Efficient Object Versioning for Object-Oriented Languages from Model to Language Integration

Thèse présentée par

Pluquet Frédéric

Sous la direction de

Messieurs Roel Wuyts et Stefan Langerman

En vue de l'obtention du grade de

Docteur en Sciences

de l'Université Libre de Bruxelles

Bruxelles, Belgique

2012

(Defended July 3, 2012)

Computers are incredibly fast, accurate, and stupid. Human beings are incredibly slow, inaccurate, and brilliant. Together they are powerful beyond imagination.

Albert Einstein

Remerciements

Ces remerciements sont rédigés en français car je désire être sûr que les personnes concernées comprennent combien je les remercie.

Merci à toi, **Magali**, qui crois en moi, qui me donnes la force de terminer ce que j'entreprends et qui es toujours là quand j'en ai besoin. Je profite de cette page qui restera pour longtemps dans notre bibliothèque pour te dire combien je suis fier de toi, de notre fille Zélie, de nous trois et ce que nous construisons chaque jour ensemble. Merci. Je t'aime.

Merci à toi, M. **Roel Wuyts**, qui m'as suivi durant toute cette aventure, qui m'as aidé lorsque j'en avais besoin, qui as laissé une liberté totale à mon esprit débordant d'idées. Travailler à tes côtés a été, pour moi, plus qu'un grand honneur.

Merci à toi, M. **Stefan Langerman**, qui as rejoint un peu après Roel le bureau de mes directeurs. Tu as su m'apporter l'éclairage nécessaire sur les côtés sombres de l'algorithmique. Tes encouragements finaux m'ont aidé à trouver la confiance dont j'avais besoin pour terminer cette thèse.

Merci à vous, ma **famille**, mes parents, mon frère, ma belle-soeur Sab, mes beaux-parents, sans qui je ne serai jamais arrivé là où j'en suis aujourd'hui. Merci à vous tous d'avoir été présents sans m'avoir trop demandé "Alors cette thèse?".

Merci à vous, mes **amis**, JB, Vivi, Greg, Virginie, Renaud, Marie, Fabien, Bastien, Armina, Cindy, Maxime, aux trogossiens et à tous ceux que j'oublie. Vous m'avez permis de m'évader lorsque j'en avais besoin.

Merci à vous, mes **collègues** avec qui j'ai pu travailler durant mon passage à l'ULB. J'ai appris énormément en votre compagnie. Nos mardisages me manquent déjà. Je tiens à remercier particulièrement Eythan Levy, Olivier Markowitch et Antoine Marot.

Merci à tous ceux qui ont participé de près ou de loin à la réalisation de cette thèse et à ce que je suis devenu.

Contents

Chapter 1

Introduction

Computer science is divided into several main domains, such as formal methods, parallel system, databases, artificial intelligence, algorithms and data structures, programming language theory and implementation. These domains are obviously linked but there are more or less wide gaps between them.

For instance, many efficient theoretical algorithms are never implemented in practice. Versioning is a good example. Data structures are called *versioned*[1] if they support access to multiple versions of that data structure [Driscoll *et al.* , 1986]. Versioned data structures make it possible to go back in time and revisit their state at some point in the past. Undo/redo functionality, the most used example of versioning implemented in almost all desktop applications, allows one to revert to previous states of the edited document. Many other applications, such as file editors [Reps *et al.* , 1983], debuggers [Lienhard *et al.* , 2008, Pothier *et al.* , 2007], execution tracers [Lange & Nakamura, 1997, Hamou-Lhadj & Lethbridge, 2004, Ducasse *et al.* , 2006] and computational geometry [Aurenhammer & Schwarzkopf, 1991, Yellin, 1992, Eppstein, 1994, Goodrich & Tamassia, 1991, Acar *et al.* , 2004, Gupta *et al.* , 1994, Agarwal *et al.* , 2003, Bern, 1988, Hershberger, 2006, Cheng & Ng, 1996, Mehlhorn *et al.* , 1994, Klein, 2005, Agarwal, 1992, Turek *et al.* , 1992, Koltun, 2001, Cabello *et al.* , 2002, Edelsbrunner *et al.* , 2004, Bose *et al.* , 2003, Bern *et al.* , 1990, Aronov *et al.* , 2006, Demaine *et al.* , 2004], also need versioning support.

Theoretical algorithms have been proposed by several research teams to make efficient versioned data structures. There are techniques that work on any kind of structure and

[1] We avoid the algorithmics term *persistent* that designate the same kind of data structures 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.

other that only work on specific structures (tables, trees, etc.). These theoretical results were validated by leading figures of research but when a developer needs to use versioning, often he will develop an ad hoc solution to the target program, mostly not optimal.

It is interesting to ask why there is such a gap between theoretical algorithms and implementation. We believe these two areas simply do not share the same goal. Algorithmic researchers study theoretical bounds. For each problem, they try to find algorithms with theoretically optimal running times. The actual implementation of these complex algorithms takes too much time: there is no programming language that allows a simple transcription of scientific article algorithms into a compilable program. Therefore, if developers need their work, they must read it and implement it.

On the other hand, developers are often happy with a code that works fast enough, even if it is not necessarily optimal. Only when the execution is too slow for their application that they will try to read scientific articles in algorithms. But these articles are often difficult to understand for developers because they are intended for the algorithmic community, and preliminary algorithmic knowledge is necessary for their understanding.

This gap is difficult to fill: knowledge to acquire in each area is substantial and it is difficult to understand precisely the problems and solutions related to each area. However the implementation of theoretical algorithms provides interesting contributions in both domains. First, it validates (or invalidates) the theoretical results in practice. For example, an algorithm with a constant upper bound might be unusable in practice if the constant factor results in a very large number of operations in the implementation.

Second, the implementation of theoretical algorithms checks the realism of the theoretical assumptions and constraints imposed on the data or system. Indeed it is common to find theoretical algorithms that work very well if certain constraints on the system or the data are checked. However these constraints can be completely unworkable in the implementation.

Third, during the actual implementation of algorithms, some theoretical issues may arise. These issues may be related to an optimization for a specific practical problem or related to the integration of the algorithm in a programming language.

This thesis builds a bridge between the algorithmic domain and the implementation domain by studying how to realize an efficient and expressive system for object versioning, i.e. versioning applied to object-oriented languages. Our system allows one to save and browse the old states of objects, manipulate the time line, while requiring no modification

of code to make it versioned.

1.1 Versioning

Human beings have always been concerned about the past. Just look at the archive rooms full to bursting in all jurisdictions and other courts. We keep track of the past because the past has value to us. We base on it to understand the present and try to guess the future.

Versioning is a discretization of the past, in which the past is divided into a set of versions. A version can be seen as a picture of the past at a given time t. Versioning is used for a long time in computer science. Take the example of the feature "undo" in the text editor that allows retrieval at previous versions of the edited document. This feature was introduced for the first time by IBM in 1976 [Miller & Thomas, 1976] and this is yet a feature we use every day.

We arrive at a point in computer history where versioning takes an important part in our lives. Just look at the number of applications that already use it. All development teams use tools such as SVN or GIT with the aim of keeping track of different versions of files and folders to work more effectively as a team. Apple has integrated directly into its OS automatic versioning of user's files by storing them on an external hard drive called "Time Machine". More recently, Apple introduced "Versions", a versioning system related application for a more detailed integration of versioning. Google has also integrated versioning for collaborative work in their projects "Google Wave" and "EtherPad". The "Dropbox" project which allows one to save files in the cloud also keeps the different versions of uploaded files.

In parallel with these developments, theoretical algorithms and data structures have been studied to store different versions of a data structure by using a minimum of time and space. First, Mark H. Overmars [Overmars, 1981] provides data structures that store or retrieve a version in linear time on the total number of versions. Five years later, Driscoll, Sarnak, Sleator and Trajan [Driscoll *et al.* , 1986] provided techniques to store and retrieve the different versions of any data structure with an minimum slowdown. These techniques are complex and require a good knowledge of algorithms to understand them.

Nowadays it is not easy to add versioning in applications for any developer. First, ver-

sioning is a crosscutting feature that modifies the code of all the application to browse the different versions and handle modification of an old version. Managing versioned information can be a difficult task if this is not well separated from the rest of the code. Moreover, a brute force versioning implementation will be inefficient in time and space. The advanced techniques of Driscoll et al. require advanced data structures that could be difficult to understand and to implement for some developers.

1.2 Object Versioning

We focus this thesis on versioning applied to object-oriented languages, i.e. to save and browse the different versions of an object-oriented application, and to allow developers to integrate, in an easy and efficient way, the versioning in any object-oriented application.

More precisely, we consider an *object* as a data set, composed of *fields*, associated with a set of behaviors called *methods*. The value of a field, that is, what it contains, is either some object or nothing. The objects form thus a graph in which each object can be connected to others through their fields.

We define the *state of a field* at time t as the value of this field at time t. By extension we speak about the *state of an object* at time t to express the states of all its fields at time t. We define the *version of the system* as the state at a given time of all objects that belong to the system.

A state or a version is *accessible* (or *saved*) if it can be retrieved. A non accessible state or version is lost and can not be browsed by the system. We speak about an *ephemeral system* to express a system in which only the last version of the system is accessible and the previous versions are lost. On the other hand we speak about a *versioned system* to define a system where several versions of the system can be saved. Common object-oriented languages, as Java and C++, are designed to produce ephemeral systems and do not allow one to build versioned systems easily.

A versioned system has always a first version, called the *root version*. A version v is created always from another version v_p, called its *predecessor*. The *version graph* of a system is the graph in which each version of the system is connected to its predecessors and successors. This graph defines how the different versions have been built over time.

In this dissertation, we study the transparent and efficient implementation of object ver-

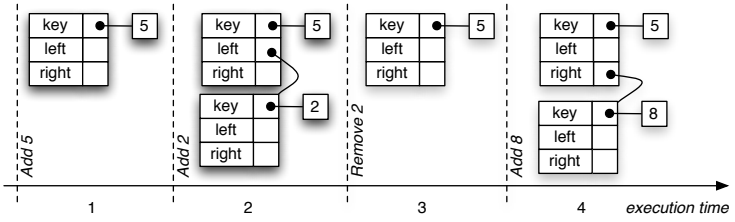


Figure 1.1: Linear Versioning Example: Four states of a tree where each state is produced from the previous one.

sioning in an object-oriented language. We study the algorithmic techniques to make its implementation efficient in time and memory space and we also study how to integrate object versioning in an object-oriented language so a developer can use it easily and accurately.

There exists several kinds of versioning: *linear* versioning, *backtracking* versioning, *branching* versioning and *confluent* versioning.

1.2.1 Linear Versioning

Linear versioning allows the browsing of old versions in read-only mode, i.e. the versions can be browsed but not updated. Only the last version can be updated, creating a new version. Linear versioning is called *partial persistence* in algorithmics [Driscoll *et al.* , 1986]. This kind of versioning is used in many geometric problems, such as the classical planar point location problem [Sarnak & Tarjan, 1986]. New generation debuggers [Lienhard *et al.* , 2008, Pothier *et al.* , 2007] also use it to save all object states (while classical debuggers use the stack to retrieve last object states).

The example in Figure 1.1 shows the states of a binary search tree with linear versioning: we start with an empty tree and we add the key 5 (state 1), we add the key 2 (state 2), we remove the key 2 (state 3) and finally we add the key 8 (state 4). These four states can be browsed but no modification on saved objects can be performed on states 1 to 3. The tree can be updated only from the state 4.

Because each version is created from the last one, the version graph in linear versioning is always a path (each version has zero or one successor).

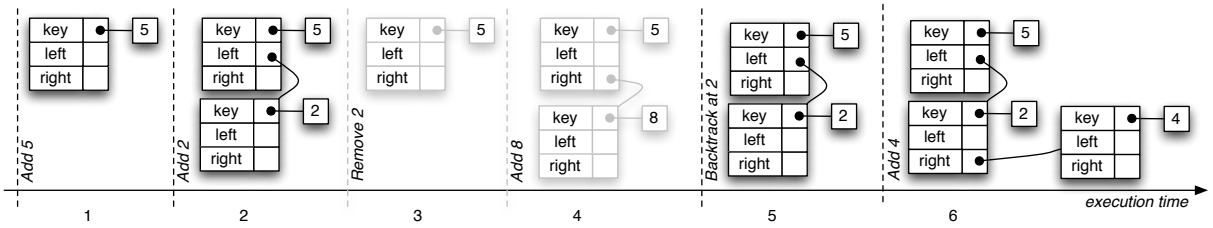


Figure 1.2: Backtracking Versioning Example: The two last states of a versioned tree are backtracked.

1.2.2 Backtracking Versioning

Backtracking versioning allows one to browse versions and delete any version after a given point in the past. It is used in many desktop applications to undo some operations, such as in most desktop editors (OpenOffice suite, Programming IDE, etc.). The user performs some operations (deletes part of the text for example). Using the "undo" functionality the user can retrieve previously saved versions of the text. The user can "redo" an operation, allowing to retrieve previously undone states. If redo operations are still possible but the user changes the text (edit a sentence for example), the redo operations are forgotten: there is no way to retrieve the redo-able states of the text. However previous versions are always available.

Figure 1.2 shows an example with a versioned tree: an empty tree in which we add successively the keys 5 (state 1) and 2 (state 2), we remove then the key 2 (state 3) and we add the key 8 (state 4). We then decide to backtrack the system until the state 2 and all states from the state 3 are simply forgotten. The last state of the tree has therefore the keys 5 and 2. The previous states 3 and 4 are no longer accessible. After this backtracking we add the key 4 (state 6).

As in linear versioning the version graph forms a path because each version is created from the last non backtracked version.

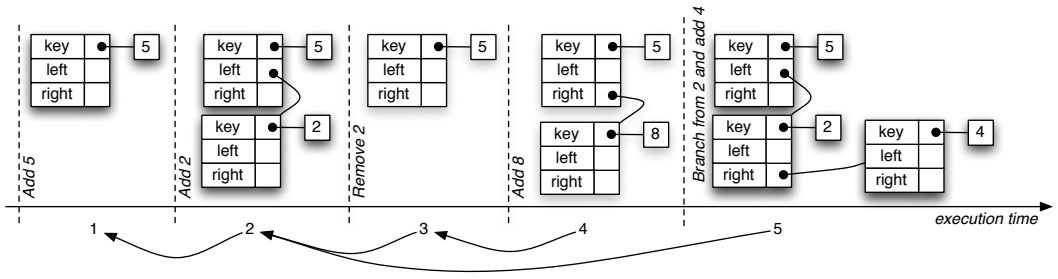


Figure 1.3: Branching Versioning Example: State 5 of this tree is created from the state 2.

1.2.3 Branching Versioning

Branching versioning allows one to retrieve old states of the system and create a branch from any state in the past. The branching versioning is called *full persistence* in algorithmics [Driscoll *et al.* , 1986].

Figure 1.3 shows an example of the branching versioning on a tree. This example starts like the one used in Section 1.2.1. After the fifth state, we return to the third state and we create a new state 6 from this state. We then add a new key 4 (state 7): the tree is therefore composed of 3 items (2, 4 and 5).

In branching versioning, each version can be created from any version. A version has therefore none, one or many successors: the version graph is therefore a tree.

1.2.4 Confluent Versioning

Confluent versioning allows one to retrieve old versions of the system, create a branch from any version in the past and merge two versions to create a new one. The confluent versioning is called *confluent persistence* in algorithmics [Driscoll *et al.* , 1994].

Figure 1.4 shows an example of the confluent versioning on a tree. The four first states are the same than in previous examples. The fifth state is created by merging the states 2 and 4. The result of the merging operation can depend on some rules of merging. Here the merging results in a tree that contains the values 2, 5 and 8.

This kind of versioning is used in the popular file versioning system Subversion (SVN).

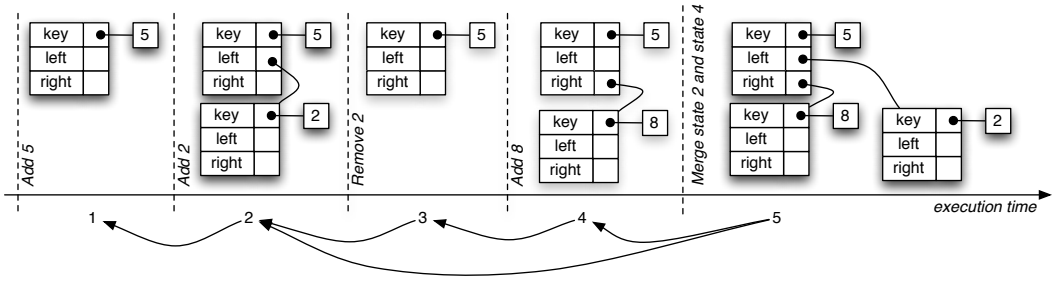


Figure 1.4: Confluent Versioning Example: State 5 of this tree is created by merging the states 2 and 4.

This kind of versioning will be not studied in this thesis; we concentrate our efforts on the linear, backtracking and branching versioning.

1.3 Problem

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

No Constraint on Application Design. We want that any existing object-oriented application can use object versioning. The classes can be defined without restriction on syntax, hierarchy building or any design constraint. Our tool must adapt to the design of the existing application and not the design of the existing application that has to adapt to our tool. Design patterns, collaboration between objects and inheriting of several classes must be not a problem when adding versioning.

Transparency. For convenience, objects can be versioned without the need for changing the implementation of their class. Transparency has two advantages. First, there is a gain of time: the system does the work for the developer. Second, there is separation of concerns between the business code and the versioning. The code can be written as any ephemeral system and the developer can concentrate efforts on the business code. We implement transparency using Aspect Oriented Programming (AOP) and bytecode transformation: when an object must be versioned, its class is instrumented automatically.

Three Kinds of Versioning. Each kind of versioning is interesting for different problems. Switching from one kind to another must be facilitated by using one common model (with few modifications for each kind) and one common application programming interface (API). The three kinds of versioning share a same model and a same API, with some addition for backtracking and branching versioning.

In-memory Object Versioning. We are interested only by in-memory object versioning, where all states are saved in the same memory space as the application. It must be designed for applications that need in-memory versioning, such as most debuggers or geometric problems. For such applications, file systems and database solutions will be too slow because of disk accesses; old states of objects must be saved and available as fast as possible. Backing up objects on a physical medium to restore them at a later time will take too long. In our implementation, all states are saved in the same memory space than the application. The data structures and algorithms are optimized to take a minimum of space. The transparent integration of object versioning allows a separation between the objects used by the application and the data structures used to store states: our system mechanisms can completely be hidden from the developer.

Efficiency. Saving old states of all fields of many objects requires space and time. But we developed two axes to minimize time and space required for each kind of versioning.

On the one hand we define a *fine-grained model of selection* where given fields of an object can be versioned while other ones are not. Moreover because different applications have different requirements, the model allows one to specify when states must be kept. This selection reduces the number of saved states and therefore improves efficiency, while leaving the possibility to save all states of the application.

On the other hand, we study efficient data structures and algorithms to reduce considerably the time to store and browse states and their space. For linear and backtracking versioning, an update has a constant cost that does not increase with the number of objects that have already been versioned. For branching versioning an update has a logarithmic cost in number of states.

For all kinds of versioning querying a past version of the object graph can be done with a cost that is logarithmic to the number of saved versions. The required space is linear in terms of states saved.

Versioning Orthogonality. Full support for object versioning ideally should support the same design principles as orthogonal persistence (that aims to transform short-lived objects to long-lived objects) because this has proven to be useful when dealing with saved object states [Atkinson & Morrison, 1995]:

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

Our system 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. Moreover our model defines rules for cohabitation of ephemeral and versioned objects.

Language Integration. The versioning mechanism should be properly integrated in the programming language, such that:

- An easy-to-use API must be provided.
- The encapsulation of data objects can not be violated by the system.
- The language tools (such as garbage collection and reflective methods) must work as expected.

We integrate our system in two object-oriented programming languages: Smalltalk and Java. For a basic usage, it only requires three primitives, making it easy to learn and use.

1.4 Existing solutions

Two partial solutions were previously proposed to introduce efficient versioning methods of Driscoll et al. in languages. The first one is the Zhiqing Liu persistent runtime

system [Liu, 1996]. The entire system is persistent and uses a persistent stack and a persistent heap to save changes. The granularity of changes to be recorded can be tuned to manage the quantity of recorded data. This solution is not flexible enough to version a subset of classes. This is a serious drawback because it is common that only a subset of all used structures for algorithms must be made versioned. The second previously existing solution is the Allen Parrish et al. persistent template class [Parrish *et al.* , 1998]. A template class `Per` is provided by the author. The author admits that the solution suffers from some problems (e.g., because of references in C++) and it is not transparent for the program since variable declaration must be modified by hand.

On the other side all previous practical attempts to save previous states in a general and transparent way lack some of the main advantages of Driscoll et al.'s efficient technique: some papers [Reiss & Renieris, 2000, Reiss & Renieris, 2001] propose techniques to trace a program, events are logged, but full snapshots of previous versions are not readily accessible. Caffeine [Guéhéneuc *et al.* , 2002] on the other hand stores previous states as prolog facts for fast future queries, but the snapshots are taken by brute force, as a copy of the entire set of objects to trace.

1.5 Inspiration Of Work

Our model is related to two other important research fields: orthogonal persistence and temporal and versioned databases.

1.5.1 Orthogonal Persistence

Orthogonal persistence aims to transform short-lived objects to long-lived objects with the maximum of ease of use for the developer. A short-lived object is typically defined as an object that is created and deleted during the lifetime of a program [Atkinson & Morrison, 1995]. A long-lived object remains available even after the program is finished: it can be saved in files, in a database or anywhere else. The benefits for the developer are the independence of the support to save objects (e.g. files or database), the independence of the types of objects to save (any object can be saved) and the freedom to manipulate short-lived and long-lived objects.

Our system and orthogonal persistence do not have the same goal but they share the three same principles [Atkinson & Morrison, 1995]: persistence independence, data type orthogonality and persistence identification.

Persistence independence states that the longevity of the data has no impact on the form of the program: short-term and long-term data are manipulated in the same way. We apply this principle to object versioning: an ephemeral object and a versioned object must be manipulated in the same way. As we will show in Chapter 5, our model can be integrated into a language in a such way that the selected objects are manipulated the same as non selected objects.

Data type orthogonality defines that all objects can be transformed into persistent objects, independently of their type: there is no object that is not allowed to be long-lived or not allowed to be transient. We will apply this principle to object versioning: any object is allowed to be versioned or not, independently of its type. Our model is completely independent from the type of the object: any object can be versioned independently of its type.

Persistence identification defines that the way to identify and provide persistent objects must be orthogonal to the universe of discourse of the system. We will show in Chapter 3 an orthogonal way to identify fields for which states must be keep.

A deeper analysis of the correlation between orthogonal persistence and object versioning is provided in Chapter 3.

1.5.2 Temporal and Versioned Databases

Temporal and versioned object-oriented databases are a good inspiration to design our system. We introduce them in this section. A deeper analysis of their correlation with our model is provided in Chapter 3.

1.5.2.1 Temporal Databases

A database is a program that aims to organize, store and retrieve data easily. The most studied databases are relational databases [Codd, 1970] and object-oriented databases [Won, 1990]. Relational databases contain a collection of tables in which entries

are interconnected by keys. Object-oriented databases are modeled around object concepts: data is managed as objects with fields, as in object-oriented languages.

Temporal databases improve classic databases by adding the notion of time to database data. The bitemporal model [Snodgrass, 1992], the most common model, manages a set of facts, i.e. logical statements that are true in real world. For example, "Albert works at iMec" is a fact. Temporal databases add time information to facts by using two kinds of time information: the valid-time and the transaction-time. The valid-time defines the period of time during which the fact is true. Outside of the valid-time the fact is considered as false. The transaction-time defines when the data is considered as available in the database. By default, the transaction-time of data covers the interval of time from the creation of the data to its deletion. The transaction-time allows one to search in the database in a previous version ("Ten years ago where did the database believe Albert worked?") and the valid-time allows time information ("Where did Albert work ten years ago?"). Notice that the valid-time and the transaction-time can reflect on the past, the present and the future.

1.5.3 Versioned Object-Oriented Databases

Versioning in object-oriented databases makes it possible save different versions of objects [Zdonik, 1984, Bjornerstedt & Britts, 1988, Beech & Mahbod, 1988, Oussalah & Urtado, 1996, Oussalah & Urtado, 1997, Rodríguez *et al.* , 1999, Khaddaj *et al.* , 2004, Arumugam & Thangaraj, 2006]. Their versioning is not global as our model, where all objects participate in a global system history. Their versioning is centralized on object graph defined as follows: Each object has a set of versions that contains its different states. New versions are created implicitly (at any change [Oussalah & Urtado, 1997] or following strategies [Oussalah & Urtado, 1997, Oussalah & Urtado, 1996]) or explicitly ([Zdonik, 1984, Bjornerstedt & Britts, 1988, Beech & Mahbod, 1988, Oussalah & Urtado, 1996, Oussalah & Urtado, 1997, Rodríguez *et al.* , 1999, Arumugam & Thangaraj, 2006]). When a new version of an object is created, the objects that refer to this object create a new version such that each object contains a set of versions that corresponds to each of its modifications and to each modification of its directly or transitively connected objects. Some techniques allow breaking this upward version propagation by configuration.

1.6 Contributions

This dissertation provides an original study of object versioning for object-oriented languages, with four main contributions:

Fine-grained model for object versioning. We provide a model for object versioning. This is the first model for object versioning that allows the developer to express with precision which fields must be versioned and which states of these fields must be kept. We define rules for the cohabitation of ephemeral and versioned objects that allow one to use object versioning in practice. Moreover we design this model such that it is completely independent of the physical support (memory, files or database) used to store states.

Efficient implementation of data structures and algorithms. We provide a way to efficiently implement data structures and algorithms, adapted from well-known algorithmic methods. We adapt them to provide real implementable data structures. We develop an efficient technique for backtracking versioning.

Language integration. We provide design principles to integrate our system in any object-oriented language in a transparent and elegant way by using object principles (e.g. polymorphism and inheritance) and tools (AOP and bytecode transformation).

Validation. We implemented our system in Java and Smalltalk. 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 [Sarnak & Tarjan, 1986]. We performed benchmarks for synthetic cases and for these applications that show that the required space and the execution time penalty are minimal. Moreover the expressiveness and easiness of the usage our system is clearly visible in the provided code of these applications.

1.7 Structure of the Dissertation

Chapter 2 presents the state of the art of persistence in data structures. Chapter 3 defines the object versioning model for object-oriented languages. Chapter 4 shows data structures and algorithms to efficiently implement the model proposed in Chapter 3. Chapter 5

focuses on the integration of object versioning in object-oriented languages. Chapter 6 validates first of all the expressivity of our model by the integration of object versioning in the three applications. Furthermore it validates the efficiency of our implementations. Chapter 7 concludes this dissertation by summarizing the contributions and suggesting future directions.

Chapter 2

Algorithmic Foundations

Saving history of objects within reasonable time and space requires advanced data structures and algorithms. Making data structures versioned (also known as *persistent* in the algorithmic domain) with efficiency has been studied from the eighties. In this chapter we present the state of the art of persistence in data structures. For the reader non familiar with the algorithmic terms, we start this chapter by a quick overview of the algorithmic tools used in the rest of this chapter and dissertation.

2.1 Basic tools

We overview in this section the algorithmic tools used in the rest of this chapter and dissertation.

Pointer Model vs RAM Model. In order to precisely analyze the execution time of algorithms it is necessary to define a model for the computer that will execute them. Two such models are used: the *pointer* model and the *random access memory* (RAM) model. In the pointer model [Ben-Amram & Galil, 1992], information is organized in a graph where each node points to a small number of other nodes. Information is accessed by following pointers from one node to another. In the RAM model [Cook & Reckhow, 1972, Aho & Hopcroft, 1974], the memory is seen as a big array where the information is stored in small boxes. Each box has an integer address. Any box can be accessed by giving its address. Whereas information is only accessible by browsing node by node in the pointer model, information in the RAM model can be browsed from the first to the last address without restriction (e.g. by computing

addresses). The RAM model is strictly more powerful than the pointer model since it is possible to simulate a pointer machine with a RAM machine. Furthermore, there are problems that require a factor $O(\log n)$ more time in the pointer model, such as sorting n integers in $[1, n]$ or accessing an arbitrary element by its index in a list of size n. Although some programming languages can be modeled with the simpler pointer model, many more require the more powerful RAM model. Therefore we chose to describe algorithms in the RAM model.

Cell-Probe Model. The *cell-probe model* is a model of computation where the cost of a computation is measured by the total number of memory accesses to a random access memory with $\lceil \log n \rceil$ bits cell size, where n is the size of the studied structure [Atallah & Fox, 1998]. All other computations are not counted and are considered to be free.

Big-O Notation. Big-O notation is a good way to describe complexity of algorithms and to compare them. This notation simplifies the exact complexity functions to show only their growth rates. The idea is that different functions with the same growth rate may be represented using the same O notation. The multiplicative members, lower order terms and additional constants are hidden. For instance, the function $f : 3 * x^3 + 5 * x + 3$ is upper bounded by $O(x^3)$. More formally, if $f(x)$ and $g(x)$ be two functions defined on a subset of the real numbers, $f(x) = O(g(x))$ if and only if there exists a positive real number M and a real number x_0 such that $|f(x)| \leq M|g(x)|$ for all $x > x_0$.

Time Complexity. The time complexity of an algorithm quantifies the number of elementary operations done by the execution of this algorithm. We express the time complexity of an algorithm by using the big-O notation in the standard model of computation (RAM).

Space Complexity. The space complexity of an algorithm quantifies the number of bytes needed in memory by the algorithm. This quantity is also described using big-O notation.

Named complexities. Some complexities are used often and have a special name. We present here three important ones: the constant, the logarithmic and the linear complexities. Let n be the size of the input.

Constant complexity. An algorithm has a constant time (resp. space) complexity if the time complexity (resp. space complexity) for any execution of the algorithm can be upper bounded by a given constant. Its big-O notation is $O(1)$.

Logarithmic complexity. The logarithm l of a number x in a base b (denoted $\log_b x$ or $\log x$ when $b = 10$) is such that $x = b^l$. For instance, $\log_{10} 100 = 2$ and $\log_2 8 = 3$. An algorithm has a logarithmic complexity if its complexity is $O(\log n)$.

Linear complexity. An algorithm has a linear complexity if and only if its complexity is bounded by $O(n)$.

Best, Worst and Average Complexities. The best, worst and average complexities express what the time (or the space) complexity of an algorithm is at least, at most and on average, respectively for one operation. For the average complexity, the execution time of the algorithm must be considered to be a random variable. Either the average is taken over an assumed probability distribution of the input, or the algorithm generates a random bits during its execution and the average is taken over those. Unless mentioned explicitly we will use the second meaning.

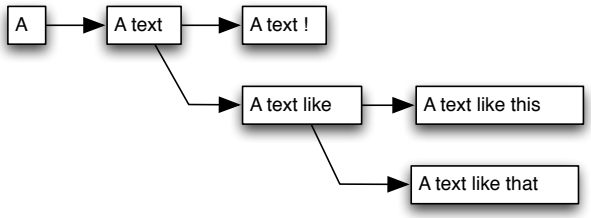
Amortized Complexity. The amortized complexity does not focus on one operation but on a sequence of operations. The idea is that some operations with high complexities can be mixed with some operations with low complexities. Therefore if we analyze the time to execute the sequence of operations versus the number of operations, the high complexity operations can be amortized by the low complexity operations.

More formally, the amortized complexity of an algorithm expresses a bound b_i for each operation i of the sequence of operations such that $\sum_{i=1}^{j} w_i \leq \sum_{i=1}^{j} b_i$ for $j = 1, .., n$, where n is the number of operations and w_i the actual complexity of the operation i. That is, after each operation, the total time spent until then is no more that the total amortized bound.

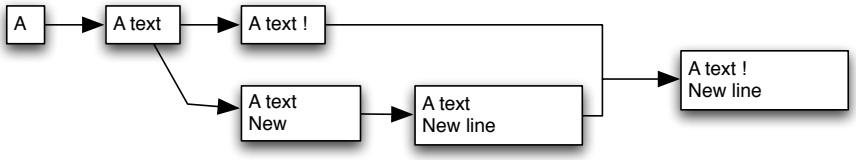
Take the example of a dynamic array. This kind of array doubles its size each time it fills up. The insertion operation has a worst case time complexity of $O(n)$ (where n is the number of elements in the array), because the array can be filled up before an insertion and a new array must be created in which all old elements must be copied. However, before the creation of a new array of size n, at least $\frac{n}{2}$ insertions have been made without resizing, taking each 1 unit of time. Charging an amortized cost of 3 to each insertion, we pay 2 extra units of time in advance for each insertion, so that when the array of size n is rebuilt, we have accumulated n credits that can be used for the rebuilding.

Note that the amortized complexity differs from the average complexity: the average complexity bounds the expected cost of one operation, without implying any bound in

(a) Partial Persistence



(b) Full Persistence



(c) Confluent Persistence

Figure 2.1: Example of usage of the three versioning with a text editor.

the worst case; the amortized complexity implies a worst case bound for any sequence of operations.

When a complexity is given without specifying its kind, we assume it is the worst case complexity.

2.2 General Persistence

The versioning of data structures is called "persistence" in algorithmics. A data structure has an initial state and at each update, a new state of this structure is generated. Classical data structures are *ephemeral*: once an update is performed, the previous state is no longer accessible. A data structure is called *persistent* if its different versions are accessible. We outline here several known methods to implement three kinds of persistence: partial persistence, full persistence and confluent persistence.

A data structure is *partially persistent* if all versions can be accessed but only the newest

version can be modified. Let the version graph be a directed graph where each node corresponds to a version and there is an edge from node V_1 to a node V_2 if an only if V_2 was created by an update operation on version V_1. In a partially persistent data structure the version graph is a path. Figure 2.1(a) shows the example of a text editor. Text entered in the application will be versioned using partial persistence. A new state is created whenever the space key is pressed. Assume that the text "A text !" is entered. Three states are then available ("A ","A text " and "A text !"); the two first ones are read-only.

A data structure is *fully persistent* if every version can both be accessed and modified. Each version has none, one or more next versions. Each version has exactly one parent version (except the first version that does not have a parent). The version graph of a fully persistent data structure is a tree. Figure 2.1(b) takes the previous example of the text editor but using full persistence. Here the user types "A text !" and then returns to the second state to create two new versions adding the text "like this". Finally the user returns to the previous state to add the word "that".

Finally a data structure is *confluently persistent* if it is fully persistent and has an update operation which combines more than one version. Each version has none, one or more next versions but also has one or more parents (except the first version that does not have a parent). The version graph is a directed acyclic graph (DAG). Figure 2.1(c) shows a text editor using the confluent persistence. The final state is created from merging two other states. The merged state takes the first line of the first state and the second line of the second state.

In the rest of this section, we show different techniques to transform any data structure into a persistent one. We focus on partial and full persistence. Confluent persistence will be mentioned but not developed in this dissertation.

2.3 Purely Functional Data Structures

The easiest way to use persistence is to select a language that is already persistent. For instance, in a purely functional language, in which there is no assignment and each structure is built from the previous ones, all data structures are intrinsically persistent, even confluently persistent. The analysis of algorithms developed in a functional model is often difficult. Okasaki [Okasaki, 1998] shows that some common data structures (e.g.

queues and red-black trees) have a functional variant that has the same efficiency as their imperative and ephemeral variant.

The functional paradigm is very interesting but it comes out of the scope of this dissertation.

2.4 Algorithms for Partial and Full Persistence

Over the years, several theoretical methods for partial and full persistence have been developed. We compare each method following several criteria:

Query time: the multiplicative factor for the time to access a field of a node of the data structure at a given version compared to the time to access the same field of the corresponding ephemeral data structure.

Update time: the multiplicative factor for the time to create one new version of the data structure.

Space: the multiplicative factor for the quantity of memory used when a persistent update is performed.

We present six existing methods. We add two new variants, easily deducible from the existing ones. We summarize their time complexities in Table 2.1.

2.4.1 Copy and Update Techniques

Some of the first work on persistent data structures was done by Mark H. Overmars [Overmars, 1981]. He studied general approaches to transform any ephemeral data structure into a persistent one. His simplest method, **update and copy**, creates a copy of the data structure after each update. When performing a query at version i, we look for the copy created just before or at i. This lookup is done in $O(1)$ by using an array to store the different versions. Once the copy is found, it can be used as-is. The query time is $O(1)$. The update time is clearly bounded by the size of the data structure: if the size of the structure grows by one at each update, the update time is bounded by $O(N)$. The overhead of space is bounded by $O(N)$.

	Partial			Full		
	UT	QT	SU	UT	QT	SU
Update and copy	$O(N)$	$O(1)$	$O(N)$	$O(N)$	$O(1)$	$O(N)$
Update List	$O(1)$	$O(N)$	$O(1)$	$O(1)$	$O(N)$	$O(1)$
Update List and copy	$O(\frac{N^2}{k}+N)$	$O(k)$	$O(\frac{N^2}{k}+N)$	NA	NA	NA
Fat node method	$O(\log n)$	$O(\log n)$	$O(1)$	$O(\log n)$	$O(\log n)$	$O(1)$
Node Copying method	$O(1)$*	$O(1)$	$O(1)$*	NA	NA	NA
Node Splitting method	NA	NA	NA	$O(1)$*	$O(1)$	$O(1)$*

Table 2.1: Complexities of all described methods. UT: update time; QT: query time; SU: Space used per update; N: total size of the corresponding ephemeral data structure; k: number of updates before a copy; n: number of updates saved for the queried/updated field; NA: Not applicable. The starred complexities are amortized.

This method is described only for the partial persistence. But the same mechanism could be used for full persistence: save all copies in an array. A separate tree of versions must also be maintained. The full persistence of this method is achieved with the same complexity than its partial version.

The second method proposed by Overmars is the **update list** that lists the modifications made in each update. To retrieve the data structure at version i, each update preceding i is replayed on the first version of the data structure, in this way rebuilding the state of the data structure at i, and performing the query. The used space is $O(1)$. The query time is bad, $O(N)$, where N is the number of updates. The update time is constant ($O(1)$).

Overmars does not show the adaptation of this technique to the full persistence but it could be done by maintaining a tree of modifications (branching at good version). To retrieve the data structure at version i, we perform each update found on the path between i and the root of the tree (starting from the root). The query time, the update time and the space overhead remain unchanged.

The third method proposed by Overmars, **update list with copy**, is a mix of the two first ones. Instead of copying the structure immediately after each update we copy the structure in its entirety only after k updates have occurred. Between two successive copies we store the list of modifications made since the last copy of the data structure. This structure uses $O(\frac{m^2}{k}+m)$ space and total update time for a sequence of m operations on a data structure of maximum size $O(m)$. To retrieve the data structure at version i we start with the copy of the structure S_j nearest before i. On S_j we perform all modifications

that have taken place up to i. In this way, we obtain a structure S'_j with the same state than S_j at version i, that can be queried. Therefore the time to reconstruct a version is $O(k)$. A large number of updates between two complete copies of the structure means a high query time and a low (average) update time while a small number gives a low query time and a high (average) update time.

2.4.2 Fat node method for Partial Persistence

Five years after [Overmars, 1981], Driscoll, Sarnak, Sleator and Tarjan [Driscoll *et al.* , 1986] study different methods to transform any ephemeral data structure into an efficient persistent one. These methods are considered to be the base of the theory of persistence.

These methods are based on the decomposition of a data structure. A data structure is seen as a graph of nodes, connected by their pointer fields. Whereas the previous techniques of persistence save the complete data structure after a certain number of modifications, the methods of Driscoll et al. focus on nodes themselves rather than on the entire data structure. All modifications done in a node are kept in this node or in some linked nodes. Note that when a complete copy of the data structure is made, the fields of all nodes have a value for each version. On the other hand, if the modifications are saved in the nodes themselves, their fields do not necessarily have a value for all versions: an update of a node does not affect necessary other nodes.

The first method is the **fat node** method. This method saves all modifications of a node in that node itself. The size of the node grows when updates are performed on its fields. The query time and the update time are bounded by $O(\log n)$, where n is the number of updates done in the node. The used space is constant per update.

An ephemeral node n is transformed into a persistent one during the version i as follows. For each field named f with a value v, an entry $\langle f, i, v\rangle$ is added in some extra fields of the node itself. When a new value v is set for a field f for a version j, if there is an entry $\langle f, j, v'\rangle$ in the node, v' is replaced by v in the entry. Otherwise an entry $\langle f, j, v\rangle$ is added.

When the value of a field f is queried for a version i, we search the extra fields for the value of the entry $\langle f, j, v\rangle$ such that j is the greater version number smaller or equal to i amongst all entries contained in the node.

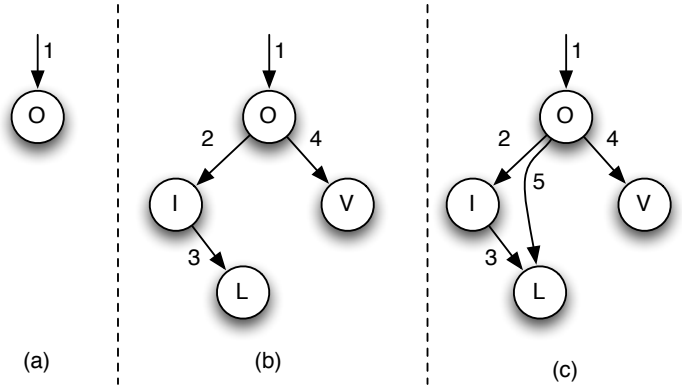


Figure 2.2: A partially persistent binary search tree built using the fat node method, in which O (step a), I, L and V (step b) are inserted followed by the deletion of I (step c). Left pointers leave the left sides of nodes and right pointers leave the right sides. Each pointer is labelled by its assignment version number.

