Executing Code in the Past: Efficient In-Memory Object Graph Versioning

Frédéric Pluquet Stefan Langerman [*]
Université Libre de Bruxelles (Belgium)
Computer Science Department
Faculty of Sciences
{fpluquet,stefan.langerman}@ulb.ac.be

Roel Wuyts
IMEC, Leuven (Belgium) and
Katholieke Universiteit Leuven (Belgium)
roel.wuyts@imec.be

Abstract

Object versioning refers to how an application can have access to previous states of its objects. Implementing this mechanism is hard because it needs to be efficient in space and time, and well integrated with the programming language. This paper presents HistOOry, an object versioning system that uses an efficient data structure to store and retrieve past states. It needs only three primitives, and existing code does not need to be modified to be versioned. It provides fine-grained control over what parts of objects are versioned and when. It stores all states, past and present, in memory. Code can be executed in the past of the system and will see the complete system at that point in time. We have implemented our model in Smalltalk and used it for three applications that need versioning: checked postconditions, stateful execution tracing and a planar point location implementation. Benchmarks are provided to asses the practical complexity of our implementation.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]: Classes and Objects, Data types and structures; D.3.2 [*Language Classifications*]: Object-Oriented Languages

General Terms Algorithms, Design, Experimentation, Languages, Performance

Keywords Object Versioning, Object-oriented Programming, Language Design

[*] Maître de recherches du FRS-FNRS

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
OOPSLA 2009, October 25–29, 2009, Orlando, Florida, USA.
Copyright © 2009 ACM 978-1-60558-734-9/09/10. . . $10.00

1. Introduction

In the algorithmic research community, data structures are called *persistent* (in the rest of this paper we will use the term *versioned*[1]) if they support access to multiple lifetime versions of that data structure [Driscoll et al. 1986]. Versioned data structures make it possible to go back in time and revisit the state of a data structure at some point in the past. There are several applications that need such versioning support. For examples, debuggers and tracers benefit from offering insight in previous states of objects.

Implementing efficient object versioning is hard because of the space and time complexity needed to save past states of potentially all fields of all objects. Furthermore the versioning mechanism should be properly integrated in the programming language. Last but not least, full support for object versioning ideally should support the same design principles than orthogonal persistence because this has proven to be useful when dealing with saved object states [Atkinson 1995]:

1. Any object, regardless of its type, can be versioned.
2. The lifetime of all objects is determined by their reachability.
3. Code cannot distinguish between versioned and non-versioned data.

This paper presents HistOOry, an in-memory object versioning system that is general enough to add versioning to any existing program and that has the following features:

- It supports the design principles outlined above: any object can be versioned, unreachable objects are garbage collected and there is no difference between versioned and non-versioned objects.
- It has a fine-grained model where certain fields of an object can be versioned while other ones are not. Moreover

[1] We avoid the word *persistent* because it has a different meaning in the object-oriented and database communities, where it is tied to long-lived data and the suspension and resuming of execution.

the model allows to specify when the state is saved, because different applications have different requirements.

- Objects can be versioned without the need for changing the implementations of their class.
- It is efficient. Versioning has a constant cost that does not increase with the number of objects that have already been versioned. Querying a past version of the object graph can be done with a cost that is logarithmic to the number of saved versions.
- It only requires three primitives, making it easy to learn and use.

HistOOry saves the versions in memory because we make the past versions of objects directly available. The reason is simple: the goal of HistOOry is to reflect on the past of objects as quickly as possible and not to backup objects on some physical medium to restore them at some later time. This leads us to a different solution than the "classical" database- or file-oriented persistency approaches.

HistOOry is implemented in Squeak/Pharo (a Smalltalk environment), using efficient structures to keep states (based on the fat node method of Trajan *et al.*[Driscoll et al. 1986]) and we have used it to build three applications using object versioning: we add support for checked postconditions to Smalltalk, implement an object execution tracer that keeps track of the states of receiver and arguments, and implement a planar point location program. Benchmarks for synthetic cases and for these applications show that the measured execution time penalty is a factor of 7.3 for a synthetic worst case example, and a factor of 2.3 in the application benchmarks.

This paper is the second in our research on adding efficient object versioning to an existing programming language. A first workshop paper [Pluquet et al. 2008] showed the basic feasibility of the idea and showed that our approach can be efficient using a first prototype. This paper revisits the algorithm (optimizing it even more), presents a much faster, robust and efficient implementation in Smalltalk and shows how to integrate and use object versioning.

The rest of the paper is structured as follows. Sec. 2 gives examples of applications that use object versioning. Sec. 3 and Sec. 4 introduce the terminology and the actual model and implementation of HistOOry. Sec. 5 revisits the applications and shows how they can be implemented, while Sec. 6 gives benchmark results. Sec. 7 discusses related work, while Sec. 8 describes some of the extensions we are working on. Sec. 9 concludes the paper.

2. Examples of Applications that Use Object Versioning

A large number of applications currently use object versioning, either explicitly or implicitly. They typically use ad-hoc solutions that rely on copying objects, which require a lot of effort to implement, are not very efficient, and are prone to errors. In this section, we present three types of applications using object versioning: capturing stateful execution traces, supporting checked postconditions and solving the planar point location problem. In Sec. 5, we revisit these examples and show how to build them with HistOOry.

2.1 Capturing Stateful Execution Traces

When reengineering legacy systems, one of the few trustable sources of information is the execution of the application itself [Demeyer et al. 2002]. Approaches exist to capture execution traces of programs and query or visualize the traces to gain understanding of the system [Lange and Nakamura 1997, Hamou-Lhadj and Lethbridge 2004].

What these approaches almost never capture (with the exception of [Ducasse et al. 2006]) is the state of the receiver or the arguments at the time the message was sent. With state information available we could for example find all messages to a particular object that have side-effects on a particular variable. Such queries can be expressed quite easily using for example object querying languages [Wuyts 2001, Willis et al. 2006, Hajiyev et al. 2006] once the state information is available.

2.2 Checked Postconditions

A postcondition is an assertion (a predicate the developer believes to be true) that describes the expected state at the end of some execution [Meyer 1992]. Several languages have support for checked assertions, assertions that are checked and that raise exceptions when they are violated. In object-oriented programming, postconditions can typically be found at the end of a method. They take the form of expressions that use the final values of objects used in a method. For example, a method that has as behaviour to count the number of elements of an array can have a postcondition expressing that this number is always positive.

Another example of a postcondition is one that expresses that the size of a collection grows by one if an element is added. To check this assertion there is a need to know the state before the method is being executed and afterwards, such that the sizes can be compared. The fact that the initial state of an object needs to be compared with the state at the end of executing a method holds true for many other examples as well.

2.3 Planar Point Location

Our previous paper [Pluquet et al. 2008] shows over 20 algorithms, whose implementation would be greatly simplified by using our object versioning system. For instance, planar point location is a classical problem in computational geometry: given a subdivision of the plane into polygonal regions (delimited by n segments), construct a data structure such that, given a query point, the region containing it can be reported.

There is a solution by Dobkin and Lipton [Dobkin and Lipton 1976] that answers queries in $O(\log n)$ time. It subdivides the plane into vertical slabs determined by vertical lines positioned at each vertex. Unfortunately, the worst-case space requirement for this structure is $\Theta(n^2)$.

Another solution, by Sarnak and Tarjan [Sarnak and Tarjan 1986] uses persistent (versioned) data structures to reduce the space to $O(n)$. A vertical line sweeps the plane from $x = -\infty$ to $x = +\infty$, maintaining the vertical order of the segment at every point in a balanced binary search tree. The tree is modified every time the line sweeps over a point, but all previous versions of the tree are kept, effectively constructing Dobkin and Lipton's structure while using a space proportional to the number of structural changes in the tree.

3. Basic Versioning Terminology

Before we introduce HistOOry, we define a number of basic terms. We define the *state* of a field as its value. At each modification of a value, a new state is generated. The state of an object is the combination of the states of its fields. We define a *history* as an ordered collection of states. A field is either *ephemeral*, which means that it only retains its last state and has no memory about its previous states, or *versioned*, which means that it can access previous states.

We have defined the state of an object as being the combination from the state of its fields, and not as a first-class construct in itself. The state of an object at any given time is the value of its fields at this time. This makes it possible to have objects that contain versioned and ephemeral fields, a feature we will use later on in the paper. In the rest of the paper, we speak about versioned fields (and versioned objects, that have all their fields which are versioned).

The versioning we are discussing in this paper allows one to save modifications of an object and to browse the modifications in a read-only mode: a new state can be created only from a last state. This is comparable to back-up systems like Mac OS-X Time Machine: it is possible to view previous versions of files that are included in the back-up, but they cannot be changed. Two other modes of versioning are studied in the algorithmic literature: full and confluent [Driscoll et al. 1986], but we will not discuss them here.

4. The Model of HistOOry

Before we delve into the details of the model that underlies HistOOry, we give an overview of the goals that drove its design.

