Aspects of Computer Science. In: Proceedings of Symposium in Applied Mathematics, vol. 19, pp. 19–32. American Mathematical Society (1967)
7. Hussmann, H., Finger, F., Wiebicke, R.: Using Previous Property Values in OCL Postconditions: An Implementation Perspective. In: Int. Workshop UML 2.0 - The Future of the UML Constraint Language OCL, York, UK, October (2000)
8. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Addison- Wesley, Reading (1995)
9. Hoare, T.: An Axiomatic Basis for Computer Programming. CACM 12(10) (1969)
10. Karaorman, M., Abercrombie, P.: jContractor: Introducing Design-by-Contract to Java Using Reflective Bytecode Instrumentation. Formal Methods in System Design 27(3), 275–312 (2005)
11. Laddad, R.: AspectJ in Action: Practical Aspect-Oriented Programming, Manning (2003)
12. Leavens, G.T., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D., Müller, P., Kiniry, J.: JML Reference Manual, Tech. Rep. 2007/02/07, Iowa State Univ. (2007)
13. Meyer, B.: Applying design by contract. Computer 25(10), 40–51 (1992)
14. Meyer, B.: Eiffel: The Language. Object- Oriented Series. Prentice Hall, New York (1992)
15. OMG, OCL 2.0 Specification, Version 2005-06-06 (June 2005)
16. Richters, M., Gogolla, M.: Aspect-Oriented Monitoring of UML and OCL Constraints. In: Proc. UML 2003 Workshop on Aspect-Oriented Software Development with UML. Illinois Institute of Technology, Department of Computer Science (2003)
17. Toval, A., Requena, V., Fernandez, J.: Emerging OCL Tools. Journal of Software and System Modelling 2(4), 248–261 (2003)
18. Van Der Straeten, R., Casanova, R.: Stirred but not Shaken: Applying Constraints in Object-Oriented Systems. In: Proc. of NetObjectDays, pp. 138–150 (2001)