Figure 2.2 shows an example of a binary search tree with the fat node method. Each node has two fields `left` and `right`. The step (c) of this example is detailed on Figure 2.3. The fields of ephemeral nodes contain the value of the last update. The extra fields of each node contain the different updates performed. When the tree is browsed in the version v, the extra fields that have a version number small or equal to v are considered: amongst them, we look for the ones that have the greater version number. For example, when we consider the field `left` of the root in the version 4, amongst the extra fields that correspond to `left`, we take the one with the version number 2, that points to the node I: 2 is the greater version number smaller or equal to 4 amongst the extra fields of the root that correspond to `left`.

If for each field f, all entries $\langle f, i, v\rangle$ are stored in a binary search tree with the version i numbers as search key, the update time is $O(\log n)$, the query time is $O(\log n)$ and the used space is $O(1)$, where n is the number of updates of the field. We show in this dissertation (Section 4.3.2, page 101) a new data structure that improves these complexities: $O(1)$ for the update time, $O(\log n)$ for the query time and amortized constant space per update.

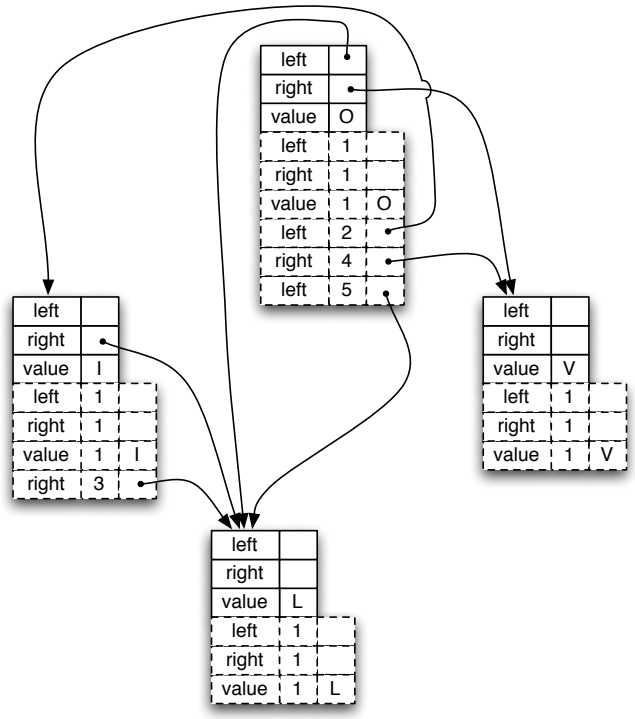


Figure 2.3: The details of each fat node of the tree at Figure 2.2 (c). The solid lined tables show the fields of ephemeral nodes and the dashed lined tables contain their extra fields (field name, version and value). We assume that all nodes are created at version 1 and linked further. The null pointers are not shown for more readability.

2.4.3 Node Copying Method

In the fat node method, a node contains all its versions. If the value of a field f_1 of a node n_1 at version i returns a link with other node n_2 and we want the value of a field f_2 of n_2 at version i, *all* versions contained in n_2 for f_2 must be considered.

Driscoll et al. [Driscoll *et al.* , 1986] introduce the **copying method** to break a node into subnodes (called copies) each containing a subset of all versions. When the value of a field of a node n_1 at version i returns a link to another node n_2, n_2 is a subnode containing versions close to the version of n_1 and therefore close to i. This technique improves the execution time for a sequence of operations performed on the same range of versions. It was inspired from *fractional cascading* [Chazelle & Guibas, 1986].

To implement the copying method, a node has to have several extra fields: e update fields, p predecessor fields and a copy pointer. A node maintains its predecessor nodes, i.e., nodes that point to it, in its predecessor fields.

The e first updates are saved in the node exactly like in the fat node method in the e update fields. When there is no more room to store a new version, the node is copied. The copy is initialized with the most recent values of fields and linked to the original node with its copy pointer. Finally all predecessors of the original node are updated to point to the copy. Note that this update for each predecessor x will fill an entry of the e update fields of x, or if they are full, might cause a copy of x which will require modifying the predecessors of x and so on.

The query of the value of a field f of a given node for version i is performed as in the fat node method (i.e. find the most recent version of f before or at i) but using the copy pointers to browse previous states if necessary.

The node copying method offers really good bounds: the amortized update time is $O(1)$, the worst-case query time is $O(1)$ (using arrays to store the value associated to version i at index i) and the amortized used space is $O(1)$. But the drawback of this method is that these bounds are respected only under a condition: p must smaller or equal than e, i.e. the maximum number of predecessors any node can have at any time should be bounded by a constant which is known beforehand.

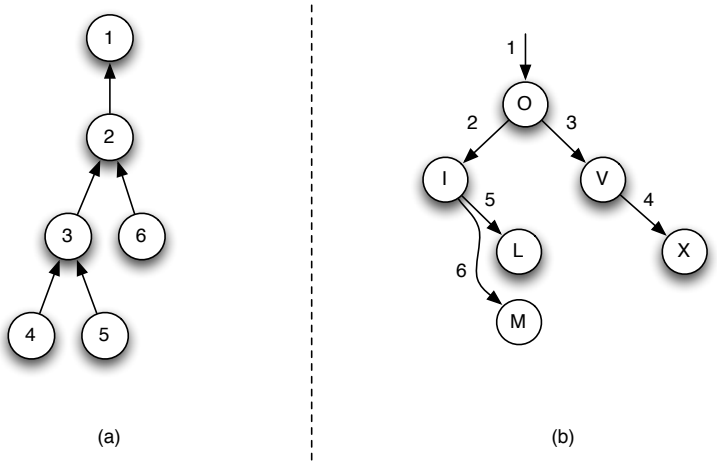


Figure 2.4: Right, a fully persistent binary search tree built using the fat node method, in which O, I, V and X are inserted followed by a branch operation at version 3 and an insertion of L, followed by a branch operation at version 2 and insertion of M. Left, the corresponding version tree. Each pointer is labelled by its assignment version number. The state of the binary tree for each version is shown on Figure 2.5

2.4.4 Fat Node Method for Full Persistence

Driscoll et al. [Driscoll *et al.* , 1986] shows that the fat node method is also applicable for full persistence with the following complexities: creating a new version of the system takes amortized constant time while storing and retrieving a field value in any version takes $O(\log n)$ time.

The management of the extra fields is the same as for the partial method, but with the difference that the natural order of integers is no longer sufficient to determine the order between versions (the versions in the version tree give only a partial order). To obtain a total order, a list L of pre-ordered versions in which each version is inserted just after its parent has to be maintained. Figure 2.4 (a) shows an example of such a version tree. Its corresponding version list is 1,2,6,3,5,4.

This list L is stored in an *order maintenance structure*, a structure that can answer if an element is before another element in $O(1)$ time. The time complexity of insertion of a new version in this structure can be bounded by $O(\log n)$ worst case and $O(1)$ amor-

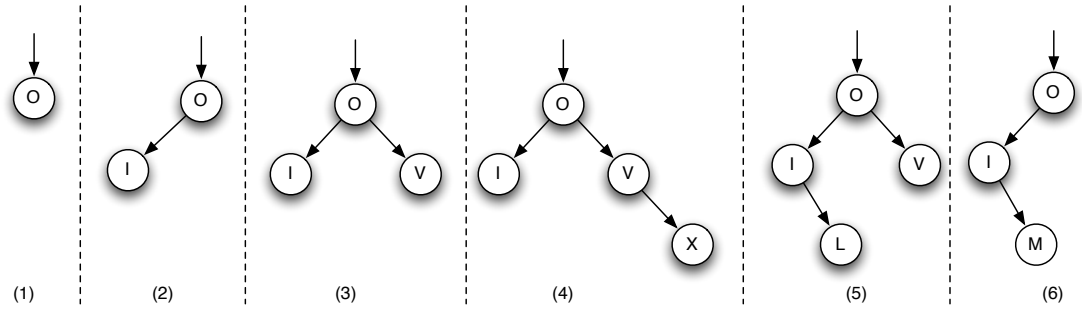


Figure 2.5: The different versions of the tree described in Figure 2.4.

tized. A more complicated solution offers constant query time and update time worst case [Dietz & Sleator, 1987].

Updates and queries of a field of a persistent node are performed similarly as in partial persistence, with two differences. First, the order between the versions is not the natural order but it is the order given by L. Second, the insertion of new versions in the version list and in a fat node can perform unwanted modifications on saved states of the data structure. Take the example of a version list 1,2,3 and a fully persistent node with a field f that has only one entry $\langle f, 1, a\rangle$. Suppose we branch the version 4 from the version 2 (the version list becomes 1,2,4,3) and add the entry $\langle f, 4, b\rangle$ in the node. When the value of f for the version 3 is asked, we look for the entry whose version number is the rightmost version number in L left of or at 3: the value b is returned, in place of a.

To solve this problem, Driscoll et al. propose the following modification of the update algorithm. Assume that the version i is already added to the version list as described before. Let $i+$ be the version after i in the version list, if a such version exists. When a field f of a fat node is updated with the value v during the version i, we add the entry $\langle f, i, v\rangle$ in the node. We denote by i_1 the first version before i and by i_2 the first after i amongst the entries of f (following the order defined by L), if such versions exists. If $i+ <_L i_2$ (or $i+$ exists but i_2 does not exist), we add an entry $\langle f, i+, v_1\rangle$, in which v_1 is the value of the entry $\langle f, i_1, v_1\rangle$. For instance, in the internal structure of the data structure of Figure 2.4 (b) shown on Figure 2.6, we must add the grey entries to maintain the consistency between the versions.

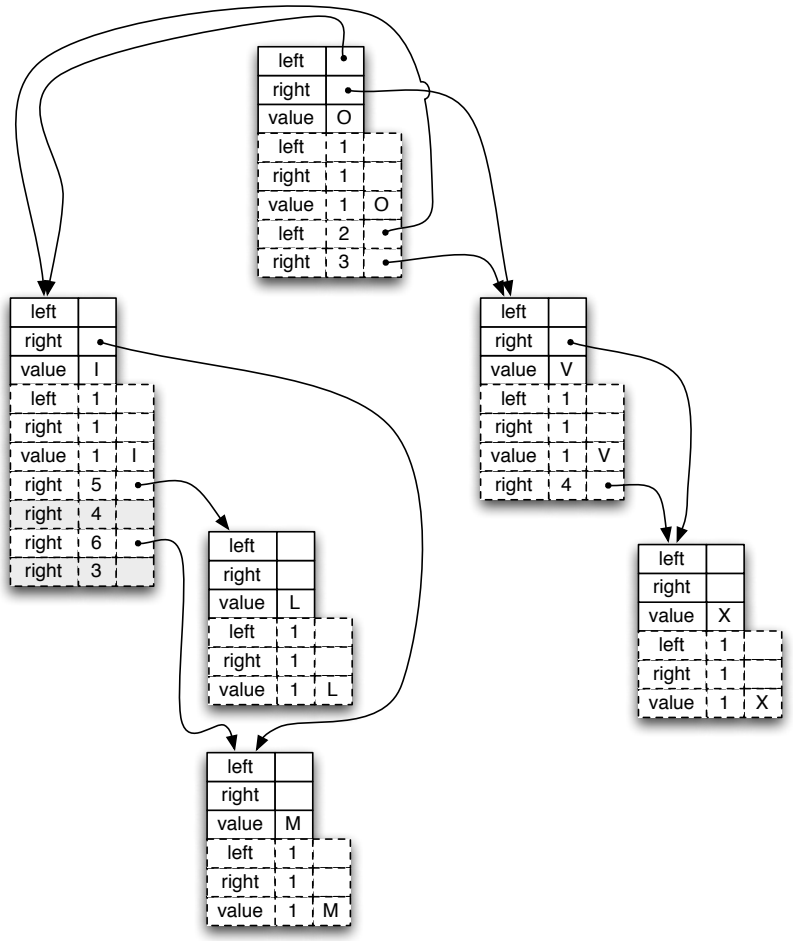


Figure 2.6: The internal structure of each fat node of our fully persistent binary search tree. The solid tables show the fields of ephemeral nodes and the dashed tables contain their extra fields (field name, version and value). The grey rows highlight the extra fields added by the system to maintain the consistence between the versions. We suppose that all nodes are created at version 1 and linked further. The null pointers are not shown for more readability.

2.4.5 Node Splitting Method

To adapt the copying method to full persistence, Driscoll et al. propose the **node splitting** method. As in the fully persistent version of the fat node method, the system maintains a global version tree in an order maintenance structure in which each version is added just after its parent. As in the copying method, each node keeps a given number of updates but once this number is reached, the node is split into two nodes containing each half of the updates. The idea is to split the nodes to obtain a tree of subnodes in which the search will be faster than in a list (as in the copying method) and where the subnodes are connected together (as in fractional cascading) to improve the access time during a sequence of operations on the same range of versions. As a result, the amortized complexities are the same as in the node copying method if the structure satisfies the same condition on the number of predecessors of every node.

2.4.6 Fully Persistent Arrays

In [Dietz, 1989] Dietz developed a fully persistent array supporting random access in $O(\log \log m)$ time, where m is the number of updates made to any version. This data structure simulates an arbitrary RAM data structure with a log-logarithmic slowdown. This slowdown is essentially optimal: fully persistent arrays have a lower bound of $\Omega(\log \log m)$ time per operation in the powerful cell-probe model [Demaine *et al.* , 2008].

2.5 Algorithms for Confluent Persistence[1]

Confluent persistence was defined by Driscoll et al. in 1994 [Driscoll *et al.* , 1994]. They also developed a data structure for confluently persistent catenable lists [Driscoll *et al.* , 1994]. In 2003, Fiat and Kaplan developed a general method to transform any pointer-based data structure into a confluently persistent one [Fiat & Kaplan, 2003]. But their technique suffers from a considerable slowdown. They also proved a lower bound on confluent persistence: some confluently persistent data structures require $\Omega(\log p)$ space per operation in the worst case, where p is the number of paths in the version DAG from the root version to the current version. Note that p can

[1] This section is inspired from the state of art in confluence persistence presented in [Demaine *et al.* , 2008].

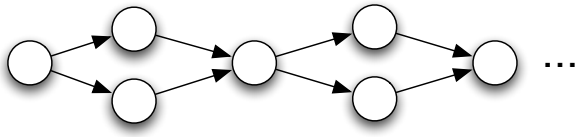


Figure 2.7: This version DAG has exponentially many paths from left to right, and can result in a structure with an exponential amount data.

be exponential in the number m of versions (Figure 2.7), resulting in a lower bound of $\Omega(m)$ space per operation in those cases. They show that no method can be significantly faster because the same node could be duplicated p times in the structure.

To remedy this, Collette et al. [Collette *et al.* , 2011] show how to transform any data structure in the pointer model into a confluent persistent one in which updates are performed in $O(\log n)$ amortized time and following a pointer takes in $O(\log c \log n)$ time where c is the in-degree of a node in the data structure. However they restrict most the structure to contain at most one copy of every node in a version. In particular, they prove that confluent persistence can be achieved at a logarithmic cost in the bounded in-degree model.

Solutions exist for specific structures, such as trees. It is possible to implement confluent persistence by designing a functional data structure, i.e. a data structure such that an update operation results in the creation of a pointer to a new version of the data structure. The most common technique for designing functional data structures is *path copying* [Okasaki, 1998, Sarnak & Tarjan, 1986]. This approach applies to any tree data structure where unmodified nodes can be shared between versions. When a node v is modified, the node v and all ancestors of v are copied. Because nodes depend only on their subtrees and the data structure becomes functional (read only), we can safely re-use all other nodes. The cost of an update in this model is d, where d is the depth of the modified node. This model is used by version control systems as SVN and many backup softwares. Figure 2.8 shows an example of path copying in a binary search tree.

In 2008, Demaine et al. [Demaine *et al.* , 2008] develop two data structures to maintain confluent persistent tries (rooted trees with labeled edges). Their first data structure maintains a confluent n-node degree-Δ trie with $O(1)$ "fingers" in each version while

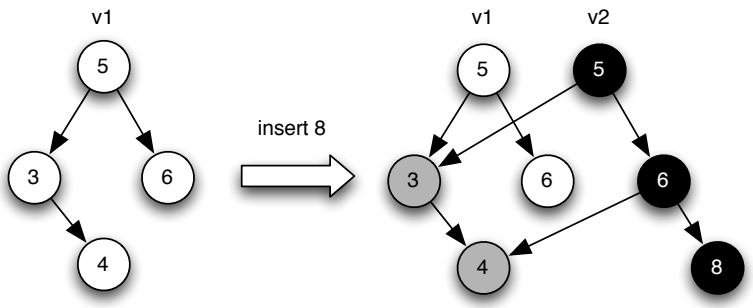


Figure 2.8: Path copying in a binary search tree. The old version (v1) consists of white and grey nodes; the new version (v2) consists of black and grey nodes.

supporting finger movement and modifications near fingers in $O(\log \Delta)$ time and space per operation. The second data structure, faster for deep updates (i.e. unbalanced tries), supports finger movement in $O(\log \Delta)$ time and space, while modifications take $O(\log n)$ time and space.

2.6 Choice of the Algorithm

To implement partial and full persistence efficiently in object-oriented languages we use the fat node method as a starting point. Although the fat method is not the best method for partial and full persistence in term of time complexities, we will use it for the following reasons.

First, we want to construct a general implementation for the partial and full persistence where the number of predecessors of a node is not fixed. Moreover we want also versioned arrays. The node copying and splitting methods are therefore not usable, due to the limit on the number of predecessors and working only in the pointer model.

In this case the efficiency of the fat node is the best we can hope for: the update time is bounded by $O(\log n)$ and the query time is logarithmic in number of versions saved in the node itself. Our implementation reduces the update time to $O(1)$ worst case for partial persistence. The method of Dietz et al. for persistent arrays could potentially improve the running time to $O(\log \log n)$ but its implementation could be considerably more complex.

Using our well-adapted data structure, the fat node method can be implemented effi-

ciently in practice, i.e. the number of operations to execute at each update or query can be really small.

2.7 Efficient Persistence for Specific Data Structures

The methods described above are applicable on any ephemeral structures, without knowing their specifics. To improve their efficiency, many studies show how to transform a specific ephemeral data structure into a persistent one. Some of the first work was done by Dobkin and Munro [Dobkin & Munro, 1980], that propose an efficient data structure to keep different versions of a list with the operations INSERT and DELETE. Knowledge of the exact data structure on which the persistence is applied allows the development of more efficient methods than the general ones. These structures are not used in this dissertation and we do not describe them.

However we present an interesting result. Tarjan and Westbrook [Westbrook & Tarjan, 1989] studied the disjoint set union-find problem with de-union operation. In this problem a data structure must keep a set of elements partitioned into given disjoint subsets. The find operation determines which subset a given element is in. The union operation combines two subsets into a single set. The de-union operation undoes the last performed union operation. The de-union operation is obviously close to the backtrack operation in versioning. Tarjan and Westbrook show algorithms that take $O(\log n / \log \log n)$ amortized time per operation, where n is the total number of elements in all the subsets. Moreover the authors prove a lower bound for the problem of $\Omega(\log n / \log \log n)$ and thus their upper bound on amortized time is tight.

Note that faster amortized union-find data structures exist, without this de-union operation, but uses nodes with arbitrary large in-degree. This suggests the restriction on the in-degree, even for partial persistence, is necessary.

2.8 Applications

The examples of usage of the persistence to solve algorithmic problems are numerous (inter alia [Aurenhammer & Schwarzkopf, 1991, Yellin, 1992, Eppstein, 1994, Goodrich & Tamassia, 1991, Acar *et al.* , 2004, Gupta *et al.* , 1994, Agarwal *et al.* , 2003,

Bern, 1988, Hershberger, 2006, Cheng & Ng, 1996, Mehlhorn *et al.* , 1994, Klein, 2005, Agarwal, 1992, Turek *et al.* , 1992, Koltun, 2001, Cabello *et al.* , 2002, Edelsbrunner *et al.* , 2004, Bose *et al.* , 2003, Bern *et al.* , 1990, Aronov *et al.* , 2006, Demaine *et al.* , 2004]). The first applications of persistence in the algorithm domain are the algorithm for planar point location by Sarnak and Tarjan [Sarnak & Tarjan, 1986] and the algorithm of Alstrup et al. [Alstrup *et al.* , 2001] for the binary dispatching problem. File versioning systems, such as SVN, are good examples of using the path copying method for confluent persistence.

2.8.1 Planar Point Location

Planar point location is a classical problem in computational geometry [Sarnak & Tarjan, 1986]: given a subdivision of the plane into polygonal regions (delimited by n segments), construct a data structure such that given a point, the region containing it can be reported quickly. Take the example of Figure 2.9. The plane is subdivided into 4 regions (A,B, C and D). A point p is placed arbitrarily on the plane. In our example, the region that contains p is B.

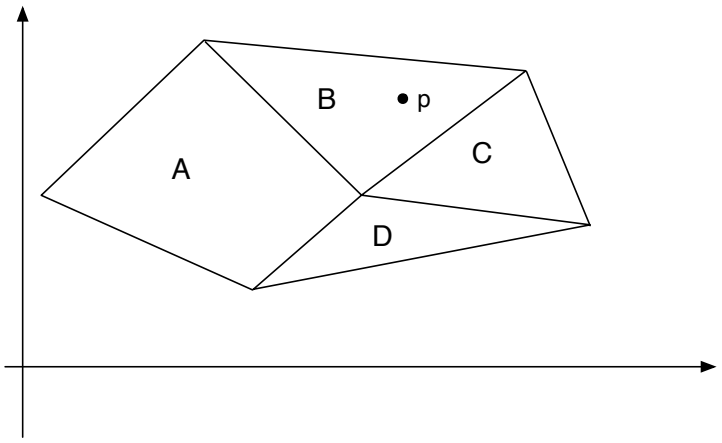


Figure 2.9: An example of planar point location instance

Dobkin and Lipton [Dobkin & Lipton, 1976] proposed a solution consisting of subdividing the plane into vertical slabs determined by vertical lines positioned at each vertex. The

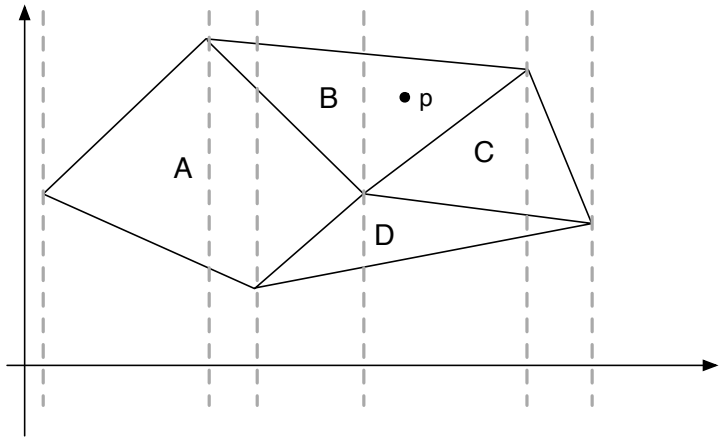


Figure 2.10: Division by vertical lines positioned at each vertex

division into vertical slabs of Figure 2.9 is shown on Figure 2.10. Within each slab, there exists a total order between line segments determined by the order in which any vertical line in the slab intersects them. Each segment is associated with the polygon just above it, and a balanced binary search tree storing the segments is constructed for each slab.

When a point is queried, its x-coordinate is used to determine which slab contains it in $O(\log n)$ time, and the binary search tree of the corresponding slab is used to locate the region containing the point, also in $O(\log n)$ time. Unfortunately, the worst-case space requirement for this structure is $\Theta(n^2)$. To solve this problem, Sarnak and Tarjan [Sarnak & Tarjan, 1986] use persistence in order to reduce the space to $O(n)$. A vertical line sweeps the plane from $x = -\infty$ to $x = +\infty$, maintaining at every point the vertical order of the segment 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.

Figure 2.11 shows the six versions of the segments tree corresponding to each vertical line. Each small graph shows the segments present in the corresponding version. When the polygon containing the point p is queried, we look for the right-most vertical line l left of p (according to the x-coordinate). In our example, this is the fourth line. There is only one version v of the segments tree that corresponds to l (the fourth version in our example). We perform a binary search in the segment tree at version v to find the segment just above

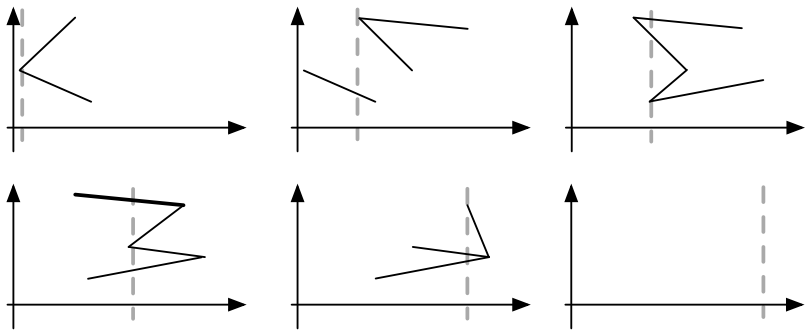


Figure 2.11: Each version of search tree for the example of Figure 2.10

the point p, using the y-coordinate of p. This segment is highlighted in the fourth version of Figure 2.11. The region B is identified as the region containing p.

2.8.2 Binary dispatching problem

In class-based object-oriented programming languages there is a hierarchy of classes. We assume that this hierarchy of classes defines also the hierarchy of types. The methods can be overloaded, i.e. a method may have different implementations for different types of its arguments. At run time when a method is invoked, the most specific implementation which is appropriate for the arguments has to be selected. The most specific implementation of a method named m with arguments $a_1, a_2, \ldots, a_n$ of types $t_1, t_2, \ldots, t_n$ is the method $m(t'_1, t'_2, \ldots, t'_n)$ such that each t'_i is t_i or the a subtype of t_i and there is no other method $m(t''_1, t''_2, \ldots, t''_n)$ such that t''_i is t'_i or a subtype of t'_i for all i.[2] The dispatching problem is to find for each invocation the most specific applicable method. In the binary dispatching problem, all implementations and invocations have two arguments.

Alstrup et al. describe a data structure for the binary dispatching problem that use $O(m)$ space, $O(m(\log\log m)^2)$ preprocessing time and $O(\log m)$ query time (where m is the number of method implementations), increasing considerably the efficiency by using full persistence.

[2] There are some ambiguous cases where the ambiguity is reported to the user.

2.8.3 File Versioning Systems

File versioning systems, such as Subversion (SVN) [Pilato *et al.* , 2008], are good examples for confluent persistence. In a such system, the user has a root directory in which files can be added, updated and deleted. All versions are kept on a server and the clients connect with it to retrieve files, create branches and merge local and distant versions. There are typically three important commands: checkout, commit and update.

Checkout copies a version of the root directory on the client computer.

Commit sends the current version of the client to the server, that puts this version the child of the current client version.

Update merges the local version with a distant version: the tool determines the difference between files and attempts to automatically merge the two versions. If automatic merging is not possible (typically when the same line of the same file was updated on the client and on the server), the update operation fails and the system asks the user for help.

Confluent persistence is required when two versions of two distinct branches are merged (by update and commit). Some conflicts can appear during the merging. Most file versioning systems try to resolve them and return the non automatically resolvable conflicts to the user. Confluent operations are using operations, e.g. copy a directory in some version into some other directory in the current version.

Note SVN is partially persistent but confluent while more modern systems such as mercurial allow several "heads" (latest versions) and so are totally persistent and confluent.

2.9 Conclusion

In this chapter we described existing methods to transform an ephemeral structure into a persistent one. We have discussed methods for partial persistence (where only the last version is editable), full persistence (new versions can be created from any existing version) and confluent persistence (creation of versions can be done from any version but also from several versions). For each method we show the complexity for the query time, the update time and the space used. Because the rest of this dissertation is focused on partial and full persistence, only those two methods were discussed in detail.

Chapter 4 will describe specific data structures and algorithms to implement efficiently persistent objects in practice.

Chapter 3

Object Versioning Model

In the previous chapter we presented the existing methods to transform any ephemeral data structure in a persistent one. In this chapter we present a model of versioning for object-oriented languages. In the next chapter we will see how this model can be implemented efficiently in object-oriented languages using the methods seen in the previous chapter. For the reader non familiar with object concepts we start this chapter by an introduction to the different terms of the object world used in this dissertation.

3.1 Object-Oriented Paradigm

In this section we explain the concepts of the object-oriented (OO) paradigm, used throughout this dissertation.

Object An object is the cornerstone of the OO paradigm. An object represents an entity of an OO application. It has an identity, fields and methods. The fields are memory cases in which other objects[1] can be stored. The fields of an object contain the data of this object. The methods of an object define its behavior that can accesses the value of fields, call other methods, invoke procedures, etc.

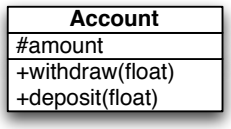
State We define a state of a field at time t as the value of this field at t. By extension the state of an object at time t is defined as the set of field states at t. The version of the system at time t is defined as the state of all objects at t.

Class A class defines a set of fields and methods shared by all objects created by it. An object created by a class is called an *instance* of this class. A class can be seen as an

[1] Or primitive values (integers, strings, etc.) when such distinction is made

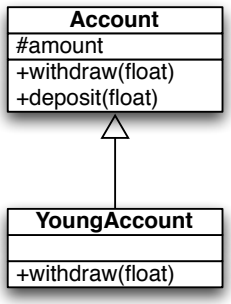
object factory: one role of a class is to create new instances and to define their common behavior.

For example, a class `Account` defines one field (`amount`) and two methods (`withdraw` and `deposit`). My account is an instance of the class `Account` with its own value for the field (e.g. `amount` = 1000000[2]). This class will be represented in UML (Unified Modeling Language [Rumbaugh *et al.* , 1999]) as follows:



Inheritence Most class-based OO languages offer a construct to allow the inheritance of a class by another class (or more in some languages) to add, share, reuse or specialize some methods. The class that inherits from another is called the *subclass* and it gets the defined fields and methods of the inherited class (named the *superclass*).

For example, we have a class `YoungAccount` that inherits from `Account` in which we redefine the method `withdraw` to avoid a negative balance. The instances of the class `YoungAccount` will have a field `amount` and a method `deposit`, inherited from the superclass `Account`, and a redefined method `withdraw`. Both classes and their relation of inheritance will be represented in UML as follows:



Message The objects collaborate using *message passing*. An object, called the *sender*, sends a message (with arguments if necessary) to another object, called the *receiver*. Method

[2]Please... Let me dream about it...

lookup is the mechanism used to find which methods of the receiver must be executed by analyzing the sent message.

Visibility of fields The visibility of a field of an object is either *public*, *private* or *protected*. When the visibility is *public*, any object can access freely the field value. When it is *private*, only the instances of the same class can access the field value. When the visibility is *protected*, only instances of the same class or of a subclass can access it. In UML, a public field is preceded by a symbol $+$, a private field by $-$ and a protected field by $\#$.

Visibility of methods The visibility of the methods of an object allows one to define which entities of an object-oriented application can call a given method of an object. A method is *public* when any entity can call this method. A *protected* method can only be called by objects of the same class or of a subclass of the receiver. A *private* method can only be called by objects of the same class than the receiver. In UML, the symbols for visibility of methods are the same than for the fields ($+$, $-$ and $\#$).

Explicit and Implicit Visibility The visibility of a field or a method is either explicit, i.e. the developer defines the visibility, or implicit, i.e. the developer declares the field or the method without specifying its visibility. In the latter case, the language has the responsibility to define the visibility. For example, in Smalltalk, all fields are private and all methods are public. Moreover, the developer has no possibility to assign another visibility. On the other hand, in Java, the developer must define the visibility of each field and each method. In C++, the developer can define the visibility for a field or a method but there is no obligation to do so: by default, fields and methods are private.

Encapsulation An object can *encapsulate* some field values by hiding them from other objects. A field f of an object o is encapsulated in o if an other object has no way to access the value of f. For example, an instance of the Smalltalk class `OrderedCollection` encapsulates an array to store its different values. There is no way to access the array outside of the ordered collection.

This important principle in object-oriented paradigm asserts that the encapsulated fields of an object will be not managed by another one, giving the responsibility of the fields maintenance to the object itself (throughout its methods).

Types Each variable of an object-oriented application has an associated type. Depending on

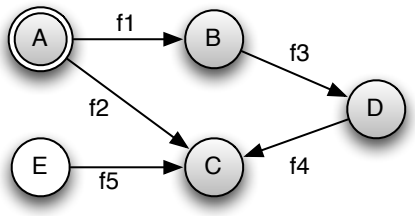


Figure 3.1: A graph of five objects. The connection graph of the object A is composed of the grey objects: A, B, C and D. E is not accessible from A and it is not in the reachable set.

the language the types can be created explicitly by the developer or implicitly by the system (e.g., where there is a bijection between types and defined classes). In a *statically typed* language, the developer must define explicitly the types for each variable and the system checks if types are compatible for each operation during compile-time and/or runtime. C++ and Java are examples of statically typed languages. On the other hand, the developer must not declare the type of each variable in a *dynamically typed* language. The majority of type checking, if any, is performed at run-time. Smalltalk and Python are examples of dynamically typed languages.

Fixed and Variable Sized Objects An object is a *fixed sized* object when it has a fixed number of fields. When it has a variable number of fields, determined at runtime, it is called a *variable sized* object. The arrays are a good example of variable sized objects: when an array is created with a specified size s, an array is created with s entries.

Object graph All objects form a graph: each object has fields, each one pointing to an object[3]. The node of the graph are the objects and a directed edge labeled by f from an object o_1 to o_2 indicates that the field f of o_1 points to o_2.

The reachable set of the object o is the set of objects including the object o and all objects pointed directly or transitively by o. The reachable set of o includes any object k such that there exists a path between o and k.

Take the example of Figure 3.1. The objects are represented as circles and their fields are the directed edges. The label of an edge is the name of the field. The reachable set

[3]In the rest of this chapter, we consider there is no primitive value but only objects, without lack of generality. When a field of an object contains nothing, the field points the special object `null`.

of the object A is composed of the grey objects: B and C are directly linked to A and D is linked to B. The object E is not in the reachable set because there is no path between A and E that follows the directed edges.

The connection depth of an object o_s in the reachable set of o_r is the number of edges on the shortest path that starts at o_r and goes until o_s. The connection depth of an object in its own reachable set is 0. The connection depth of an object that does not belong to a reachable set is positive infinity ($+\infty$).

The connection depths for the example of Figure 3.1 in the reachable set of A are 0 for A, 1 for B, 1 for C, 2 for D and $+\infty$ for E.

Simple and Complex Object A *complex object* o is an object linked with other objects by structural or existential dependences [Oussalah & Urtado, 1997], i.e. a subset of the reachable set of o is semantically connected to o. For example, an instance of the Smalltalk class `OrderedCollection` is a complex object: it manages an other object (an array) in which the different values are put. The management of the array is the responsibility of the ordered collection.

When an object is not complex, it is a simple object. For example, a Smalltalk array is a simple object. Other objects (as ordered collections) use it to store data and the array does not manage itself the objects it contains.

3.2 Model requirements

The main goal of our work is to provide an efficient framework that supports object versioning in memory. The first steps of its development shows us two important things. First, although we will use efficient algorithms and data structures, there is a cost in both time and space because saving different states of a field will always be greater than directly putting a value in a field. Also, keeping all states of all fields is often not necessary: keeping only states of interesting fields at interesting time is sufficient for most applications. Both observations lead us to think that it would be interesting to only keep useful states and to not lose time and space to save unused information.

Therefore we need a model to record interesting parts of the past of an application and to browse this past. More precisely the model should address the following requirements.

The model must be general enough to be used in any kind of object-oriented application. The model can not be defined under constraints imposed by application structures or domains.

The model must be fine-grained enough to select exactly the interesting parts of the past of the application. Having a more fined-grained model will allow the developer to describe with more precision the selectable entities, resulting in a system that saves states more efficiently in time and space.

The selected parts can be unselected to stop the collection of states and to not lose time and space with useless data.

The model must be defined for linear, backtracking and branching versioning. A unique model for the three kinds of versioning unifies shared concepts in one place.

The model must offer an automatic way to select the interesting parts to facilitate the selection of parts. It must follow the next principles:

1. the automatic selection must be **expressive** enough to select automatically only the needed objects.

2. the automatic selection must not violate the **object-oriented principles**: the encapsulation of the definition of the automatic selection for each object must be respected. The introduction of the model in object-oriented languages can not violate their principles.

The model must offer a unified way to browse the saved states. In a model in which only some parts of the system have some values for the past, the system must behave coherently while browsing the past of selected and non selected parts of the system.

To the best of our knowledge, no prior system offers an expressive and fine-grained control of the object versioning that respects these requirements. A review of the state of the art is given in Section 3.10.

3.3 Object Versioning Model

This chapter defines our first contribution: an expressive and fine-grained model for object versioning. This model defines how to record states of interesting parts of an object-oriented application at interesting times and browse them in an object-oriented system, including support for fixed and variable sized objects. This model is defined for linear, backtracking and branching versioning. Finally this model focuses on the selection of interesting parts to record and it is independent of the manner to efficiently record states (e.g. using files, databases or only in memory).

The model can be subdivided into three actions:

1. **Recording**. The first action is recording the changes of the system. We decompose this action in two crosscutting actions.
   (a) *Fine-grained selection of what to save*. To be the most expressive as possible, we fix the level of granularity of what to save from a field of an object. In this model, 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).
   (b) *Fine-grained selection of when to save*. Each modification of a field of an object generates a new state of this object. This model provides a simple mechanism to select at which time states must be kept.
2. **Browsing**. The second action is browsing the previously recorded states. We also define a number of rules to make cohabitation of ephemeral and versioned objects possible (Section 3.5).
3. **Manipulate the time line** The user can also control the time line to forget a part of the history (in backtracking versioning) or create new branches from old versions (in branching versioning).

In the next sections, we will define our model in a global way: all common concepts shared by the three kinds of versioning are explained in the two following sections. The particularities linked with each type of versioning are explained in Section 3.6. The general view of the model is shown in Figure 3.2. Each part of this picture will be described in the next sections.

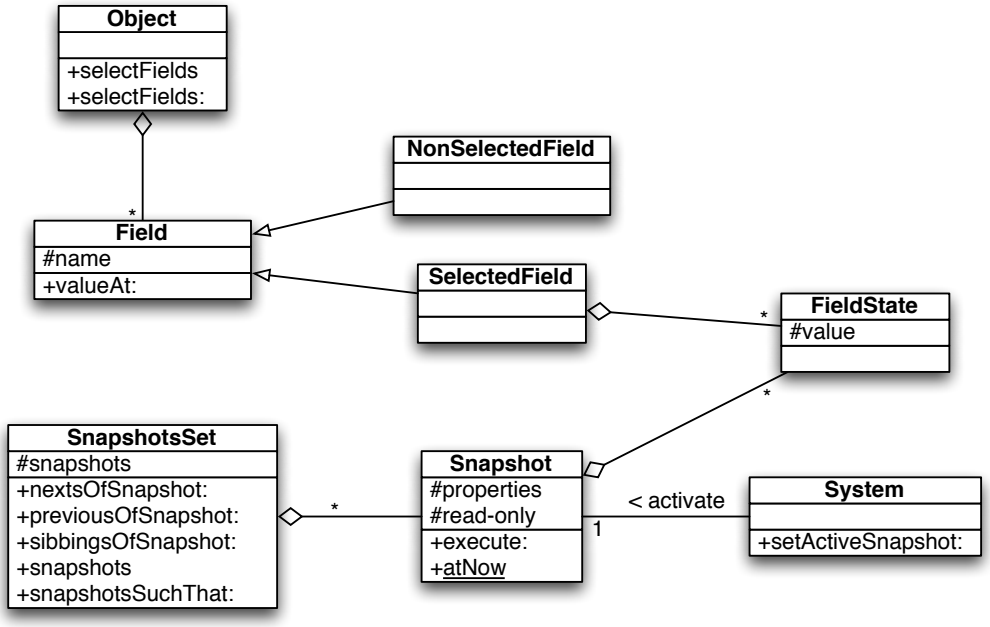


Figure 3.2: General Model

The model focuses on the state on an object. An object is seen as a set of fields. A field can be *selected* or not (Section 3.4.1). A *non selected field* is *ephemeral*, i.e. only its last value is accessible. On the other hand, a *selected field* (named also *versioned field*) is a field for which different states will be saved, i.e. different objects will be assigned to it. A selected field has one or more states, each of them storing a value. A *snapshot*, representing a version of the system, contains a value for the previously selected fields (Section 3.4.2). Snapshots are taken when it is necessary and the past can only be browsed using snapshots (Section 3.5). We introduce finally the snapshot set (Section 3.7) to manage different sets of snapshots.

We conclude this chapter by a discussion of the proposed model. We motivate our choices. We describe its advantages and its disadvantages. We compare it with other existing models.

3.4 Recording Model

The main idea is simple: save only interesting parts of the system, i.e. states of interesting fields of interesting objects at interesting times. The challenge is to provide a model that allows one to specify these parts easily. We decompose this challenge into two: determine which fields of which objects must be considered as interesting and determine at which times versions must be recorded. In this section, we define a model combining the fine-grained selection of fields and the fine-grained selection of states.

3.4.1 Selection of Fields

The model gives full control over *what* gets saved. The level of granularity of what to save is a field of an object: the smallest element that can become versioned is a single field of a single object.