1. **Recording and browsing all the available states of any object** in an object oriented system, including arrays and other kinds of collections. This goal can be subdivided into two crosscutting concerns:
   (a) *Fine-grained selection of what to save*. The level of granularity of what to save in HistOOry is the field of an object. This means that the smallest element that can become versioned is a single field of a single object, even though most applications will choose to version more elements (for example all objects and all of their fields of some classes of interest). We have taken care of making it easy for the developer to choose what becomes versioned. As will be discussed later, we also defined a number of rules to make cohabitation of ephemeral and versioned objects possible.
   (b) *Fine-grained selection of when to save*. Each modification of a field of an object generates a new state of this object. HistOOry provides a simple mechanism to select which states must be kept.
2. **Being efficient**. We have based our solution on an efficient data structure that reduces the time and space needed to store and retrieve the object history information (discussed in Sec. 4.2). It allows us to save the state in constant time (not dependent on the number of states previously saved) and retrieve the states of a field with a time complexity that is logarithmic to the number of stored states for this field. Besides the theoretical advantages of the particular algorithm we chose, we have also taken care to implement it in proper object-oriented style. For example, all states of an object are saved in the object itself. If an object is no longer used in the system, that object and all objects used to save its history will be garbage collected. As shown in Sec. 6, our implementation results in a slowdown that never exceeds a factor of 7.3, regardless of whether we keep a single state or a hundred thousand states.
3. **Ease of use**. We wanted to integrate object versioning in an object-oriented language in such a way that it is easy to version certain parts of an implementation, as well as to use the versioning information. In our approach, no modifications to existing code are necessary to version objects. Moreover it is only necessary to learn 3 primitives to use HistOOry.

4.1 Recording and Browsing Object States

The first goal of our model is to be general enough to have the possibility to record any state of any object in any object oriented system and then browse them.

4.1.1 Snapshots: When to Save Fields

The developer that uses HistOOry to make an application with versioned objects has full control of *when* states of objects are saved.

This is analogous to using a camera. Whenever the user presses a button, a snapshot is taken, remembering what was visible at that time, while life goes on. This is in contrast to a video camera, that saves a constant stream of images. While the latter can be interesting at times (and can be done in our approach as well), lots of applications that need object versioning are better served with explicitly taking snapshots than with capturing a huge stream of changes.

We can illustrate this with a concrete example. Suppose that we have an implementation of a balanced tree. While debugging the tree data structure itself, a developer is interested in seeing all the states the tree goes through while adding an element, including internal node rotations and low-level changes in the collections that store the data in the tree nodes. However, while debugging an application that uses the tree that developer might only be interested in seeing consistent states of the tree (the state of the tree after element insertions and deletions), without the internal workings of the tree. For the first application, it is necessary to keep all state changes of all objects making up the tree data structure. In the second application, we only want to take snapshots after elements are inserted or deleted.

4.1.2 Selection and Deselection: What Fields to Save

A developer has full control over *what* gets saved when a snapshot is taken.

This is analogous to putting a filter on the lens of the camera. While a camera without a filter will always take snapshots of the complete scene, lens filters will reduce the amount of information in a scene and only select items of interest. Filters can be changed at any time and change the results of pictures taken after the new filter is installed.

By default HistOOry makes all fields of all objects ephemeral. In our camera analogy this is comparable with putting the lens cap on: when you take a snapshot you will not see anything.

At any given point in time the developer can *select* what fields become versioned. This is comparable to replacing the lens cap with a filter. When a snapshots is taken, it will only save the states of versioned fields. States of ephemeral fields are not stored and can therefore not be looked at later on.

It is also possible to make a versioned field ephemeral again by *deselecting* it. Deselecting a field means that future snapshots will not save the state for that field. Old states are still available but no additional state will be saved: the value of the last state is overwritten at each update. A deselected field can be obviously re-selected.

In contrast with systems where all objects are always versioned, our model gives the developer fine-grained control. This has a number of advantages. Firstly, the object versioning is clearly visible in the code because the developer explicitly indicates which objects are versioned. Secondly, the system does not lose time and space to save modifications of objects that will never be used. Thirdly, this choice means that the developer still has the possibility of keeping everything.

4.1.3 Browsing States

Previous sections explained how states can be saved by using selection and snapshots. We illustrate this with a concrete example. Suppose we are building a library system to model the borrowing of books. It uses a class `Book` that has three fields: `title` contains a pointer to a string that represents the title of the book, `state` contains a pointer to a string describing the state of the book ("clean" or "dirty") and `borrower`

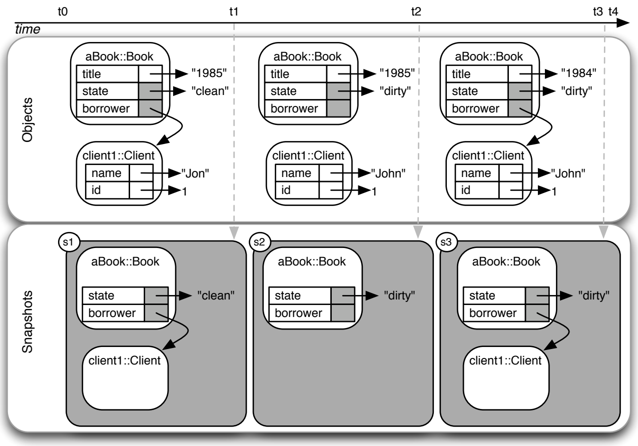


Figure 1. Pedagogical depiction of taking three snapshots of a fine-grained selected objects graph

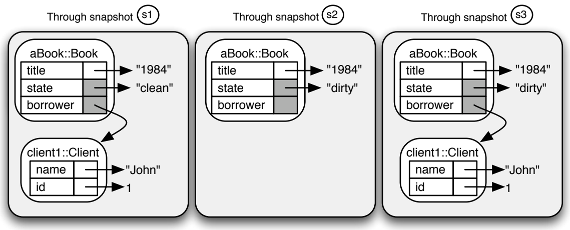


Figure 2. Browsing the past from the time *t4* through the three snapshots taken in Fig. 1

contains a pointer to a client. A client is described by two fields: `name` (a string) and `id` (an integer). We select only the fields `state` and `borrower` of `Book` to be versioned. Other fields remain ephemeral because it is unnecessary in this application to keep the older versions of the title of a book or the name of the user: only the present values of these fields are available.

Fig. 1 shows the evolution of a particular book instance, namely the novel ”1984” that will be borrowed by a client named ”John”. The book and the client are introduced in the library system between the time *t0* and *t1*. At time *t1* the book is badly titled ("1985"), its state is set to be clean and a client with name ”Jon” is set to be the current borrower. We take a snapshot *s1* at this time *t1*.

Sometime later the client comes back to the library to return the book in a dirty state. He mentions also that his name is ”John” instead of ”Jon”. A new snapshot *s2* is taken at this time *t2*.

Three days later the client comes back to borrow the same book again. During the loan registration, the library employee corrects the title of the book. A snapshot *s3* is taken at this time *t3*.

The top part of Fig. 1 shows objects states at each timestamp. We have colored the versioned fields in grey, while the ephemeral ones are white. The second line shows saved values in each snapshot. Only the values of selected fields are saved.

Snapshots reify the state of the selected part of a system at the time the snapshot was taken. Code can then be executed in the context of the snapshot. That code sees the saved part of the system exactly like it was when the snapshot was taken. The objects seen in each of the snapshots from the time *t4* are shown in Fig. 2. When fields are accessed, three things can happen:

- The field was selected before the creation of the snapshot. In that case the stored past state is returned. For example, asking the value of the borrower of the book in the second snapshot gives `NULL`.
- The field was selected after the snapshot was created. This means that we try to access the past state of a field before it was saved for the first time. We raise an exception.
- The field was not selected (it is therefore an ephemeral field) and therefore no past state exists. We return the present value of the state (all white fields in the figure have their present value). Sec. 4.3 shows other examples in which this choice it is very practical.

A modification of a versioned field while executing code in the context of a snapshot results in an exception being thrown. A modification of an ephemeral field changes that field, which is normal because the snapshot actually sees the present object. Changing the name of the ephemeral user field either in the present or in the context of a snapshot would therefore change the present value.

4.2 Being Efficient

Saving multiple states of object graphs and efficiently retrieving them requires an advanced data structure. Luckily several algorithmic results are known for this problem. After carefully reviewing the available algorithms, we decided to implement the *fat node method* [Driscoll et al. 1986] that can transform any ephemeral data structure into a partially versioned one.

In the rest of the section we first outline the fat node method itself, we then describe the structure that stores the states and then discuss our implementation.

4.2.1 Fat Node Method

The data structure must remember the states of versioned fields of objects. A first method could be to simply save all objects in the system whenever a snapshot is taken. This consumes a lot of space because there is no sharing of common state across snapshots, but makes it very easy to browse the states at a certain point in time. Another approach could be to remember every single update to a field. This approach reuses states between different snapshots, but makes it very hard and costly to reconstruct the complete past system in a consistent way. Moreover, it must keep all past states to remain consistent, making it costly when only few snapshots are taken.

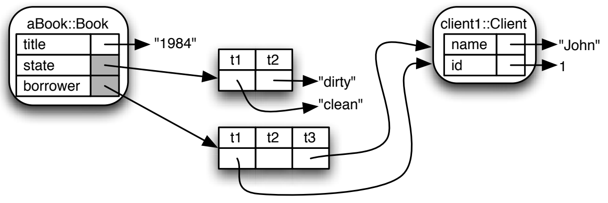


Figure 3. Internal structure in HistOOry for the example shown in Fig. 1 at time *t4*.

The fat node method in a sense combines the two previous methods. Like the second method, it keeps all the past values of a versioned field within that field itself, but like the first method it only remembers a single value per snapshot (and not all the intermediate values that do not belong to snapshots). Last but not least, it makes it very easy to use the past in a consistent way.

The key principle is that when a field is marked as versioned, it remembers its previous values. Instead of having only a reference to the present value of the field, a reference to the previously snapshotted values is kept, together with bookkeeping information that makes it easy to reconstruct the complete system at a particular point in time.

Fig. 3 shows how this works for the example shown in Fig. 1 for the time *t4*. Ephemeral fields are not changed and therefore directly store references to objects. Versioned fields on the other hand contain all of the snapshotted values associated with their time. The figure seems to imply that the values are stored in a simple table, which is an abstract view. The actual data structure used to keep and access the snapshotted values is detailed in Sec. 4.2.2 where we show how the approach functions.

Two kinds of bookkeeping information are kept. First of all there is a single global version number that is kept for the system as a whole. Second, each state (remembered value of a field) remembers when it was added by keeping a version number. This works as follows. When updating a versioned field with a new value v, the version number of the previous state of this field is considered: if it is equal to the global version number, the value saved by the state is replaced by v. If not, a new state with the value v and a version number equal to the global version number is created. The global version number is incremented only when the system needs to save a new global state. Taking a snapshot therefore boils down to just incrementing the global version number, which is very cheap.

4.2.2 States Structure

The data structure that actually stores the states of the fields is composed of chained arrays (see Fig. 4), not a simple table. Chained arrays offer good performance: new states

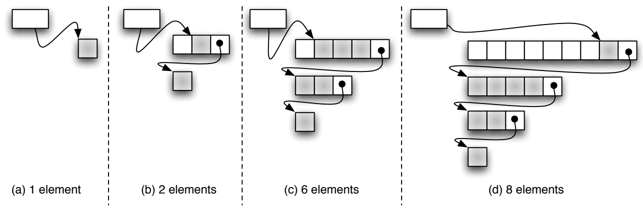


Figure 4. Structure to save versions of a field.

can be added in $O(1)$ time, the last version can be accessed in $O(1)$ and the search of a state for a given version number is bounded by $\lg m + 2$ in the worst case where m is the number of states in the arrays. Moreover the space used is $O(m)$.

We observed that consecutive retrievals of the same version of an object always have to traverse the chained data structure. Therefore, we decided to add a cache. The cache holds a single key-value pair consisting of the last version of the field retrieved and the corresponding state. Consecutive retrievals of the same version therefore no longer traverse the chained arrays but immediately return the object. This simple cache results in good practical performance because it is lightweight (only a single value is kept and only a single version number is compared) and corresponds to most practical usage scenarios.

4.2.3 Implementation

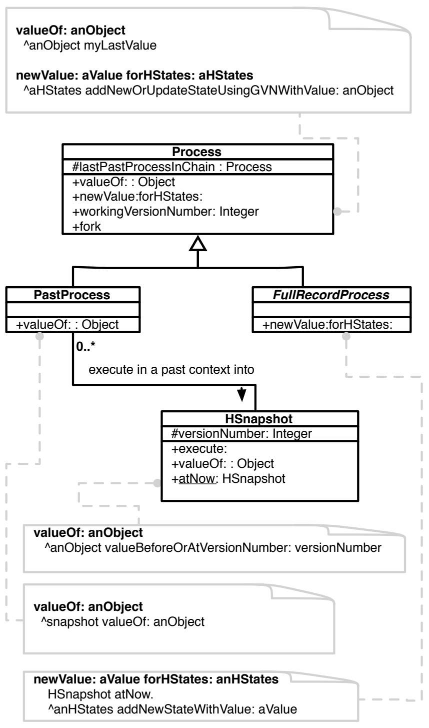
To explain how our implementation works, we first show a small part of the implementation of class `Book` from the library system introduced in Sec. 4.1.3, namely the setter and getter methods for the `borrower` field (in Smalltalk):

```
Book>>borrower
        "getter method that returns the borrower of a book"
        ^borrower

Book>>borrower: aClient
        "setter method that sets the borrower of a book"
        borrower := aClient
```

When the class `Book` has none of its fields selected for versioning, it is left untouched. But when we select the field `borrower`, HistOOry transparently modifies code that accesses fields. Accessing a field will defer to the active process instead of directly asking the object. Setting a field will send a HistOOry specific message to the object contained in the field. The resulting code does the following (we show later on that this is actually done with bytecode rewriting, not source rewriting, but it is easier to show the corresponding source code):

```
Book>>borrower
        "getter method that returns the borrower of a book"
        ^Processor activeProcess valueOf: #borrower
```



```
valueOf: anObject
  ^anObject myLastValue

newValue: aValue forHStates: aHStates
  ^aHStates addNewOrUpdateStateUsingGVNWithValue: anObject
```

```
valueOf: anObject
  ^anObject valueBeforeOrAtVersionNumber: versionNumber
```

```
valueOf: anObject
  ^snapshot valueOf: anObject
```

```
newValue: aValue forHStates: anHStates
   HSnapshot atNow.
   ^anHStates addNewStateWithValue: aValue
```

Figure 5. Processes class diagram

```
Book>>borrower: aClient
        "setter method that sets the borrower of a book"
        borrower replacedBy: aClient atOffset: 3 of: self
```

From the getter method it is clear that class `Process`[2] plays an active role to control the accessing and retrieving of versioned information. By transferring control to instances of process we can execute code in the past in one process while executing code in other processes in the present.

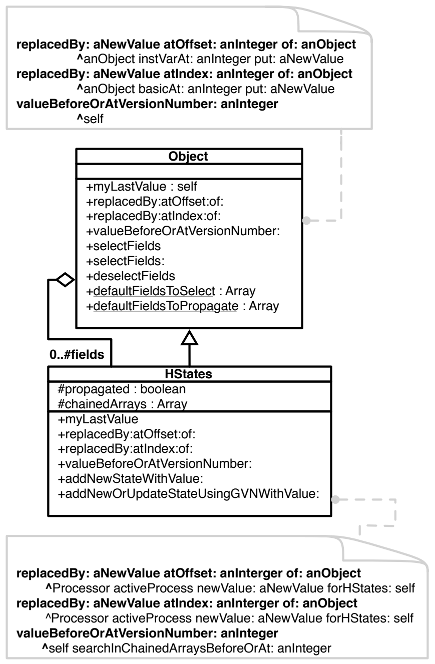
Fig. 5 describes the class `Process` and the two methods we extended it with:

- `valueOf:` returns the last value of the given object;
- `newValue:forHStates:` asks to the given instance of `HStates` to update the last state or add a new state with a given value (as described at end of Sec. 4.2.1).

By default, these methods result in the same behaviour as standard Smalltalk, but via an indirection. This is the overhead that is present as soon as fields are selected (see Fig. 7(b)), wether the versioned information is used or not.

We can now implement other `Process` classes, as shown in Fig. 5, with specific behaviour. The class `PastProcess` is the process class that is used when executing code in a particular past state. It overrides the method `valueOf:` to fetch the value from the `HSnapshot` object that will be detailed next. Another example is `FullRecordProcess`, a process that automatically snapshots just before any modification of

[2] The class `Process` is an already defined class in Smalltalk, representing the different processes executed in a Smalltalk image.



```
replacedBy: aNewValue atOffset: anInteger of: anObject
            ^anObject instVarAt: anInteger put: aNewValue
replacedBy: aNewValue atIndex: anInteger of: anObject
            ^anObject basicAt: anInteger put: aNewValue
valueBeforeOrAtVersionNumber: anInteger
            ^self
```

```
replacedBy: aNewValue atOffset: anInterger of: anObject
      ^Processor activeProcess newValue: aNewValue forHStates: self
replacedBy: aNewValue atIndex: anInterger of: anObject
      ^Processor activeProcess newValue: aNewValue forHStates: self
valueBeforeOrAtVersionNumber: anInteger
      ^self searchInChainedArraysBeforeOrAt: anInteger
```

Figure 6. Object and states class diagram

a field (meaning that absolutely every change is captured even when the user does not snapshot explicitly).