At the start of the system, all fields of all objects are ephemeral, i.e. only the last value of the fields is available. When a new value is put in a field, the old value is simply overridden. At any time, a field can be *selected*, expressing the wish to save its different values from the time of selection until the field is unselected. When a field is selected, it is no longer ephemeral and its next values will be saved. The selection of a field has no impact on other fields: other ephemeral fields remain ephemeral and only their last value is available. We stress that selecting a field does not retrieve states of this field created before the selection.

To illustrate the selection of fields, Figure 3.3 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 states S1 and S6. The book is badly titled ("1985") at state S2, its state is clean (S3) and a client is created with an id equals to 1 (S5) and named "Jon" (S6). This client is set to be the current borrower (S7). A few days later the client comes back to the library to return the book (S8) in a dirty state (S9). He also mentions that his name is "John" instead of "Jon" (S10). Three days later the client comes back to borrow the same book again (S11). During the loan registration, the library employee corrects the title of the book (S12).

If a field is not selected, only its last value is available. In our example, we select only the two fields `state` and `borrower` of the book. We think it is not necessary, in the context

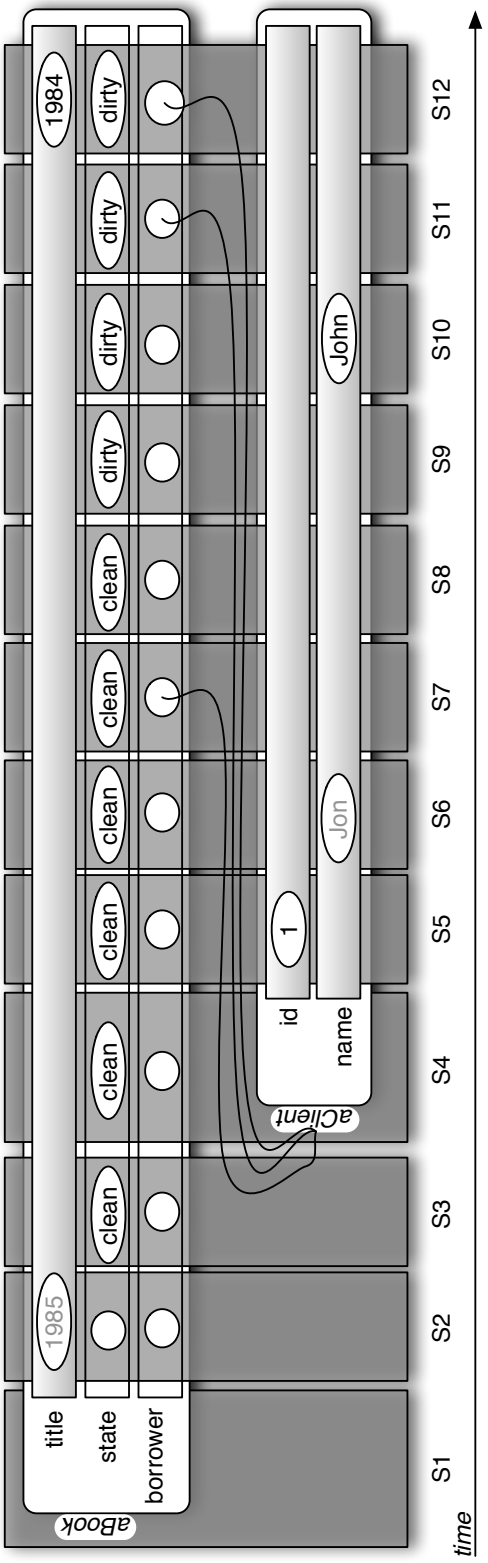


Figure 3.3: Detailled Object Model Example

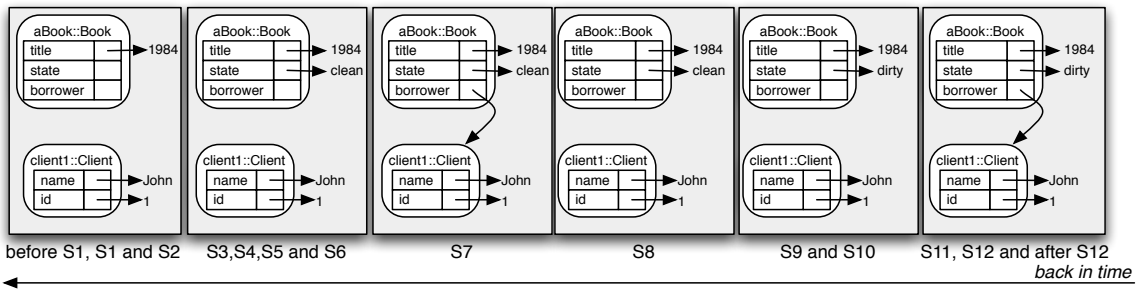


Figure 3.4: Detailled Object Model Example 2

of the application, to keep the different values of `title` of the book and of `id` and `name` of the client. This is a development choice. Every field could be selected but we do not care about the old values of the other fields. After the creation of the book at S1, we select only both fields of the book and we execute each step of the program. The different values of both fields will be saved (in this example all values are saved but we will see in Section 3.4.2 how to control exactly at which time states must be saved for selected fields). For each non selected field only its last value is accessible: the old values "1985" for the title and the name "Jon" for the name of the client are lost.

Figure 3.4 shows the different versions of the system *seen from S12*. The `title` of the book and the `name` and the `id` of client never change throughout the versions: the fields are not selected, only their last value is available. On the other hand the `state` and the `borrower` of the book were selected just after the creation of the book and their old values are all available. The browsing model is defined in Section 3.5.

It is possible to stop the collection of the states of a selected field in three ways:

- A selected field can be *paused*. Pausing a field means that the next values of this field will not be saved while old states are still available. No additional state will be created. A paused field can be de-paused by a re-selection and it retrieves its state "selected".
- A selected field can be *deselected*. The next values of a deselected field will not be saved. Its old states are kept but are not available while the field is deselected. A deselected field can be re-selected retrieving the kept states before the deselection. The difference between a paused field and a deselected field is visible while browsing old versions (Section 3.5).

- Finally a selected field can be transformed into an *ephemeral* field again: its old states are deleted and there is no way to retrieve them.

Pause and deselect a selected field allow one to stop recording the state of this field and so a finer selection of the states to keep.

Take the example of a chat application where conversations are versioned. If the user want to be in a off-the-record (OTR) conversation, i.e. in which the conversation contents are not versioned, the selection of the conversations must be paused. The next snapshots will not save the new states for this conversation.

The deselection of a field is interesting when you must browse the past of a part of selected fields but remain in a writing mode for some other ones. Take the example of an array of strings and a console window on which we can print the different states of this array. We select the array and the console window to get all of their own states. We change some values of the array, we display these changes on the console and we take snapshot of each change. When we browse the past using the different taken snapshots, we can follow the different modifications done on the array and on the console. But if we browse the system throughout a given snapshot and we ask to display the state of the array on the console, the system denies this operation: the console is also in a past state and can not be modified. To use the console as an ephemeral one throughout a snapshot, we can deselect it: the previous states of the console will be hidden throughout the snapshots and the console can be used as an ephemeral one.

We conclude this section by summarizing the different selection states of a field and the possible transitions between them (Figure 3.5). A field can either be ephemeral (old states are unavailable), selected (old states are available), paused (states before the pause are available) or deselected (old states are kept in memory but there are not available for the user). An ephemeral field can be selected but it can neither be directly paused nor deselected. All other transitions between two states are accepted. The consequences of transitions on the old states will be described in the next sections.

3.4.2 Selection of States

The developer has full control of *what* is saved thanks to the selection of fields. The developer has also full control of *when* states of fields are saved. This section yet describes

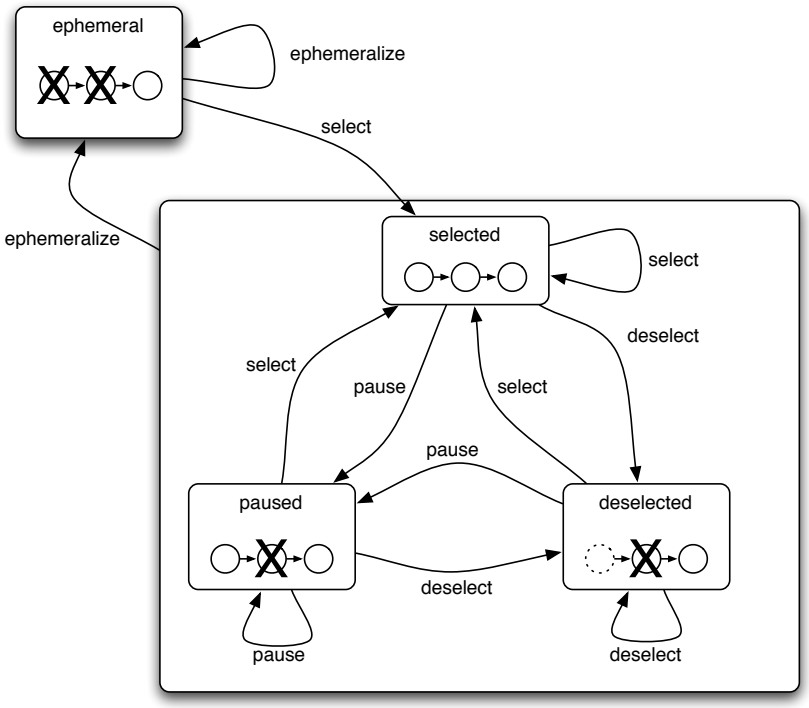


Figure 3.5: The different selection states of field and their possible transitions.

concepts shared by all kinds of versioning and a specification for each kind of versioning will be made in Section 3.6.

3.4.2.1 Snapshots

As explained in the previous section, the selection of fields is the first action offered by our model. A field contains exactly one value at any time. When a new value is assigned, its old value is replaced by the new one. As for an ephemeral field, the value contained in a selected field is always its last assigned value. Its old states will be stored in snapshots.

The second action is taking snapshots. A snapshot will contain the values of previously selected fields. This is analogous to using a camera. Whenever the user presses a button, a snapshot is taken, remembering what was visible at that time. 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), many applications that need object versioning are better served by explicitly taking snapshots than by capturing a huge stream of changes.

We 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 states 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.

A snapshot can be seen as a dictionary object (hashmap) in which each field is associated with exactly one value. For a given field, a snapshot returns the corresponding value. A snapshot can be *writable* or *read-only*. A writable snapshot allows one to erase the value associated with a field by a new value and to add new fields. A read-only snapshot does not allow any modification.

There is only one *active* snapshot at one time. This snapshot can be writable or read-only. When the active snapshot is writable, fields can be selected and the values of the selected fields can be modified. If this is a read-only one, no selection of fields and no modification

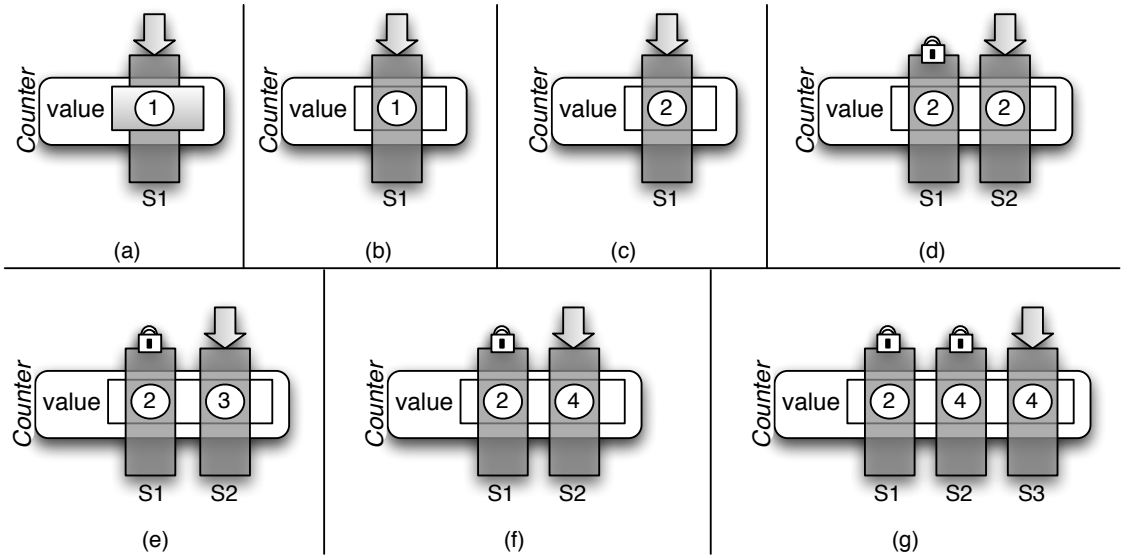


Figure 3.6: Example of usage of snapshots

of the selected fields are allowed. At the start of the system, the active snapshot is set to a writable snapshot.

When a field is selected, a new entry that associates the field with its value is added to the active snapshot. When a new value is assigned to a selected field, the value stored in the field and the entry for this field in the snapshot are updated by the new value. The values of the non selected fields are not saved in the active snapshot: they are still ephemeral and only their last value is accessible. A new value put in a non selected field replaces the previous one.

The successive different values of all selected fields are put in the active snapshot: each new value erases the previous one put in this snapshot. When the current values of the selected fields must be remembered, a new snapshot must be taken.

A new snapshot can be created only if the active snapshot is writable. A new snapshot is created as follows. The active snapshot becomes read-only and the newly created snapshot becomes the active one. This new snapshot is initialized with a copy of the values of the previous active snapshot. A link is made between the new active snapshot and the previous one: the previous active snapshot is called the **direct predecessor** of the new one while the new one is a **direct successor** of the previous active snapshot. Once a snapshot is set

read-only, it will never be writable.

Figure 3.6 shows an example of the usage of snapshots. We follow 7 steps of the evolution of the object `Counter`, that has one field `value`. This value will be incremented and we will save the state of this field only when its value is even. In each step of the figure the active snapshot is pointed to by the arrow. At the first step, the active snapshot is the single snapshot S1 and the value of the counter is initialized to 1. At step (b) the field `value` of the counter is selected. Its current value is saved in S1. Because its value is not even, we do not take a new snapshot. The value is incremented. Because a new value is put in a selected field, this new value will be saved in the active snapshot, erasing the previous saved value 1 for this field (step (c)). Now the current value of the field is even. We want to keep this state to browse it further: we take a new snapshot S2 (step (d)). S1 becomes read-only (note the lock above the snapshot) and a new writable snapshot S2 is created with a copy of the value of the selected field. S2 becomes the active snapshot. At step (e) we increment the value. The new value 3 erases the previous one in the active snapshot S2. Note that the value saved in S1 remains untouched by the new modification. The new value is odd and we do not keep this state. We increment the value (step (f)). The new value 4 erases the previous one in the active snapshot. This new value is even, we take a new snapshot S3 (step (g)). S2 becomes read-only and S3 becomes the active snapshot with a copy of the value saved in S2. The interesting states of the field `value` are saved by taking snapshots only when the current value of the field was interesting.

3.4.2.2 Stop Collection of States

As explained in Section 3.4.1 collecting states of a selected field can be stopped in three ways: make the field ephemeral again (all saved states are deleted), deselect the field (the field is seen as an ephemeral field but its old states are kept) or pause the field (the saved states are kept but no more states will be saved). In this section we explain the impact of these three transitions on the snapshots.

Ephemeralizing When a selected field is transformed into an ephemeral field again, all entries for this field in available snapshots are deleted. The field is henceforth ephemeral and there is no way to retrieve the deleted states.

Deselecting When a selected field is deselected, it retrieves the same behavior as an ephemeral field (its new values are not saved in the active snapshot and the read of its value returns always the last assigned value) while its previous values in the snapshots remain untouched.

Pausing When a selected field is paused, no more state will be saved while its old states are still accessible.

To achieve deselection and pausing, only the value stored in the field will be updated: the value saved in the active snapshot for this field remains untouched. When a new snapshot is taken, the field values stored in the previous active snapshot are copied in the new active snapshot as explained before. This copy includes also the non updated value of the paused/deselected field while the new value is in the field itself.

As an ephemeral field, the value of the deselected field can be updated when the active snapshot is writable or read-only without restriction. As a selected field, the value of the paused field can be updated only when the active snapshot is writable. Changing the value of a paused field in a read-only active snapshot throws an error.

When the deselected or paused field is selected once again, the value contained in the field is stored in the active snapshot as during the selection of an ephemeral field.

Figure 3.7 shows an example of deselection and selection once again. The first step shows the states of the counter at the last step of the previous example. There are two horizontal lines. The line at the top shows the different values saved in snapshots, as in Figure 3.6. The line at bottom shows the value stored in the field itself. This second line has been hidden in Figure 3.6 because the value contained in a selected field is always the same one than in the last snapshot. But when a field is deselected, both values (in the field and in the snapshot) could be different.

We deselect the selected field `value` (step (a)). The values in the field and in the active snapshot are the same one. At step (b) we increment the value. Only the value in the field is updated. The stored value in the snapshot remains untouched. We take a new snapshot S4 (step (c)). The value 4 is copied in the new snapshot while the value stored in the field is always 5. At step (d) we increment the value. The field that is deselected has value of 6 in itself but the value stored in the active snapshot remains unchanged. Finally

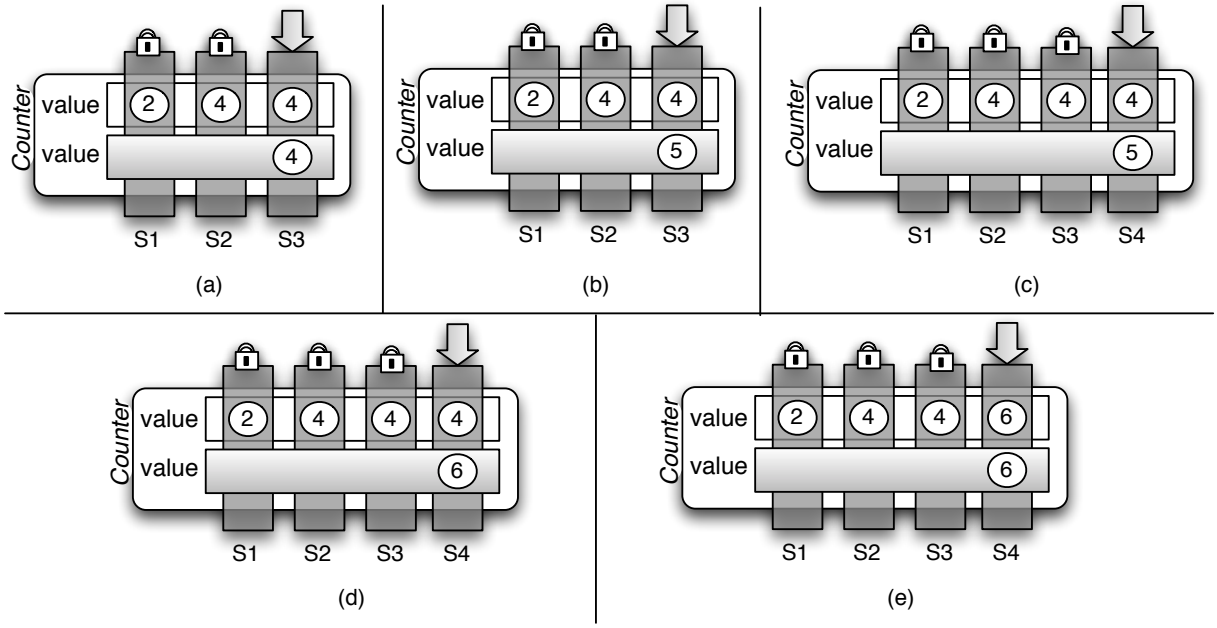


Figure 3.7: Example of deselection and selection once again

we select the field once again (step (e)). The current value of the field is stored in the active snapshot. The old stored value is replaced by the new one. The updates of the value between the deselection and the re-selection is no more accessible: the value 5 is not stored in the snapshot S3 while the value 4 is stored in this snapshot.

This example would be identical when pausing a field. The last value is kept in the field and not the active snapshot. When a snapshot is taken, the value is copied from the active snapshot to the new one, including the non updated value of the paused field. While pausing and deselecting fields have the same effect when recording states, the difference those two operations will be significant in the browsing model (Section 3.5).

Transforming a selected field into an ephemeral one again, pausing it or deselecting it stops the collection of the states. The re-selection of the field restarts it.

3.4.2.3 Snapshots Are More Than Snapshots

We saw in the previous sections how snapshots constitute a powerful mechanism to save the interesting parts of the system. But snapshots are more than simple dictionaries of

values of fields: they also have properties. The properties of a snapshot make it possible to attach information to a given snapshot. For example, the system does not save the creation time of the snapshot, but the user can add the creation time in the properties of each snapshot and retrieve it later on. The user can also attach other relevant information. For example in a text editor, when a snapshot is taken, a description of the saved operation ("key pressed", "copy/paste", etc.) can be added in the properties of the snapshot.

As we will see the next section, snapshots are central to our model: they make it possible to select the states of selected fields at desired times but they will also be used to perform timeline operations such as browsing the past, backtracking versions (in backtracking versioning) and creating new branches from a snapshot (in branching versioning).

3.5 Browsing Model

Selecting fields and taking snapshots are the mechanisms used to save the states of the interesting fields at interesting times. In this section we describe how to browse the saved states.

As explained, taking a snapshot saves the current value of selected fields. Each saved state corresponds to one snapshot and a snapshot contains a value per selected field. Snapshots are the doors to the past.

To browse recorded values stored in a snapshot s, s can be activated, i.e. the active snapshot becomes inactive and snapshot s becomes the active one.

When the value of a field is read, the selection of the field is considered:

- If the field is ephemeral, its value is directly returned.
- If the field is deselected, the value contained in this field is returned independently of the active snapshot.
- If the field is selected, the couples (field, value) saved in the active snapshot S_a are considered. If an entry exists for the field, the corresponding value is returned. If there is no entry for this selected field, i.e. the field has been selected during the activation of a snapshot taken after S_a, an error is thrown.

- If the field is paused, the value to return depends on the active snapshot. If the active snapshot is writable, the value contained in the field is returned. If the active snapshot is read-only the value is looked up though as for a selected field: if the active snapshot has an entry for the field, the corresponding value is returned; otherwise an error is thrown.

Figure 3.8 shows a representative example of browsing states. At the start of the system (step (a)), we have 4 counters. We select the field of counters 2, 3 and 4. We put respectively 1, 2, 3, 4 in the field of each counter. We take a new snapshot S2. We put 5 in the counter 1, 6 in the counter 3 and 7 in the counter 4. We create a new counter 5, we select it and we put 8 in its field. We take a new snapshot S3. We put 9 in counter 1 and 10 in counter 2. We deselect the counter 3 (depicted with two lines for counter 3) and we put the value 11 in the field of this counter. We pause the counter 4 and we put the value 12 in this counter. Finally we put the value 13 in the counter 5. We have 5 counters with different selections of their field: ephemeral, selected at S1, selected at S1 with a deselection at S3, selected at S1 with a pausing at S3 and selected at S2.

In (b), we want to browse the system at the snapshot S1. Therefore the active snapshot becomes the snapshot S1 (the arrow points the snapshot). We get the value associated with the field of each counter. The first counter has an ephemeral field. Only its last value is available: 9 is returned. The second counter has a selected field and there is a value associated with this field in the snapshot S1 (the value 2): 2 is returned. The third counter has a deselected field. The value saved in the snapshot S1 is ignored and the value in the field itself must be considered: the value 11 is returned. The fourth counter has a paused field: the active snapshot is considered. The active snapshot is read-only: the value saved for this field in S1 is returned (the value 4). The last counter has a selected field but no associated value in the snapshot S1. When the value is read, an error is thrown.

Snapshots are the doors to the past. By activating a read-only snapshot, the read value of a selected field returns the value saved during the writable activation of this snapshot. Ephemeral fields always return their last value.

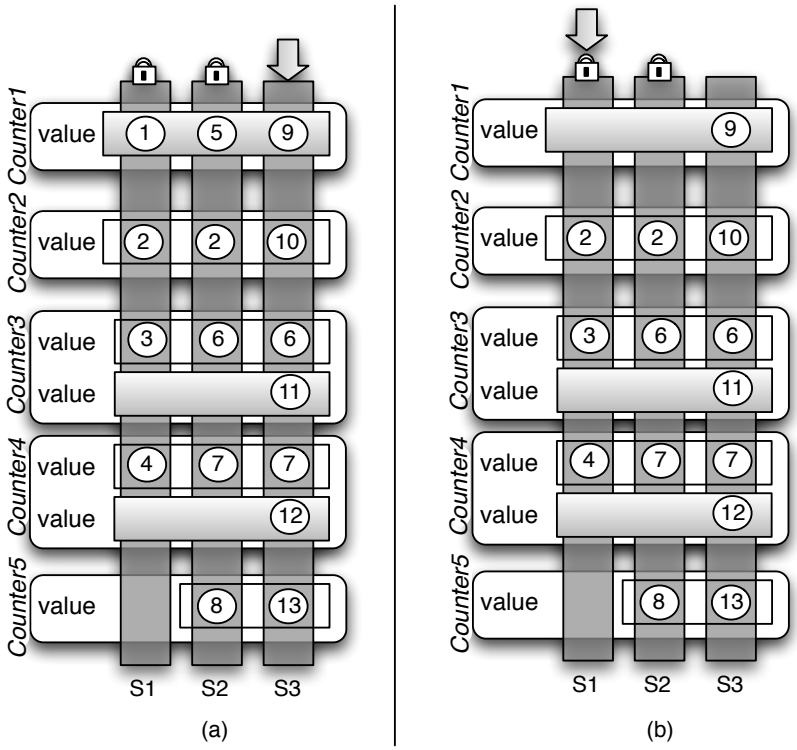


Figure 3.8: Example of browsing

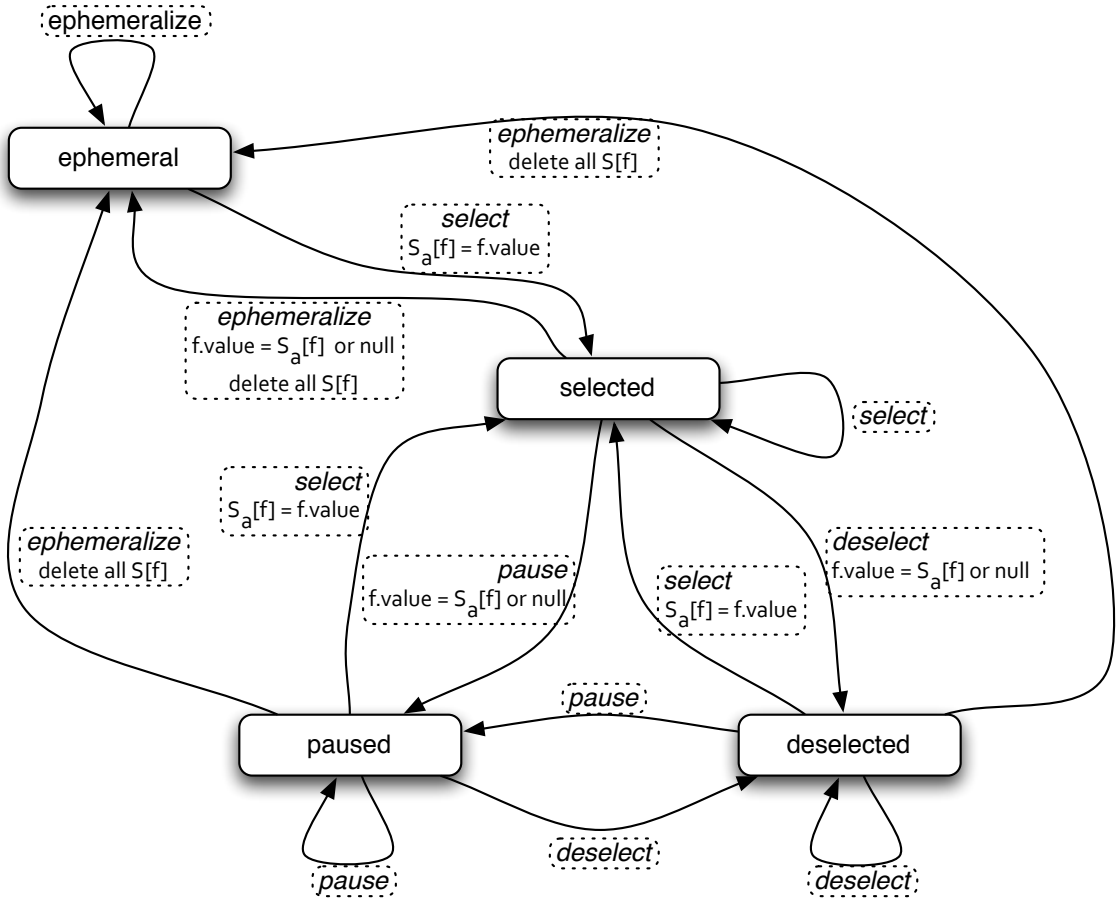


Figure 3.9: Graph depicting the transitions that are possible between the field states.

		ephemeral	selected	deselected	paused
Read-Only S_a	get	f.value	S_a[f] or error	f.value	S_a[f] or error
	set	f.value = new Value	error	f.value = newValue	error
Writable S_a	get	f.value	S_a[f] or error	f.value	f.value
	set	f.value = new Value	S_a[f] = newValue	f.value = newValue	f.value = newValue

Figure 3.10: Definition of read and store operations.

Recording and Browsing

We summarize the different operations possible on fields in Figures 3.9 and 3.10.

The state diagram in Figure 3.9 extends the state diagram of Figure 3.5 with the operations for each transition. In this diagram, the symbols are the following: f represents the field, $f.value$ the value contained in this field, $S[f]$ all values of f saved in snapshots and $S_a[f]$ the value of f in the active snapshot. The selection of a field (ephemeral, paused or deselected) copies the actual value of f in the active snapshot. When a selected field is transformed into an ephemeral field, a paused field or a deselected field, the value in the active snapshot for this field (or the object `null` if there is no entry for the field in the active snapshot – see Section 3.6.2 for an example) replaces the field value. Note that a selected field can be transformed into an ephemeral, deselected or paused field only if the active snapshot is writable: the value will always be the last one in linear and backtracking versioning but the value will depend of active snapshot in the branching versioning (in which several writable snapshots can co-exist). When a selected, paused or deselected field is transformed into an ephemeral one, all values of this field are deleted from all snapshots it appears in.

Figure 3.10 shows the rules used when a field is read or stored while the active snapshot is writable or read-only. The used symbols are the same as for Figure 3.9. The symbol $newValue$ represents the new value put in the field. The behavior of an ephemeral or a deselected field is independent of the active snapshot writability: the read returns the value contained by the field and the store replaces this value by the new one. The read of a selected field returns the value contained in the active snapshot for this field. If there is no entry for this field in the active snapshot, an error is thrown. Writing in a selected or a paused field is disallowed when the active snapshot is read-only (an error is thrown). If the active snapshot is writable, the new value is associated with this selected field in the active snapshot. Finally the paused field is a mix between a selected field when the snapshot is read-only and a ephemeral field when the snapshot is writable.

3.6 Three Variants of Versioning

The two first sections of this chapter explain the general model for object versioning. In this section we specify the model for linear, backtracking and branching versioning.

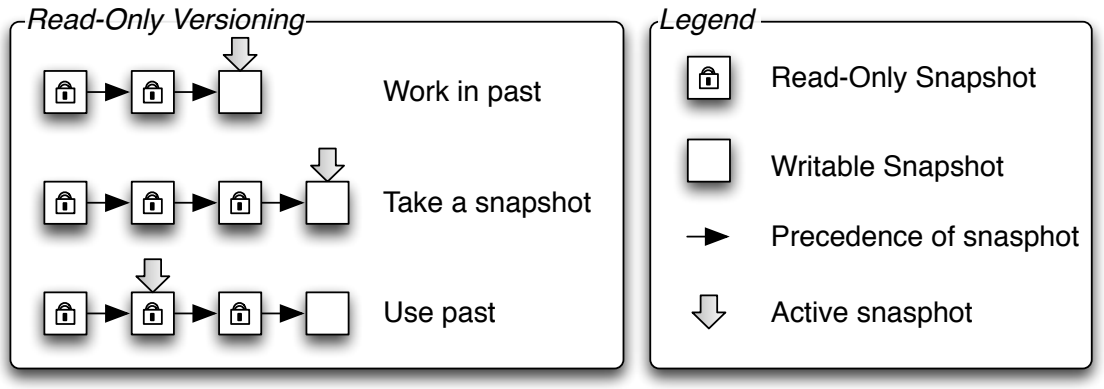


Figure 3.11: Linear operations with snapshots

3.6.1 Linear Versioning

In linear versioning there is always only one active snapshot at a time. At the start of the system, a snapshot is created by the system and it is activated. Fields can be selected at any time. All modifications of values of selected fields can only be done on a writable snapshot, i.e. the last created snapshot. All other snapshots are in a read-only mode (Figure 3.11). When a new snapshot is taken, the active snapshot becomes read-only. The new snapshot is writable, is initialized with a copy of all saved values of the selected fields of the active snapshot, and becomes the active snapshot. All modifications of values of selected fields are saved in the active snapshot, erasing previously saved values in this snapshot for those selected fields. The non selected fields can be modified independently of the writability of the active snapshot.

To browse past states, any read-only snapshot can become the active one. When the value of a selected field is read, we look for an entry for this field in the active snapshot. If there is a such entry, the associated value is returned. If not, an error is thrown. No modifications on selected or paused fields are allowed while the active snapshot is read-only.

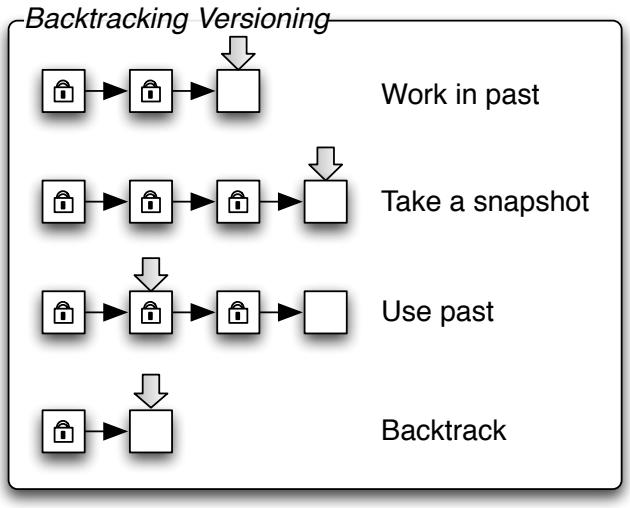


Figure 3.12: Backtracking operations with snapshots

3.6.2 Backtracking Versioning

As shown in Figure 3.12, the selection of fields and states and the browsing defined in the model for linear versioning remain the same in the backtracking versioning. However, the backtracking operation is added: all snapshots taken after the active snapshot S_a are deleted and a new snapshot is taken. All values contained in the deleted snapshots are unavailable for browsing.

The backtracking versioning is based on the undo and redo functionalities found in most text editors and web browsers. Take the example of a web browser. The web browser saves the state of the current page (i.e. the URL, the position of the scroll bars, etc.) after any click that changes the current state. To be consistent a first state of an empty page is created at the opening of the web browser.

The saved states of the browser can easily be retrieved by using the *back* button of the web browser. The *back* button lets users traverse states antichronologically (starting from the last one to the first one). A second button is as important as the back button: the *forward* button. This button performs the opposite action of the back button: traverse states chronologically. If there are four saved states (s1, s2, s3 and s4) and the back button is pressed 3 times, the state s1 is shown to the user. If the forward button is then pressed, the state s2 is shown. This behavior can be achieved with linear versioning because we

only browse the past states, and not modify them.

Backtracking versioning is needed as soon as a click on the displayed page is done and we are not in the last saved state. In our example, the displayed page shows the state s2 and a click is performed somewhere inside the page: the states s3 and s4 will be deleted and a new state s5 will be created. The available states will be s1, s2 and s5. We performed a *backtrack* to s2 and the deleted states s3 and s4 are called *backtracked states*.

The undo and redo features are realized similarly for a text editor (Figure 3.13). We have an object `aDoc` with a field `text`. We select it. We chose to take a snapshot after each space typed by the keyboard. In the editor we enter the sentence "a text T ". The step (a) shows the three snapshots taken (one per typed space). An undo (respectively a redo) operation replaces the active snapshot by its previous (resp. next) snapshot in the list, when a such snapshot exists. If we perform an undo, the active snapshot becomes S2 (step (b)). If we refresh the editor after the undo operation, the value of the field `text` will be asked. The field is selected and there is an associated value in S2: "a text " will be displayed. A second undo sets S1 as active snapshot and displays "a " in the text editor. If we perform a redo operation S2 becomes yet once the active snapshot and "a text " is displayed (step (d)). The value of the text follows the undo and redo operations, according to the active snapshot. When the text is updated, the active snapshot is either writable or read-only. Writable text indicates that the active snapshot points to the last state and that the update can be performed. The modifications of the value of the selected field are saved in the writable snapshot.

If the active snapshot is in read-only mode, an error is thrown. The system catches this error, backtracks until the active snapshot: all snapshots after the active snapshot are deleted and a new snapshot S4 is taken. The active snapshot is now writable and the modification on selected fields can be made. The backtracked snapshots are deleted: no redo operation to retrieve deleted snapshots is possible. In our example, the active snapshot is always S2 (step (d)) and we type the letter "S" in the text editor. The editor tries to put the new value "a text S" in the field. The active snapshot S2 is read-only and an error is thrown. The system catches this error and backtrack the system until S2. The snapshot S3 is deleted and a new snapshot S4 is created (step (e)). The value "a text S" can now be put in the field (step (f)).

The deselection and reselection of fields have the same behavior than in the linear ver-

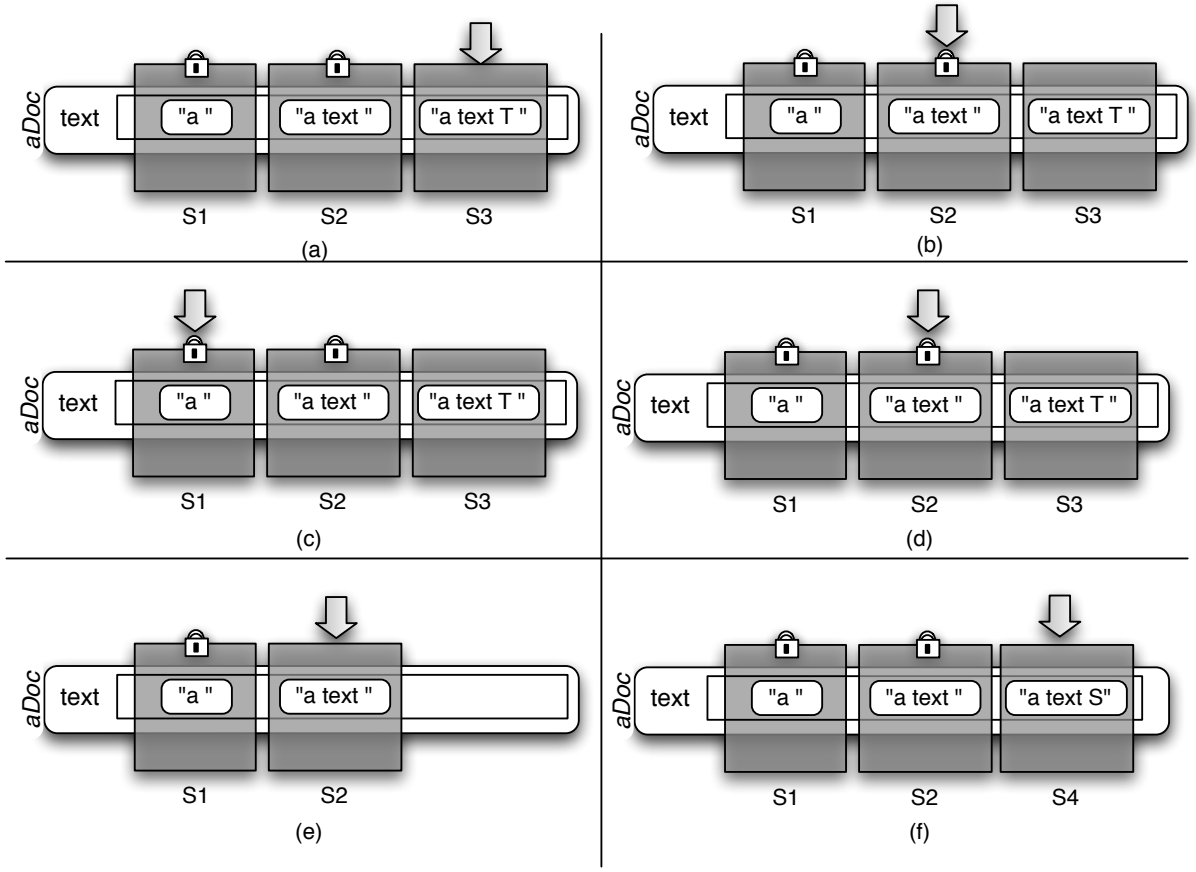


Figure 3.13: Backtracking in a text editor: undo and redo functionality

sioning. Notice that a deselected field can have several states saved in the snapshots created before its deselection. If some of them are backtracked before the reselection, the reselection does not retrieve these backtracked states.

Fields Selected in Backtracked Snapshot

A field can be selected at any time when the active snapshot is writable. Let s be the snapshot during which the field f is selected. If s is backtracked there is no more snapshot that has an entry for f. For any active snapshot the read of f will throw an error. But if a new value is stored in f, this value will be put in the active snapshot and reading it will return this value.