The indirection through the `Process` classes allow us to change the semantics of reading and writing of fields. This is actually done in close interaction with the class `HStates`. A versioned field contains an instance of the class `HStates`. This class contains four important methods:

- `myLastValue` returns the last value contained in the chained arrays data structure that keeps all snapshotted values and was discussed in Sec. 4.2.2;
- `replacedBy:atOffset:of:` is called when this instance of `HStates` will be replaced by a new value at a given offset[3] of a given object. Depending on the current process, it adds a new couple (version number, value) in the chained arrays or it updates the last value;
- `replacedBy:atIndex:of:` is the same as the previous one but for a given index of a given object;
- `valueBeforeOrAtVersionNumber:` returns the value contained in the chained arrays associated with the highest version number before or equal to the given version number.

Now that we have introduced the `Process` and `HStates` classes and show how they collaborate to read current or past fields by going through the appropriate process class, it is time to look at the writing of fields. When selecting fields the setter method will not directly set the value of a field, but instead it sends the message `replacedBy:atOffset:of:` to the object in the field. When this field is selected, then the object will actually be an instance of `HStates` and the value is saved. When the field is not selected, the field contains the actual object itself that needs to be replaced. To be transparent, we extend the Smalltalk root class `Object` with the same interface than the one used by `HStates`, as shown in Fig. 6. These implementations do the following:

- `myLastValue` returns itself;
- `replacedBy:atOffset:of:` puts itself at the given offset of the given object;
- `replacedBy:atIndex:of:` puts itself at the given index of the given object;
- `valueBeforeOrAtVersionNumber:` returns itself.

As mentioned before we refrained from doing these modifications to the source code because it would be slower and the developer would then be exposed to the internal workings of our system when seeing the rewritten code. We also did not want to modify the virtual machine because that would be significantly more difficult and a user would need to use our modified virtual machine. Therefore we chose to directly manipulate bytecodes to implement our algorithm. We decided to use Smalltalk because we could use the excellent ByteSurgeon tool [Denker et al. 2005] and had better reflection support.

We could have implemented our approach in statically typed languages like Java or C# as well. We believe that the results would be comparable to the Smalltalk implementation, but more difficult to achieve.

4.3 Language Integration

The previous sections explain the concepts of our model and how we made it efficient to store the data. This section shows how this data can be used effectively by integrating it in an object-oriented language.

As discussed before, the developer can select what fields become versioned, take snapshots, and browse the saved state. For these three basic operations, we give the developer three primitives:

1. `selectFields` selects all fields of the receiver to be in the next snapshot (essentially versioning the complete object);
2. `HSnapshot atNow` takes a new snapshot and returns it;
3. `execute: aBlock`, sent to a snapshot, permits to execute the specified code block at the time when the snapshot was taken.

These primitives are actually an embedded domain specific language. Implementation-wise we just had to add a method `selectFields` to the `Object` root class of

[3] The offset of a field is the order place of this field in the object. For instance, the borrower of a Book instance being the third field, its offset is 3.

Smalltalk and create a `Snapshot` class to turn snapshots into first-class objects.

The following subsections give a number of examples to show HistOOry in action.

4.3.1 Example of Basic Usage

This example shows how we can track the changes to a particular object, namely a Squeak/Pharo package. The example code first finds the package object named `Kernel`, makes it a versioned object, changes its name to `Test`, takes a snapshot, and renames it once more to `NewKernel`. We then print the current name of the package on the transcript, which shows `NewKernel`, as expected. Then, we do the same, but execute it in the context of the saved snapshot. This time the transcript prints `Test`, again as expected.

```
|package s|

package := PackageInfo named: 'Kernel'.
"Gets the package named 'Kernel'"

package selectFields. "Selects this object"

package packageName: 'Test'. "Renames the package"

s := HSnapshot atNow. "Takes a snapshot"

package packageName: 'NewKernel'.
"Renames the package again"

Transcript show: package packageName.
"Prints 'NewKernel' "

s execute:
    [Transcript show: package packageName].
    "Prints 'Test' "
```

There are several interesting things in this example.

- The class `PackageInfo` is one of the system classes core to the Squeak/Pharo Smalltalk language, and not one of our own classes. Yet, it is versioned simply by sending it the `selectFields` message. This code illustrates that the original implementation of an object (or its class) does not need to be changed. Behind the scenes, our bytecode rewriting tool takes care of instrumenting the code to keep track of all changes to the fields of this object and puts in place our data structures.
- When an ephemeral object is versioned, it is exactly the same object and can be continued to be used exactly like any other object. The reason is that we do not change the object itself but update its class, which ensures that there is no difference except for the fact that its state is saved when snapshots are taken.
- The developer is responsible for taking snapshots. By default, the system will not save anything. It is the role of the developer to determine which states are important.
- Code can be executed in the context of a snapshot by using the `execute:` message and passing the code to be executed in a block.
- Ephemeral objects and versioned objects can live together. In our example, the object `Transcript` is ephemeral while the package object is versioned. This is possible because versioned fields return the saved value at the time of the snapshot while ephemeral fields return their present value.

4.3.2 Selection Protocol

Up until this point we have primarily talked about how to make individual fields versioned, and just mentioned that when an object is versioned, its fields are versioned. In practice however it is important to give the developer good control over what fields of what objects are versioned, and this cannot be done with a single message like the `selectFields` used above.

In fact there is a more refined protocol to let developers decide what is versioned, consisting of three methods: `selectFields`, `selectFields:` and `defaultFieldsToSelect`.

The `selectFields` message by default versions all fields in the object. However, a developer can control this default behavior by overriding the method `defaultFieldsToSelect` and indicating what fields are selected when the message `selectFields` is sent. This method overriding is a practical way for establishing the default choices for what gets saved. Method `defaultFieldsToSelect` is implemented in `Object`, the root class, and returns all fields of the receiver object. The fields are collected by using reflection and it is therefore not needed to override this method on each class that just wants to indicate that it too has fields to include.

If a developer wants to deviate from the default, the message `selectFields:` can also be used. It takes as argument the fields that need to be versioned, regardless of what is specified in method `defaultFieldsToSelect`.

In our Smalltalk implementation, we added methods to existing classes (for example the three selection protocol methods to the root class `Object`). The object versioning is nicely integrated in the language, resulting in a small embedded domain-specific language. Moreover we implemented our language extension using *class extensions*[4] (also called *open classes* [Millstein and Chambers 1999]). Other languages could use their particular language features to integrate a versioning model, for example through library calls, method annotations, AOP-style *inter-type declarations* [Kiczales et al. 2001], macros, *etc*.

[4] A *class extension* is a method that is defined in a module, but whose class is defined in another module.

4.3.3 Propagation

The previous section talked about selecting individual fields. Of course what happens a lot is that a field itself contains an object that you also want to be versioned.

Take for example the class `Set`. This class has two fields: `array` (the basic collection that stores the actual elements in the set) and `tally` (a number that indicates the position of the last element in the array). When the array is full, a new larger array replaces it (in which the old values are copied). To make a `Set` instance, as a whole, versioned, we can send it the message `selectFields`. The result is that both fields are versioned and, therefore, changes to these fields will be saved when taking snapshots. However, because the `array` field is itself an object and that object was not versioned, we will not actually be able to revert to the previous contents of the set instance but merely remember changes to the pointer itself.

To solve this problem, we can make the array variable itself versioned, for example by doing the following:

```
|s|
s := Set new.
s add: 1; add: 2.
s selectFields.
(s instVarNamed: #array) selectFields.
1 to: 100 do: [:each | s add: each. (s instVarNamed: #array)
  selectFields].
```

We need to send the message `selectedFields` after each insertion. This ensures that if the array has grown, the new ephemeral array that resulted from the growth is immediately versioned.

Because this code is tedious too write and frequently needed we support it directly through either a message or a class extension. The message `propagateFields:` can be used to make the selection propagate, meaning that the previous code snippet can be rewritten as follows:

```
|s|
s := Set new.
s add: 1; add: 2.
s selectFields.
s propagateFields: {#array}.
1 to: 100 do: [:each | s add: each].
```

As an alternative the method `defaultFieldsToPropagate` can be implemented on a class to indicate what fields should be propagated when the object is versioned. In our example, we could add it to the class `Set` as follows:

```
Set>>defaultFieldsToPropagate

  ^NHArray with: #array
```

When an instance of the `Set` class is versioned, all values of the field `array` will also be selected for versioning.

The class `NHArray` is a "non HistOOrizable" implementation of the class `Array` that we defined in the HistOOry package. No state of its instances will be saved. This class therefore avoids unnecessary indirections to the `Process` classes hierarchy.

With the method `defaultFieldsToPropagate` added to class `Set`, our code snippet can be written as follows:

```
|s|
s := Set new.
s add: 1; add: 2.
s selectFields.
1 to: 100 do: [:each | s add: each].
```

We stress that this solution is again transparent for the original code: the original code is not changed, but it is extended with one method residing in another package.

4.3.4 HPastObject

Suppose that we have made an application versioned and have taken a number of snapshots. Then, we want to send messages to an old version of some object. In the very first example, we have seen that this can be done by sending the message `execute:` to the snapshot, passing the code to execute in that snapshot as an argument. The following code example illustrates this.

```
...
((s execute: [aSet size]) < aSet size)
    && (s execute: [aSet includes: 0])
...
```

To make it easier to repeatedly send messages to a previous state of a single object, we have provided a proxy-based mechanism that redirects messages sent to it to the old state of the object. The next code snippet shows this mechanism in action.

```
...
oldSet := HPastObject on: aSet during: aSnapshot.
oldSet size < aSet size
    && oldSet includes: 0
...
```

The implementation of the class `HPastObject` is pretty straightforward. It captures all messages sent to it by overriding the method `doesNotUnderstand:` [Ducasse 1999]. In that method it sends the message to the object to the snapshot provided when an instance of the class was created.

5. Implementing Versioned Applications Using HistOOry

Sec. 2 listed a number of applications that, explicitly or implicitly, use object versioning. This section shows how they can be implemented using HistOOry.

5.1 Capturing Stateful Execution Traces

In this example, we show how we can very easily build an execution tracer that is stateful: it saves the messages that are sent, including the state of the receiver before and after sending the message. Therefore, trace analyzers can not only find patterns on the order and nesting of the messages sent, but they can also take the state of the receiver into account (for example to find all messages that have side effects).

In Smalltalk, execution traces can be captured fairly easily by using *method wrappers* [Brant et al. 1998] to instrument code. The instrumented method will be replaced by a method wrapper where we can add hooks to trace the activation of methods. The following example shows the key part of the implementation of this technique. The wrapped method first calls `traceEntryIn:on:`, then it calls the original method, and finally it calls `traceExitOf:on:`[5].

```
MyWrapper>>run: aSelector with: arguments in: aReceiver
  |answer|
  self traceEntryIn: aSelector on: aReceiver.     "before call"
  answer := aReceiver withArgs:arguments executeMethod:
        originalMethod.
  self traceExitOf: aSelector on: aReceiver.    "after call"
  ^answer
```

The previous implementation only captures the messages being sent. It is easy to extend it to save the state of the receiver before and after sending the message, turning it into a stateful sequence tracer: we make the receiver versioned by sending it the message `selectFields`.

```
MyWrapper>>run: aSelector with: arguments in: aReceiver
  |answer|
  aReceiver selectFields.
  self traceEntryIn: aSelector on: aReceiver at: HSnapshot atNow.
  answer := aReceiver
                withArgs:arguments executeMethod: originalMethod.
  self traceExitOf: aSelector on: aReceiver at: HSnapshot atNow.
  ^answer
```

Note that repeatedly sending the message `selectFields` is harmless. For each method call, both states of the receiver are saved by the snapshots. These snapshots will then be used to retrieve the state of the receiver at a given time.

This section showed how, with a minimum of effort, an execution trace was extended with support for saving the states of the objects.

5.2 Checked Postconditions

Checking postconditions frequently requires one to compare the states of the receiver before the method is being executed with the final state at the end of the method execution. We show how we have extended Smalltalk with support for checked postconditions by using HistOOry.

The developers needed a mechanism to make it possible to specify the postconditions they would like to have checked. We opted to do this by extending the Smalltalk class `BlockContext`, the class implementing delayed code evaluation, because it is available in all Smalltalk implementations. An alternative could have been to add the postcondition using method annotations, but these only exist in a number of Smalltalk implementations, with different internal implementations.

[5] `traceEntryIn:on:` and `traceExitOf:on:` are auxiliary methods that store information about the messages that were sent, such as the timestamp.

An example of using the postconditions in Smalltalk is given below. It adds a postcondition for the method `swap:with:` of class `SequenceableCollection` (one of the abstract classes in the Collection hierarchy). The postcondition verifies that the elements were indeed swapped by comparing the identities of the objects:

```
SequenceableCollection>>swap: oneIndex with: anotherIndex
    "Move the element at oneIndex to anotherIndex, and vice–
      versa."
    [

      | element |
      element := self at: oneIndex.
      self at: oneIndex put: (self at: anotherIndex).
      self at: anotherIndex put: element

    ] postCond: [:old |
        (old at: oneIndex) == (self at: anotherIndex) and: [
        (old at: anotherIndex) == (self at: oneIndex)]]
```

The original code of the method is put into a Smalltalk block (the square brackets). In the rest of the explanation, we will call this block the *method block*. The postcondition is specified as another block that is given as argument to the `postCond:` message sent to the first block. We will call this the *postcondition block*. The postcondition block takes one argument (*old*) that represents the state of the system before the execution of the method body. In the postcondition block, messages are sent to `old` to retrieve values from before the execution of the method block and to self to retrieve the current values.

We implement the method `postCond: aBlock` as an extension of the `BlockContext` class. The block that receives the message is the method block. The argument block is the postcondition block. It makes the receiver versioned, takes a snapshot, creates a `HPastObject` object to make it easy to refer to the past states, and then executes the method block and the postcondition block.

```
BlockContext>>postCond: aBlock

    | old snapshot value |
    self receiver selectFields.
    "makes the receiver versioned"

    snapshot := HSnapshot atNow.
    "snapshot before executing the method block"

    "create a HPastObject"
    old := HPastObject
            on: self receiver during: snapshot.

    "execute the method block"
    value := self value.

    "execute postcondition block"
    self assert: (aBlock value: old).

    "return the result of the method block"
    ^value
```

This implementation is fairly straightforward. The only tweak is the creation of a `HPastObject` object for the receiver in the old state and passing it to the postcondition block. The result is that the code in the postcondition can directly send messages to the "old" receiver, as explained in Sec. 4.3.4.

Sometimes postconditions need access to other objects, for example to arguments of the method. We therefore added a second method, `postCond: aBlock withObjects: aSetOfObjects`, where the objects for which we need to access past states are passed explicitly. The difference with the previous postcondition is that the argument passed cannot be a `HPastObject`, because that only makes it easy to send messages to a single object in the past. Instead the argument is a regular snapshot.

```
BlockContext>>postCond: aBlock withObjects: aSetOfObjects
    | snapshot value |
    "make arguments versioned"
    aSetOfObjects do: [ :each | each selectFields].
    snapshot := HSnapshot atNow.
    value := self value.
    self assert: (aBlock value: snapshot).
    ^ value
```

We can use this more elaborated postcondition mechanism to check that after adding a collection to another collection the size of the argument is unchanged while the size of the new collection is the sum of the initial collection sizes.

```
OrderedCollection>>addAll: aCollection
    [
        self addAllLast: aCollection
    ]
      postCond: [:snapshot |
        "the size of aCollection must not change"
        (snapshot execute: [aCollection size] = aCollection size)
      and: [

        "self size = oldSelf size + aCollection size"
        ((snapshot execute: [self size]) + aCollection size) = self
      size. ]
      ]
      withObjects: {self. aCollection}.

    ^ aCollection
```

We showed in this section how we can add checked postconditions to Smalltalk by extending the `BlockContext` class with two methods.

5.3 Planar Point Location

To illustrate how HistOOry can simplify the implementation of complex data structures, we implemented a *random treap* [Seidel and Aragon 1996], a randomized binary search tree. This structure is a mix of a tree and a heap where each node has a key and a random priority. At each insertion, node rotations ensure that constraints on the keys and the priorities hold. We implement this structure using several classes: a class `RandomTreap` that inherits from a class `Treap` and has as its root an instance of a class `TreapNode`. The instances of `TreapNode` have the attributes `key`, `priority`, `left` and `right`. The two last attributes contain either the default value `nil` or an instance of `TreapNode`.

To turn this structure into a versioned random treap, we simply extend the classes `Treap` and `TreapNode` with the following methods:

```
Treap>>defaultFieldsToPropagate
    ^NHArray with: #root

TreapNode>>defaultFieldsToPropagate
    ^NHArray with: #left with: #right
```

The following code is placed in a class `PlanarPointLocation`, that implements a solution to the planar point location problem. It stores a set of points. In the construction of the point location data structure, each point of the set is swept by the sweepline, its outgoing segments are added to the treap, the incoming ones are removed and a snapshot is taken and associated with this point.

```
PlanarPointLocation>>constructRTreap
  | linkedInfo |
  rtreap := RandomTreap new.
  rtreap selectFields.
  self allPointsDo:
     [ :aPoint |
     aPoint incomingSegmentsDo: [ :segment | rtreap deleteKey:
        segment ].
     aPoint outcomingSegmentsDo: [ :segment | rtreap putKey:
        segment ].
     aPoint associatedSnapshot: (HSnapshot atNow) ]
```

When a location query of a point p is considered, the slab containing p is determined, searching the rightmost point to the left of p in the points of the plane. This point l is the left point of the slab. Then the snapshot associated with l is used to browse the treap at the time where only the relevant segments were present. The treap is then used normally, inside the block executed through the snapshot, to locate the point.

```
PlanarPointLocation>>searchPoint: aPPLPoint
     | thePoint linkedInfo |
     thePoint := self lastPointBefore: aPPLPoint.
     ^thePoint snapshot execute: [rtreap keyEqualOrJustBefore:
        aPPLPoint]
```

This section again showed how a data structure can be made persistent without much difficulty and without changing the existing implementation. The next section will look at the efficiency of the approach.