A selected field can be deselected at any time when the active snapshot is writable. Let s_s be the snapshot during which the field f is selected and s_d the snapshot during which the field f is deselected. If s_s is backtracked before the field is deselected, it is possible there is no entry for f in s_d. As described in Figure 3.9 the object `null` will be set as value of f [4].

3.6.3 Branching Versioning

Branching versioning adds the branching operation to linear versioning (Figure 3.14).

A new snapshot can be taken when the active snapshot is any snapshot, regardless of whether it is writable or read-only. If the active snapshot is writable, the creation of a new snapshot is the same than in linear and backtracking versioning. If the active snapshot s is read-only, we create a new snapshot initialized with a copy of the values of the selected fields of s. This new snapshot is writable and it becomes the active snapshot. The branching operation is the fact to take a snapshot from a read-only active snapshot.

A consequence of the introduction of the branching operation is that a read-only snapshot has one or more next snapshots whereas in the linear and the backtracking versioning, there

[4] When an object is created in many programming languages, the default value of its fields is the object `null`. Put the object `null` is thus a choice for our model dicted by our implementation experience. We could also raise an error each time the entry is used but, considering that an object contains selected and not selected fields, use the object `null` as value of selected fields when their versioned value is not clearly defined allows us to use the object even throughout a snapshot taken before it was selected.

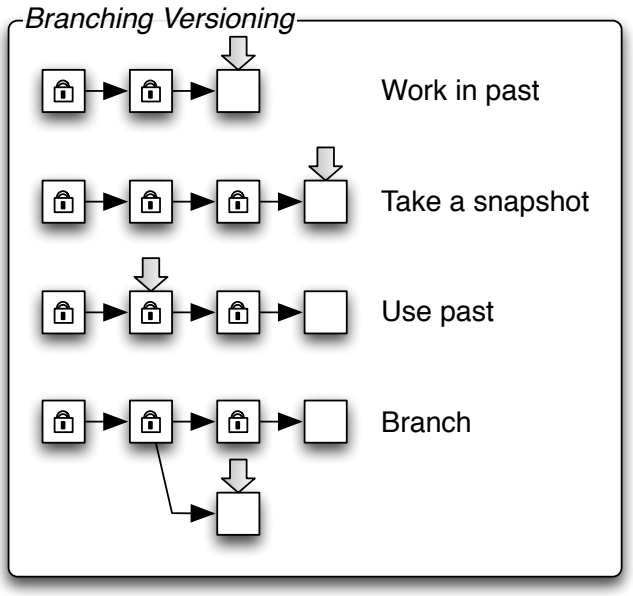


Figure 3.14: Branching operations with snapshots

is only one next snapshot for each read-only version. With branching versioning, snapshots form a tree in which the root is the initial active snapshot of the system. All snapshots that have no next snapshots (i.e. the leaves of the tree) are writable. Whereas there is only one writable snapshot at any time in the linear and the backtracking versioning, several writable snapshots can coexist in branching versioning.

Other operations are done analogously to linear versioning: the active snapshot is considered to determine the behavior to adopt when a selected field is read or a new value is stored in it.

The selection and deselection of a field as described in our model can be surprising in the branching model. When a field is selected, the current value of the field is stored in the active snapshot. Let f be the field and let s be this active snapshot. Notice that the value of f can be asked only when the active snapshot belongs to the subtree rooted at s: the other snapshots do not contain a value for f. When f is deselected, the value of f is set to the value stored in the active snapshot (if there is no entry for f in the snapshot the object `null` is stored in f). When the field is reselected again, its current value is stored in the active snapshot, independently of wether the active snapshot has an entry for f or not.

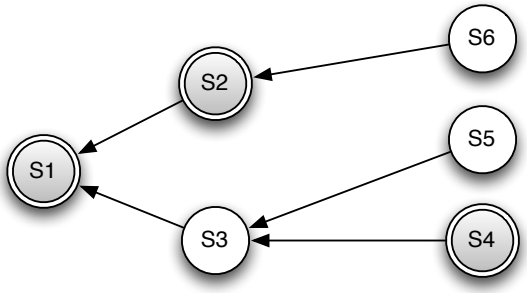


Figure 3.15: A tree of snapshots. The grey snapshots belong to the same snapshot set.

3.7 Controlling the snapshots

All snapshots are objects and can therefore be stored in a data structure, such a list or a tree (also named a collection). For example, in a distributed text editor, two users edit a text concurrently. We use linear versioning to add a view on the different versions of an user text. At each important modification (e.g. press the space bar or create a new paragraph), the system takes a snapshot, links it to the user that performed the modification and stores it in a global set of snapshots. This set is thus subdivided into two subsets: those linked to the first user and those linked to the second user. When the user wants to see the different versions of its text, we must browse snapshots only linked to this user.

To manipulate easily collections of snapshots, we introduce *snapshot sets*. A snapshot set is an ordered set that stores snapshots. A snapshot can be contained in none, one or more snapshot sets. The purpose of a snapshot set is to answer questions about the order between the snapshots it contains:

All snapshots returns all snapshots of the set;

Root snapshots returns all snapshots without parent in the set;

Previous snapshot returns the first parent, contained in the set, of a given snapshot on the path from the given snapshot to the root, if a such snapshot exists;

An example of snapshot set is shown in Figure 3.15: among the six snapshots of the system, three grey snapshots belong to a snapshot set. The snapshot S_1 is the root

snapshot and it is the previous snapshot for S_2, S_3, S_4 and S_5. The previous snapshot of S_6 is the snapshot S_2.

Note that when several snapshots must be returned as the result of a query of root snapshots, they are returned in a new snapshot set for convenience. The snapshot sets are used in all kinds of versioning. Note that for linear versioning and backtracking versioning, the root snapshots returns at most one snapshot.

3.8 Automatic Selection

The versioning model we propose gives users very fine-grained control in selecting for versioning what fields of what objects. However the drawback is that using the model can become very tedious when many fields of many objects need to be selected. Therefore we added automation facilities on top of our fine grained model.

As mentioned in the introduction of this chapter the automatic selection must respect the following two principles:

1. the automatic selection must be **expressive** enough to select automatically only the needed objects.

2. the automatic selection must not violate the **encapsulation of states of objects**.

To illustrate both principles, we give an example of a dictionary object. A dictionary object[5] keeps couples of objects, namely keys and associated values. Among other operations, the value can be retrieved given the key. Many efficient implementation of dictionaries exist. We use here one object-oriented implementation (inspired from the Smalltalk implementation) where many objects are manipulated.

Our dictionary object has two fields (Figure 3.16) that store an array and a number that represents the number of elements in the array. When a new entry $\langle$`key`, `value`$\rangle$ must be added in the dictionary, an association with the objects `key` and `value` is created by the dictionary and put in the array at a well-chosen place (using a hash value created from the

[5]Sometimes a dictionary is also called *hashtable*, when the key is hashable.

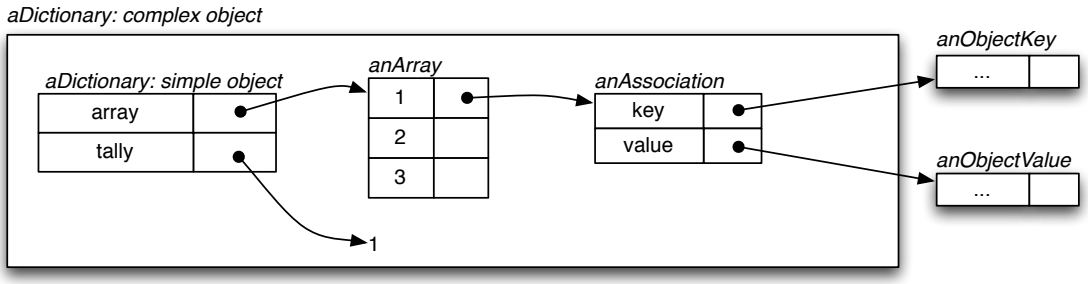


Figure 3.16: An example of a complex object that encapsulates objects: a dictionary with 1 association.

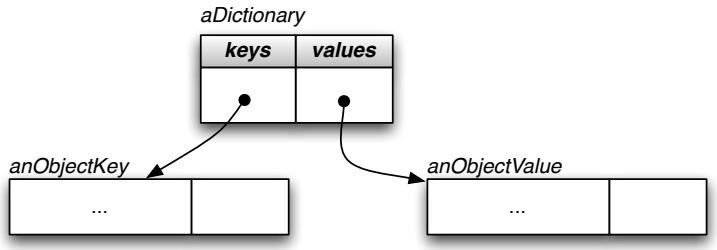


Figure 3.17: The dictionary seen from the outside: the array and associations are encapsulated and hidden to the other objects.

key object with collision management). When the array is full and a new association must be added, a larger array is created by the dictionary in which the associations of the old array are moved. The dictionary encapsulates the array and the associations: the dictionary creates and manipulates them and no other object than the dictionary can access the arrays and the associations. Any other object can only access the key or value objects and see the dictionary as shown in Figure 3.17.

Take the example of an application using a dictionary and wanting to save states of the dictionary at different times (i.e. the different objects used as keys and values) to know the order of their insertion in the dictionary, the deleted objects, the subsequent values for keys, and so on. The different internal states of the objects used as keys and values are not important here: we want to know the different states of the dictionary only, i.e. the different arrays put in the field `array`, the different associations put in any array and the different pointers put in the associations.

The automatic selection must select the arrays and its associations automatically when the dictionary is selected. If not, the developer would have to select each individual field one at a time.

In many cases, the dictionary itself will be encapsulated by an other object. Take the example of an object $Repository$ that keeps information about employees (Figure 3.18). To optimize the search on the name, the object uses a first dictionary where only the first letter of the name is used as key. For each letter used as key, a dictionary keeps each object that represents an employee associated with its name. The first dictionary and all sub-dictionaries are encapsulated in the object $Repository$. When the repository is selected, all dictionaries must be selected to keep consistent states of a repository. The values of the main dictionary (i.e. the sub-dictionaries) must be selected too.

The first principle of the automatic selection expresses that if we are only interested in the history of the dictionary, the past of the objects used as keys and values is not relevant and we must not be forced to select these objects if we want to select only the dictionary. On the other side, it must also be possible to express that we want to select the dictionary and all objects put in as keys and values as needed for the main dictionary in the example of the employees repository.

The second principle says that the objects encapsulated in an object (such as the arrays and the associations of a dictionary and all dictionaries of a repository) must be managed

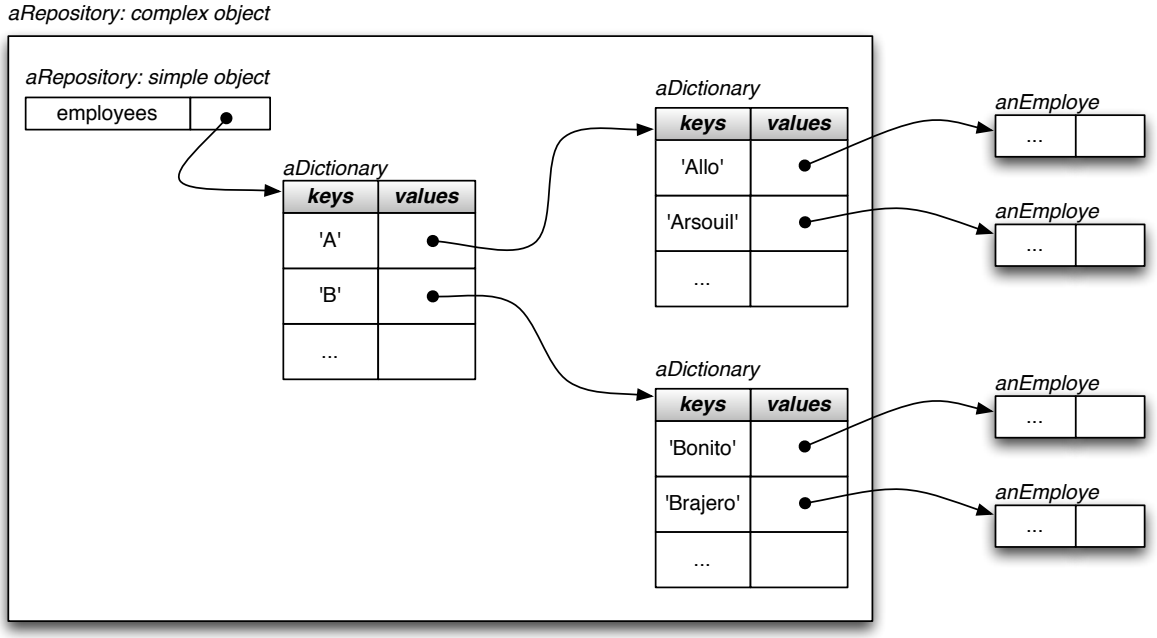


Figure 3.18: An object `Repository` with some values. The dictionaries are seen from outside for more readability.

inside this object and not from the outside. The selection of the arrays and the associations of a dictionary therefore must be encapsulated in the dictionary object. For example the repository object must not specify that arrays and associations of the main dictionary must be selected: these are managed by the dictionary and the repository sees the dictionary as a black box.

In the next section we extend our model with an automatic selection mechanism that respects these two points.

3.8.1 Automatic Object Graph Selection

To automatically select fields, we define for each object a selection depth. This selection depth is an integer between -1 and positive infinity and it specifies the distance to select objects in the reachable graph of the object. A **wanted selection depth** D_o and a **selection configuration** must be defined for each object o. Moreover a selection configuration for each object specifies a **wanted selection depth** d_f and a **selection operator** $\oplus_f$ per field f.

The selection operator will be used to know how to combine the object and field selection depths. There are two possible operators: fixed and sum. The operators are defined as follows:

Fixed returns d_f,

Sum returns the sum of D_o and d_f.

By default, each object o has a selection depth D_o equals to -1 and each field has a configuration $\langle 0, sum \rangle$.

Definition 3.8.1.1 *A* **path** *from an object o_1 to an object o_n is defined as an ordered sequence o_1,f_1,o_2,f_2,...,o_n such that:*

- *o_1,o_2,...,o_n are objects of the system*
- *f_i is a field of the object o_i*
- *o_{i+1} is the value of the field f_i.*

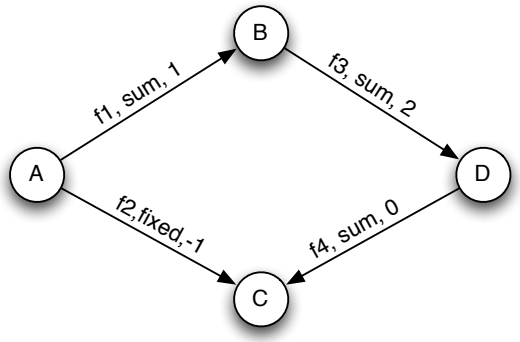


Figure 3.19: Four objects with their selection configuration.

Definition 3.8.1.2 *The* **weight of a path** p *(with a sequence* o_1,f_1,o_2,f_2,...,o_n*), denoted* $w(o_1, f_1, o_2, f_2, ..., o_n)$*, is the composition of the selection depth* D_{o_1} *and wanted selection depth of the fields along the path using the operator, i.e.*

$$w(o_1, f_1, o_2, f_2, ..., o_n) = (D_{o_1} \oplus_{f_1} f_1 \oplus_{f_2} f_2 ... \oplus_{f_{n-1}} f_{n-1}) - (n-1)$$

Note the substraction of $n-1$ to take into consideration the number of browsed fields on the path.

Definition 3.8.1.3 *The* **weight between two nodes** o_1 *and* o_n*, denoted* $w(o_1, o_n)$ *is the maximum weight over all paths from* o_1 *to* o_n *such that all partial paths have a positive weight, i.e.* $w(o_1, f_1, ..., o_i) >= 0$ *for all* i *from* 2 *to* $n-1$*. If no such path exist,* $w(o_1, o_n) = -\infty$*.*

Definition 3.8.1.4 *The* **weight of a node** o*, denoted* $W(o)$*, is the maximum weight over* o *and the set* $Objects$ *of all objects of the system, i.e.* $W(o) = max(w(s, o))$ *with* $s \in Objects$*.*

An object o is selected if $W(o) \geq 0$. A field f is selected if its object o (which it belongs) is selected and $W(o) \oplus_f d_f \geq 0$.

Figure 3.19 shows an example of configuration for the objects A, B, C and D of Figure 3.1 (page 44). This configuration must be read as follows. When the object A is selected, the field f_1 must be selected and the field f_2 must not (the operator fixed will always return the wanted selection depth -1 of f2 and it will then be never selected). The field selection depth of f_1 is 1 and its operator is `sum`. The field f_3 of B has a field selection depth of 2

and the operator `sum`. The field f_4 of D has a field selection depth of 0 and the operator `sum`.

The wanted selection depth associated with its operator of a field of an object o expresses the depth in the reachable set of o until which objects must be selected. For instance, consider the field f_3 in Figure 3.19 with its wanted selection depth of 2 and the operator `sum`. If B is selected with a wanted selection depth 0, the sum of 0 and 2 equals 2: all objects that have a connection depth smaller or equal to 2 in the reachable set of B must be selected.

We show now, throughout an example, how local configuration of each object allows a partial and encapsulated automatic selection.

3.8.2 Example

We show here how to configure the dictionary to select automatically its different encapsulated objects (arrays and associations).

First, we configure the object `Association`. When such object is selected, we want to keep the different values of both fields `key` and `value`. The configuration of an association is $\langle\langle key, 0, sum\rangle, \langle value, 0, sum\rangle\rangle$.

Second, we configure the arrays. When an array is selected, we want to keep the different values put in all its indexed fields. The configuration depends on the size of the array: $\langle\forall e \in array : \langle e, 0, sum\rangle\rangle$.

Thirdly the dictionary. When a dictionary is selected, we want to keep the different arrays (depth 0), the different associations put in each array (depth 1) and the different states of each association (depth 2). The different states of the field `tally`, that is the number of entries in the dictionary, must also be kept. The configuration of a dictionary is $\langle\langle array, 2, sum\rangle, \langle tally, 0, fixed\rangle\rangle$.

The selection operation for all configuration is the operator `sum` to propagate the selection automatically. We illustrate this choice using our example of the employees repository.

When the repository is selected, the main dictionary and all sub-dictionaries must also be selected. To express that in our model it suffices to set the configuration of the repository as $\langle\langle employees, 2, sum\rangle\rangle$. Depth 2 expresses that we want to keep the pointers to all dictionaries used as main dictionary (depth 0), the internal changes of each main dictionary

(depth 1) and the internal changes of each dictionary put in each main dictionary (depth 2).

Figure 3.20 extends Figure 3.18. Each field is represented by a row with 3 boxes: the field name, the object configuration for this field (wanted selection depth and selection operator) and its value (a pointer or a primitive value). For more readability we put boxes around the conceptual complex objects (repository and dictionaries) and their encapsulated objects. We set the wanted selection depth of repository to 0, i.e. we want to select enough objects to save the internal states of the repository. The labels with a circle on arrows are the weight of the path from the start to the objects pointed by fields. We see that all fields in the repository are well selected (they belong to an object with a positive weight) while the employe fields will be not selected.

There are two important related remarks. First, the meaning of the depth is respected from the point of view of the complex objects: the selection depth of 2 of the field *employees* expresses that the main dictionary and all dictionaries put in this dictionary must be selected. We see that the weight of partial paths decreases at the end of each complex object: the weight of the path from the repository to the employees objects pointed by the fields `employees` is 2, the one to the objects pointed by the fields `value` of associations maintained by the main dictionary is 1 and 0 for the objects pointed by the fields `value` of sub-dictionaries. If each object defines correctly how to select its encapsulated objects when it is itself selected, the other objects can select an object as a black box and their configurations mean "select this object and all objects attached directly to it" with a depth of 2, including the encapsulated objects.

The second remark is that the repository can be seen itself as a black box now: if we want to keep its internal states, we set its wanted selection depth to 0. If we want to keep also the internal states of the objects linked directly to it (i.e. the employees), we set its wanted selection depth to 1: the employees will be selected because the weight of the path from the repository to the employees is 0.

3.8.3 Discussing Automatic Selection

The automatic selection mechanism respects the encapsulation of objects by encapsulating the selection configuration. Each object has the responsibility to define its own selection

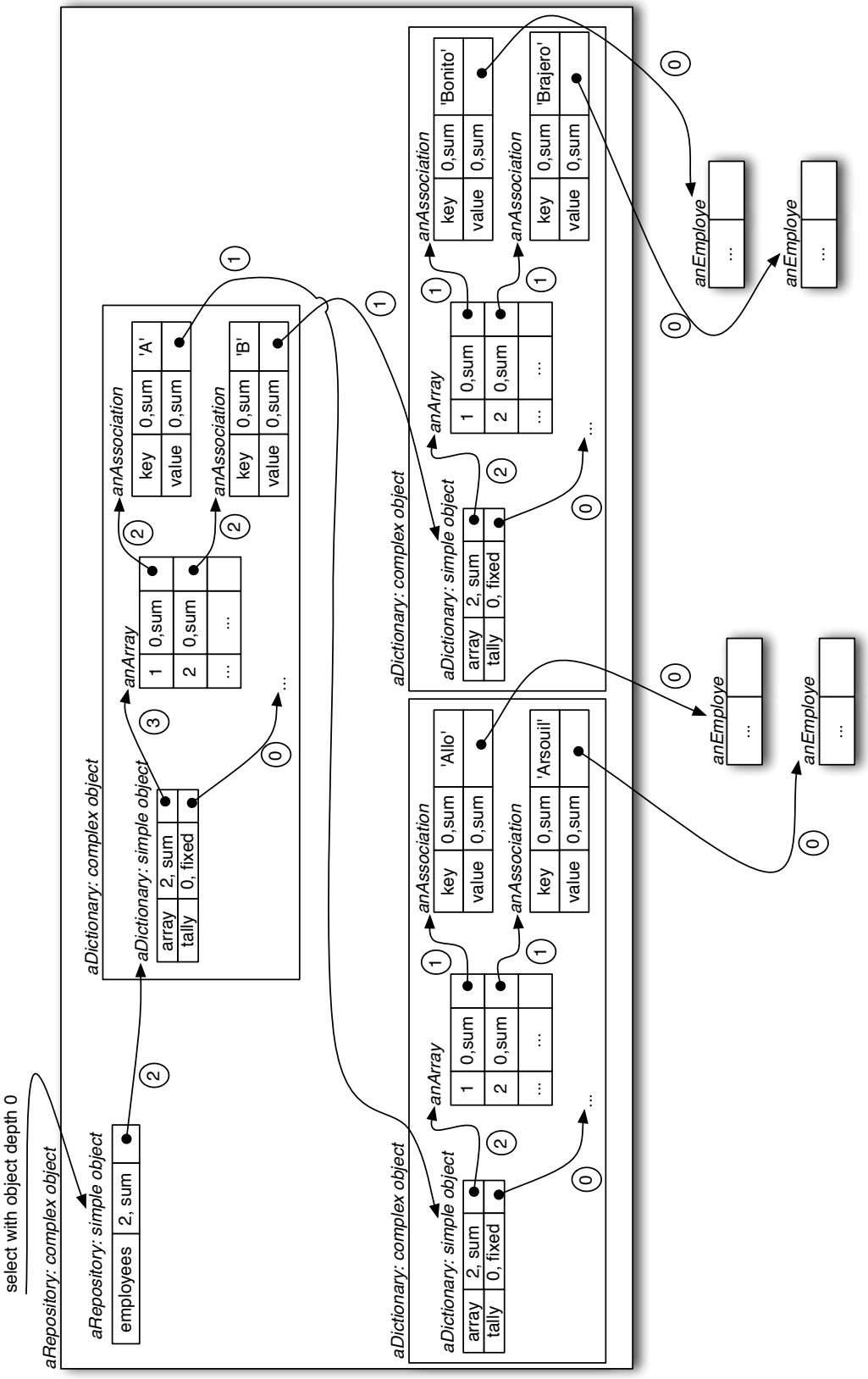


Figure 3.20: Repository with field selection depth and current field depth.

configuration, i.e. what must be selected when is itself selected. This configuration is not global but it is distributed among objects. This implies a fine-grained automatic selection. The automatic selection makes it possible to select the minimal number of fields that are needed to save the interesting fields for all objects of the application.

Due to the encapsulation of selection configuration, the complex objects remain black boxes: external objects do not know the internal structure of the object and how it manages its encapsulated objects. When a complex object, such as a dictionary object, is asked to be selected, we can assume that the encapsulated objects will be selected correctly.

The other point is the automatic propagation of the selection. When an ephemeral object o is stored as new value of a selected field f, it will be selected automatically. It makes it possible to select the objects in the reachable set of o. This important principle propagates the selection automatically by the system.

Note that each object has the possibility to specify which objects must be selected when it is selected itself. For example, the dictionary selects its arrays and associations. An external object can select this dictionary with the assurance that the internal structures will be automatically selected. An external object can also select this dictionary with a wanted selection depth of 1 to select the internal structures and the objects directly linked from it, i.e. all values used as keys and values. One restriction of our model is that there is no way for an external object to have information on how objects are directly connected with one specific object: each selection configuration indicates the distance in the reachable set until which the objects must be selected automatically. For instance, the objects used as values and keys in a dictionary are at distance 1 from the external object and the external object that points to the dictionary cannot distinguish between keys and values thus. There is no way to select only the keys or only the values.

In this chapter, we only focus on the model. In the next chapter that focuses on the efficient implementation of our model, we discuss different ways to implement efficiently the automatic selection.

3.9 Special cases

The partial selection of the objects graph is a powerful mechanism to select with precision the interesting fields of all objects of the system. In this section we finish the presentation

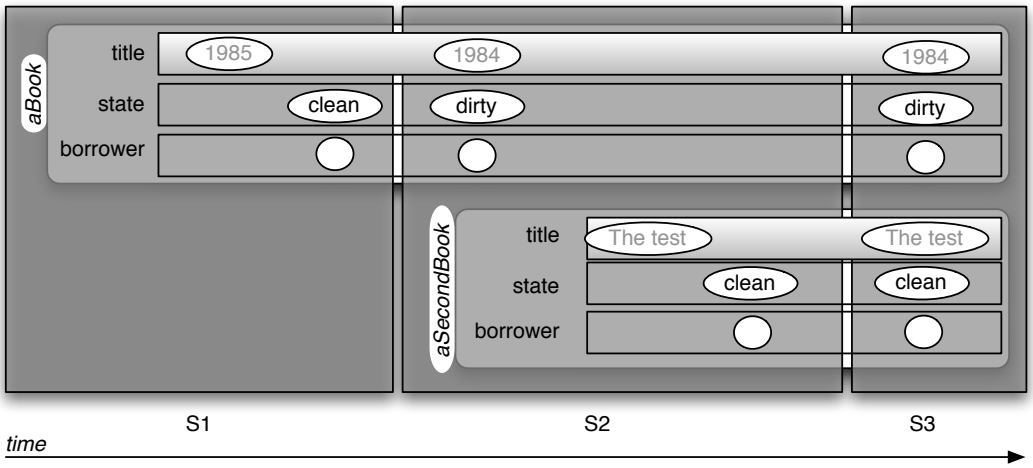


Figure 3.21: The book "The test" is selected after the first snapshot.

of our model by analyzing a few special cases of our model.

3.9.1 Selection After Snapshot

Selecting fields and taking snapshots are two independent actions that the developer can perform as wanted. The developer can select some fields, take snapshots, select some other fields, take snapshots and so on.

For example (Figure 3.21), we show another use case of the previous library example. We create a book titled 1985 in a clean state and we select its fields `state` and `borrower`. We take a snapshot (S1). We correct its title (1984) and set its state to dirty. We create also a new book titled "The test" for which we select the same fields and we take a snapshot (S2). The snapshot S3 is the active and writable.

When we browse the different versions of this little application (Figure 3.21), what happens when we ask the value of the second book fields throughout the snapshot S1? For the title, that is straightforward: the value of the title, this field being not selected, is returned. But the state and the borrower are selected and no value for them in the snapshot S1.

In our model we chose to throw an error when a selected field is queried throughout a snapshot taken before its first state.

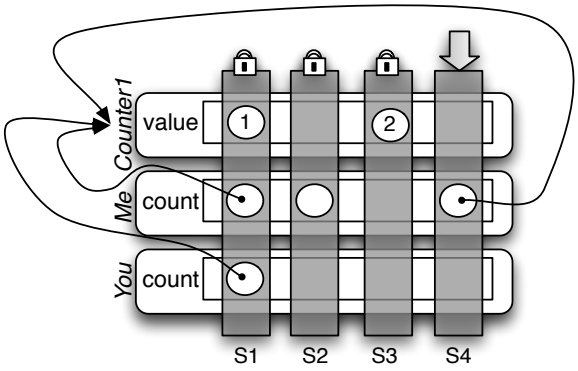


Figure 3.22: Take an object from past to present but with a new value.

3.9.2 From Past to Present with Modifications

Sometimes the user might want to use an object found in a read-only snapshot in a writable snapshot. For example, assume two objects "me" and "you" with one field `count` that points to the same object: a counter "Counter1" with a value initialized to 1 (Figure 3.22, S1). A snapshot is taken. The object "me" removes its pointer to the counter (S2). A new snapshot is taken. The object "you" increments the counter by 1 and a snapshot is taken (S3). The user browses the past to find the first non `null` value for the field `count` for the object "me". It is found in the snapshot S1. The reference to the counter is reassigned to the counter of the object "me".

When the object "me" reads its counter in S4, the value is 2 and not 1 as in the snapshot S1. There are two possible user scenarios:

1. the user wanted only the reference to the counter and a new value is not a problem for him. Nothing more need to be done.

2. the user wanted to retrieve the last value of the counter for the object "me". There are two solutions:

   - A (deep) copy of the counter throughout the snapshot S1 is assigned to the object "me" in S4. A copy creates a new object with the same fields pointing to the same object. The identities of the counter and the copy are different. A deep copy of an

object o creates a copy of the object graph of o. All objects are therefore copied with a new identity.

- A proxy object is used. This proxy keeps a reference to the subject object ("counter1") and a snapshot (S1). When a message is sent to the proxy, it changes the active snapshot to S1, sends the message to "counter1", replaces the previous active snapshot and returns the answer of the sent message. The proxy is a view on an object throughout a given snapshot. No copy is done and the identities of all objects are saved.

The first solution does not keep the identity of objects while the second solution preserves them. The second solution is better if it is necessary to keep objects identity in the application. But this second solution is also more complicated to implement (see Section 5.4.2, page 166).

We do not restrict our model to the first or the second case: depending on context of application, both can are useful. Our model permits both and the developer can choose exactly what (s)he want to retrieve from past.

3.10 Related Work

Our model is related to two main research fields: orthogonal persistence and temporal and versioned databases.

Orthogonal persistence aims to transform short-lived objects to long-lived objects with the maximum of ease of use for the developer. A short-lived object is typically defined as an object that is created and deleted during the lifetime of a program [Atkinson & Morrison, 1995]. A long-lived object remains available even after the program is finished: it can be saved on files, in a database or anywhere else. The benefits for the developer are the independence of the support to save objects (e.g. files or database), the independence of the types of objects to save (any object can be saved) and the freedom to manipulate short-lived and long-lived objects.

The databases store and retrieve data efficiently on physical media. Relational databases (organized by tables) and object databases (structured around the object-oriented concepts)

are the most frequently used storage media. A temporal database adds time information to database data. A versioned database keeps different versions of its data.

Each type of database has several models for selection of objects and states with complex object integration or not. In the following subsections we compare our model with database models on the following characteristics.

Selection of fields The parts of the system to be versioned can be defined (e.g. all instances of a class, a particular object or a field of an object). To our knowledge there is no database model that is able to select individual fields of objects. The finest granularity is a single object. The example of the book in the library (see Section 3.4.1) can not be realized without selecting the complete book. The fine granularity of our model is a major contribution of our dissertation.

Selection of states States at given times can be kept while states at other times can be forgotten.

Complex Objects Integration The models can define specific rules to facilitate the versioning of complex objects. A complex object is an object linked with other objects by structural or existential dependences [Oussalah & Urtado, 1997]. A complex object is an object o with a subset s of its transitively connected objects. The subset of objects is composed only of objects p such that there is a path between o and p following the pointers of objects contains in s. The objects in s are often semantically defined.

Selection Propagation Selection propagation automates the definition of the parts to be selected in the system. The propagation can be related to the complex objects integration.

Global and local versioning Most of the time the object versioning is global to the system. Some models support local versioning, in which versions for a subset of versioned objects can be created.

3.10.1 Orthogonal Persistence

Orthogonal persistence saves objects on physical support. It does not care about versions of objects: only the last version is kept. But there are some commonalities with our approach: the selection of objects to save, the complex object integration and the propagation of selection.

Orthogonal persistence follows three principles [Atkinson & Morrison, 1995]: persistence independence, data type orthogonality and persistence identification.

Persistence independence states that the longevity of the data has no impact on the form of the program: short-term and long-term data are manipulated in the same way. We apply this principle to object versioning: an ephemeral object and a versioned object must be manipulated in the same way. We will show in Chapter 5 that our model can be integrated into a language in a such way that the selected objects are manipulated the same as non selected objects.

Data type orthogonality defines that all objects can be transformed into persistent objects, independently of their type: there is no object that is not allowed to be long-lived or not allowed to be transient. We apply this principle to object versioning: any object is allowed to be versioned or not, independently of its type. Our model is completely independent from the type of the object: any object can be versioned independently of its type.

Persistence identification defines that the way to identify and provide persistent objects must be orthogonal to the universe of discourse of the system. This principle corresponds to the selection of fields in the object versioning.

3.10.1.1 Selection of fields

The principle of persistent identification imposes that the mechanism to identify (i.e. to select) objects to be persistent must be orthogonal to the rest of the system. To achieve this goal an often-used technique is *identification by reachability*, i.e. the objects to be made persistent are identified by the system by computing the transitive closure of all objects reachable (by following pointers) from some persistent root or roots [Atkinson & Morrison, 1995]. The roots are objects defined by the system. This technique can be reused for object versioning: the objects to select are all objects in the transitive closure of all objects reachable from some root(s).

Our model allows the identification by reachability. If the selection configuration of an object selects all its fields (this is the default behavior), it suffices to select any object considered as root with an infinite selection depth (or the biggest integer managed by the system). The selection depth must simply be greater than the total number of objects in the system. The system will then select all objects in the transitive closure of all objects

reachable from the root.

Our model offers a mechanism that is even more expressive than identification by reachability: a different selection configuration can be expressed for each object; an object can omit some of its fields from selection if necessary. With this expressivity, an automatic selection of a subset of objects of the reachable set of a root is possible: all objects of the reachable set of a root are not necessarily selected automatically.

3.10.1.2 Complex Objects Integration

By definition, identification by reachability does not manage complex objects: there is no way to define which objects must be selected when an object (considered as a complex object) is selected itself. All the transitively connected objects will be selected.

In our model each object can define its own selection configuration. We shown that it allows one to select properly complex objects, such as dictionaries.

3.10.1.3 Selection Propagation

Identification by reachability propagates the selection of objects automatically: when a non persistent object is connected to a persistent one (i.e. there is a path between the root and this object following pointers), the non persistent object becomes automatically persistent. In our model the non selected objects added in the reachable set of the root (by a new link) will be automatically selected.

One difference between object selection configuration (from our model) and selection by reachability as defined for orthogonal persistence is that a selected object that becomes not reachable from the root (because the pointer to it is removed) will remain selected while the object will be no longer persistent in the orthogonal persistence. Our model focuses on the minimal set of objects to be selected and does not focus on their deselection. Notice that a manual deselection, pausing or ephemerization remains possible.

3.10.2 Temporal Databases

A database is a program that aims to organize, store and retrieve data easily. The most studied databases are relational databases [Codd, 1970] and object-oriented

databases [Won, 1990]. Relational databases contain a collection of tables in which entries are interconnected by keys. Object-oriented databases are modeled around object concepts: data is managed as objects with fields, as in object-oriented languages.

Temporal databases improve classic databases by adding the notion of time to database data. The bitemporal model [Snodgrass, 1992], the most common model, manages a set of facts, i.e. logical statements that are true in real world. For example, "Albert works at iMec" is a fact. Temporal databases add time information to facts by using two kinds of time: the valid-time and the transaction-time. The valid-time defines the period of time during which the fact is true. Outside of the valid-time the fact is considered as false. The transaction-time defines when the data is considered as available in the database. By default, the transaction-time of data covers the interval of time from the creation of the data to its deletion. The transaction-time allows one to search in the database in a previous version ("Ten years ago where did the database believe Albert worked?") and the valid-time allows time information ("Where did Albert work ten years ago?"). Notice that the valid-time and the transaction-time can reflect on the past, the present and the future.

The changes of prices of some products following a business calendar is a good example of the usage of the valid-time and transaction-time. On December, 16th 2010 the user enters different prices of a product for the year 2011 in a database:

ProductId	Price	Valid-Time	Transaction-Time
1	80	01/01/2011 - 01/31/2011	12/16/2010 - 12/21/2010
1	100	01/31/2011 - 30/06/2011	12/16/2010 - $+\infty$
1	70	07/01/2011 - 07/15/2011	12/16/2010 - $+\infty$
1	50	07/16/2011 - 07/31/2011	12/16/2010 - $+\infty$
1	100	08/1/2011 - $+\infty$	12/16/2010 - $+\infty$
1	75	01/01/2011 - 01/15/2011	12/22/2010 - $+\infty$
1	60	01/16/2011 - 01/31/2011	12/22/2010 - $+\infty$

Each valid-time indicates the period of time during which the price will be active. The transaction-time starts at the day of the encoding and stops at the deletion of this data (if the transaction never stops it is noted by the infinity sign). Notice that the price 80 firstly entered was split into two prices 75 and 60 the December, 22nd 2010. The transaction-time of the first price is bounded between the 16th and the 21st.

When a query of price is done with a given valid-time t_v and a transaction-time t_t the price with a valid-time that includes t_v and a transaction time t_t is returned. A request for the valid-time January, 22th 2011 and the transaction-time Jaunary, 17th returns 80 while the same request with the transaction-time January, 23th returns 75.

Temporal databases and our model are very close. We can export data from our model to temporal database and vise versa: data columns and the valid-time become fields in our model and each line of tables is transformed into an object. These objects are selected and snapshots replace the transaction time. For instance, the first five lines of price-example can be transformed into five objects with three fields: ProductId, Price and Valid-Time. We select them and we take a snapshot. We tag this snapshot with the current date (here 12/16/2010). When the first line is split into two new lines (on 12/22/2010), we modify the first object and we create a new one to contain the information of the last line and we take a snapshot. We tag also this snapshot with the current date (12/22/2010).

However our model and temporal databases do not share the same goal: temporal databases add time to data where our model saves the previous states of objects. In a temporal database, the user associates time information with their data and the user defines the meaning of this time information in the application. In our model, the user defines at which times states of which objects must be saved. Moreover, at our knowledge, temporal databases only allow linear versioning while our model is applicable for linear, backtracking and branching versioning.

3.10.3 Databases Schema Versioning

Schema versioning [Li, 1999, Edelweiss & Moreira, 2005] studies the schema evolution of databases and the impact of this evolution on the data. The schema of a database is the definition of the structure of this information. For instance the classes and their relations (inheritance, etc.) defines the schema of an object-oriented database. When the schema evolves, its different versions can be saved. For example, when a new class is added the system stores this event in a table. The developed techniques to solve this recurrent problem are specific to the schema database and they are not comparable with our model.

3.10.4 Versioned Object-Oriented Databases

Versioning in object-oriented databases makes it possible save different versions of objects [Zdonik, 1984, Bjornerstedt & Britts, 1988, Beech & Mahbod, 1988, Oussalah & Urtado, 1996, Oussalah & Urtado, 1997, Rodríguez *et al.* , 1999, Khaddaj *et al.* , 2004, Arumugam & Thangaraj, 2006]. Their versioning is not global as our model, where all objects participate to a global system history. Their versioning is centralized on object graph defined as follows. Each object has a set of versions that contains its different states. New versions are created implicitly (at any change [Oussalah & Urtado, 1997] or following strategies [Oussalah & Urtado, 1997, Oussalah & Urtado, 1996]) or explicitly ([Zdonik, 1984, Bjornerstedt & Britts, 1988, Beech & Mahbod, 1988, Oussalah & Urtado, 1996, Oussalah & Urtado, 1997, Rodríguez *et al.* , 1999, Arumugam & Thangaraj, 2006]). When a new version of an object is created, the objects that refer to this object create a new version such that each object contains a set of versions that corresponds to each of its modifications and to each modification of its directly or transitively connected objects. Some techniques allow breaking this upward version propagation by configuration.

In our model, the states of each object are saved in snapshots. These snapshots are global to the system and have a view on the past of the whole of the system. Each modification does not require propagation of versions but in certain cases, propagation of selection.

The selection propagation has been studied in these papers. Most of the time selection by reachability is used. But in [Oussalah & Urtado, 1996, Oussalah & Urtado, 1997], the authors propose another technique to propagate selection by semantical rules and strategies. These techniques seem expressive. We could add their technique as a layer on our model but unfortunately the authors do not present any details about a possible efficient implementation and we doubt of its achievement. On the other side, our model can be implemented efficiently (as shown in the next chapter). Finally these techniques are globally defined: rules can not be defined for particular instances and the encapsulation of selection propagation in each object by these techniques seems really difficult.

3.10.4.1 Selection of fields

Selection of fields is similar to our model. Some fields can be selected while other ones remain non selected. The rules for versions browsing are not always well defined.

3.10.4.2 Selection of states

Selection of states is either implicit (all states or by upward propagation) or explicit (by manual version creation or via some rules and strategies).