6. Measurements

This section presents performance benchmarks for HistOOry. It first gives general measurements about the time and space needed for a number of synthetic examples. Then, it shows measurements for the stateful tracer and the checked postconditions discussed in Sec. 5. All tests were performed on an iMac 2.4 GHz Intel

Core 2 Duo with 2 gigabytes of RAM and using an empty image of Squeak/Pharo for developers (version 0.1-101166dev08.11.6).

6.1 General Measurements

We start with some general measurements: the space required and the time required to save and retrieve states.

6.1.1 Space Required

To show the size required by HistOOry, we create an object with a single field (with integer 0 as initial value) and we make this field versioned. We then increment the field and take a snapshot, and repeat this.

Fig. 7(a) shows the size taken by the data structure in the field after each update. The size grows in steps: every jump corresponds to the creation of a new array in the chain of arrays that store the actual states when the last array is full.

6.1.2 Execution Times for Saving States

We want to show the overhead in execution time when HistOOry is saving states. Therefore, we measure the average time required to update a field with an integer value. Fig. 7(b) shows four different benchmark results for this case, depending on how HistOOry is being used:

1. The code is executed in a pure Squeak/Pharo image, without HistOOry.
2. The code is executed in a Squeak/Pharo image where the methods are instrumented by HistOOry, but nothing is selected and no snapshots are taken.
3. The field is selected, but no snapshots are taken.
4. The field is selected and snapshots are taken after each update.

When observing the plot, we first of all note that the execution time plots are nearly flat. This indicates that, as expected, the execution cost when using HistOOry does not depend on the number of states that are saved (the cost is always a constant overhead).

The overhead cost in the first case, where HistOOry is not used at all, is zero. This is normal because in that case no instrumentation is done and the code runs without any modification. This is an important point, because it shows that you only pay for the features of HistOOry when you need it.

Instrumenting a class (the second case) adds an overhead of about a factor of 2 that must be paid by all instances created from this class, whether their fields are selected or not. As explained in Sec. 4.2.3, the reason for this cost is that all methods of this class are instrumented to redirect reads and writes of instance variables to the `Process` hierarchy.

Selecting a field (the third case) shows that the overhead grows to a factor of about 5.6 and 6.7 when fields are selected but no snapshots are taken.

An overhead between 6.7 and 7.3 is visible when a field is selected and snapshots are taken.

Having an application run 7.3 times slower might seem like a big price to pay. However this example is a synthetic example where literally each operation results in an assignment that needs to be stored. In practice this is often not the case: not every single operation is an assignment (sending a message, for example). Fig. 8, for example, shows the average execution times per insertion in a random treap, again for a number of use cases:

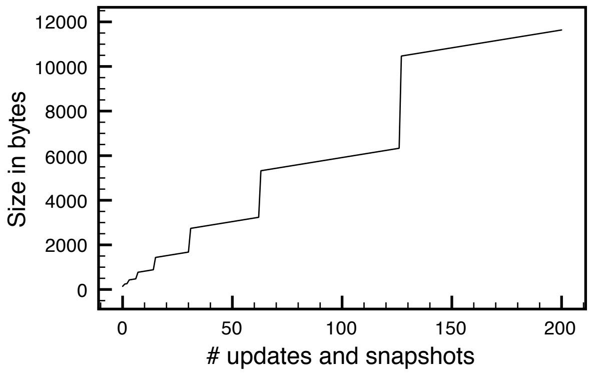
1. Treap not instrumented.
2. Treap instrumented but none of its fields selected.
3. All fields of the treap selected, and no snapshots are taken.
4. All fields selected, and snapshots taken after each insertion.
5. All fields selected and snapshots taken after every change (including for example the internal rebalancing happening in the treap)

The overall curves remain similar: they still show that for this more complex data structure the cost is constant and does not depend on the number of states being saved. Moreover we can see that the biggest execution time overhead is now only about 2.3, which is much better than the 7.3 times in the synthetic example.

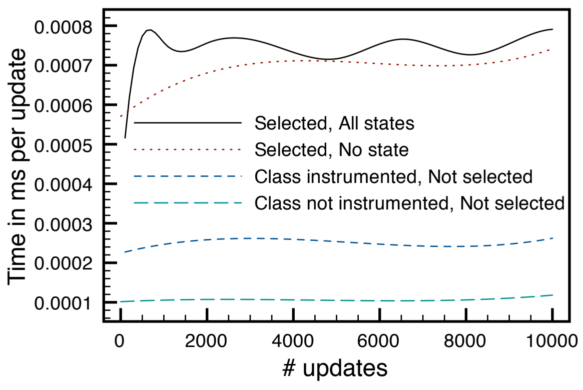
6.1.3 Execution Times for Retrieving States

We show the cost to retrieve a saved state, depending on the number of states that were saved. Therefore, we select a field and update it a fixed number of times n, each time followed by taking a snapshot. Then, we take the total time to inspect all states saved by the snapshots and divide this time by the number n. This gives us the average execution time to access a single state. Fig. 9 shows the result, on a logarithmic scale. The curve is logarithmic as expected (it is the theoretical complexity of the algorithm), indicating that our implementation is correct. Note that the peaks are again the result of an allocation of a new array in the chained arrays that keep the past states.

To show the importance of the introduction of the cache described in Sec. 4.2.2 we performed an experiment with a random treap in which we insert 1000 values and we take a snapshot we call s. We insert a given number of new values and after each insertion we take a snapshot. Finally we take the time to retrieve the 1000 initial values through the snapshot s. We did this experiment with and without the cache. Fig. 10 shows the time as Y-coordinate and the number of snapshots taken after s as X-coordinate). The cache reduces the lookup time by a factor of 2 in this example.



(a)



(b)

Figure 7. (a) Numbers of update vs. total size of a field. (b) Execution times when updating a field for four different usage scenarios of HistOOry.

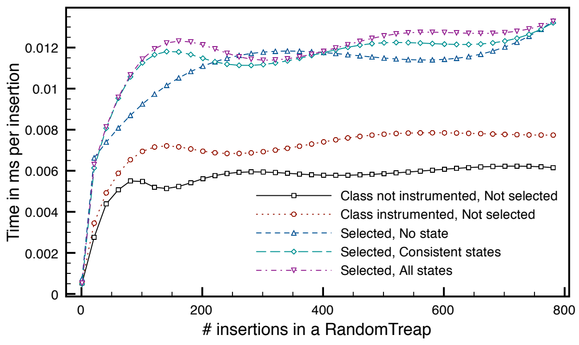


Figure 8. Execution times for saving states in a random treap data structure.

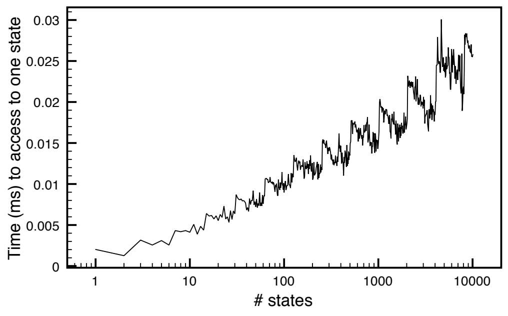


Figure 9. Execution time for retrieving saved states (logarithmic scale).

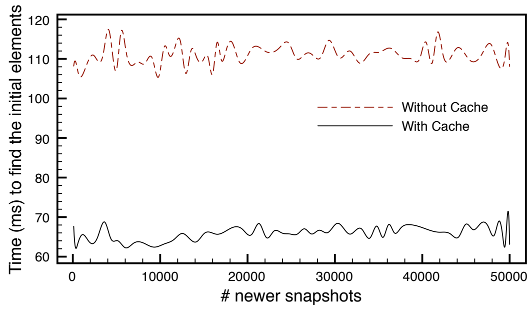


Figure 10. Time with and without cache to retrieve 1000 values in a versioned random treap from a snapshot s depending on the number of snapshots taken after s.

6.2 Capturing Stateful Execution Traces

This benchmark shows the performance of our stateful execution tracer. We let the tracer record the execution trace for inserting a number of elements in a random treap data structure (recording the entry and exit of all methods of the three treap classes) and measure the execution time needed to produce that trace. Dividing this number by the number of elements that were added gives us the average time per insertion. We do the experiment without any tracing, for a stateless tracer that does not keep any state, and for a stateful tracer that uses HistOOry as described in Sec. 5.1.

Fig. 11 shows the results. Transforming a stateless tracer into a stateful tracer only adds a slowdown of a factor of 1.3. Not only was it very easy to upgrade the stateless tracer, the performance is also feasible for the added functionality.

6.3 Postconditions

In Sec. 5 we showed how we added checked postconditions to Smalltalk, and gave examples on two methods. This section shows how much this addition costs for each of these methods.