3.10.4.3 Complex Objects Integration

Most of the time, the versioning of complex objects in versioned databases is defined by one step selection (the developer can select the directly connected objects) or selection by reachability. Our model is more expressive: the selection depth of our model selects objects in a more precise way. For example, we can save the different states of a dictionary without saving the connected values and keys, which is not possible in versioned databases.

In [Oussalah & Urtado, 1997], the authors try to fix the problem of complex object versioning by rules and strategies for selection propagation. As we already said, the authors do not present any details about a possible efficient implementation and we doubt of its achievement.

3.10.4.4 Selection Propagation

Using selection by reachability new objects connected to versioned objects becomes versioned as well. In most cases (except [Oussalah & Urtado, 1996, Oussalah & Urtado, 1997]), there is no way to specify if the propagation must be done or not.

3.10.4.5 Global and local versioning

The versions are local to the reachable set of an object. There is no global versioning as in our model.

Our model allows one to maintain several snapshot sets. The meaning of each snapshot set is given by the developer. For example, we can have a snapshot set for all versions of the system and a snapshot set for each object that contains the different snapshots relative to the object modifications.

3.11 Discussion

In this section we discuss our design choices.

3.11.1 Field Granularity

In our versioning model we focus on the fields of objects and not on objects themselves. We think our decision to use a very fine granularity, namely the field, is a great improvement for the developer: we let the possibility to keep the history of a complete object but with the possibility to keep only a part of this object.

There is some cases where the selection of fields is not only useful but even necessary. In a system, as Smalltalk, where many objects co-exist, there are special objects that can be linked to a huge number of objects. For instance in Pharo (a Smalltalk implementation), the object `SmalltalkImage` points transitively to all objects of the system. If the object `SmalltalkImage` is pointed by a field f of an object o and we select o with a big depth (to select all object graph for example) and we do not want to select all objects of the system, we must specify that the field f must be not selected when o is selected.

Notice that objects are not versioned but fields can be versioned. As a consequence, a reference to an object is always the same in any snapshot.

3.11.2 Snapshots versus Version Numbers

In the related works discussed in the previous section, version numbers play a central role. All operations at certain point in the past are identified by a version number. But a version number is an integer and all information about this point in past must be saved somewhere else in the system.

In our model the version numbers are encapsulated in snapshots and they are hidden from developer. This little difference allows one to keep information about a past instant in a unique object, usable as any object of the system. The developer has a real object which he can use, manage and extend as necessary. Moreover, the developer can attach properties to a snapshot to specify additional information. Finally snapshot sets allow for easy organization of different saved versions of the system.

3.11.3 Global and Method Variables

Our model focuses on fields of objects and does not study other kinds of variables, such as global and method variables (the temporary variables used during the execution of a method). This is a design choice.

These variables could easily be versioned as well. Global variables can be collected in a global selected object in which each global variable will be a field.

Method variables are linked to the method to which they belong. If the method is reified (i.e. there exists an object that represents this method, as in Smalltalk) the different method variables can be maintained in a selected dictionary put in the object that reifies the method. For example, in Pharo, the class `CompiledMethod` holds such a dictionary that can be selected.

3.11.4 Concurrent Accesses

To avoid concurrency problems, we assume the following operations are atomic:

- access the global version number;
- access the list of versions (backtracking versioning);
- access the tree of versions (branching versioning);
- access a snapshot entry.

In the implementation we have achieved this with semaphores.

3.11.5 Transactions

We finish by discussing transactions. Transactions make it possible to change objects in a context hidden from other contexts (concurrent executions for example). To truly perform the changes they must be *commited*. Non commited changes can also be forgotten by a *rollback*.

Our model and transactions do not share the same goal. Our model saves the history of objects, independently of the context and visibility of changes. However the link between transactions and our model is that transactions can be implemented using the object versioning as a first layer of the implementation design. Implementing transactions using object versioning is a future work of this dissertation.

3.12 Conclusion

This chapter presents our first contribution: a fine-grained model of object versioning for object-oriented languages. This model is centered around the developer: only the developer knows which versions to keep for its application and our model provides mechanisms to express which parts of history of an application must be saved. This model is designed to be fine-grained, expressive and compatible with linear, backtracking and branching versioning.

The model focuses on fields, not on objects. A field is either ephemeral or versioned. An ephemeral field keeps only its last value: old values are deleted. A versioned field keeps its different states. An object can mix ephemeral and versioned fields.

The choice of versioned fields is let to the developer. The developer selects the fields to be made versioned. Our model supports manual and automatic selection. The automatic selection follows a configuration defined in each object by the developer: any object has the responsibility to select correctly its fields and the values pointed by its fields when the object is selected itself. A mechanism of selection depths make possible to select only a part of the reachable set of an object. Because the configuration of automatic selection is encapsulated in each object our model allows selection of complex objects.

Once fields are selected, the developer takes snapshots to save the current state of these selected fields: when a snapshot is taken, it saves the current values of selected fields, as a partial backup of the system in which only the states of fields being selected are saved. Deselection and pausing mechanisms stops the collection of new states for given fields.

The developer uses the snapshots to navigate on the time line, i.e. browse the past, return to present, backtrack states and create a new branch. In our model the developer selects a snapshot to be the active one. The activation of a snapshot without successors (named writable) allows the collection of new states. The activation of an old snapshot (named read-only) allows the retrieval of the values of selected fields saved in this snapshot.

Using backtracking versioning, the backtrack operation takes an old snapshot and deletes all states of selected fields after this snapshot. Using branching versioning, taking a snapshot s while an old snapshot s_o is activated defines s_o as the parent of s, i.e. a new branch is created form s_o.

In the next chapter we show how to implement efficiently this model for each kind of versioning.

Chapter 4

Efficient In-Memory Object Versioning

In the previous chapter, we described a model for object versioning for linear, backtracking and branching versioning. This model can be implemented in several ways, for example by storing states in files, in a database or directly in memory. In this chapter we focus on storing in memory. The two main challenges when storing states in memory are the size of the information about the past which can grow fast in memory and the time to save this information and retrieve it. Our goal is to find techniques to save the states of fields only in memory with a good efficiency in time and in space.

This chapter focuses on the study of algorithms and data structures we developed to implement efficiently the three kinds of versioning in memory. The next chapter focuses on the integration of the model into an object-oriented language.

The presented algorithms use basic operations such as creating an object and capturing read and store of a field. These operations are realized in constant time in most of languages. Moreover we assume that the active snapshot is a global variable that contains a snapshot. To activate a given snapshot, we put this snapshot in that global variable. The time to access the active snapshot is constant. In the next chapter, we will see that the activation can be implemented in two different ways (the global activation and the thread activation) to grow the expressivity.

The time to free memory depends on the language. For example, when a garbage collector is used, the operation should take amortized constant time. We assume in this chapter that this operation is done in constant time but our analysis can easily be adapted to correspond to a specific language.

This chapter is structured as follows. First, we give some applications that need the in-memory object versioning. Second, we define how to implement efficiently each operation

for each kind of versioning. The automatic selection is studied in a separated section because it is transversal to the other operations. We finally discuss the possibility of automatic deselection and we conclude.

4.1 In-Memory

In many cases, versions of an application can be kept in memory only. Obviously, this data can be saved in an external database or in a file. But memory has the advantage that it is faster than databases and files. This efficiency is even crucial for applications that depend essentially of the time to store and retrieve versions, as file editors and debuggers. Long response times make the tool unusable.

Moreover, as will be shown in this chapter, each state will be stored in memory only once. Our benchmarks (see Chapter 6, page 197) show that storing 10^5 states of an integer variable (stored on 8 bytes) takes only about 5.3 megabytes for linear and branching versioning and about 30.5 megabytes for branching versioning. If we consider also that the memory size of sold computers grows each day, storing states in memory is not a problem until the number of states becomes very large.

4.1.1 Undo/Redo

The undo functionality allows one to retrieve past states of an application. The redo functionality allows one to return on states browsed via undos. Any modification on a past state deletes the saved states that follow it.

These functionalities are present in the majority of applications nowadays: editors for texts, music, images and movies. The changes from the opening of the document are saved until its closing. Once closed, the saved states are forgotten by the application.

Most of the time these functionalities are implemented by storing the different undoable states only in memory without using any file or database.

4.1.2 Debugger

Many object-oriented languages offer a debugging tool to help finding bugs. The debugger allows one to browse the contexts of each method call on the stack (temporal variable, etc.) and to execute a program step by step. Most of the time the old states of objects that participate to the execution are not saved. Each value erased in a field by the debugger is lost.

The debuggers of the new generation [Pothier *et al.* , 2007, Lienhard *et al.* , 2008] save the old objects states. These tools are more powerful than simple debuggers: not only the last values of fields are present but all the story of the execution is available.

These kind of debuggers are most of the time implemented only in memory so as not to hurt the efficiency of the debugged program.

4.2 A First Solution

We can implement our model directly following strictly its definition: in each snapshot, we put a dictionary that keeps fields and their associated values. When a snapshot is taken, a snapshot dictionary is created with a copy of the dictionary of the old active snapshot. If we assume that the insertion and the lookup in a dictionary can be performed in constant time, the complexities of each operation is constant, except for taking snapshots (see Table 4.1): copying all states of the active snapshot into a new snapshot takes time and space linear in number of selected fields. These complexities are acceptable only if the number of snapshots and/or the number of selected fields is really small. But this configuration is not the most common case.

We want a solution acceptable for any configuration. Many tradeoffs of complexities exist. After an analysis of common applications (see Chap. 6), we think is more important to record (and take snapshots) efficiently the states than browse efficiently the past: browse the past is often seen as an option on the existing application and this option must not decrease *too much* the execution time of the ephemeral application. Obviously the size of the states must be minimized.

The following sections describe how to implement efficiently the three kinds of versioning.

	Active snapshot							
	Read-Only				Writable			
Take a snapshot	NA				$O(\#fields)$			
	State of the field				State of the field			
	NS	S	D	P	NS	S	D	P
Select field	NA	NA	NA	NA	$O(1)$	$O(1)$	$O(1)$	$O(1)$
Deselect field	NA	NA	NA	NA	$O(1)$	$O(1)$	$O(1)$	$O(1)$
Pause field	NA	NA	NA	NA	$O(1)$	$O(1)$	$O(1)$	$O(1)$
Read field	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
Store field	$O(1)$	NA	$O(1)$	NA	$O(1)$	$O(1)$	$O(1)$	$O(1)$

Table 4.1: Time Complexity for Snapshots Based. NS: Non selected. S: Selected. D: Deselected. P: Paused. NA: Non available.

For the three kinds of versioning, the same ideas are used to achieve efficiency: put the different states of a field in the field itself and order states of fields following a global data structure (list or tree of versions).

4.3 Linear Versioning

In this section we propose efficient data structures and algorithms to implement our linear versioning model. Our solution is an efficient adaptation of the fat node method of Driscoll et al. [Driscoll *et al.* , 1986], described in Section 2.4.2. We adapt their terminology to fit object-oriented languages: nodes of Driscoll et al. are objects, composed of fields that point to other objects.

Driscoll et al. add some extra entries to objects to store the different modifications done on this object. Each object becomes more or less fat depending on the number of saved states. Each entry is composed of the field name, the version number and the value.

We group the changes of a field in a data structure and we put this data structure in this field. Therefore each selected field keeps its own data structure that contains only the different states of this field. Whereas the value of a selected field is saved in each snapshot in the model defined in previous chapter, the efficiency of the model implementation is based on the fact that only the modification of the value of a selected field will be saved in the data structure put in this field. If the value of a selected field is updated two times while

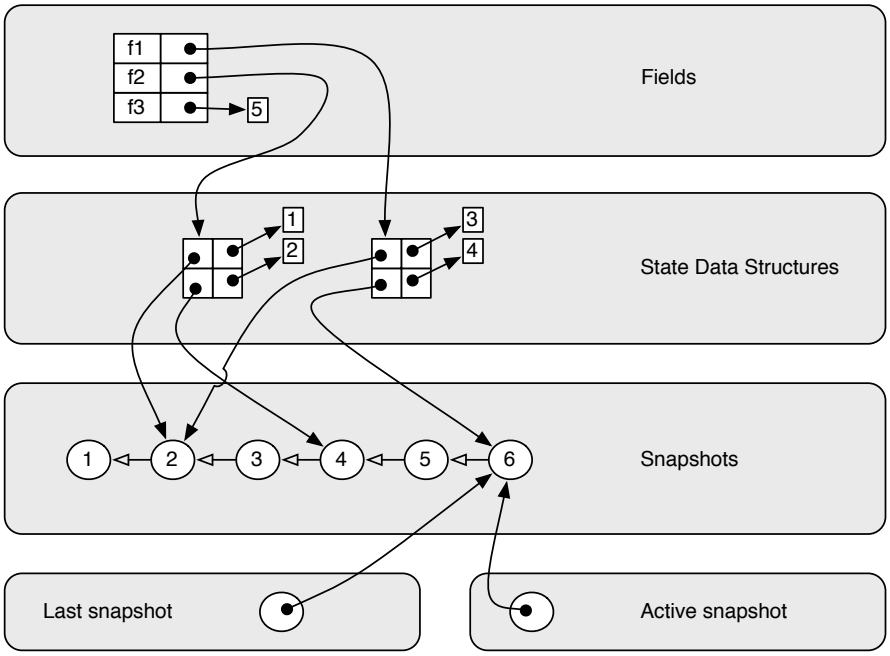


Figure 4.1: The structure of the technique for linear versioning.

the user takes one hundred snapshots, only two states will be saved in the data structure.

4.3.1 Structure Overview

Figure 4.1 shows an overview of the data structures. An object is composed of fields ($f1$,$f2$,$f3$). These fields can be either ephemeral ($f3$ points to the integer 5) or selected ($f1$ and $f2$ point to data structures with states). A selected field has a data structure in which its different states will be stored. Each state associates a snapshot with a value. Finally the snapshots are unique in the system and any field state points to one of them. In this example $f1$ and $f2$ are selected during the snapshot 2 (their first state points to the snapshot 2). The value of $f1$ and $f2$ were respectively 1 and 3. The value of $f1$ is updated at snapshot 4 with the value 2 and the value of $f2$ is also updated at snapshot 6 with the value 4.

The efficiency of the linear versioning depends essentially on performance of data structures that keep states. We develop the *chained array*, a data structure that adds a state

	Active snapshot							
	Read-Only				Writable			
Take a snapshot	NA				$O(1)$			
	State of the field				State of the field			
	NS	S	D	P	NS	S	D	P
Select field	NA	NA	NA	NA	$O(1)$	$O(1)$	$O(1)$	$O(1)$
Deselect field	NA	NA	NA	NA	$O(1)$	$O(1)$	$O(1)$	$O(1)$
Pause field	NA	NA	NA	NA	$O(1)$	$O(1)$	$O(1)$	$O(1)$
Read field	$O(1)$	$O(\log m)$	$O(1)$	$O(\log m)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
Store field	$O(1)$	NA	$O(1)$	NA	$O(1)$	$O(1)$	$O(1)$	$O(1)$

Table 4.2: Time Complexities for linear versioning. NS: Non selected. S: Selected. D: Deselected. P: Paused. m: number of states for this field. NA: Non available.

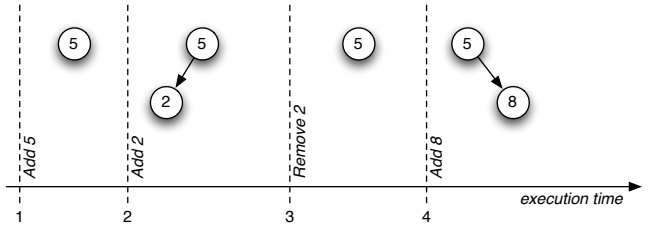


Figure 4.2: Example of a binary search tree with 4 versions.

in $O(1)$, accesses the last state in $O(1)$ and retrieves an old state in $O(\log m)$ time, where m is the number of saved states in the field. Moreover taking a snapshot takes constant time.

We define now the implementation of operations we described for linear versioning, i.e. taking a snapshot, select, deselect and pause a field, read and store a value. Their time bounds are summarized in the table 4.2.

To illustrate the next sections, we use a detailed example of a binary search tree. Figure 4.2 shows an example of four versions of a binary search tree (add 5, add 2, remove 2, add 8). Figure 4.3 shows this example in an ephemeral object form. Each object has three fields `key`, `left` and `right`. The fields `left` and `right` point respectively left and the right subtrees if such trees exist.

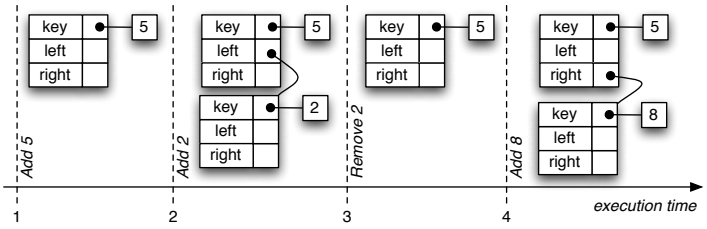


Figure 4.3: The example of Figure 4.2 with nodes seen as objects and their fields `key`, `right` and `left`.

4.3.2 Data structure to keep states

The article of Driscoll et al. [Driscoll *et al.* , 1986] explains that a binary search tree can be used to store the different states of a field, retrieving any state in $O(\log m)$, where m is the number of states saved in the tree. However, this efficiency is respected only if the tree is well-balanced. In the family of well-balanced trees (e.g. red-black tree), their complexity in time (insertion in $O(\log m)$ worst case) and their implementation efforts pushed us to seek a better and easier-to-implement solution. In this section we describe *chained arrays*, a data structure with the following complexities:

This data structure is composed of chained arrays (see Figure 4.4). It stores an extensible array where new elements are appended at the end in $O(1)$ time (assuming constant time memory allocation), and where the number of pointers to follow and the number of comparisons to be performed during a search are both bounded by $\lg n$ in the worst case where n is the number of elements in the array. The space used is $O(n)$.

The structure is composed of a linked list of $\lfloor \lg n \rfloor + 1$ arrays of exponentially decreasing sizes $2^{\lfloor \lg n \rfloor}, 2^{\lfloor \lg n \rfloor - 1}, \ldots, 1$. Each array stores (value, snapshot) pairs in decreasing order of snapshot version number, and all arrays are completely filled except maybe the frontmost and largest array. The frontmost array maintains the position of the snapshot with the largest version number (the last snapshot added).

When storing the first value in an empty structure during initialization, an array of size one is created and the pair (value, snapshot) is stored (Fig.4.4 a). When a new snapshot is taken and the field is updated a second version is generated, an array of size $2{+}1$ (2 place for states and one pointer to the next array) is created and linked with first array (i.e. the last element of the new array points to the old array); the new version is stored along with

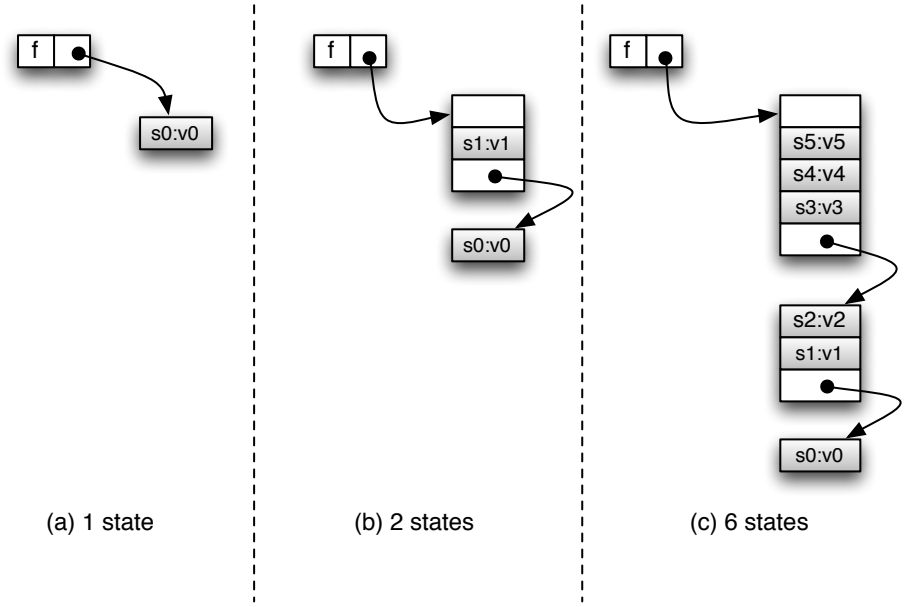


Figure 4.4: The chained arrays: the data structure to store efficiently states of a field

its snapshot at the end of this array (Fig.4.4 b). Further changes fill the frontmost array from back to front until the array is full. When the array is full and the next update occurs, a new array is created whose size is twice that of the previous array, and is linked with that previous array by the last element; the new version is stored in the before last position of the new array (see Fig.4.4 c).

The structure always maintains a pointer to the last state in order to access it in constant time (read and write).

Finally chained arrays lookup a value for a given snapshot in $O(\log m)$ time, where m is the number of stored states (see Algorithm 1). The algorithm is divided into two parts. First, we find the array in the list such that contains the searched state. Because the insertion follows the natural order of version numbers, the version number of the oldest state in this array must be no larger than that of the queried snapshot version number. If no such array is found, an error is thrown because the active snapshot predates the first state saved of this field. Second, we perform a binary search on this array to find the state with the greater version number inferior or equal to the queried version number.

Algorithm 1 ChainArrays:valueAt(vn)

```
array ← lastArray
while array <> NULL AND array[array.size − 1].versionNumber ≤ vn do
  array ← array.nextArray
end while
if array == NULL then
  throw ERROR
end if
return binarySearch(array, vn)
```

Chained arrays offer a good performance for linear versioning: creation takes $O(1)$ time, new states are added in $O(1)$ time, the last version is accessed in $O(1)$ and search the state of a given version number is bounded by $O(\log n)$ in the worst case where n is the number of states in the arrays. Moreover the space used is $O(n)$.

4.3.2.1 Size Bound

We demonstrate in this section the size is upper bounded by $O(n)$. We suppose to be general enough that when a new array is appended to an array of size m, the new array has a size $(1+\epsilon)m$. The total size of the structure to save n elements is :

$$n \leqslant \sum_{i=0}^{k} (1+\epsilon)^i \tag{4.1}$$

in which k is unknown.

The equation 4.1 can be transformed as follows:

$$n \leqslant \frac{(1+\epsilon)^{k+1} - 1}{\epsilon} \tag{4.2}$$

$$(1+\epsilon)^{k+1} \geqslant \epsilon \cdot n + 1 \tag{4.3}$$

$$(k+1)\log(1+\epsilon) \geqslant \log(\epsilon \cdot n + 1) \tag{4.4}$$

$$k \geqslant \frac{\log(\epsilon \cdot n + 1)}{\log(1+\epsilon)} - 1 \tag{4.5}$$

$$k \geqslant \log_{(1+\epsilon)}(\epsilon \cdot n + 1) - 1 \tag{4.6}$$

The first integer k respecting this condition is :

$$k = \lceil \log_{(1+\epsilon)}(\epsilon \cdot n + 1) - 1 \rceil \tag{4.7}$$

This number can be bounded by :

$$\lceil \log_{(1+\epsilon)}(\epsilon \cdot n + 1) - 1 \rceil \leqslant \log_{(1+\epsilon)}(\epsilon \cdot n + 1) \tag{4.8}$$

If we replace this k in 4.2, we have :

$$n \leqslant \frac{(1+\epsilon)^{\lceil \log_{(1+\epsilon)}(\epsilon \cdot n+1)-1 \rceil+1}-1}{\epsilon}$$

and the total size is limited by (using 4.8):

$$\frac{(1+\epsilon)^{\lceil \log_{(1+\epsilon)}(\epsilon \cdot n+1)-1 \rceil+1}-1}{\epsilon} \leqslant \frac{(1+\epsilon)^{\log_{(1+\epsilon)}(\epsilon \cdot n+1)+1}-1}{\epsilon} \tag{4.9}$$

$$\leqslant \frac{(1+\epsilon)(\epsilon \cdot n+1)-1}{\epsilon} \tag{4.10}$$

$$\leqslant (1+\epsilon)n + 1 \tag{4.11}$$

4.3.3 Taking a snapshot

The different states of a field are saved in the field itself rather than in each snapshot as described in the model. Therefore a snapshot contains only two pieces of information: a version number and a dictionary of properties (usable for the user to store extra information about the snapshot).

Starting the system, two global variables are created: `activeSnapshot` and `lastSnapshot` (Figure 4.1, page 99). We initialize them by creating a new snapshot with the version number 1.

When a new snapshot is taken, a snapshot is created. Its version number is the version number of the last snapshot incremented of one. Both global variables `activeSnapshot` and `lastSnapshot` are updated with new snapshot. It is not needed to keep precedence links between snapshots: the total order of snapshots follows the natural order of their version number.

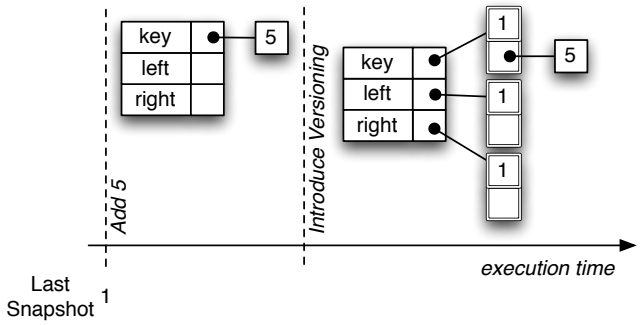


Figure 4.5: Selection of three fields

As explained in the model, a snapshot can be taken only if the active snapshot is writable, i.e. we are in the last version of the system. It is not necessary to store a boolean in each snapshot to know if the writability of a snapshot: only the snapshot pointed to the variable `lastSnapshot` is writable. All other ones are read-only.

The complexity in the worst case of taking a snapshot is $O(1)$.

4.3.4 Selecting Fields

When a field f is selected, we replace its current value by a new chained array to store its different states. This data structure is initialized with a new state that associates the current value of f and the active snapshot. Therefore the snapshot of the first state stored in any chained array is the snapshot during which the field has been selected.

Because the value of selected fields is a specific data structure and the non selected fields has other values (any other objects), it is sufficient to determine at run time if a field is selected or not. We discuss about the implementation of this part in statically and dynamically typed languages in Section 5.1.1.1, page 142.

Figure 4.5 shows the selection of the root of our binary search tree. Each field is selected: their value is replaced by a new chained array with a state associating the current snapshot 1 and the value of the field. Null pointers are omitted for more readability.

The complexity of the creation of the chained arrays being $O(1)$, the selection takes a

worst case constant time[1].

4.3.5 Deselecting and Pausing

When a selected field f is deselected or paused, a boolean variable in the corresponding chained array is updated. In the model described in Chapter 3, we use the value of the field to put the ephemeral value of the field while the states remain untouched in the snapshots. Here the value of field is already used to keep the data structure. Therefore we add a variable `ephemeralValue` to the data structure to store this value. This variable receives the value of the last state each time the field is paused or deselected.

When a paused field (respectively a deselected field) is deselected (respectively paused), the booleans used to know its selection state are updated. No other operation is necessary.

Because we can access the last state in constant time, deselecting and pausing a field also takes constant time.

4.3.6 Reselecting

When a paused or a deselected field is reselected, we look at the chained arrays d stored in this field. If the last state saved in d is associated with the active snapshot, its value is replaced by the value contained in the variable `ephemeralValue`. Otherwise we add in d a new state that associates the value `ephemeralValue` with the active snapshot.

Reselection takes constant time.

4.3.7 Ephemeralizing

A selected, paused or deselected field can be transformed into an ephemeral field again by deleting all its saved states. If the field is selected, the value of the last saved state replaces the chained array in the field. If the field is paused or deselected, the value of `ephemeralValue` becomes the value of the field.

This operation takes constant time.

[1] As said in the introduction of this chapter, we discuss the automatic selection in a further section (Section 4.7).

4.3.8 Storing a Value in a Field

A new value is stored in a field f. This field can be ephemeral, selected, deselected or paused while the active snapshot can be read-only or writable.

If this field is ephemeral, the store is performed as in any ephemeral system: the new value replaces the old one.

If the field is selected or paused, the active snapshot is considered. If this is a read-only one, an error is raised: a modification on a selected or paused field is not allowed when the active snapshot is not the last one.

If the active snapshot is writable, we inspect the value of f, i.e. an instance d of chained arrays. Here we distinguish a paused field and a selected field. If a new value is asked to be stored in a paused field while the active snapshot is writable, the value of `ephemeralValue` in d is updated to the new value. This is done in constant time.

If the field is selected and the active snapshot is writable, we compare the snapshot associated with the last state stored in d and the snapshot in the global variable `lastSnapshot` : either their version numbers are equal or the version number of the last snapshot is greater[2]. If they are equal, 2 values are stored in the same field during the same snapshot: the value of the last state is replaced by the new value. The old value is forgotten and there is no way to retrieve it. On the other hand, if the version numbers are different, a new state is added in d that associates the new value with the active snapshot. This is done in constant time in both cases.

If the field is deselected, the new value replaces the `ephemeralValue` in the chained arrays, independently of the writability of active snapshot.

Accessing the last state and inserting a new state in a chained array are done in $O(1)$ time (Section 4.3.2). Other operations (accessing to the active and last snapshots and comparing integers) are done in constant time too. Therefore storing a value in a field takes constant time.

Figure 4.6 shows how the selected fields of the root of our binary search tree evolves throughout snapshots and updates. Take the example of the evolution of the field `left` of the root node. After take a snapshot 2, the node 2 is added: the chained array contained

[2]Note that the version number of the last snapshot can not be smaller than any version number of existing states.

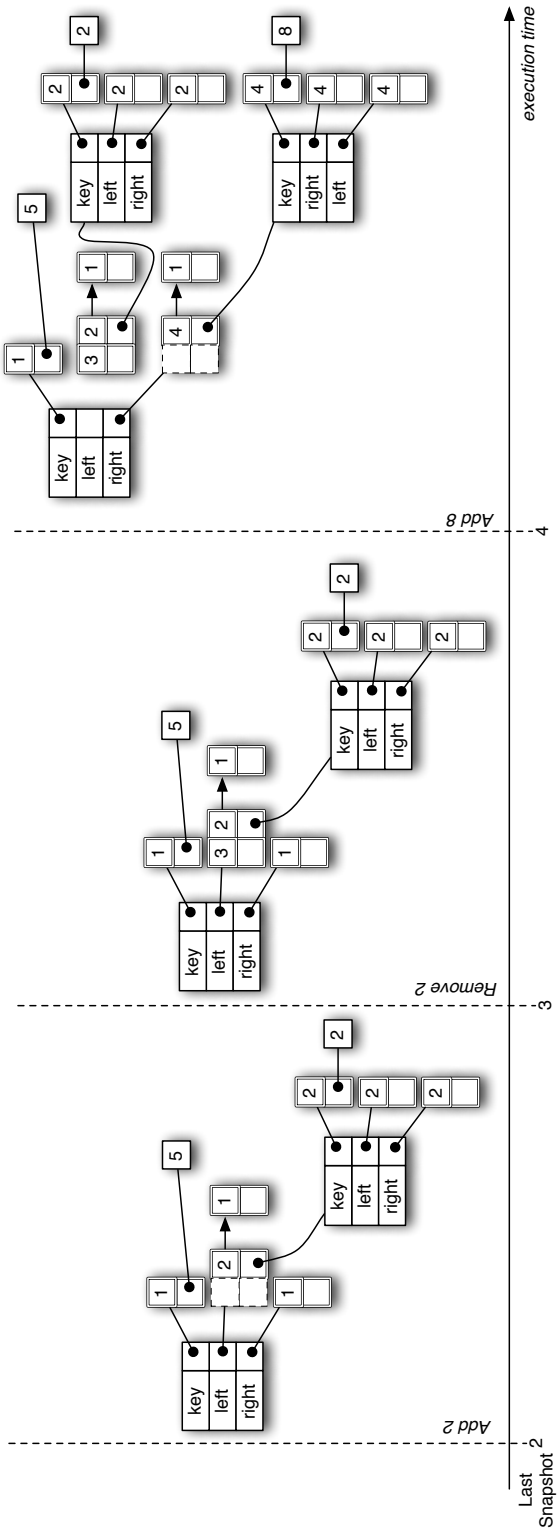


Figure 4.6: Evolution of the internal mechanism of the selected binary tree for the following operations: take a snapshot 2, add 2, take a snapshot 3, remove 2, take a snapshot 4 and add 8.

in the field is extended to store the state that associates the snapshot 2 and the pointer to the node 2. After a new snapshot 3 and the deletion of the node 2 (by putting the object `null` in the field), the chained array is filled with the state that associates the snapshot 3 and the object `null`.

4.3.9 Reading a Field

When the value of a field is read, this field can be ephemeral, selected, deselected or paused while the active snapshot can be read-only or writable.

If the field is ephemeral, the current value of the field is returned.

We consider now the field is selected or paused. It contains a chained array d that keeps its different states. If the active snapshot is read-only, we consider the version number v associated with the active snapshot. The value to return is associated to the newest state with version number smaller or equal to v. This operation takes $O(\log m)$ time where m is the number of states in the data structure linked with this field.

The active snapshot is the writable one and the field is selected, the value associated with the last state in d is returned. This operation is done in constant time. If the field is paused, the value of the `ephemeralValue` stored in d is returned.

If the field is deselected, the value of the variable `ephemeralValue` in the chained arrays of this field is returned, independently of the writability of the active snapshot. This is done in constant time.

4.3.10 Cache

We observed that consecutive retrievals of the same version of a field always have to traverse the chained data structure (Figure 4.7). Therefore, we decided to add a cache. The cache holds a single key-value pair consisting of the last retrieved value of the field and the corresponding snapshot. Consecutive queries for the same snapshot therefore no longer traverse the chained arrays but immediately return the value. 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.

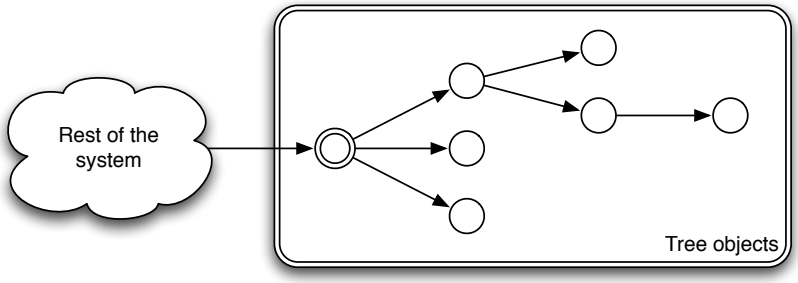


Figure 4.7: Example in which the cache is crucial

4.3.11 Discussion

In this section we showed how our linear versioning model can be implemented efficiently in time and in space. This technique is inspired from Driscoll et al. [Driscoll *et al.* , 1986].

The data structures used by Driscoll et al. to store states in objects are not clearly defined: general trees are evoked without further precision. We introduce chained arrays to reduce yet the time to save states in practice.

This technique has been developed in order to obtain an efficient implementation. Selecting fields, taking snapshots and saving new states take constant time worst case: recording the past of an existing application has a constant slowdown, regardless of the number of snapshots/states saved. Moreover, the simplicity of chained arrays allows one to implement them easily and with a very small number of operations. Notice that the time to access ephemeral fields is nearly the same time without linear versioning. The cache mechanism we introduce is lightweight and reduces the complexity on successive read accesses of the same version number.

Browsing the past takes more time (logarithmic on number of states) but we made the realistic assumption that this operation will be less frequent than the recording process.

4.4 Backtracking Versioning

To our best knowledge there is no study to transform efficiently any ephemeral object into a backtrackable one. In this section we propose a new method to implement efficiently

	Active snapshot	
	Read-Only	Writable
Take a snapshot	NA	$O(1)$
Backtrack	$O(s)$ $O(1)$*	$O(1)$

	State of the field				State of the field			
	NS	S	D	P	NS	S	D	P
Select field	NA	NA	NA	NA	$O(1)$	$O(1)$	$O(1)$	$O(1)$
Deselect field	NA	NA	NA	NA	$O(1)$	$O(\log n)$ $O(1)$*	$O(1)$	$O(1)$
Pause field	NA	NA	NA	NA	$O(1)$	$O(\log n)$ $O(1)$*	$O(1)$	$O(1)$
Read field	$O(1)$	$O(\log n)$	$O(1)$	$O(\log n)$	$O(1)$	$O(\log n)$ $O(1)$*	$O(\log n)$ $O(1)$*	$O(\log n)$ $O(1)$*
Store field	$O(1)$	NA	$O(\log n)$ $O(1)$*	NA	$O(1)$	$O(1)$*	$O(\log n)$ $O(1)$*	$O(\log n)$ $O(1)$*

Table 4.3: Time complexities for backtracking versioning. NS: Non selected. S: Selected. D: Deselected. P: Paused. NA: Non available. s: Number of taken snapshots. Starred complexities are amortized.

the model of backtracking versioning we described in the previous chapter. The achieved time complexities are shown in Table 4.3. The size is bounded by the number of saved states and the number of backtracks.

We explain how efficiently take a snapshot, select, deselect, pause and ephemeralize fields, read a field, store a new value in a field and backtrack to a read-only snapshot. These algorithms and data structures are based on linear versioning.

To illustrate each operation we use example of a binary search tree (Figure 4.8). A tree is built at steps 1 and 2 with two nodes with respectively the keys 5 and 2. After the snapshot 3 the node with the key 2 is removed from the tree. A snapshot 4 is taken and a new node with the key 8 is added. The user asks then to backtrack the system at snapshot 2. Therefore the system forgets all states saved from the snapshot 3 (these steps are in grey). The last available state of the tree is at snapshot 2 with the keys 5 and 2. Finally a new snapshot 6 is taken and a node with the key 4 is added in the tree. The final state of the tree is composed of three nodes with the keys 2, 4 and 5.

Figure 4.9 shows the same example but with ephemeral objects, composed of three fields: the key, the left subtree and the right subtree. It is how the developer sees the system

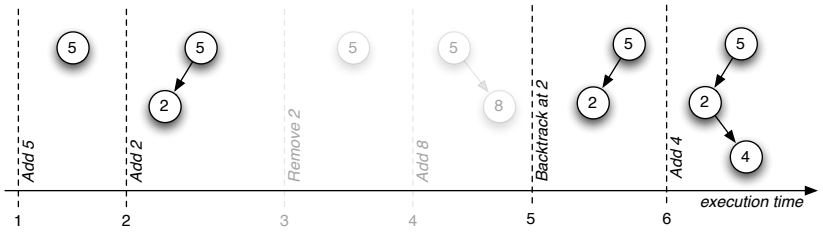


Figure 4.8: An example of backtracking on a tree.

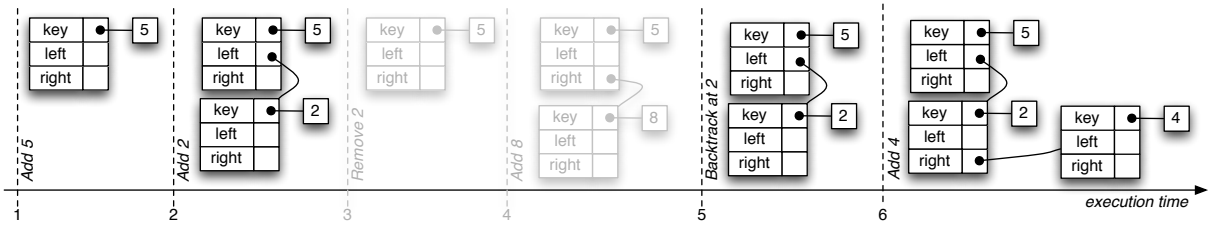


Figure 4.9: The same example seen from Figure 4.8 with objects.

without the internal mechanism we will explain in this section. `Null` pointers are omitted for more readability.

4.4.1 Structure Overview

To be efficient, we use a technique close to the linear versioning. Figure 4.10 shows the global structure of our solution on a synthetic example. Ephemeral fields store their value like in any ephemeral system. A selected field saves its states in a chained array. The two variables `lastSnapshot` and `activeSnapshot` keep the same usage as in linear versioning.

The developer takes snapshots as in linear versioning. All snapshots taken after a snapshot s are here named the successors of s.

The developer can take a read-only snapshot s and ask the *backtrack* of the system to s, i.e. delete all field states saved by the successors of s. We call s the target snapshot of the backtrack. All successors of s are then marked as *backtracked* and the variables

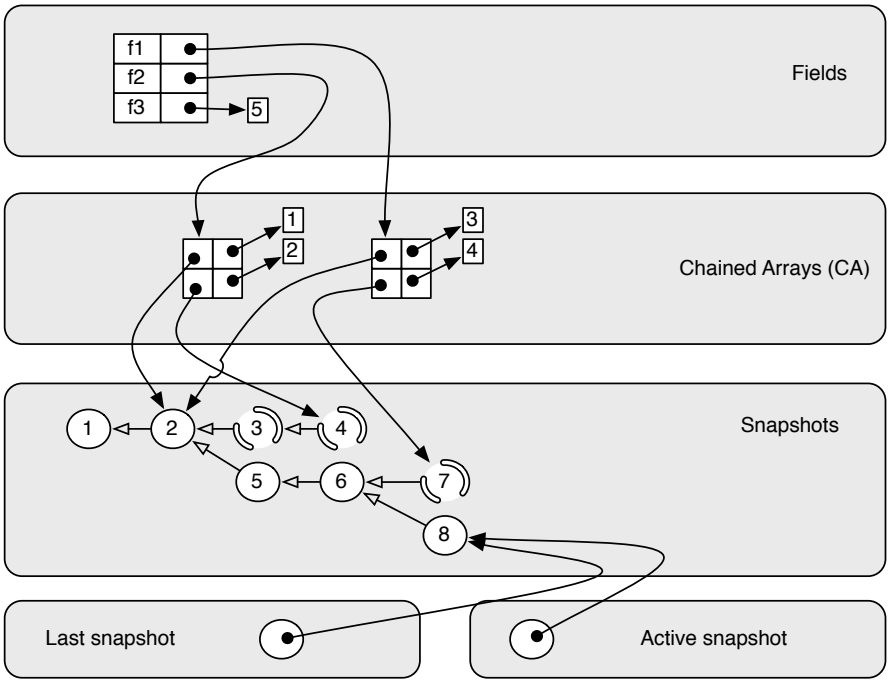


Figure 4.10: The structure of the technique for backtracking versioning. This example is the result of the following sequence of operations. The fields f1 and f2 contain respectively the values 3 and 1. They are selected at snapshot 2. Two snapshots (3 and 4) are then taken. The value 2 is put in f2. A backtrack to the snapshot 2 is asked. Three snapshot 5, 6 and 7 are taken. The value 4 is put in the field f1. A backtrack is performed on snapshot 4. A new snapshot 8 is taken.