6.3.1 swap:with:

The method `swap:with:`, defined on class `SequenceableCollection`, swaps the place of the

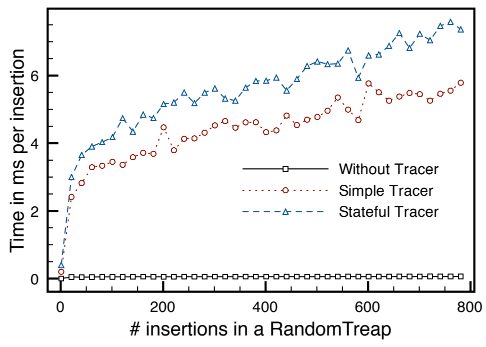


Figure 11. Execution times for adding elements in a treap without tracing, with a stateless tracer, and with a stateful tracer.

elements on the indices given as argument. For our experiment, we create collections of different sizes (ranging in size from 1 to 800 elements). We add either simple integers or array objects of 100 elements pointing to `nil`). We then perform 10,000 swaps at random indices and take the total time. Dividing this total time by 10,000 gives us the average execution time per swap.

We perform the experiment with three implementations of the `swap:with:` method: the original Smalltalk method, the method with a checked postcondition based on HistOOry and shown in Sec. 5.2, and the method where we add a checked postcondition based on doing a copy of the receiver before executing the swap, as follows:

```
SequenceableCollection>>swap: oneIndex with: anotherIndex
    "Move the element at oneIndex to anotherIndex, and vice–
      versa."
    | element old|
    old := self copy.     "copying the receiver before doing the
      swap"
    element := self at: oneIndex.
    self at: oneIndex put: (self at: anotherIndex).
    self at: anotherIndex put: element.
    self assert: ((old at: oneIndex) = (self at: anotherIndex) and: [
         (old at: anotherIndex) = (self at: oneIndex)])
```

Fig. 12 shows the results. It shows that an implementation that uses copies has an execution time that grows linearly with the size of the collection (and quickly, depending on the size of the data structure). The implementations based on HistOOry have a constant cost that does depend neither on the number of elements in the collection nor on the kind of the elements (integers or arrays). HistOOry does not take full copies of the receiver. When the first swap is performed, the fields are selected and a snapshot is taken. For all the following snapshots, the collection is already instrumented and everything is in place. Only a snapshot must be taken, which boils down to incrementing the global version number, before executing the normal body of the method.

The copying approach is faster than the HistOOry based approach for smaller collection sizes. The reason is simple:

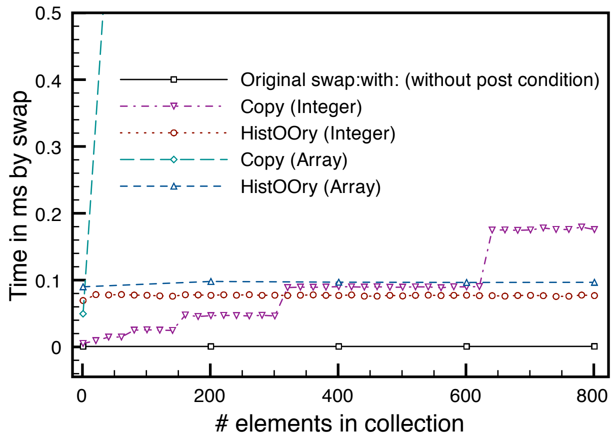


Figure 12. Postconditions (swap:with:): Number of elements in collection vs. time per swap.

the performance of a copy depends on the number of elements in the collection. It is obvious that the performance is better for small collections. HistOOry offers a low constant cost for any number (and any kind) of elements in the collection but this constant cost is higher than the simple copy operation for small collections.

6.3.2 addAll:

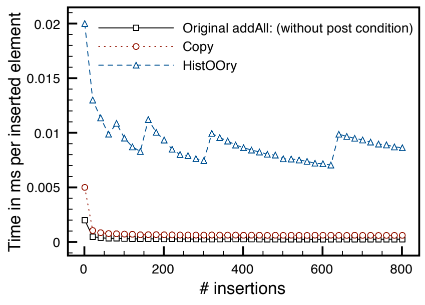
The method `addAll:`, defined on class `OrderedCollection`, adds all elements in the argument collection to the receiver collection. We compare two different scenarios. In the first scenario, we add a collection of a given number of elements to an empty receiver collection (and divide by the size of the argument collection to get an average per single insertion). In the second scenario, we add collections containing a single element, again starting from an empty collection.

We compare the original implementation with an implementation that has postconditions based on HistOOry as shown in Sec. 5.2 and with an implementation that copies the receiver state before executing the body of the method, as follows:

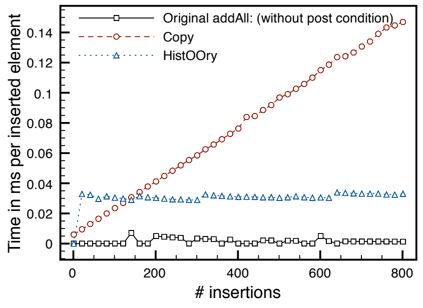
```
OrderedCollection>>addAll: aCollection
        "Add each element of aCollection at my end. Answer
      aCollection."
        |ans dcs dcc|
        dcs := self copy.
        dcc := aCollection copy.

        ans := self addAllLast: aCollection.
        self assert: ((dcc size = (aCollection size)) and: [
                  ((dcs size) +  aCollection size) = self size])
        ^ans
```

Fig. 13 shows the results for both applications, with adding the larger collections shown in Fig. 13(a) and adding collections of size 1 shown in Fig. 13(b). The results are similar to the previous experiment: we again see linear execution



(a)



(b)

Figure 13. Postconditions (addAll:): (a) Numbers of elements in collection to add vs. time (ms) per insertion. (b) Number of elements added one by one vs. time (ms) per insertion.

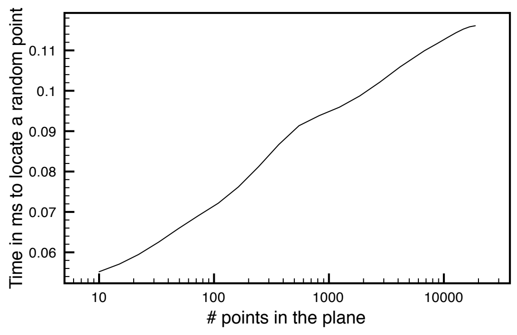


Figure 14. Planar Point Location

time for the implementations based on copying and bounded execution time for the HistOOry-based implementations.

6.4 Planar Point Location

Fig. 14 shows the time used to search the polygon in which a randomized point is. This curve is logarithmic as expected.

7. Related work

We have split the related work in three subsection. We first discuss approaches that are directly related to ours, then look at applications that use object versioning internally and therefore have an implicit versioning system built-in, and finally look at object versioning in Java.

7.1 Similar work

In [Marquez 2007], Marquez describes an orthogonal object versioning system in Java, providing long-lived versioned objects. Unfortunately, only an overview of the system is given in the 4 page paper, without details about the language design . Neither the actual data structures used nor the performance of the system are given. The project seems to have stopped in 2005, and it is no longer available, so we were not able to inspect their approach ourselves and properly compare it to our work. In contrast, we tried to be very clear about what we did and how, and provide detailed numbers so that future approaches can properly compare their results with ours.

In [Bertino et al. 1998], Bertino *et al.* extend the Object Database Management Systems model with the notion of time. Their model is formally defined and complete. But no details about its implementation are given. The conclusion section mentions in one sentence that B^+-trees are used, without more detail. Again we cannot do a proper comparison of this approach with ours.

ObjectFlow [Lienhard et al. 2009] is a tool to follow the flow of an object through a system, from its creation to its destruction. Amongst other things it keeps track of assignments made to its fields. This means that all states of these fields are kept, which is similar to what HistOOry does.

While this sounds similar, ObjectFlow also differs on several key points from HistOOry because its goals are different. First of all ObjectFlow fixes the scope of what objects are versioned to the process (thread): any object manipulated in the process is versioned. Secondly it always records all state changes. Note that both of these design choices make perfect sense in the context of ObjectFlow, so their approach does not offer options to change this. Because we have a general approach we have options to decide what fields to save and when. Thirdly ObjectFlow is implemented in the virtual machine while HistOOry is implemented in the source code. Both of these choices have advantages and disadvantages. Implementing a very complex data structure in the lower-level languages used in the virtual machine is not trivial. On the other hand it would probably be even faster than our current implementation. Finally HistOOry and ObjectFlow share that all states are kept in the object space, permitting an automatic garbage collection of no-longer-used states.

7.2 Applications Using Object Versioning

One category of applications that frequently use object versioning are advanced debuggers [Pothier et al. 2007, Lienhard et al. 2008, Feldman and Brown 1989, Boothe 2000] and model checking tools that are based on execution, such as Java Pathfinder [Visser et al. 2000]. The Omniscient Debugger, for example, executes a program and remembers the states objects went through to give the possibility to the developer to return at any point in the execution's past. Other information is also saved during execution, such as the method calls and the method return values.