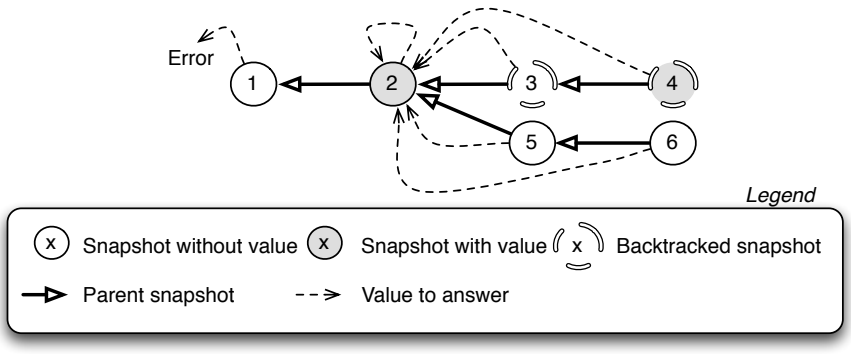


Figure 4.11: Example of the states of a field, with values and without values

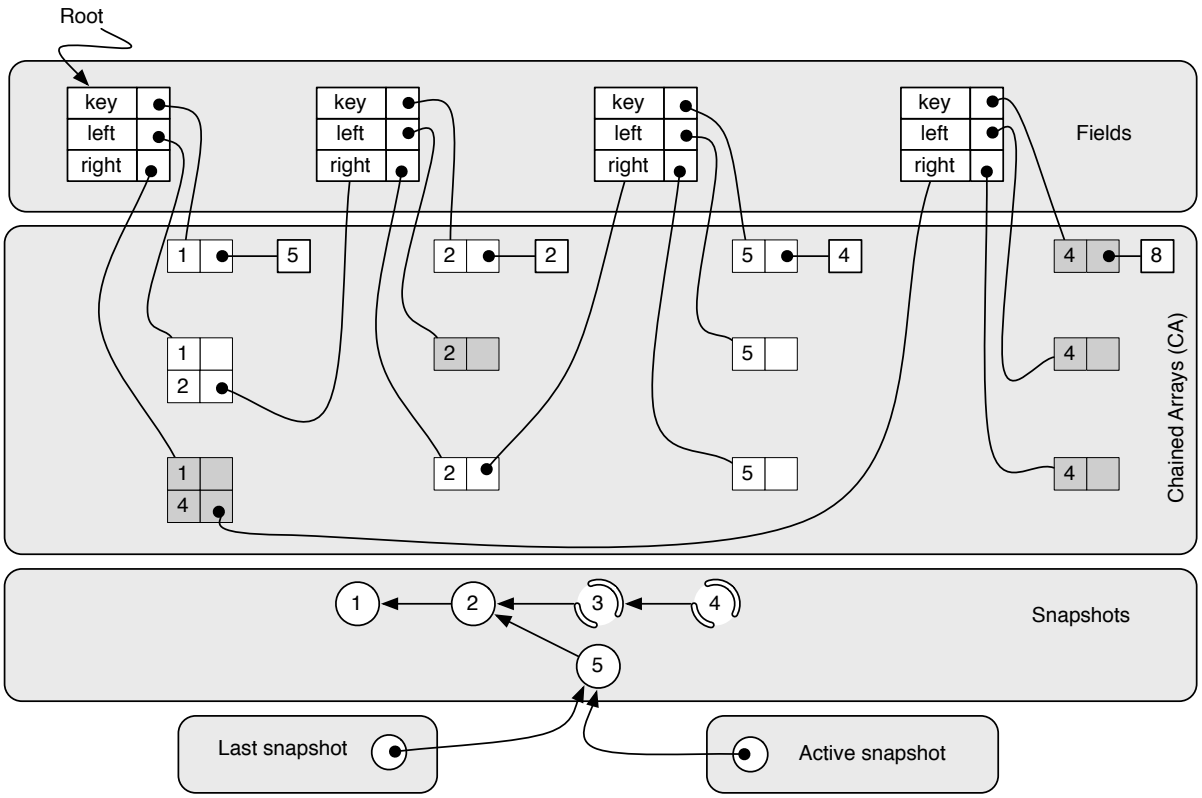


Figure 4.12: The structure of the system for the final state of Figure 4.9. To avoid a large number of arrows and improve the readability, the number of the snapshots replace the arrows from the chained arrays to the corresponding snapshots.

`activeSnapshot` and `lastSnapshot` are updated to point to s.

When a value is asked for a given snapshot s, the system returns the value associated with the most recent snapshot such that it is created before or at s and it is not backtracked. If there is no such snapshot, an error is thrown. Figure 4.11 shows the result of queries for the value of a field that has only two states at snapshots 2 and 4.

Notice all snapshots form a tree because of backtracks: a branch appears after each snapshot used as target for a backtrack. However there is always only one path from the last non backtracked snapshot to the first snapshot of the system. The snapshots that do not belong to this path have been backtracked.

When a backtrack of the system is performed, we could browse all chained arrays in fields to remove their possible backtracked states but it would take linear time in term of selected fields. In most cases, this would be too expensive. We prefer to delete the backtracked states of a field only when this field is accessed. Therefore the backtracked states remain in the chained array until it is accessed. Because we postpone the deletion of backtracked states, we must keep enough information about backtracks to find the states to remove when a field will be accessed. To achieve this, when a backtrack to a snapshot s is asked, snapshot s and all its successors are marked as backtracked.

Before each access to a chained array, the states corresponding to backtracked snapshots must be removed from the structure. To determine whether such states exist, we must examine the snapshot s associated with the last state. If s is not marked as backtracked then the structure can be accessed without further modification. By construction, if the snapshot is marked as backtracked, then the structure must be cleaned. We must then find the most recent snapshot s_o such that it is created before s and it is not backtracked. We remove then all the states associated with snapshots created after s_o. A complete description of this operation is given in Section 4.4.6.1 (page 119).

In next sections we explain how to implement all possible operations. To illustrate each operation, we show how an empty system evolves step by step to get the final step of Figure 4.9 presented in Figure 4.12.

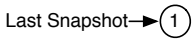


Figure 4.13: The system is initialized with a first snapshot.

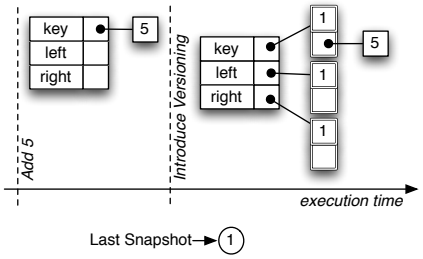


Figure 4.14: An ephemeral node (at left) is selected (at right).

4.4.2 Initialization

At initialization of the system, a first snapshot is created (Figure 4.13). The variables `activeSnapshot` and `lastSnapshot` point at it. This is done in constant time.

4.4.3 Selecting Fields

When a field is selected, the active snapshot must be writable. This field can be ephemeral, selected, deselected or paused.

If we select an already selected field, nothing must be done.

When an ephemeral field is selected, a new chained array is created and initialized with one state that associates the current value of the field with the active snapshot. This chained array is put as value of the field. Figure 4.14 (at left) shows how the root node of the tree of Figure 4.9 becomes versioned (at right): a chained array is put in each selected field. We use integers for snapshots to avoid too numerous arrows in figures. The chained arrays are initialized with a state that associates the current value of the field with the last snapshot (the 1 in each array). This operation is performed in constant time.

If the field is deselected or paused, we first clean the possible backtracked states in the chained array in $O(1)$ amortized time (see Section 4.4.6.1, page 119). Once cleaned, the

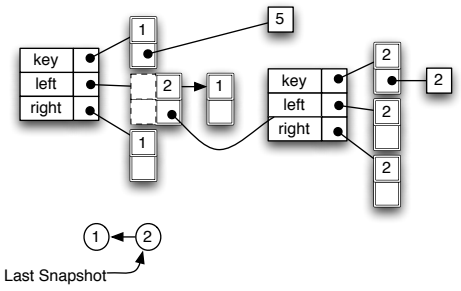


Figure 4.15: Add the node with the key 2. The chained array of the field `left` of the root is extended as well to add the new state.

selection of a deselected or a paused field is performed like in linear versioning ($O(1)$).

4.4.4 Taking a snapshot

When a snapshot is taken, the variables `lastSnapshot` and `activeSnapshot` are replaced by the new snapshot. This operation remains unchanged from the linear versioning and its time is always bounded by $O(1)$.

4.4.5 Storing a Value in a Field

The store of a new value in a paused, deselected or ephemeral field is performed exactly as in linear versioning. No cleaning of the structure must be done because we do not access it (the variable *ephemeralValue* is directly used for paused or deselected field) and it is performed in constant time.

Before the store of a new value in a selected field, we must clean the possible backtracked states contained in the chained array in $O(1)$ amortized time (see Section 4.4.6.1). Once cleaned, the operation is done like in linear versioning ($O(1)$).

In Figure 4.15 the snapshot 2 is taken. For the three next operations (adding node 2 (Figure 4.15), taking snapshot 3, removing node 2 (Figure 4.16), taking snapshot 4 and adding node 8 (Figure 4.17)), chained arrays are updated exactly like explained in

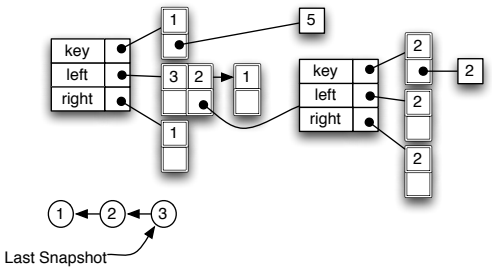


Figure 4.16: The node with the key 2 is removed and a state, that associates the snapshot 3 and the object `null`, is added in the chained array of field `left` of the root.

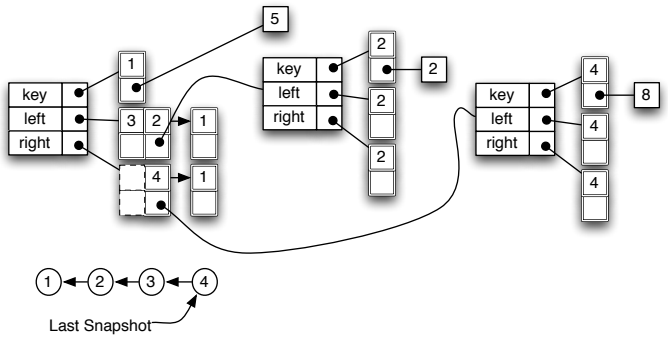


Figure 4.17: Add the node with the key 8. The chained array of the field `right` of the root is extended as well to add the new state.

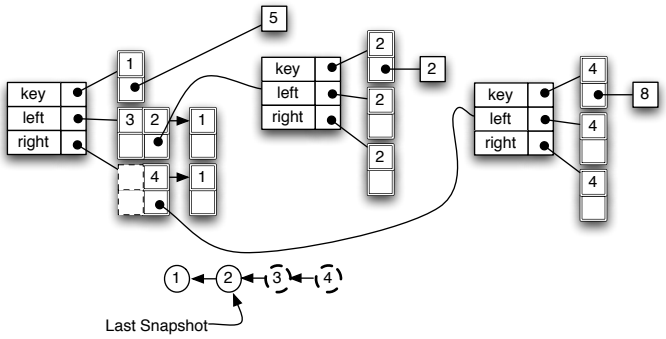


Figure 4.18: The system is backtracked at snapshot 2. The snapshots 3 and 4 are marked as backtracked and the active and last snapshots pointers are updated to snapshot 2. The chained arrays remain untouched.

Section 4.3 for linear versioning.

4.4.6 Backtrack

When a backtrack at snapshot s is asked, all successors of s are marked as backtracked. The backtrack operation is clearly bounded by $O(s)$ where s is the number of taken snapshots.

Figure 4.18 shows the system of our tree example after the backtrack to the snapshot 2. The snapshots 3 and 4 are marked as backtracked (their outline is dashed). The last and active snapshots point at snapshot 2.

4.4.6.1 Cleaning Backtracked States in a Chained Array

When a chained array is accessed, its possible backtracked states must be removed (Algorithm 2). To find out if there is such a state, we look at the snapshot s associated with the last state. If it is not marked as backtracked, then we have nothing to do. Otherwise we remove efficiently all states associated to a backtracked snapshot in a chained array (lines 5-9). First, we delete arrays such that the snapshot of the earliest state is associated with a backtracked snapshot. The earlier state is the first state added in the array, placed at the second last position (the last position is for the link to the next array). Therefore

Algorithm 2 Backtracking Versioning:cleanStates(f)

```
1: s ← f.lastStates.snapshot
2: if s is not backtracked then
3:    return
4: end if
5: array ← f.chainedArray.lastArray
6: while array[array.size - 1].snapshot is backtracked do
7:    chainedArray.lastArray ← array[array.size]
8:    array ← chainedArray.lastArray
9: end while
10: chainedArray.freeIndex ← binarySearchIndex(array)
```

the first array with the snapshot of the earliest state associated with a non backtracked snapshot becomes the last array of the chained arrays. Second, we perform a binary search on it to find the states to keep inside this array.

Cleaning the backtracked states in a chained array is bounded by $O(\log m)$ worst case and $O(1)$ amortized where m the number of states stored in the chained array.

4.4.7 Reading a Field

Reading an ephemeral or a deselected field is performed exactly as in linear versioning. This operation is done in constant time.

For a selected or a paused field, the backtracked states must be cleaned if necessary. Once cleaned, the read is then realized as in linear versioning. Both operations are done in $O(\log m)$ time, where m is the number of states in the chained arrays.

In Figure 4.19, we show the result after a new snapshot and the insertion of 4 in the tree. The grey chained arrays are accessed (read or store) during the insertion. The states taken after the snapshot 2 are removed. The white chained arrays are not accessed during the insertion. They contain always non pertinent states (e.g. the state of the field `right` of the root points to the old node with the key 8). They will be cleaned only when they will be accessed.

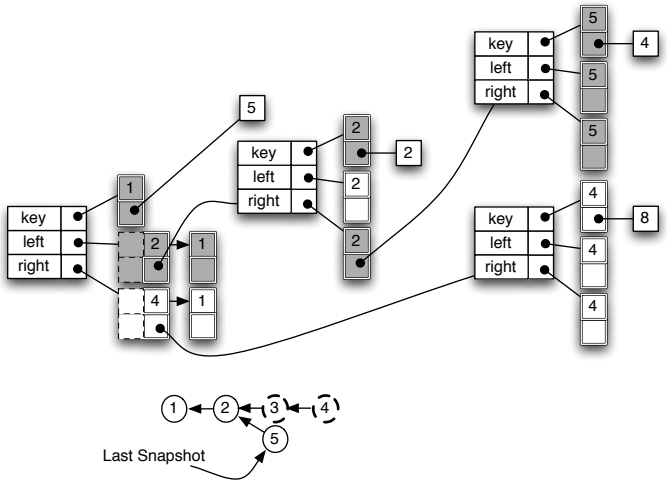


Figure 4.19: Add a node with key 4. The chained arrays of selected fields accessed during the operation are cleaned of backtracked states (they are grey). The white chained arrays have been not accessed during the operations.

4.4.8 Deselecting, Pausing and Ephemeralizing a Field

The deselection, pause or ephemeralization of a deselected or paused field is performed exactly as described in linear versioning (in constant time).

If the field is selected, the states of the chained array must first be cleaned of possible backtracked states ($O(b+\log m)$). Once cleaned, the last state can be queried in constant time, as defined in linear versioning.

4.4.9 Cache

The cache is implemented exactly as in linear versioning. However, the cache must be flushed each time backtracked states are deleted from the chained array.

4.4.10 Discussion

We presented in this section the first efficient implementation of general backtracking versioning. It allows the transformation of any ephemeral data structure into a versioned structure such that its states can be saved and backtracked efficiently.

	Active snapshot							
	Read-Only				Writable			
Take a snapshot	$O(1)$*				$O(1)$*			
	State of the field				State of the field			
	NS	S	D	P	NS	S	D	P
Ephemeralize field	NA	NA	NA	NA	$O(1)$	$O(\log m)$	$O(1)$	$O(1)$
Select field	NA	NA	NA	NA	$O(1)$	$O(1)$	$O(\log m)$	$O(\log m)$
Deselect field	NA	NA	NA	NA	$O(1)$	$O(\log m)$	$O(1)$	$O(1)$
Pause field	NA	NA	NA	NA	$O(1)$	$O(\log m)$	$O(1)$	$O(1)$
Read field	$O(1)$	$O(\log m)$	$O(1)$	$O(1)$	$O(1)$	$O(\log m)$	$O(1)$	$O(1)$
Store field	$O(1)$	$O(1)*$	$O(1)$	$O(1)$	$O(1)$	$O(\log m)$	$O(1)$	$O(1)$

Table 4.4: Time complexities for branching versioning. NS: Non selected. S: Selected. D: Deselected. P: Paused. NA: Non available. Starred complexities are amortized.

We design our solution with a practical point of view. There is no slowdown when we work with the ephemeral fields. The time and space complexities are the same than the linear versioning if no backtrack is performed. The backtrack operation takes amortized constant time and backtracked states are removed only when necessary.

We think these complexities are well designed for a practical usage of backtracking versioning: operations take time only when there are used.

4.5 Branching Versioning

The branching versioning is the most complex kind of versioning studied in this document. Driscoll et al. [Driscoll *et al.* , 1986] describes the fat node method for the full persistence, a technique to transform any ephemeral structure into a versioned structure allowing branching operation. We explained it in Chapter 2 (Section 2.4.4, page 28).

In this section we describe how to implement the fat node method, specially designed for our model, with the complexities of Table 4.4.

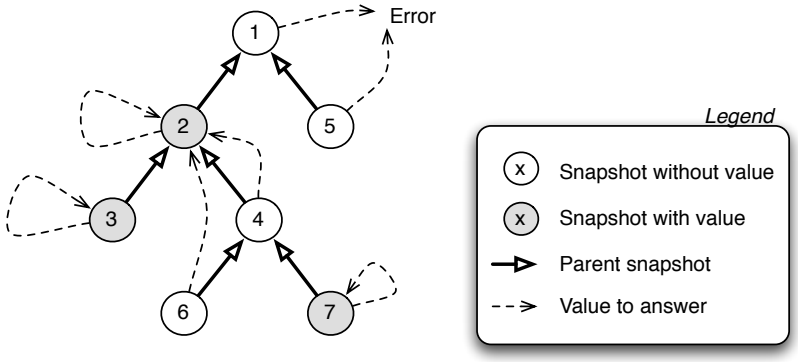


Figure 4.20: Example of the states of a field, with values and without values

4.5.1 Structure Overview

A list of snapshots, like for linear versioning, is not enough for the branching versioning because multiple snapshots can be created from a given snapshot. They are naturally organized in a tree, the *global tree of snapshots* (GTS)[3], where each snapshot s is a child of the snapshot from which s is created.

On the other hand, as in linear and backtracking versioning, each field keeps its own states in a data structure stored directly in this field. These states are maintained in a local tree to find quickly a state for a given snapshot. When the user activates a read-only snapshot s and queries the value of a selected field, the system must return the value contained in the field during s.

Figure 4.20 shows an example of a GTS. The three grey nodes highlight the snapshots for which there is an associated value for a field f. The local tree put in f will be composed only these three nodes. The dashed arrows points the value to return when f is queried and the snapshot at origin of the arrow is the active one. For instance, when the snapshot 6 is the active one, the value saved for the snapshot 2 must be returned: the snapshot 2 is the lowest ancestor of the snapshot 6 such that there is a value for f.

The GTS offers only a partial order between snapshots. However we need a total order between all snapshots to order states in local trees. A list, the *global list of snapshots* (GLS), will define a total order. This list pre-order versions by inserting each version just after its

[3]GTS is usually called *global version tree* in the literature but for consistency reasons, we chose GTS.

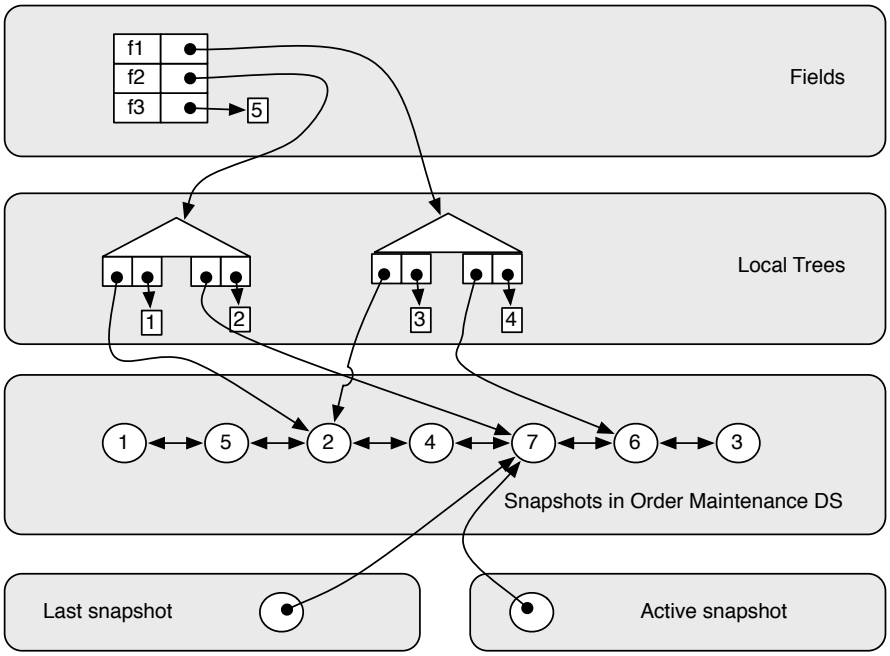


Figure 4.21: Structure of the technique for the branching versioning

parent. The version tree of Figure 4.20 has the corresponding version list 1,5,2,4,7,6,3.

The GLS is stored in an Order Maintenance structure (OM) [Bender *et al.* , 2002], i.e. a structure that answers a query on precedence between two elements in $O(1)$. To achieve it, the list is transformed into a doubled-linked list where each node can access its direct neighboring nodes in constant time. A label is attached to each node: these labels are ascending integers from the first node to the last node of the list. To know if a node precedes another one in the OM, we compare their labels: if a node A is before the node B, the label of A should be smaller than the label of B. When a new snapshot must be added, the labels of nodes must be updated whenever a new snapshot is added between two snapshots with two successive labels (the difference of the two labels equals 1): range of nodes around the inserted position must be relabelized [Bender *et al.* , 2002] to ensure logarithmic amortized time for the OM insertion. This bound can be improved to $O(1)$ amortized time by using a technique of two-level data structure, where the bottom level has $\Theta(\log n)$ elements and the top level has $\Theta(n/\log n)$ [Bender *et al.* , 2002].

Because the GTS and the GLS contain same information, the system must maintain only

the GLS. Figure 4.21 shows the global structure of our method. A selected field will contain a local tree in which its different values will be associated with the corresponding snapshot in GLS. The active and last snapshots point to a snapshot as in linear and backtracking versioning.

4.5.2 Initialization

At initialization of the system, the GLS is composed of snapshot 1, that is the first snapshot in the system. The active and last snapshot point at it.

4.5.3 Taking a snapshot

A snapshot s_n can be taken when the active snapshot s_a is a read-only or a writable snapshot. The active snapshot s_a is replaced by the new snapshot s_n. We add the snapshot s_n before the snapshot s_a in the global list. This operation takes amortized constant time and a worst case constant space [Bender *et al.* , 2002].

4.5.4 Data Structure to Keep States

Each selected field keeps its own states in a complete search tree. In our implementation we use 2-4 trees to be efficient [Cormen *et al.* , 2001]. However we use a complete binary search tree in this section for more readability. Each leaf keeps one state while each internal node has a node of the GLS as key. We denote by T_f the local field tree associated with the field f.

Each leaf keeps pointers to the previous leaf and the next leaf. This operation is done in constant time during the creation of a new leaf. That allows us to access the direct neighboring leaves in constant time during the insertion of a new value in the local tree.

4.5.5 Selecting an Ephemeral Field

When an ephemeral field f with a value v is selected with the active snapshot s_a, a new empty local tree T_f is created in which the value v is added as described in Section 4.5.8.

Moreover we add a variable `ephemeralValue` to T_f, used when the field is deselected or paused. This operation takes constant time worst case.

4.5.6 Deselecting, Pausing and Ephemeralizing a Field

When a selected field f is deselected or paused, we use two booleans as in linear and backtracking versioning to register this selected state. Moreover we read the value v for the active snapshot in T_f as described in Section 4.5.9 in logarithm time and we store v in the variable `ephemeralValue`.

If a selected field is ephemeralized, we read the value v for the active snapshot in T_f and we put this value in the field itself, replacing the local tree.

If a paused or deselect field is ephemeralized, we put the value `ephemeralValue` in the field itself.

4.5.7 Selecting Deselected and Paused Fields

When a deselected or paused field is reselected at snapshot s_a, we store the value `ephemeralValue` associated to the snapshot s_a in T_f as described in Section 4.5.8. It takes logarithmic time.

4.5.8 Storing a Value in a Field

If the field is ephemeral, the new value is put in the field itself in constant time.

If the field is deselected, the new value is stored in the variable `ephemeralValue` in constant time.

When a new value v is put in a selected or paused field f at snapshot s_a, we lookup the key s_a in T_f, using the order defined by GLS, until we are either on a leaf l or completely at right of all leaves. If l points to s_a, the value associated in l is replaced by v.

If we are at right of all leaves or the snapshot of l is not s_a (i.e. there is no state for s_a in T_f), we add a leaf that associates s_a with v. This leaf is added either just before l in T_f or at right of all leaves. We also maintain two pointers in this leaf to allow access its direct neighbors in constant time. If we inserted the leaf before another, we must check if

the insertion does not interfere with the states already saved. We need three snapshots: the snapshot s_p of the previous leaf, the snapshot s_n of the next leaf and the snapshot s_+ in GLS just at right of the snapshot of l. These snapshots are accessed in constant time. If s_+ is before s_n in GLS (or if s_+ exists but s_n does not exist), we must add a new leaf with the value of previous leaf associated to the snapshot s_+.

If we use a 2-4 tree, updating the value of a selected field takes $O(\log n)$ time, where n is the number of states saved in T_f.

4.5.9 Reading a Field

If the field is ephemeral, its value is returned. If the field is deselected, the value in `ephemeralValue` is returned. If the field is paused and the active snapshot is writable, the value in `ephemeralValue` is also returned.

If the field is selected or if the field is paused and the active snapshot s_a is read-only, we look at the value of the field f at snapshot s_a in T_f using the snapshot s_a and the order defined by GLS. This search returns a leaf l. If the snapshot of l is before s_a in the GLS, an error is thrown. Else the value contained in l is returned. Because the comparison of two nodes in the GLS is performed in constant time, this operation takes a $O(\log m)$ time where m is the number of states in T_f.

4.5.10 Cache

We use the same mecanism of cache than for other kinds of versioning. A pair snapshot-value is maintained in each local tree. It is updated each time we browse the tree to found a value for a given snapshot s. Before performing a search in the local tree, if the cached snapshot is the same than the queried snapshot, the cached value is returned in constant time. Otherwise the tree is browsed and the result is cached.

The cache is flushed each time an update is performed in the tree.

4.5.11 Discussion

We seen in the previous sections the theoretically efficient data structures and algorithms to keep states for all kinds of versioning.

We study now the other elements of our model, i.e. the snapshot sets and the automatic selection.

4.6 Snapshot Sets

The snapshots can be grouped in snapshot sets. A snapshot set supports the following queries:

Add/Remove snapshot to manage which snapshots are contained in the set;

All snapshots returns all snapshots of the set;

Root snapshots returns all snapshots without parent in the set;

Previous snapshot returns the first parent, contained in the set, of a given snapshot on the path from the given snapshot to the root, if a such snapshot exists;

We show in this section efficient structures of snapshot sets for each kind of versioning.

4.6.1 Linear Versioning

In linear versioning, a snapshot set stores its snapshots in a double linked list ordered on their version numbers. Such a list inserts new snapshot, deletes a snapshot and retrieves previous snapshots in worst-case $O(1)$ time and s space, where s is the number of snapshots in the snapshot set.

4.6.2 Backtracking Versioning

In backtracking versioning, a snapshot set is also maintained in a double linked list as in linear versioning. But the snapshots that belongs to a snapshot sets can be divided into two distinct sets: the backtracked snapshots and the others.

Backtracked snapshots are considered as deleted for the user but if they are added in some snapshot sets, the system also must delete them from snapshot sets. To avoid a complete browsing of all snapshot sets for each performed backtrack, we remove the backtracked snapshots while we browse the list for a required operation: when we browse an node in the list, we remove the backtracked snapshots. This clean operation adds a constant time to each operation. All operations are so always performed in worst case $O(1)$ time.

4.6.3 Branching Versioning

In branching versioning, snapshots of a snapshot set are maintained as in local field tree (without the value associated to snapshot in leafs).

The complexities are exactly the same: all operations are performed in worst-case $O(\log s)$ time and takes $O(s)$ space, where s is the number of snapshots in the snapshot set.

4.7 Automatic Object Graph Selection

The selection of an object must be considered now. As explained in the model in Section 3.8, a wanted selection depth is defined for each object (denoted D_o) and for each field (denoted d_f). Each field f has also an operatior $\oplus_f$ that defines how to compute the field selection depth (fixed or sum). To select an object o, the user changes the wanted selection depth of this object. Thanks to the wanted selection depths and the operators, the selection will be automatically propagated to the object graph of o. As defined in 3.8.1.4 (page 76), if $W(o)$ is the maximum of the greatest path weight (i.e. the path where the wanted selection combined with field operators is the greatest between two given objects) from any object of the system to o, an object o will be selected if $W(o) \geq 0$ and a field f is selected if its object o_f (which it belongs) is selected and $W(o_f) \oplus_f d_f \geq 0$.

We must find an efficient algorithm that takes as input the object graph with a selection detph for each object and for each field and that must determine if a field is selected or not.

We first provide an offline algorithm, i.e. an algorithm in which all datas (the object graph, the selection configuration, etc.) are known before its execution. We then provide an online algorithm, where an update of a selection depth does not require a complete pass

on the graph to determine which objects must be selected.

4.7.1 Offline Algorithm

We present first an offline algorithm where the object graph, with the selection depths and field operators, are fixed before the execution of the algorithm. An update of one of these data requires a complete new execution of the algorithm.

Actually we are looking for the greater weight for each object from a given object, i.e. search the path from an object to each connected object such that the combinaison of operators and selection depths on fields is the bigger. If the only operator is the sum, this problem is known as the longest path problem with cycles and this problem is NP-hard [Karger *et al.* , 1993], i.e. there is no known algorithm that resolves this problem in a polynomial time. There exists many studies about the approximation of the longest paths [Vishwanathan, 2000, Zhang & Li, 2007] or the classification of graphs for which the problem is not NP-hard [Feder & Motwani, 2005, Gabow, 2004, Gabow & Nie, 2008].

To solve this problem in polynomial time, we can split it into two: find the longest cycle-free paths and detect cycles.

The problem of longest path in an acyclic graph can be reduced to the problem of the shortest path by exploiting the fact that maximizing a positive value is the same as minimizing a negative value. If G is the graph considered for the problem of the longest path, we construct the graph H such that H contains the same vertices and arcs the same as G but where each arc weight is the negative. We can then use a shortest path algorithm that accepts negative weights but that considers only the nodes with positive label. You can use a slightly modification of the "FIFO label-correcting algorithm" that works with both operators and that consider only nodes with positive label. It can be shown that the algorithm works with both operators and with the same runtime complexity $O(nm)$, where n is the number of vertices and m the number of edges.

We present this algorithm applied on an object system (see algorithms 3 and 4), where nodes become objects and arcs become fields. As a first step, the weight $W(o)$ of each object o is initialized to positive infinity. For each object selected, we then browse its object graph range by range thanks to a FIFO data structure to compute the minimum negative

Algorithm 3 propagateOn: H

```
1: W(i) ← +∞ for each object i of G
2: for each object o of G do
3:    if D_o > −1 and W(o) < D_o then
4:       propagateOn: G from: o
5:    end if
6: end for
```

Algorithm 4 propagateOn: H from: s

```
1: FIFO ← {s}
2: while FIFO is not empty do
3:    remove an element o from FIFO
4:    for each field f of o in H do
5:       v ← f.value
6:       if W(v) > W(o) ⊕_f d_f − 1 then
7:          W(v) ← W(o) ⊕_f d_f − 1
8:          if W(v) ≥ 0 and v ∉ FIFO then
               add object v to the rear of FIFO
9:          end if
10:      end if
11:   end for
12: end while
```

weight for each object of this object graph.

After the execution of this algorithm, each object with a weight $W(o) \leq 0$ are selected.

To detect negative cycles, we can use two different properties. The first property is that the weight of an object can not be smaller than $-n * C$, where n is the number of objects of the graph and C the highest field selection depth across the graph. If a weight falls below this limit then there is a cycle of negative weights. [Ahuja *et al.* , 1993]

The second property is related to the FIFO algorithm itself. In this algorithm, each object will be extracted from the FIFO list at most $n - 1$ times to be treated if there is no cycle. If an object is processed at least n times then there is a cycle of negative weights. [Ahuja *et al.* , 1993]

When a negative cycle is detected, we can offer two alternatives: either the algorithm returns an error, or the algorithm continues by dealing only with the nodes respecting the two properties.

4.7.2 Online algorithm

The offline algorithm requires a complete pass on the graph to determine which objects are selected. We study now an online algorithm that takes as input the graph of objects with the current weight of the object and a new selection depth of a given object. This algorithm must determine in an efficient way which objects must be transformed into selected objects and which objects must be deselected.

But find a such algorithm for this problem is complex: one update can have an impact on the entire system. The difficulty is to find the good trade off between the time to select an object (and the part of its object graph), to deselect the objects that no longer need to be selected and the space required to perform quick selections and deselections.

We propose an algorithm that resolves part of this problem: the automatic selection without automatic deselection. This algorithm (see Algorithm 5) is simple to understand, gives good experimental results but it is theoritically exponential. The idea is to start from the object with the updated selection depth, browse its reachable graph in a depth-first search and update the weight of the object until there is no weight to update. This algorithm allows the propagation of the selection but not the propagation of the deselection.

This algorithm takes as input an object o and a selection depth D. Each field f (with its

field selection depth d_f and its operator $\oplus_f$) in the object graph of o is followed to update the weight of the connected objects.

Algorithm 5 selectedObject: o withDepth: D

```
1: if o is not yet marked then
2:    mark: o
3:    if W(o) < D then
4:       W(o) ← D
5:       for each field f of o do
6:          aDepth ← W(o) ⊕_f d_f
7:          if aDepth ≥ 0 then
8:             selectObject: f.value withDepth: aDepth − 1
9:          end if
10:      end for
11:   end if
12:   unmark: o
13: else
14:   Detection of cycle
15: end if
```

To summarize these few lines, the weight of an object is updated and the objects that have a connection depth smaller than D in the reachable set of the value of the field are automatically selected.

To understand the algorithm the best way is to start from lines 4 to 10. The weight of an object is initially -1. At line 4, the weight of the considered object is replaced by the new one. Between lines 5 and 10, we browse the different connected objects to o. The selection depth to apply to these objects are computed with the weight of o, the selection depth of the respective field f and its associated operator. The recursion allows the execution of the same treatment on the connected objects of o and so on.

Note (line 3) that this process is done only if the current weight of the object o is smaller than the passed depth. This condition optimizes somewhat the algorithm: if an object has already been selected with a weight that equals or is greater than the passed depth $aDepth$, the objects that have a connection depth smaller or equal to $aDepth$ are already selected and it is therefore not necessary to browse these objects one more time.

The last lines to explain deal with cycles. During the selection of an object, the same object could be selected twice. Take the example of an object o pointing to itself by the

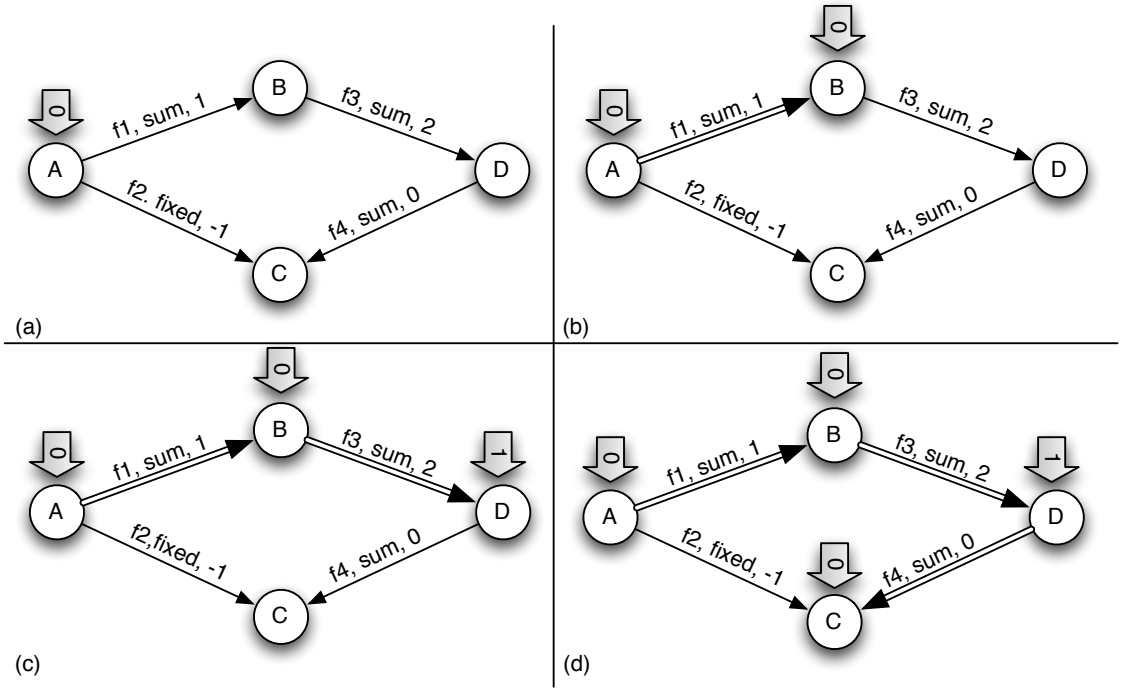


Figure 4.22: Steps of the selection of the object A with an object selection depth of 0.

selected field f, that has a depth of 1 with the operator *sum*. When o is selected with a depth of 1, selecting f with a depth of 2, selecting o, that is the value of f, with a depth of 1, and so on. To avoid these problems, only the first selection will be considered. We add a mark on the object (line 2) before selecting the rest of the reachable set and we remove it after (line 12). A condition is simply added at the start of the algorithm to detect marked objects (line 1). When a negative cycle is detected we can raise an error or do simply nothing.

Figure 4.22 shows the steps of the selection of the object A with a wanted selection depth 0. The step (a) shows the initial version of the system. Each object has a weight set to -1, to indicate that they are not selected (they are omitted on the figure for more readability). When the object A is selected with a wanted selection depth 0, all of its fields are considered. The field $f1$ has the operator `sum`. It is applied with the weight 0 and the field selection depth 1 minus 1, that equals to 0. As shown in the algorithm 5, the object B, that is pointed to by the field $f1$, must thus be selected with a depth 0 (step (b)).

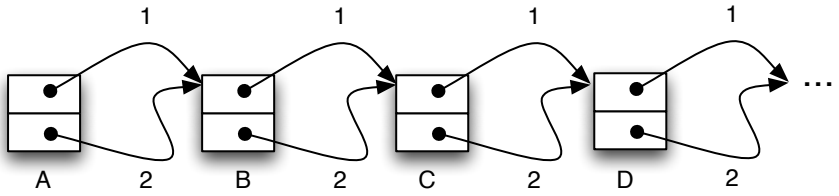


Figure 4.23: An inefficient example of automatic selection

The field $f3$ is considered: the operator `sum` applied on 0 and 2 (the field selection depth) minus 1 gives 1. The object D, pointed by $f3$ must be so selected with an object selection depth of 1 (step (c)). The fields of D are considered: the field $f4$ has the operator `sum` and a depth 0. The operator applied to 1 (the object selection depth on D) and 0 minus 1 gives 0. The object C must be selected with an object selection depth 0 (step (d)). The object C has no fields. The process of selection is finished for C, returning to the process of selection of D. It is also finished (all fields are handled) and it returns to the process of selection of B. B is also finished and we return to A. The combinaison of the weight of A (0) and the selection depth of the field $f2$ (-1) minus -1 equals -2: the value of the field must not be selected; the selection of A is finished.

This algorithm is unfortunately exponential. Take the example of Figure 4.23. All objects have two fields: the first field has a selection depth of 1 and the second field has a selection depth of 2. When the object A is selected with an object selection depth of 100, the configuration is considered: the first field is followed with a depth 100, selecting B, the value of this field, with a depth of 100. The fields of B are considered and its first field is selected with a depth 100, selecting C, the value of the first field of B, with a depth 100 and so on. When the first field and the connected objects are selected, the second field is used to select D with a depth of 101 (object depth (100) + field depth (2) - 1). Because this new depth is bigger than the previous one, the 99 objects connected this field will be browsed to select the connected objects. Once finished with C, we return to the selection of B: the second field of B must be browsed with a depth 101, reselecting the objects connected from C with an one-more-unit depth. This algorithm takes so an exponential time.

We use this exponential algorithm in our implementation because it is simple to implement and does not require other data structures. Moreover the experimental runtime

performances are very good for a well thoughout configuration of depths. Note that we could use the labeling algorithm from the selected node instead and obtain an update in time polynomial in the number of changed labels. It will be probably faster than your algorithm in practice but it is more complicated to implement and requires the maintenance of a FIFO data structure.

4.8 Online Automatic Deselection

Automatic selection allows one to be sure that when you select an object, all objects that will need to be versioned will be versioned automatically. The online and the offline algorithms implements the automatic selection. On the other hand, we have studied the offline automatic deselection that, when an object o is deselected, automatically deselect all objects selected because o is selected.

Online automatic deselection raises an interesting algorithmic question. Is there an efficient data structure such that the longest paths can be recomputed quickly when the wanted depth of an object is updated at runtime?

4.9 Conclusion

In this chapter we show how to implement efficiently linear, backtracking and branching versioning, the snapshot sets and the automatic selection. Our methods are an adaptation of fat node method of Driscoll et al. [Driscoll *et al.* , 1986].

We improved the original fat node method for the linear versioning to save states in constant time. For that, we developed the chained arrays, a data structure used to store the states of a field in the field itself. The complexities of chained arrays are particularly well adapted for linear versioning and they are implementable without too much effort. All operations are done in constant time, except query a field for a past state that takes $O(\log s)$ time worst case, where s is the number of states saved for this field.

We introduced a new method for backtracking versioning, based on our linear versioning method. We designed our solution so that the backtrack operation has an impact on the complexities of linear versioning only if the operation is used. Moreover the backtrack

operation is defined such that the fields with some backtracked states are cleaned ($O(\log s)$) only if they are accessed.

We described the fat node method [Driscoll *et al.* , 1986] for branching versioning adapted for our model. The snapshots are maintained in an order maintenance structure, named the global list of snapshots (GLS), to introduce a total order between snapshots. The order comparison between two snapshots is realized in constant time through the GLS. Each selected field keeps a local tree with its own states. Retrieval a state of a field for any snapshot takes $O(\log s)$ time worst case.

The field granularity of our model and the developed methods allow the introduction of versioning in an ephemeral application without any cost: there is no theoretical slowdown when ephemeral fields are used; the cost of versioning, we tried to minimize in this chapter, comes only when fields are selected.

Finally our methods are designed for a practical general usage of versioning: there are no constraints on the object graph on which the versioning is applied. Our method respects the expressivity of our model and it can be applied on any existing application, without restriction.

Chapter 5

Language Integration

In the two previous chapters, we designed an expressive model for object versioning and we studied algorithms to implement it efficiently. We implemented the three kinds of versioning in Smalltalk and linear versioning in Java. We call this library *HistOOry*. During this work, we discover that the definition of our model and efficient algorithms and data structures are not sufficient to achieve a smart integration of object versioning in these languages. In this chapter we show how to integrate *transparently* object versioning in any object-oriented language by providing a *simple* but *expressive* API.

The object versioning must be as transparent as possible, i.e. adding the object versioning in a base code must imply the fewest possible changes in this base code, to let the base code be always readable. This transparency can be implemented using aspects (used for Java) and bytecode transformation (used for Smalltalk).

The versioning is intrinsically simple: it allows one to save states and retrieve them. The Application Programming Interface (API) must be also simple to be quickly understandable by any developer. A too-complicated API will be never used by developers. The API is based on the model operations, e.g. take a snapshot and select of fields. The API uses overriding and polymorphism to integrate versioning with a minimum of modifications in the existing code.

The expressivity of the model must be respected. The simplicity of the API must not reduce the expressivity. We can improve this expressivity by attaching an active snapshot to each process or adding some intermediate objects.

5.1 Selection API

We present in this section an API to select fields and states in object-oriented languages. This API is used in the rest of this chapter to describe other parts of the integration. Moreover we show how the states data structures can be stored in statically typed languages in which types of selected fields are fixed at compile time, forbidding our states data structures as values.

5.1.1 Select fields

The class diagram Figure 5.1 shows an overview of the complete integration. This diagram will be described throughout this chapter. Each object has a protocol to let developers decide what is versioned, consisting of four methods only: `selectObject`, `selectObjectWithDepth:`, `selectField:withDepth:` and `selectionConfiguration`.

By default, the message `selectObject` sets the wanted depth of the object to 1. Another wanted selection depth can be specified by using the message `selectObjectWithDepth:`. Alternatively, the default behavior can be redefined by overriding the method `selectionConfiguration` : the message `selectObject` uses it to select appropriate fields. This method overriding is a practical way for establishing the default choices for what gets saved. Method `selectionConfiguration` can be implemented in `Object`, the root class in Smalltalk and Java, and returns all fields of the receiver object with a default wanted selection depth of 1 for each field. The fields are collected by using reflection and it is therefore not needed to override this method on each class that only wants to indicate that it too has fields to include.

If a developer wants to deviate from the default, the message `selectField:withDepth:` can also be used. It takes as argument the field that needs to be selected and its wanted selection depth, regardless of what is specified in method `selectionConfiguration`.

In our Smalltalk implementation, we added methods to existing classes (for example the four selection protocol methods to the root class `Object`) using *class extensions* (also called *open classes* [Millstein & Chambers, 1999]), i.e. a method that is defined in a module, but whose class is defined in another module. Other languages could use their particular

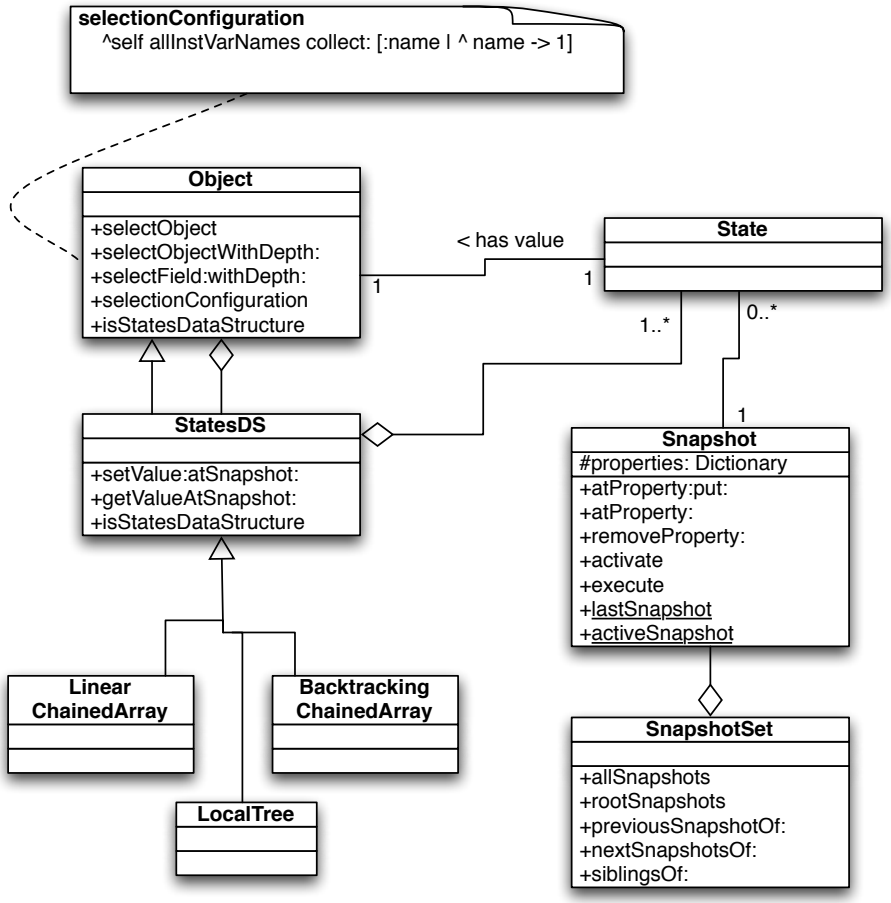


Figure 5.1: Class diagram of the object versioning integration

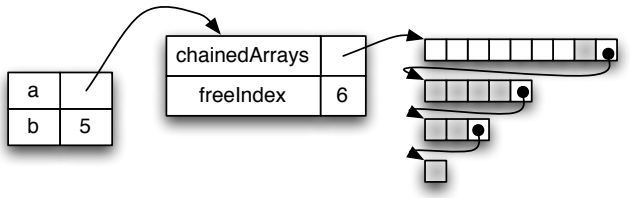


Figure 5.2: Dynamic typing: Object with a selected field a and a non selected field b

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.* The object versioning is nicely integrated in the language, resulting in a small embedded domain-specific language.

5.1.1.1 States Data Structure

When an ephemeral field is selected, we create a specific data structure that we put in the field itself, as described in Chapter 4. This structure is a chained array for linear versioning, a modified chained array for backtracking versioning and a local tree for branching versioning. These data structures are all subclasses of the generic class `StatesDS` (see Figure 5.1) and therefore share the same API:

- `setValue: anObject atSnapshot: aSnapshot` sets the object `anObject` as value for the snapshot `aSnapshot`. If a value is already present for this snapshot, the old value is replaced by the new one. If the snapshot `aSnapshot` is read-only, an error is thrown.
- `getValueAtSnapshot: aSnapshot` returns the value associated to the snapshot `aSnapshot` following the rules defined in the model (Chapter 3).

This common interface allows the abstraction of the kind of versioning for accesses to states. The rest of our API is defined using these two methods, allowing usage of any kind of versioning with a common API. The examples in the rest of this chapter are expressed in linear versioning, unless another kind of versioning is explicitly mentioned.

Practically, when we say that a data structure is set as value in the field, that can be really done only in one of the following cases:

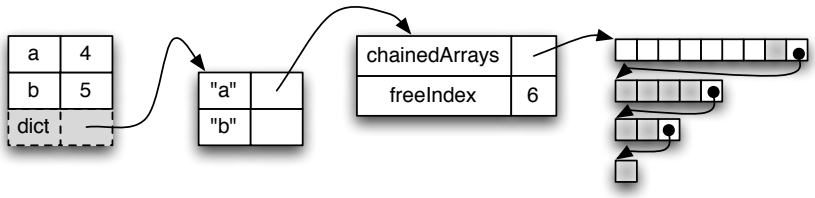


Figure 5.3: Static typing: Object with a selected field a and a non selected field b

- the field is *dynamically typed*. A dynamic typed field can accept any type of object as value and therefore the field can accept our data structure. Figure 5.2 shows an object with dynamically typed fields. When field a is selected, a chained array is created and is put as value of the field.

- the field is *statically typed* and the type of our data structure is a *subtype* of the static type of the field. Take the example of the class `MyClassA`, defined in Java as following:

  ```
  class MyClassA{
      protected Object a;
  }
  ```

  Our data structure can be put in the field a because any type in Java is a subtype of `Object`.

On the other hand, if the field is statically typed and the type of our data structure is not a subtype of the field type, the data structure can not be put in the field. For example, the Java compiler does not accept to put our data structure in the field b of the following class:

```
class MyClassB{
    protected String b;
}
```

The type of our data structure is not a subtype of the type `String`.

To avoid this problem in statically typed language, we add in each selected object a new field initialized with a dictionary. This dictionary associates name of fields with the corresponding states data structure. Figure 5.3 shows a statically typed object. When the field a is selected with linear versioning, a data structure is created and put in the dictionary

with the corresponding key "a". Note that if the language allows one to get for the field an index in place of a name (0 for the first field, 1 for the second one, and so on), an array can be used to minimize the lookup time.

While a such dictionary is necessary for statically typed languages, it can deteriorate the efficiency, as we will see in benchmarks of Chapter 6.

5.1.2 Snapshots and Snapshot Sets

The developer can choose the states to keep by taking snapshots sending the message `newAtNow` sent to the class `Snapshot`. This message returns an object that represents the snapshot. The properties of a snapshot can be managed by the methods `atProperty:put:`, `atProperty:` and `removeProperty:`.

When a snapshot must be activated, the message `activate` is sent to the snapshot object. The active snapshot and the last snapshot are accessible by the static methods `activeSnapshot` and `lastSnapshot` of the class `Snapshot`.

```
s1 := Snapshot newAtNow.
"some instructions".
s2 := Snapshot newAtNow.

s1 activate.
"Snapshot activeSnapshot == s1".

Snasphot lastSnapshot activate.
```

The snapshots must be managed by the developer, as any other object. The developer can use instance of class `SnapshotSet` to easily manage a set of snapshots. Instances of class `SnapshotSet` can understand the following messages: `allSnapshots`, `rootSnapshots` and `previousSnapshotOf:`. Their definitions follow the description of Section 3.7, page 70.

5.2 Transparency

In our model, accesses to a field have another meaning than in any classical ephemeral system. As explained in the previous section, a selected field will contain a data structure to

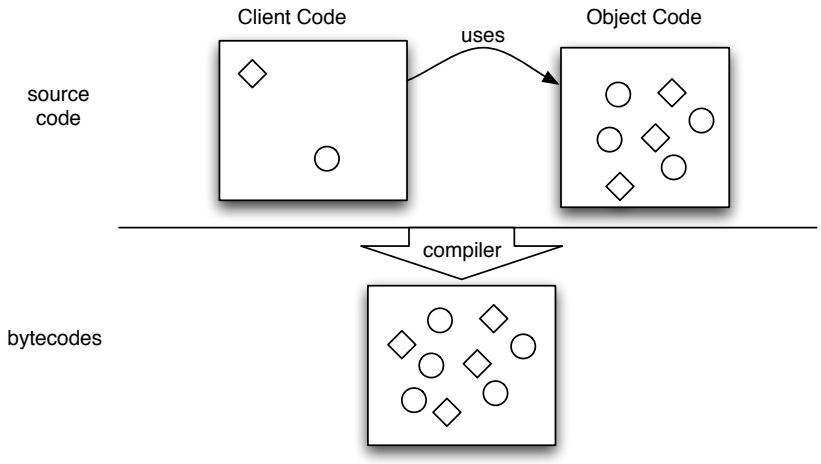


Figure 5.4: An application without versioning: client code uses objects and all the code is compiled to bytecodes. Diamonds and circles represents field accesses (read and write). The client code is the code that benefits of transparency because it uses in the same interface ephemeral and versioned objects.

keep states. Store a new value and read the value (that depends on the active snapshot) of a selected field are transformed into operations on this data structure. Each field access must therefore be instrumented so that accesses can be trapped and this behavior implemented. As we will study in this section, this instrumentation can be realized with or without *transparency*.

We divide the application code into two categories: the client code, i.e. a code that uses versioned objects, and versioned objects code. This separation is just a view on the code. A client code can be a versioned objects code if it itself uses other versioned objects. Figure 5.4 shows an ephemeral code: client code uses object code. Diamonds represent getter and circles represent setter. Field accesses are mainly done by the object code but client code can also access directly to public fields.

Transparency for the client code allows the usage of versioned and non versioned objects in the same way. The versioning is orthogonal to the object type: the client code uses versioned objects as ephemeral objects. If necessary, the client code can select fields, take snapshots and activate snapshots.

Transparency on versioned objects code adds the versioning mechanism in the versioned

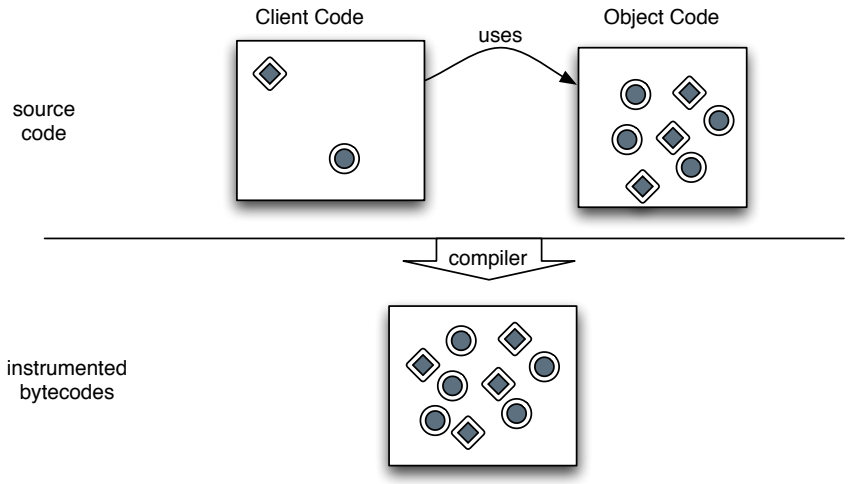


Figure 5.5: Manual instrumentation: the field accesses (diamonds are field reads and circles are update accesses) are manually instrumented. The instrumentation is shown by the black filling of accesses.

code in such a way that the original code remains untouched. During or after the compilation of the original code, a tool will instrument automatically the accesses to selected fields found in the original code. This separation of the original code and the versioning exempts versioning mechanisms from the business code, as we will show on examples in the following sections.

In this chapter we present three solutions to instrument field accesses: manually, using aspects and by transforming bytecodes.

5.2.1 No Transparency

The first solution is the manual insertion of code to intercept the accesses to the selected fields. The idea is to modify the code to access fields of versioned objects. Figure 5.5 shows the manual instrumentation of Figure 5.4: all accesses are instrumented directly in the code (accesses are black filled).

Take the example (in Smalltalk) of a simple class `Counter` that has an instance variable `myCounter` and two accessor methods (`setCounter:` and `getCounter`).

```
Counter>>setCounter: anInteger
```

```
    myCounter := anInteger

Counter>>getCounter
    ^myCounter
```

The versioning functionality is added by instrumenting each field access as follows. We inspect the current value of the field. If it is a states data structure, i.e. the field is selected, we forward the access to the data structure passing the active snapshot. If the field is ephemeral the access is performed normally. For instance, the class `Counter` would be instrumented as follows.

```
Counter>>setCounter: anInteger
    myCounter isStatesDataStructure
        ifTrue: [myCounter setValue: anInteger atSnapshot: Snapshot activeSnapshot]
        ifFalse: [myCounter := anInteger].

Counter>>getCounter
    myCounter isStatesDataStructure
        ifTrue: [^myCounter getValueAtSnapshot: Snapshot activeSnapshot]
        ifFalse: [^myCounter].
```

The message `isStatesDataStructure` is defined on all objects by extension of the class `Object` and the class `StatesDataStructure`:

```
Object>>isStatesDataStructure
    ^false
StatesDataStructure>>isStatesDataStructure
    ^true
```

The original code is clearly modified to add versioning: manual versioning is not transparent for the versioned objects code. For the client side, objects are used in the same way, independently of whether they are versioned or not, as long as fields are not directly accessed. For instance, we present a client code that uses ephemeral and versioned instances of this class.

```
    |c1 c2 s1 s2|
    c1 := Counter new.
    c1 setCounter: 1.
    c1 selectObject.

    c2 := Counter new.
    c2 setCounter: 10.
```

```
s1 := Snapshot newAtNow.

c1 setCounter: 2.
c1 setCounter: 3.

c2 setCounter: 11.

s2 := Snapshot newAtNow.

c1 setCounter: 4.
c2 setCounter: 12.

Transcript show: c1 getCounter. "print 4"
Transcript show: c2 getCounter. "print 12"

s1 activate. "Snapshot activeSnapshot == s1".
Transcript show: c1 getCounter. "print 1"
Transcript show: c2 getCounter. "print 12"

s2 activate. "Snapshot activeSnapshot == s2".
Transcript show: c1 getCounter. "print 3"
Transcript show: c2 getCounter. "print 12"

Snasphot lastSnapshot activate.
Transcript show: c1 getCounter. "print 4"
Transcript show: c2 getCounter. "print 12"
```

This code creates two instances of the instrumented class `Counter`. The first instance is versioned while the second one is ephemeral. They are then assigned different values separated by snapshots, as follows:

$c1 \leftarrow 1; c2 \leftarrow 10$; snapshot $s1$;

$c1 \leftarrow 2; c1 \leftarrow 3; c2 \leftarrow 11$; snapshot $s2$;

$c1 \leftarrow 4; c2 \leftarrow 12$.

We can see that both are manipulated in the same way (using the accessor methods). Note that this code takes and activates snapshots but it uses objects independently of their versioned state and of the active snapshot: at the end of this code, the same two sentences are executed four times but with different active snapshots. The versioned counter returned the state corresponding to the active snapshot while the non versioned counter returns always its last assigned value.

5.2.1.1 Discussion

The manual insertion solution has several drawbacks. The instrumentation is not automatic. Therefore it is easy to forget to instrument some accesses. Moreover the visibility of fields must be taken in consideration because it defines the scope of the accesses to instrument.

- If a field is *private*, only the methods of the class that defines this field must be instrumented.

- If a field is *protected*, the methods of its class and the methods of all subclasses of its class must be instrumented.

- If a field is *public*, the methods of its class and the methods of all subclasses of its class must be instrumented. Moreover all the code must be analyzed to find the usage of this field: a public field can be accessed from anywhere in the code. The client code of a versioned object must therefore be modified to integrate versioning mechanism: there is no transparency for the client code.

Moreover the original code is strongly modified to integrate versioning. The readability of the code becomes really bad. Take the example of the getter and setter methods in the counter example: the instrumented code is much less explicit than the original one! The maintenance of the code is significantly more complex.

The two next instrumentation techniques (aspects and bytecode manipulation) allow for fully transparent versioning.

5.2.2 Full Transparency using Aspects

Manual instrumentation is not transparent enough for the developer: the base code of an application must be modified to integrate versioning functionalities. In this section we show a fully transparent implementation of object versioning through Aspect-Oriented Programming (AOP).

5.2.2.1 Aspect Oriented Programming

Aspect-oriented programming is a modularization mechanism that allows a program to be split between (functional) base code, and so called cross-cutting behavior that needs to be applied throughout the base code [Kiczales & Hilsdale, 2001].

Take for example an application that implements a number of data structures (vectors, balanced trees, etc.). For helping with debugging, the developers want to keep a log file that shows whenever elements are deleted from these data structures. A good solution to implement this behavior using a non-AOP language would be to implement a logging facility, and to change the delete functionality in the data structure implementations to call this logging facility. An alternative would be to call the logging facility in the code that uses the data structures. In both cases however, the logging code that is only there for the purpose of debugging is added to the base program (either in the data structures themselves or in the code that uses the data structures).

Using aspect-oriented programming, the data structures and the client code are written without taking the logging code into account. The logging code is implemented in its own module (an *aspect*), that contains the logging facility itself as well as expressions that indicate where this logging facility needs to be called. The base program and this logging code are then composed by a so-called *weaver*, that produces the final program that does logging. An aspect implements the behavior that needs to be called, and specifies when the behavior needs to be called. In our example, we could decide to call the *log* functionality as last statement in the implementation of any delete procedure in any of our data structures (which corresponds to the first manual solution). We could also decide to execute the *log* functionality after every call to a delete procedure, corresponding to the second solution.

An aspect language hands a developer a number of points (*join points*) in the execution of the program where code can be called (the *advice code*), and a language to use them. Such language typically supports quantifiers and wildcard expressions that make it easy to specify global criteria. In our example, the second approach needs to express *'After any call to a method named delete, call the following piece of code: ...'*. Exactly what join points are offered depends on the aspect language. Typically code can be executed before, after or around the execution of behavior (calling a function, constructing an object, etc.). Aspect languages typically also offer support to add elements to existing code (e.g., methods, fields, interfaces, if AOP extends OOP).

5.2.2.2 Java Specific Implementation Details

Java is an object-oriented language offering a set of classes in different packages (e.g., `Vector`, `Set`, `Tree`) [Eckel, 2002]. The root of the hierarchy of all classes is the `Object` class, providing the minimum behavior for any object. To have an uniform and transparent mechanism to store values of a field, our different structures that keep states of this field (for linear, backtracking and branching versioning described in Chapter 4) store only values of type `Object` (a value can be put in a typed-`A` field if the type of value is `A` or a subtype of `A`), as well as any subclass of `Object` (i.e. all classes of the Java system).

A special case must be considered for Java's *basic types* (e.g., `int`, `double`, `float`). The basic-typed values are not objects, i.e. no message can be sent to them and their type is not a subtype of `Object`. However dual classes of the basic types exist in the system (class `Integer` for `int`, and so on). When a basic-typed value is placed in a typed-`Object` field a cast is automatically operated by the system and a new object of the associated class is created.

```
class MyClass {
    Object anObject = 5;
}
```

In this code, the `int` value `5` is put in a typed-`Object` field. A new object of the class `Integer`, containing the information `5`, is created and put in the field. This transformation is transparent for the developer and hard coded in the virtual machine. Because our data structure accepts `Object` values, the basic type is treated as any other object: our versioning mechanism works fine even with basic-typed values. But an additional cost in time is required for the conversion from the basic type to an object when a basic type value must be saved in a state, thus reducing versioning efficiency. We show this slowdown impact in our benchmarks at Section 6.2.2.

As said in a previous section, fields in Java are statically typed, thus our states structure can not be stored directly in place of those fields. Furthermore AspectJ only provides the names of the fields accessed. Because of this, we have to create a dictionary in each object for mapping the field name to the structure storing its states (named `fieldsAndStates` in following codes). When a field in a versioned object is accessed, a lookup in the corresponding dictionary is performed.

After performance tests the Java standard library `Hashtable` class seems to be inefficient

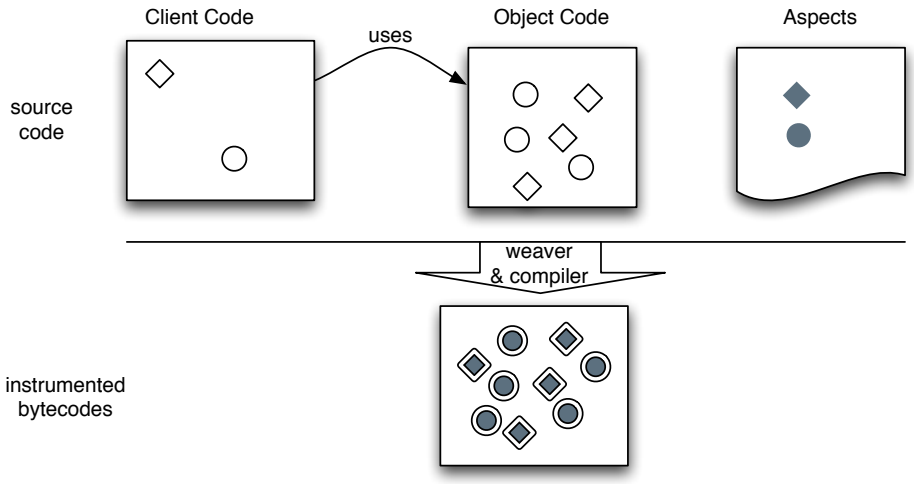


Figure 5.6: The instrumentation is defined in an aspect. The base code is untouched and bytecodes are well instrumented by the weaver and compiler passes.

for our solution: the hash function, applied on each access, is too slow where a simple binary search is really faster. We implemented a simple dictionary where keys are attributes names and values point to the states structure.

5.2.2.3 Transparent Versioning with AspectJ

Our implementation uses the aspect-oriented AspectJ system to add versioning to existing Java programs without needing to change these programs. Figure 5.6 shows the concept: we define the versioning instrumentation in an aspect, without touching the base code. Once weaved and compiled, the produced bytecodes are instrumented as wanted.

In order to use AspectJ to make classes versioned, the developer writes an *aspect declaration*. For example, the following AspectJ code makes all classes in a package `treap` versioned (note the wildcard expression `treap.*`):

```
declare parents: (treap.*) implements PObject;
```

Note that other criteria could be used, such as explicitly enumerating classes or selecting a number of classes based on their name. Technically, the aspect declaration updates the existing class to add our `PObject` interface to it. AspectJ installs all necessary wrappers

to classes implementing the `PObject` interface, adds an instance variable in these classes, initializes them, and finally extends them with the methods to access old states of fields of instances (see Figure 5.1). Note that this solution is transparent. The existing structure is made versioned with the aspect declaration, which is not part of the ephemeral implementation. The rest of the aspect is used to manage the states.

Adding a new variable to contain a dictionary in each versioned object:

```
public FieldsAndStates PObject.fieldsAndStates = new Dictionary();
```

Declaration of the pointcuts of setters and getters of versioned objects, but not in the `aspects` package:

```
// declaration of pointcut setters with 1 arg
pointcut setters(PObject t):
    // all updates of PObject implementors
    set(* PObject+.*)
    // not in 'aspects' package
    && ! within(aspects.*)
    // put the target in the variable t
    && target(t);

// same for all read operations
pointcut getters(PObject t): get(* PObject+.*)
                                && ! within(aspects.*)
                                && target(t);
```

Definition of the advice code after each update of a field of a versioned object (we ask to save the new value for the set field):

```
after(Object newValue, PObject t) : setters(t)  && args(newValue) {
    String fieldName = thisJoinPoint.getSignature().getName();
    String dict = t.fieldsAndStates;

    if(dict.hasKey(fieldName)){
        dict.at(fieldName).setValueAtSnapshot(
            newValue, // the new value stored in field
            Snapshot.activeSnapshot()
        );
    }
}
```

Definition of the advice code around each read on a field of a versioned object:

```
Object around(PObject t) : getters(t) {
```

```
    // retrieve the states of the field
    OrderedStates states = t.getStatesFor(
                thisJoinPoint.getSignature().getName());
    if(states == null)
          return proceed(); // return value in field if there is no entry for this field in the
     dictionary
    // search the good version of field in respect to active snapshot
    return states.getValueAtSnapshot(Snapshot.activeSnapshot());
}
```

Note that, because of AspectJ limitations, arrays can not be made versioned in this way: AspectJ does not offer a mechanism to instrument accesses to the elements of an array.

Notice that the field selection is well done at runtime: a field is considered as selected only if an entry for this field exists in the object dictionary, i.e. when a field is selected, a state data structure is added as entry in the object dictionary for this field. However the selection of a field of an object o is only possible if the instrumentation has been done by AspectJ on the class of o. The developer must so choose at compile time which classes could have versioned instances. A simple solution is to instrument all classes but this solution introduces a slowdown for each field access in the system, even for non selected fields. A tradeoff must be so defined by the developer between the instrumentation cost at runtime and selection capabilities at runtime.

5.2.3 Transparent versioning with Bytecode Manipulation

In Smalltalk, Python or Java, the compiler transforms source code into bytecodes. The bytecodes are a limited set of operations understandable by the virtual machine (Java VM or Smalltalk VM). A fully transparent integration of versioning can be achieved by bytecode transformation [Denker *et al.* , 2005, Tanter *et al.* , 2002]. The idea is that the base code is compiled as usual and that the generated bytecodes are modified to instrument field accesses (Figure 5.7).

To illustrate the bytecode transformation we take the example of the Smalltalk method in the class `Counter` defined as follows.

```
Counter>>counterPlusOne
    ^counter + 1
```

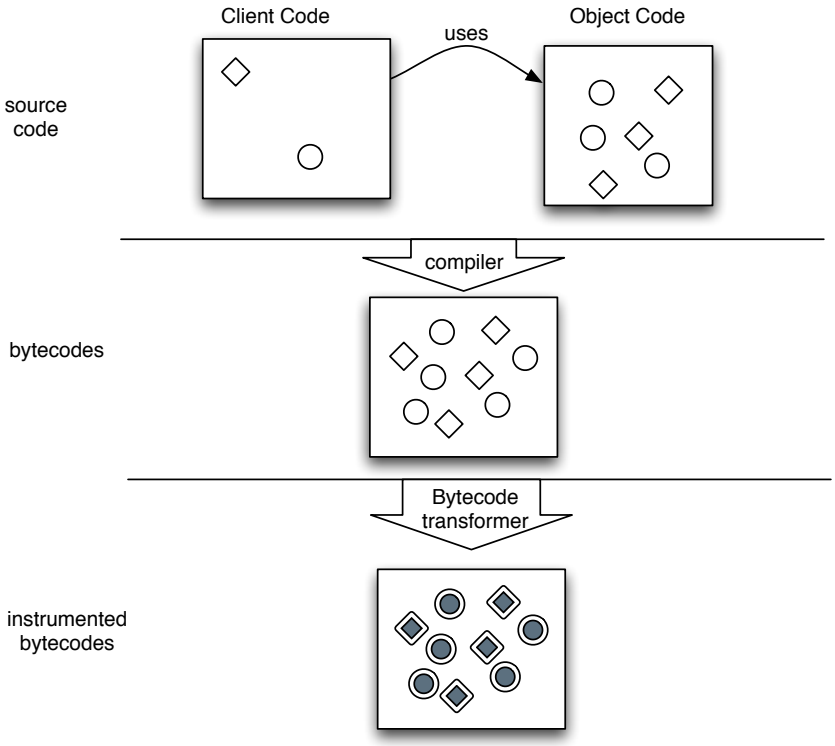


Figure 5.7: The base code is compiled as usually: the produced bytecodes are not instrumented (they are not black filled). We then manipulate the bytecodes to instrument them.

A manual instrumentation of this code should produce the following code.

```
Counter>>counterPlusOne
    ^(counter isStatesDataStructure
        ifTrue: [counter getValueAtSnapshot: Snapshot activeSnapshot]
        ifFalse: [counter]) + 1
```

Once compiled these two methods are transformed into the following sequences of bytecodes. We add their explanation (note that the Smalltalk virtual machine uses a stack to push operands of operations):

```
<00>  ⟶  push the first field of the receiver (self)
<76>  ⟶  push constant 1
<B0>  ⟶  send +
<7C>  ⟶  return top of stack
```

```
<00>  ⟶  push the first field of the receiver (self)


<D1>  ⟶  send isStatesDataStructure
<9A>  ⟶  if false: jump 5 instructions
<00>  ⟶  push the first field of the receiver
<42>  ⟶  push class Snapshot
<D1>  ⟶  send: activeSnapshot
<D0>  ⟶  send: getValueAtSnapshot:
<7C>  ⟶  jump 1 instruction
<00>  ⟶  push the first field of the receiver


<76>  ⟶  push constant 1
<B0>  ⟶  send +
<7C>  ⟶  return top of the stack
```

We see that the difference between the two sequences of bytecodes is the insertion of a block of 8 bytecodes just after the read access of the field. A similar block of bytecodes should be produced for write access. It is therefore possible to integrate the versioning in any sequence of bytecodes by adding blocks of bytecodes after field access bytecodes.

We implement this technique in Smalltalk because the bytecodes are easily accessible and editable in Smalltalk: each method is reified by an object `CompiledMethod`, maintained in the class where the method is defined. The bytecodes are contained in this object. We retrieve therefore the base bytecodes and we instrument them by adding bytecodes blocks for versioning.

The advantages of this technique are multiple. The instrumentation is completely hidden for the developer (like with aspects): the business code of the application uses object versioning without instrumenting getters and setters manually. The instrumentation of the bytecodes is faster than the instrumentation of the source code: bytecodes are a well-known limited set of instructions while source code is open to personalization (variable name, spaces and tabs, etc.). Work with bytecodes allows one to escape syntactic problems linked to source code.

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.

5.2.3.1 Example of Basic Usage

The following example versions the fields of a particular Squeak/Pharo package, i.e. an object that represents a package of code. It 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 with the saved snapshot as active snapshot. This time the transcript prints `Test`, again as expected.

```
|package s|

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

package selectObject. "Selects this object"

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

s := Snapshot newAtNow. "Takes a snapshot"

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

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

s activate.

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. It is versioned simply by sending it the `selectObject` 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 version fields of this object.
- The instrumentation is transparent: 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.
- 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 last value.

5.3 Improve Expressivity

In this section we see how to improve the model expressiveness in its concrete implementation. First, we attach the active snapshot to each individual process, allowing processes to work on different states of the system concurrently. One process can save states while another one can browse old states.

Second, we show how to add indirections through objects to personalize the versioning operations. As example we show how to take a snapshot after each change made in the system and collect these snapshots for a future usage.

5.3.1 Active Snapshot

The active snapshot plays an important role in our model: the semantic of an access to a selected field changes according to the active snapshot. Until this section we only

showed the usage of one active snapshot at a time. The active snapshot is *global* when there is only one active snapshot for the complete system. A modification of the active snapshot has an impact on the entire application: all accesses to fields follow the unique active snapshot.

The expressivity of the model can be improved by attaching an active snapshot per process. The active snapshot is named *process attached*. When a field is accessed, the active snapshot in the current process is therefore considered. The expressivity of the model is augmented because different processes can have different active snapshots: one process can save new states while another one can inspect old states.

There are several ways to attach an active snapshot to a process: it depends of the tools provided by the language. For example, the task is easy in Smalltalk: everything is an object, even internal processes and the processes scheduler. The processes are reified as instances of the class `Process` and the current process is accessible by the following code.

```
Processor activeProcess.
```

We extend the class `Process` by a field `activeSnapshot` and methods to access it. The methods `activeSnapshot` of the class `Snapshot` must be redefined to use the active snapshot of the current process, as in the following code.

```
Snapshot(class)>>activeSnapshot
    Processor activeProcess activeSnapshot.
```

The last snapshot is always managed globally and points to the last created snapshot in the system.

We believe that this technique could be implemented in statically typed languages like Java or C# as well but with more difficult implementation work. The minimum requirements are that processes must be reified in the application and that they must be extensible to be adapted to our needs.

Fork

When a process is forked, the link to the active snapshot of the father process is copied in the child process. Both processes can therefore manage their active snapshots independently.

For an easy usage of versioning, the active snapshots of processes are not independent

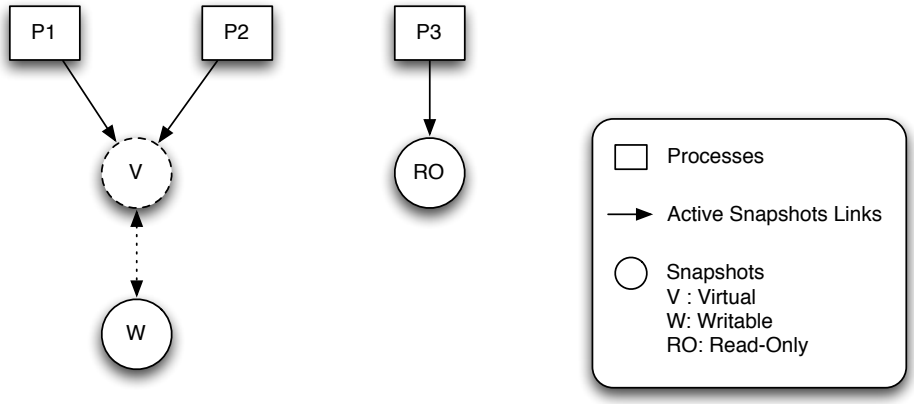


Figure 5.8: Management of active snapshots with processes

for writable active snapshots. When, a snapshot s_n is taken in a process with a writable active snapshot s_a, all processes with the active snapshot s_a are updated with the new active snapshot s_n. The idea is that all processes that originally have an active snapshot s_a work in a writable context; if their active snapshot is not updated to s_n, they are suddenly in a read-only context. On the other hand, the processes with a read-only active snapshot are not updated to stay in their past context.

Updating the processes to replace the active snapshot s_a by s_n can be realized in constant time as follows (see Figure 5.8). Each writable snapshot keeps a link to a *virtual snapshot* and the virtual snapshot keeps a link to the real snapshot. When a writable snapshot is assigned as active snapshot of a process the virtual snapshot is stored. Processes with the same writable snapshot therefore keep the same virtual snapshot while processes with a read-only snapshot point directly to the real snapshot. When the active snapshot is queried for a process, the real snapshot (found via the virtual snapshot if necessary) is always returned. When a new snapshot s_n is taken and the current process points to a virtual snapshot v_s, v_s is updated to point s_n and s_n is updated to point to v_s.

We specify that locks must be maintained on sensible variables as described in Section 3.11.4 (page 92) to avoid mutual modifications between processes.

5.3.2 Finer Configuration at Runtime

Our model can be summarized as follows: we select fields and we take snapshots. A snapshot saves the values of selected fields. But the expressivity of our model has some limits. For example, we can not express easily that we want to take a snapshot after every update of a selected field. Or yet take a snapshot after each update of a selected field but only if the last snapshot was taken after ten seconds ago.

It is difficult to express how to take a new snapshot after each new state of any object. However it can be achieved easily during the instrumentation of field accesses. The actual transformation of the code of a selected object is performed as follows.