The difference between these applications and HistOOry is that HistOOry was from the ground up designed to be a language extension to make it easy to remember and use object states, paired with an infrastructure to do so efficiently. The applications all have their own implementation that is application-specific and only meant to keep those states needed by the application.

HistOOry can therefore be seen as a general layer that any of these applications could have used.

This would have eased the implementation of these approaches, because developing a full fledged performant object persistence mechanism is not trivial. On the other hand, HistOOry is a general-prupose object versioning framework and some applications will still benefit from having specific structures and algorithms optimized for their particular usage.

Finally, software transactional memories [Shavit and Touitou 1995] can be also considered to be an application of object versioning. Transactional memories can be decomposed in three parts:

1. When a transaction begins, the states of interesting objects are saved;
2. During the transaction, modifications of states performed in the transaction are not visible outside the transaction;
3. A transaction is finished when either the code of the transaction executed without problem and the modifications are commited, or because an error occurred and a rollback of the saved states occurs.

From this breakdown it becomes clear that HistOOry is currently not really suited to support software transactional memories. One reason is that the visibility of the states in HistOOry is global. The second is that a rollback is not directly supported. While such functionality could be built on top of HistOOry we think that the performance and ease-of-use will suffer. A modified version of HistOOry that retains the data structure but allows to keep local changes would be interesting. We feel that it would enhance software transactional memories with the ability to have internal fine-grained data-driven rollback where the past states of unaffected variables can be retained over executions while other ones are recomputed.

7.3 Object Versioning in Java

We mentioned that this paper is the second in our research on efficient object versioning. The first paper presented the first-ever published implementation of the fat node method of [Driscoll et al. 1986]. It relied on AspectJ to instrument changes to fields. It scaled very well, due to the properties of the chosen algorithm, but also had a big overhead and was not very robust.

This paper revisits the algorithms and data structures, adding the cache to substantially improve performance (as shown by Fig. 10 in Sec. 6.1.3). We also did a complete reimplementation in Smalltalk, where we directly manipulate the byte codes to have less overhead than relying on an aspect-oriented programming framework. This implementation is also more robust and practical. The Java version, for example, uses one global variable to determine whether or not we are browsing or recording old states. If old states are browsed in a thread, any update of a versioned object in any thread raises an error. Our new implementation (described in Sec. 4.2.3) uses the Smalltalk processes to save or browse state local to a thread. Last but not least we have fully integrated HistOOry in a language, and used it in a number of applications.

8. Future Work

HistOOry implements a partial versioning model that only provides viewing of stored states. We are currently working on a fully versioning model that removes this limitation and makes it possible to create new states in the past by modifying saved states. Algorithms for this use advanced data structures and pose several interesting implementation challenges. Like in this work, we want to have a completely object-oriented transparent solution that is efficient and well-integrated.

We also want to make some more algorithmic improvements. One possible improvement is to introduce the possibility of supporting multiple "local" snapshots instead of having only one global snapshot. This has the potential to reduce the number of states that need to be saved.

Last but not least we will develop more applications that use HistOOry, for example a stateful debugger with built-in query capabilities that can take advantage of the past states.

9. Conclusion

This paper introduced HistOOry, an efficient in-memory object versioning system. The efficiency is due to our object-oriented implementation of, and changes to, an efficient data structure to keep past states. From the practical point of view, we have shown how existing applications can be made object versioned without much effort, simply by either sending them a message or using a *class extension* to override a default method. Regardless of this choice, fine-grained control is offered on what fields of an object are versioned, and when exactly the states are saved. Therefore, our solution is general enough to support applications that need object versioning but have different needs. Debuggers might want to save every single state change of a lot of objects, while other applications like an execution tracer might want to only record certain states for certain messages being sent. Even though it is general, our solution only requires three basic primitives, making it easy to learn and use. Properly integrating it in the language, like we did in Smalltalk, makes it easy to transform existing applications that do not use object versioning into ones that have such support. We have shown how to do this by extending the Smalltalk language with checked postconditions, by extending an execution tracer to become stateful, and by implementing a planar point location program. Benchmarks show that the overhead for storing states is constant, and does not scale with the number of states that need to be stored. It is our hope that by presenting HistOOry applications that currently use ad-hoc and inefficient object versioning implementations will be able to take advantage of our approach.

Acknowledgments

We thank the reviewers as well as Yann-Gaël Guéhéneuc for their helpful comments on drafts of this paper.

References

Malcolm Atkinson. Orthogonally persistent object systems. Nov 1995. URL `http://citeseer.ist.psu.edu/411649`.

Elisa Bertino, Elena Ferrari, Giovanna Guerrini, and Isabella Merlo. Extending the odmg object model with time. In *In Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 41–66, 1998.

Bob Boothe. Efficient algorithms for bidirectional debugging. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 299–310, New York, NY, USA, 2000. ACM. ISBN 1-58113-199-2. doi: http://doi.acm.org/10.1145/349299.349339.

John Brant, Brian Foote, Ralph Johnson, and Don Roberts. Wrappers to the rescue. In *Proceedings of ECOOP'98*, pages 396–417, 1998.

Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. *Object-Oriented Reengineering Patterns*. 2002. ISBN 1-55860-639-4.

Marcus Denker, Stéphane Ducasse, and Éric Tanter. Runtime bytecode transformation for smalltalk. *Computer Languages, Systems & Structures*, 32(2-3):125–139, 2005.

D. Dobkin and R. Lipton. Multidimensional searching problems. *SIAM Journal of Computing 5*, pages 181–186, 1976.

James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, pages 86–124, 1986.

Stéphane Ducasse. Evaluating message passing control techniques in Smalltalk. *Journal of Object-Oriented Programming (JOOP)*, 12(6):39–44, 1999.

Stéphane Ducasse, Tudor Gîrba, and Roel Wuyts. Object-oriented legacy system trace-based logic testing. In *Proceedings of CSMR'06*, pages 35–44, 2006.

Stuart I. Feldman and Channing B. Brown. Igor: a system for program debugging via reversible execution. *SIGPLAN Not.*, 24(1):112–123, 1989. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/69215.69226.

Elnar Hajiyev, Mathieu Verbaere, and Oege de Moor. Codequest: Scalable source code queries with datalog. In *Proceedings of ECOOP '06*, pages 2–28, 2006.

Abdelwahab Hamou-Lhadj and Timothy Lethbridge. A survey of trace exploration tools and techniques. In *Proceedings IBM Centers for Advanced Studies Conferences (CASON 2004)*, pages 42–55, 2004.

Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *Proceedings of ECOOP'01*, pages 327–353, 2001.

Danny B. Lange and Yuichi Nakamura. Object-oriented program tracing and visualization. *Computer*, 30(5):63–70, 1997.

Adrian Lienhard, Tudor Gîrba, and Oscar Nierstrasz. Practical object-oriented back-in-time debugging. In *ECOOP '08: Proceedings of the 22nd European conference on Object-Oriented Programming*, pages 592–615, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-70591-8. doi: http://dx.doi.org/10.1007/978-3-540-70592-5_25.

Adrian Lienhard, Stéphane Ducasse, and Tudor Gırba. Taking an object-centric view on dynamic information with object flow analysis. *Comput. Lang. Syst. Struct.*, 35(1):63–79, 2009.

A. Marquez. Orthogonal object versioning in an odmg compliant persistent java – extended abstract, 2007. URL `http://www.cs.adelaide.edu.au/~idea/idea7/PDFs/marquez.pdf`. Presented at the School of Computer Science, University of Adelaide, Australia.

Bertrand Meyer. Applying design by contract. *IEEE Computer (Special Issue on Inheritance & Classification)*, 25(10):40–52, 1992.

Todd Millstein and Craig Chambers. Modular statically typed multimethods. In *Proceedings of ECOOP '99*, pages 279–303, 1999.

Frédéric Pluquet, Stefan Langerman, Antoine Marot, and Roel Wuyts. Implementing partial persistence in object-oriented languages. In *Proceedings of the Workshop on Algorithm Engineering and Experiments (ALENEX08)*, 2008.

Guillaume Pothier, Éric Tanter, and José Piquer. Scalable omniscient debugging. *SIGPLAN Not.*, 42(10):535–552, 2007. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/1297105.1297067.

Neil Sarnak and Robert E. Tarjan. Planar point location using persistent search trees. *Commun. ACM*, 29(7):669–679, 1986. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/6138.6151.

Raimund Seidel and Cecilia R. Aragon. Randomized search trees. *Algorithmica*, 16(4/5):464–497, 1996.

Nir Shavit and Dan Touitou. Software transactional memory, 1995.

Willem Visser, Klaus Havelund, and Guillaume Brat. Model checking programs. In *Automated Software Engineering Journal*, pages 3–12, 2000.

Darren Willis, David J. Pearce, and James Noble. Efficient object querying for java. In *Proceedings of ECOOP'06*, pages 28–40, 2006.

Roel Wuyts. *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation*. PhD thesis, Vrije Universiteit Brussel, 2001.