```
Counter>>setCounter: anInteger
    myCounter := anInteger

" will be transformed into "

Counter>>setCounter: anInteger
    myCounter isStatesDataStructure
        ifTrue: [myCounter setValue: anInteger atSnapshot: Snapshot activeSnapshot]
        ifFalse: [myCounter := anInteger].
```

The instrumented code detects our data structure and either it sends the message `setValue:atSnapshot:` to the data structure or it performs the ephemeral field update. To take a snapshot after each new state we can change the instrumentation to produce a personalized code as follows:

```
Counter>>setCounter: anInteger
    myCounter := anInteger

" will be transformed into "

Counter>>setCounter: anInteger
    myCounter isStatesDataStructure
        ifTrue: [
            myCounter setValue: anInteger atSnapshot: Snapshot activeSnapshot.
            Snapshot newAtNow
        ]
        ifFalse: [myCounter := anInteger].
```

A second solution is to modify the method `setValue:atSnapshot:`. The disadvantage of both solutions is that the base code we provide for versioning must be modified by the

developer to include the desired behavior.

To provide a dynamic way to personalize the versioning process we propose a modification of the base mechanism: add an indirection to another object that will be placed between the field access and the state data structure. This object can be viewed as a limited meta object protocol (MOP).

A MOP is an interpreter of the semantics that is open and extensible [Kiczales & Rivieres, 1991]. Usually an field access is interpreted directly by the program executer (e.g. a virtual machine). When a MOP is introduced, a field access is no longer interpreted directly by the program executer but sent to the MOP to ask when the program executer must interpret this access. For example, a simple MOP can express that when the value of integer field is queried, the value must be doubled before being returned. Thus, a MOP is an object in which the behavior of basic operations is defined outside of the compiler or the interpreter.

To improve the expressivity of the model, we introduce a limited MOP between field accesses and the state data structures. This MOP introduces a point of configuration and it allows one to define at runtime different behaviors at each field access. This MOP is reified as an instance of the class `MOP`. This instance can be retrieved by the message `getInstance` on class `MOP`. Field accesses are instrumented by forwarding the field access to our MOP as follows.

```
Counter>>setCounter: anInteger
    ^MOP getInstance newValue: anInteger forField: #myCounter ofObject: self
```

The MOP is between the field access (access to counter) and the data structure: instead of performing directly the ephemeral or versioned update, we ask to the instance returned by the message `getInstance` to manage this update.

Our limited MOP must implement two messages[1]: `newValue:forField:ofObject:` for setters and `valueForField:ofObject:` for getters.

The benefit of this indirection is centralized into the method `getInstance`. This method can be implemented in several ways. The most simple implementation uses a class variable to store and retrieve an instance of the class MOP *or an instance of a subclass of the class MOP*. The basic implementation of the message `valueForField:ofObject:` of the class

[1]More messages could be implemented to manage, for example, selection and deselection of fields. We show here only two messages to illustrate the idea.

MOP, that saves states or performs the ephemeral update as usually, follows.

```
MOP>>newValue: oldValue forField: aFieldName ofObject: anObject
    | oldValue |
    oldValue := anObject valueInField: aFieldName
    oldValue isStatesDataStructure
        ifTrue: [oldValue setValue: newValue atSnapshot: Snapshot activeSnapshot]
        ifFalse: [anObject setValue: newValue inField: anObject].
    ^newValue
```

Because the message `getInstance` returns a MOP instance, any developer can create a subclass of MOP, redefines the messages `newValue:forField:ofObject:` and `valueForField:ofObject:` as desired. For example, we can take a snapshot after each new state by redefining this method `newValue:forField:ofObject:` in a subclass `MOPRecordAllStates` of the class `MOP` as follows.

```
MOPRecordAllStates>>newValue: oldValue forField: aFieldName ofObject: anObject
    | oldValue |
    oldValue := anObject valueInField: aFieldName
    oldValue isStatesDataStructure
        ifTrue: [
                oldValue setValue: newValue atSnapshot: Snapshot activeSnapshot.
                takenSnapshots add: Snapshot newAtNow.
                ]
        ifFalse: [anObject setValue: newValue inField: anObject].
    ^newValue
```

A new snapshot is taken when a new value is put in the states data structure. This snapshot is added in the snapshot set `takenSnapshots`, initialized at the creation of the MOP and accessible from methods defined in `MOPRecordAllStates`.

To use this MOP, the developer create a new instance of `MOPRecordAllStates` and put it in the variable of the class `MOP`. At each state modification, the message `newValue:ForField:ofObject:` is called on this instance at each instrumented field update, taking a snapshot after each versioned update.

Thank to this indirection, the developer can create any subclass of MOP to personalize as desired the instrumented field accesses.

Process Dependent Behavior The expressivity of our model is improved by the indirection to the MOP. But we can also use the current process (called also thread) to personalize

furthermore the behavior of instrumented field accesses. For example, we want to take a snapshot after each versioned update in a process but not in the concurrent processes.

To do that we can attach a MOP instance to each process, accessible by a message `getMOP` defined on class `Process`, and implement the method `getInstance` as follows.

```
MOP(class)>>getInstance
    ^Processor activeProcess getMOP
```

The returned MOP is clearly dependent of the current process. The following code shows an example of the automatic collection of all states of a binary search tree for the states created in the process that execute the code. The other process will always use the default MOP.

```
|t|
t := BinarySearchTree new.

"t must be versioned"
t selectObject.

"save the current MOP"
oldMOP := Processor activeProcess getMOP.
"create a MOP to record all states and collect snapshots"
mop := MOPRecordAllStates new.

"active the new MOP"
Processor activeProcess setMOP: mop.

"put the first 100 integers in the tree"
1 to: 100 do: [:i | t add: i].

"reactive the old MOP to stop the automatic record"
Processor activeProcess setMOP: oldMOP.


"retrieve and browse the collected snapshots"
mop takenSnapshot do: [:s |

    "activate the read–only snapshot"
    s activate.

    "make queries on the tree: for example print the structure of the tree"
    t printStructure.
]
```

5.3.3 Discussion

These expressivity improvements are really useful in practice to express exactly what we need to save and retrieve for a particular application. But the consequence of these modifications is a constant slowdown factor on efficiency (see our benchmarks in Section 6.3, page 189): each field access requires more operations to be executed. There is therefore a tradeoff between the benefits of improved expressivity and a worst efficiency.

5.4 Shortcuts to Browse Past

During our experiments we found that adding two shortcuts in the API substantially improved the readability.

5.4.1 Execute Block of Code Throughout a Snapshot

It is useful to retrieve old states to use them in present. Take the example of the following code where `s` is an old snapshot and `aSet` is a set of integers.

```
...
s activate.
oldSize := aSet size.
oldIncludesZero := aSet includes: 0.
Snapshot lastSnapshot activate.

(oldSize < aSet size)
      && oldIncludesZero
...
```

In this code, we compare the current size of `aSet` with its old size at snapshot `s`. Moreover, we need to know if the set contained the integer 0 at snapshot `s`. To do that we activate `s`, we store in two variables the old data, we reactivate the last snapshot and we perform the comparison. The fact that the query of these old information is separated from its usage in the condition makes the code somewhat difficult to understand. For more readability, we extend the class `Snapshot` with the message `execute:` as follows.

```
Snapshot>>execute: aBlock
    |oldActive returnedValue|
    oldActive := Snapshot activeSnapshot.      "save the active snapshot"
```

```
self activate.                          "active the snapshot receiver"
returnedValue := aBlock value.          "execute the block of code and saves the
returned value"
oldActive activate.                     "reactive the old active snapshot"
^returnedValue                          "return the value of the block"
```

With this method, the example can be transformed into a more readable form as follows:

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

5.4.2 PastObject

To make it easier to repeatedly send messages to a previous state of a single object, we can implement 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 := aSet viewAt: aSnapshot.
oldSet size < aSet size
      && oldSet includes: 0
...
```

The message `viewAt:` returns an instance of a class `PastObject`. The implementation of this class is pretty straightforward in Smalltalk. 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.

5.5 Reflective methods

Reflection is the process by which a computer program can observe and modify its own structure and behavior at runtime [Kiczales & Rivieres, 1991]. Reflective methods are methods that allow a such reflection process. For example in Smalltalk, the methods `instVarAt:` and `instVarAt:Put:` allow the read and store of the value of a given field from anywhere in the code, even if the field is private. Reflective methods are used by

language tools, like debuggers. In Smalltalk, most of reflective methods are primitives that directly invoke procedures from the virtual machine. When applied to a selected field, our data structure is therefore shown to the user.

This reflection breaks the principle of transparency of our instrumentation. Therefore we chose to hide completely our versioning mechanism by redefining these methods. However we aliased the original methods to use them when it is really necessary.

5.6 Garbage collection

In several languages, a garbage collector is provided to automate the heap management [Jones & Lins, 1996]. On a very high level a garbage collector works as follows. Some objects are designated as roots of the system and they can not be remove automatically of the system. All objects that are not contained in the reachable set of the roots are considered inaccessible. These inaccessible objects are removed automatically from memory by the garbage collector at key instants (e.g. when the memory is full or before querying all instances of a given class). Many algorithms exist to efficiently detect inaccessible objects. We do not explain these algorithms because they are not necessary to understand the rest of this section.

Conceptually an object can be garbage collected only if it is inaccessible in our versioned system, i.e. there is no way to access it from the roots, *even through an old state*. Indeed, all objects that participate to the accessible past of the application must be kept in memory. The versioning data structures must be implemented in such a way that the states of inaccessible objects are also automatically garbage collected. Figure 5.9 shows a general view of our different techniques explained in Chapter 4. This figure focuses on references to objects by each object. Versioned objects keep references to other objects by their ephemeral fields and to states data structures (i.e. chained arrays or local trees) via their selected fields. States data structures keep references to objects (values of states), to snapshots and to global data structures (e.g. B-links in backtracking versioning). Global data structures (i.e. pointers to active and last snapshots and global list of snapshots) and snapshot sets keep pointers to snapshots. Finally the client code takes snapshots, keeping them directly in variables or snapshot sets and manipulate objects.

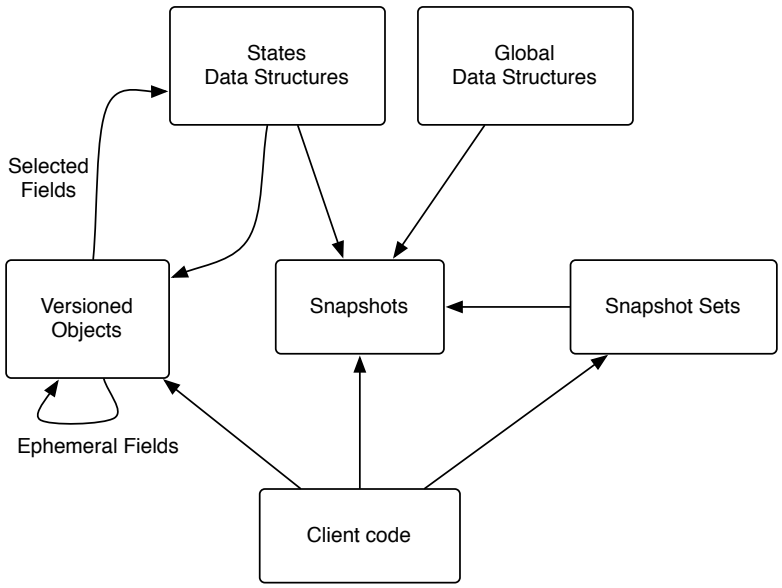


Figure 5.9: Arrows show the references kept by objects.

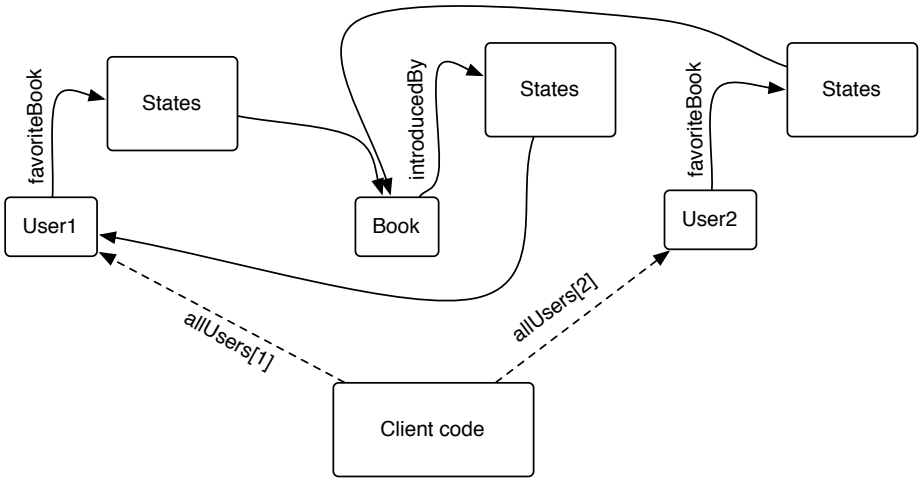


Figure 5.10: Users, book and their states will be garbaged collected when the dashed pointers are removed.

This organization was engineered in such a way that the garbage collection algorithm should not be modified (independently of the used algorithm). If our states data structures are not hidden from the garbage collector, an object can be garbage collected only when there is no reference on it. Take the example of Figure 5.10. Two users have their selected field `favoriteBook` that points to a particular book. The state of the selected field `introducedBy` of the book points the user `User1`. Only the users are accessible from the roots used by the garbage collector (in the client code). If the link from the client code to the user `User2` is deleted, the user `User2` and its states data structures will be automatically deleted: there is no more way to access them from the roots. If the link to the user `User1` is also deleted, the user `User1`, the book and their states will be garbage collected.

5.7 Discussion

This chapter proposes some design ways to integrate object versioning with object-oriented languages. They are based on our experience in Java and Smalltalk but these languages are used as examples: concepts can be applied to any object-oriented language, with more or less effort.

Our API consists of a small number of classes and methods, easy to learn and use by developers, as we show on real applications in the next chapter. It is compatible with the three kinds of versioning. The expressivity of our model is respected and can be extended easily without API modifications.

We integrate versioning using the concept of transparency, i.e. ephemeral and versioned objects must be used in the same way. We can make a link with the principle of *persistence independence* of the orthogonal persistence [Atkinson & Morrison, 1995] discussed in Section 3.10.1 (page 84). This transparency allows the developer to write code as using ephemeral objects and insert versioning controls (e.g. take snapshots) where this is needed. As opposite example, Caffeine[Guéhéneuc *et al.* , 2002], a non efficient[2] object versioning layer for Java, stores previous states as prolog facts for fast future queries. This integration is non transparent for the developer: the different states of an object are manipulated in Prolog and not in Java. This is contraignant for the developer: a new language must be

[2]Snapshots are taken as a copy of the entire set of objects to trace, resulting as a poor efficiency.

learned, including syntax and semantic. In addition, the developer must play with multiple paradigms (declarative, logic and object) in the same application, with the difficult task to translate data from one environment to another. We show that our integration allows us to use the same language and also the same business code if aspects or bytecodes manipulation are used.

Chapter 6

Validation

In previous chapters, we showed how we have implemented our framework HistOOry in two languages (Java and Smalltalk) by using the same model and same data structures and algorithms but different instrumentation tools for a transparent integration (aspects and bytecodes transformation). In this chapter we validate our implementations from two sides. First, we show how the expressiveness and the simplicity of HistOOry allow one to write powerful real applications. Second, we show the time and space efficiency of our implementations by analyzing, for each kind of versioning, the cost of each operation (e.g. to take a snapshot), the cost of the usage of instrumentation tools, the cost to attach active snapshots to processes and the cost of the usage of MOPs. We finally show the efficiency of HistOOry in real applications.

All tests for Java were done on a Dual 2 GHz PowerPC G5 with 2 Go DDR of memory, using the NetBeans IDE 5.5 with version 1.5.0_06 of Java and the AspectJ Development Environment (AJDE) version 1.5.0[1]. The following parameters were used: `-Xms1024m` and `-Xmx1024m` (the size of stack is exactly 1 gigabyte) and `-Xnoclassgc` (no automatic garbage collector). We disable the garbage collector to avoid parasite behavior during the performance tests. A manual garbage collection is performed before each test to clean the stack. Java is a dynamic language and has many features to improve its performance (Just In Time compilation, Hotspot dynamic compilation, etc.)[2]. The Java Just In Time(JIT) compiler is a real challenge for algorithm analysis: a read of a variable takes 40ns when the compiler is enabled. In the same conditions two reads take 45ns as total time. The sum of individual times is thus not equal to the time of combined operations. On the other hand this property is respected without enabling the compiler. Therefore we chose to disable the

[1] `http://www.netbeans.org`, `http://java.sun.com`, `http://aspectj-netbeans.sourceforge.net`
[2] `http://www-128.ibm.com/developerworks/library/j-jtp12214/`

compiler, in order to collect more coherent data.

All tests for Smalltalk were done on a MacBook 2.4 GHz Intel Core 2 Duo with 4 gigabytes of RAM and with an empty image of Squeak/Pharo for developers (version 0.1-101166dev08.11.6).

6.1 Case Study

In this section we show that the fine-grained model and its transparent implementation allows us to build complex real applications with minimum effort. We show three real applications with different goals, that use our efficient and fine-grained implementation to save and retrieve states of given objects. We start by stateful execution traces that save the different messages sent during the execution but also the state of each message receivers before and after a method execution. Our second example shows an implementation of postconditions in Smalltalk that allows the verification of assertions at the end of a method execution using the state of the receiver before and after the method execution. We finally present an easy way to implement the solution proposed by Sarnak et al. [Sarnak & Tarjan, 1986] for the planar point location problem.

This section stresses on the expressiviness of our framework. The efficiency of these applications is studied separately in Section 6.6 of this chapter.

Studied applications use only linear versioning. Examples using other kinds of versioning are not necessary to show how complex applications can be built with HistOOry: the three kinds of versioning share the same API to which the backtrack and branching operations are added. Examples of using these two operations are already given throughout the previous chapters.

6.1.1 Capturing Stateful Execution Traces

In the context of reengineering of 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 & Nakamura, 1997, Hamou-Lhadj & 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 could be expressed quite easily using for example object querying languages [Wuyts, 2001, Willis *et al.* , 2006, Hajiyev *et al.* , 2006], if only it would be possible to have the state information available.

This section shows 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 cannot 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. This will replace the instrumented method with a wrapper method where we can add hooks to trace the activation of methods. The following example shows the key part of this implementation. The wrapped method first calls `traceEntryIn:on:`, then it calls the original method, and finally it calls `traceExitOf:on:`. These two methods `traceEntryIn:on:` and `traceExitOf:on:` are auxiliary methods that store in a centralized data structure information about the messages that were sent, such as the timestamp.

```
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 centralized data structure can therefore be queried to retrieve information about the execution trace.

But 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. To do so we make the receiver versioned by sending it the message `selectFields`.

```
MyWrapper>>run: aSelector with: arguments in: aReceiver
  |answer|
  aReceiver selectObject.
```

```
self traceEntryIn: aSelector on: aReceiver at: Snapshot atNow.
answer := aReceiver
            withArgs:arguments executeMethod: originalMethod.
self traceExitOf: aSelector on: aReceiver at: Snapshot atNow.
^answer
```

For each method call, both states of the receiver are saved by the snapshots (by `Snapshot atNow`). These snapshots can be stored in the centralized data structure to be used to retrieve the state of the receiver at a given instant in the execution trace.

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

6.1.2 Postconditions

A postcondition is an assertion (a predicate the developer believes to be true) that describes the expected state at the end of 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. Note that in order 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.

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's execution. We show how we have extended Smalltalk with support for checked postconditions.

We needed a mechanism to make it possible for developers for specifying 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 avail-

able throughout Smalltalk. An alternative could have been to add the postcondition using method annotations.

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 checks 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)]]
```

In the example it can be seen how the original code of the method is put into a Smalltalk block (note 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*. Note how the postcondition block takes one argument (*old*) that represents the state of the system before the execution of the method's 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.

To implement this we extended the `BlockContext` class with the method `postCond: aBlock`. 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 `PastObject` object (see Section 5.4.2) 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 selectObject.
    "makes the receiver versioned"
```

```
    snapshot := Snapshot atNow.
    "snapshot before executing the method block"

    "create a PastObject"
    old := PastObject
              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
```

Note that this implementation is fairly straightforward. The only tweak is the creation of a `PastObject` 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 Section 5.4.2.

Sometimes postconditions need access other objects, for example to arguments as well. To support this we 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 `PastObject`, 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 selectObject].
    snapshot := Snapshot 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: [

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

    ^ aCollection
```

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

6.1.3 Planar Point Location

Planar point location described in Section 2.8.1 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 quickly.

To solve this problem, Sarnak and Tarjan [Sarnak & Tarjan, 1986] use persistent (versioned) data structures in order to reduce the space to $O(n)$. A vertical line sweeps the plane from $x = -\infty$ to $x = +\infty$, maintaining at every point the vertical order of the segment 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.

In order to illustrate how our framework can simplify the implementation of complex data structures, we implemented a *random treap* [Seidel & Aragon, 1996], a randomized binary search tree. This structure is a mix of a tree and an heap, each node keeping a key and a random priority. At each insertion rotations could be done to respect both constraints. We implement this structure using several classes: a class `RandomTreap` inherits of a class `Treap`, having as 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 in a versioned one, we simply extend the classes with the following methods:

```
Treap>>selectionConfiguration
    "Select the root node with a depth of 2 to get the different pointers and the states of the
    differents roots"
    ^NVArray with: {#root->2}

TreapNode>>selectionConfiguration
    "Select the left and right nodes with a depth of 2 to get the different pointers and the
    states of nodes"
    ^NVArray with: {#left->2.#right->2}
```

The class `NVArray` is a *non versioned* implementation of the class `Array` that we defined in the HistOOry package. No old state of its instances will be saved. This class is never instrumented and it avoids therefore unnecessary treatments to improve the efficiency of our framework.

Once we have a versioned random treap, the following code can be placed in a class `PlanarPointLocation`, storing 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 random treap, the incoming segments are removed and a snapshot is taken and associated with this point.

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

When a location query of a point p is considered, the slab containing p is determined, searching the rightmost point at the left of p among the points of the plane. This point l is the left point of the slab. We use then the snapshot associated with l to browse the treap at the time where only the relevant segments are 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]
```

In summary this section showed how we can implement easily a planar point location solution in Smalltalk by transforming an ephemeral random treap into a versioned one.

6.2 Time Efficiency Benchmarks

We start our benchmarks by analyzing the execution time of basic operations defined in our model: select a field, take a snapshot, store a value and query a field. The Smalltalk implementation is shown first.

We take a class `MyCounter` with one field `counter`.

```
MyCounter>>getCounter
    ^counter

MyCounter>>setCounter: anInteger
    ^counter := anInteger
```

We instrument manually these methods as follows:

```
MyCounter>>getCounter
    ^counter isStates
        ifTrue: [ counter valueForActiveSnapshot ]
        ifFalse: [ counter ]

MyCounter>>setCounter: anInteger
    counter isStates
        ifTrue: [ counter forActiveSnapshotSetValue: anInteger ]
        ifFalse: [ counter := anInteger ].
    ^anInteger
```

A comparable instrumentation is made for the Java benchmarks.

6.2.1 Smalltalk

To perform these benchmarks we take an empty image of Pharo and we load HistOOry with the kind of versioning we want to test. We use the tools provided by Pharo to perform

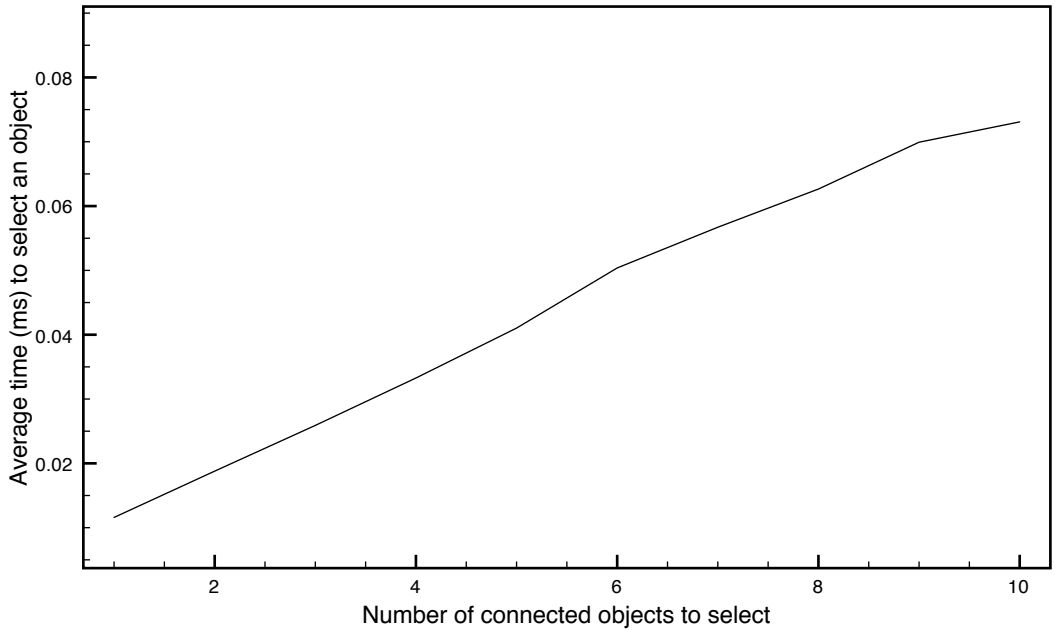


Figure 6.1: Average time to select an object vs. the number of reachable objects to select too.

tests and collect the different execution times (using the method `timeToRun` of the class `BlockClosure` that returns the number of milliseconds to execute a block of code).

Select a field We analyze the time to select a field (Figure 6.1). This operation replaces the current value by a states data structure and it takes constant time for any kind of versioning if the automatic selection is not used. Otherwise the object selection depth plays an important role in the time to select one object. We take the example of an object (the *root* object) with n reachable objects, i.e. the root object is linked to one object, this latter is linked to another object and so on until the $(n-1)$th object is linked to the nth object. We vary the selection depth of the root object between 0 and n. We take the average time (we run t times the benchmark and we take the average time) to select the root object. As expected the time is linear in term of selected objects.

Take snapshot We analyze the execution time to take snapshots for each kind of versioning. We collect the total time t to take n snapshots and we calculate the average time t/n to

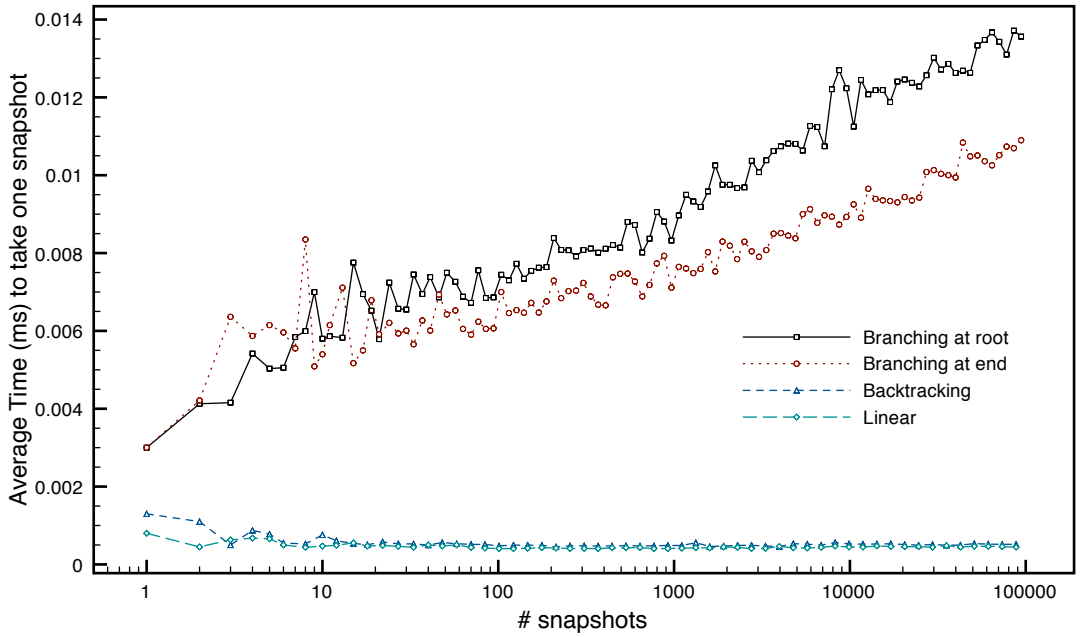


Figure 6.2: Average time to take a snapshot vs. the number of taken snapshots.

take one snapshot. We present the result for each kind of versioning in Figure 6.2 with a log scale for the number of taken snapshots.

For linear and backtracking versioning, only the last and active snapshots variables must be replaced by the new one. This operation is obviously independent of previously taken snapshots: it is always constant and it takes a minimum of time.

For branching versioning, the snapshots are stored in an order maintenance structure that inserts theoretically a new snapshot in amortized $O(\log n)$ time, where n is the number of taken snapshots. We differentiate between two cases: each new snapshot is branched either from the last one (snapshots form a path) or from the first snapshot of the system (snapshots form a tree of height 2 where the root is the first snapshot of the system and all other snapshots are branched directly from the root). Both are constant in time in practice, as expected from the theoretical results. We notice also that branching all states at root is two times slower than creating a branch from the last taken snapshot. After some investigation we found that the difference comes from a larger number of relabelings of the global list of snapshots (GLS) when snapshots are branched from the root.

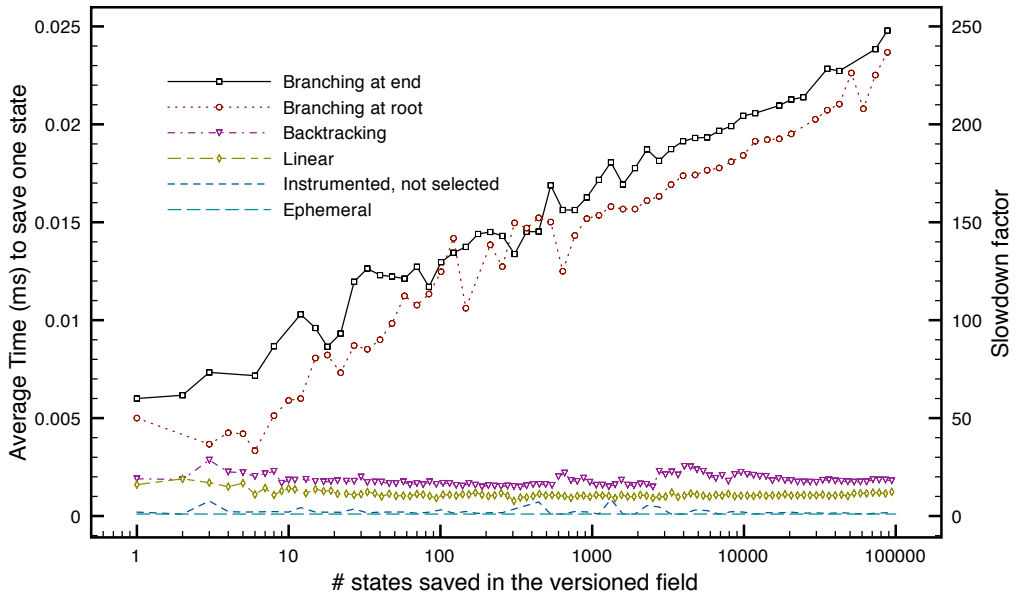


Figure 6.3: Average time to store a value in a selected field vs. the number of saved states for this selected field.

Store To evaluate the efficiency to store and query field states for each kind of versioning, we take the simplest implementation: a manual instrumentation of field accesses, global variables to store active and last snapshots and no MOP usage. The impact of the instrumentation by aspects and bytecode transformation, the usage of process to get the active snapshot and the usage of MOPs are studied separately in the following sections. As a result the slowdown of each part of the system is clearly visible.

To obtain the average time to update the value of a field, we take one object with one field. We first accumulate the total time t to update n times the field, each one followed by a new snapshot. We finally calculate the average time per insertion $t_n^{s,t}$ (t/n). We vary n from 1 to 10^5.

We use the same technique to get the average time t_n^t to take only snapshots without updating the field. The difference t_n^s between $t_n^{s,t}$ and t_n^t gives us the real average time to update a given field. Figure 6.3 shows this result for each kind of versioning. The left axis shows the average time in milliseconds to store one state while the right axis shows the

slowdown on the system versus an ephemeral set that takes 0.0001 millisecond according to our tests (the lowest curve shows the time to store a value in an ephemeral field).

First, we test the impact of the instrumentation on an ephemeral field: we take an ephemeral object, we manually instrument its methods and we test the update time without selecting its fields. The slowdown is about 1.5. That means that an update of a ephemeral field by an instrumented method is 1.5 times slower.

Next we analyze time to store a new state in linear and backtracking versioning: they are clearly independent of the number of already saved states. As expected the curves are constant. Moreover their slowdown factors are only about 7 for linear versioning and 20 for backtracking versioning. Backtracking versioning is slower than linear versioning because a verification of possible backtracked states must be performed before each store operation.

For branching versioning, we perform the same tests in two different situations: we branch at end (snapshots form a path) and we branch from the first snapshot (snapshots form a tree of height 2). In both cases the curves are logarithmic as expected. The slowdown factor for storing for branching versioning range from a factor 50 to a factor 250.

Having applications run respectively 7, 20 and 250 (for 10^5 states) times slower might seem like a big price to pay. However these tests are synthetic tests 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). In Section 6.6 we show that these slowdowns are reduced in real applications: for instance, store values in a versioned random treap using linear, backtracking or branching versioning is respectively only 2.7, 2.7 and 5.3 times slower than an ephemeral treap. These slowdowns are more acceptable.

Query We test now the average time to query the value of an ephemeral or selected field. Figure 6.4 shows the result for each kind of versioning in different situations. The left axis shows the average time in milliseconds to query one state while the right axis shows the slowdown on the system versus an ephemeral query that takes around 0.0001 millisecond (the lowest curve on figure). We describe this graph from the lowest to the higher curve.

The first curve is the time to query an ephemeral field: this is obviously constant (around 0.0001 ms). The second curve shows the time to query an field but with manually instru-

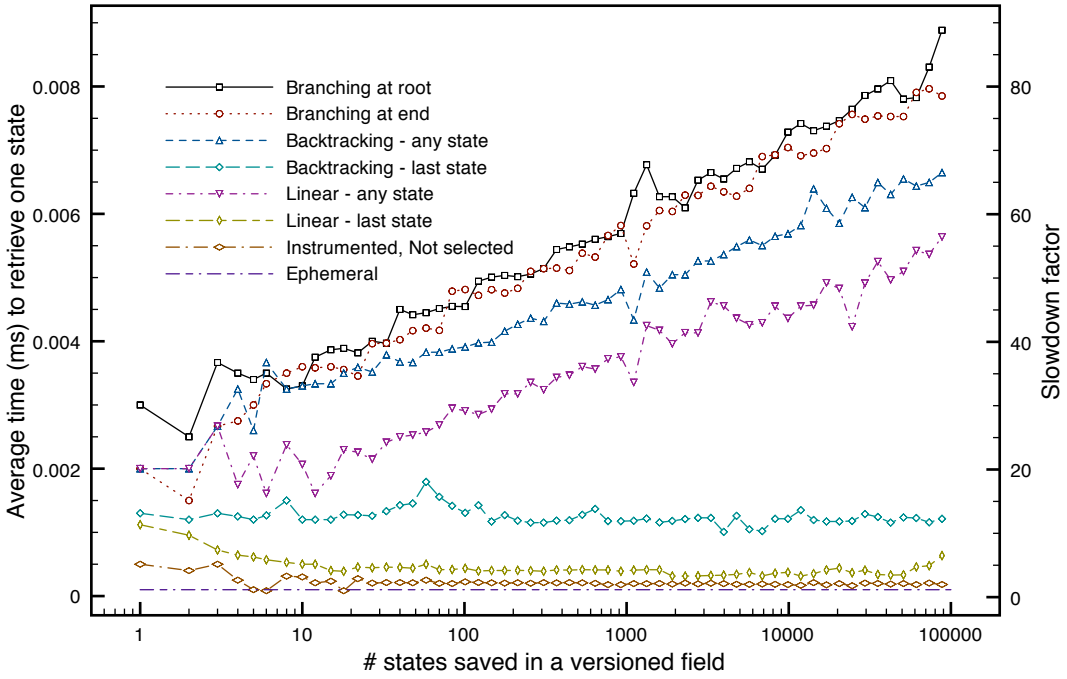


Figure 6.4: Average time to retrieve a state of a selected field vs. the number of saved states for this selected field.

mented accesses. When the value of this field is queried, a comparison on the type of the value is performed: the value is not a chained array and this value is returned directly. The slowdown factor is about 2.

The third and fourth curves show the average times to query the last state of a versioned field in linear and branching versioning. Chained arrays allow constant time to get the last state and the curve looks constant as expected: the number of saved states in the field has no impact on the query time. The slowdown factor is about 3.2 for linear versioning and about 11.4 for backtracking versioning.

The last four curves show the cost to retrieve any saved state for each kind of versioning, depending on the number of states that were saved. To obtain these times we selected a field and updated it a fixed number of times n (from 1 to 10^5), each store followed by taking a snapshot. Then we took the total time to query the n states and we divided this time by n. It gives the average execution time to access a single state. Both first curves are for linear and backtracking versioning and both last ones for branching versioning (snapshots branched at root or at end of the path). The four curves are logarithmic as expected, indicating that our implementation is correct. From 1 to 10^5 saved states, the linear (resp. backtracking) versioning has a slowdown factor between 20 and 57 (resp. 67) while the branching versioning has a slowdown factor between 20 and 90.

Discussion As expected the theoretical bounds are respected in our implementation. The linear versioning is the simplest kind of versioning and also the most efficient. It is closely followed by the backtracking versioning, that requires only few operations more than for linear versioning. The branching versioning is more complicated and consumes therefore more time.

Implementation Details For linear and backtracking versioning, we first implemented the chained arrays such that each state is encapsulated in an object (with the two fields `versionNumber` and `value`). By using profiling tools, we observed that creating an object for each state and asking its version number and/or the value are time consuming tasks. We decided to break the well-defined object-oriented code to put the version numbers and the values in two linked arrays (one for the version numbers and one for the values): the version number at index i of the array of version numbers is associated with the value at index i of the array of values. This implementation considerably increases the performance

of the data structure. However we optimized this data structure even further. Because accessing two arrays of n elements takes more time than accessing a single array of size $2n$, we decide to store it as such: a single array where version numbers precede their corresponding values.

6.2.2 Java

We implemented the linear versioning in Java as well. Many results for the Java implementation are similar to the just presented results for Smalltalk. Therefore we chose to focus on the particularities of Java in this section. A fine-grained analysis of overhead induced by aspects instrumentation is proposed in Section 6.4.2.

Store The first test measures the storing time. Like for Smalltalk we manually instrument methods of an object with one field and we collect the time to add n states. We divide this time by n to get the average time per store. We take five cases: an ephemeral field, a selected field storing its states in chained arrays (and taking snapshots or not) and a selected field using Java vectors to store its fields (and taking snapshots or not). We remind that Java vector begins with an array of m elements. At each insertion, if the array is full, a double sized array is created, the full array is copied into the new one using `System.arraycopy(...)` and the element is inserted in the first free place in the array.

Figure 6.5 shows the average time to store one value in the field for each of the five cases. The times are reported on the left axis while the slowdowns compared to the ephemeral store (around 0.0001 second) is on the right side. For each structure the time per insertion is constant as expected. The chained arrays result in a slowdown from 5 (update the last state) to 8 (add a new state) while the vector slowdowns vary between 16 and 34. The chained arrays are therefore well designed for our versioning purpose.

Query We next measure query time: we update a field n times and we take the average time to query states (see Figure 6.6). If the field is ephemeral, the query time is constant and takes about 0.000045ms. When the field is selected, we distinguish two cases: states are stored either in chained arrays or in vectors. We report the average time to get the

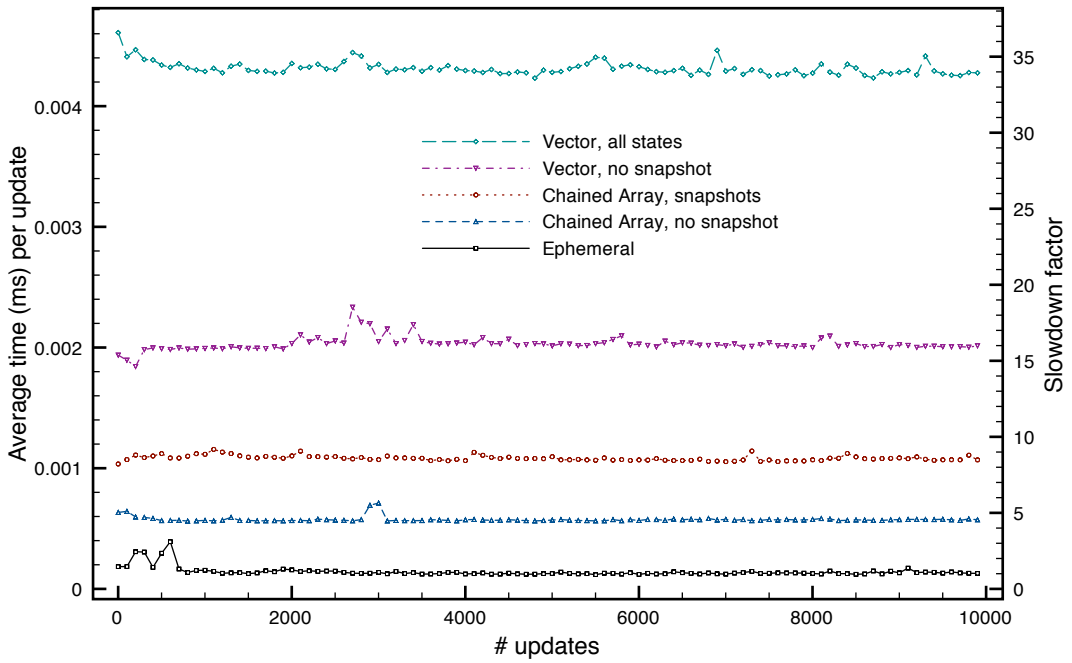


Figure 6.5: Number of updates vs. average time per update.

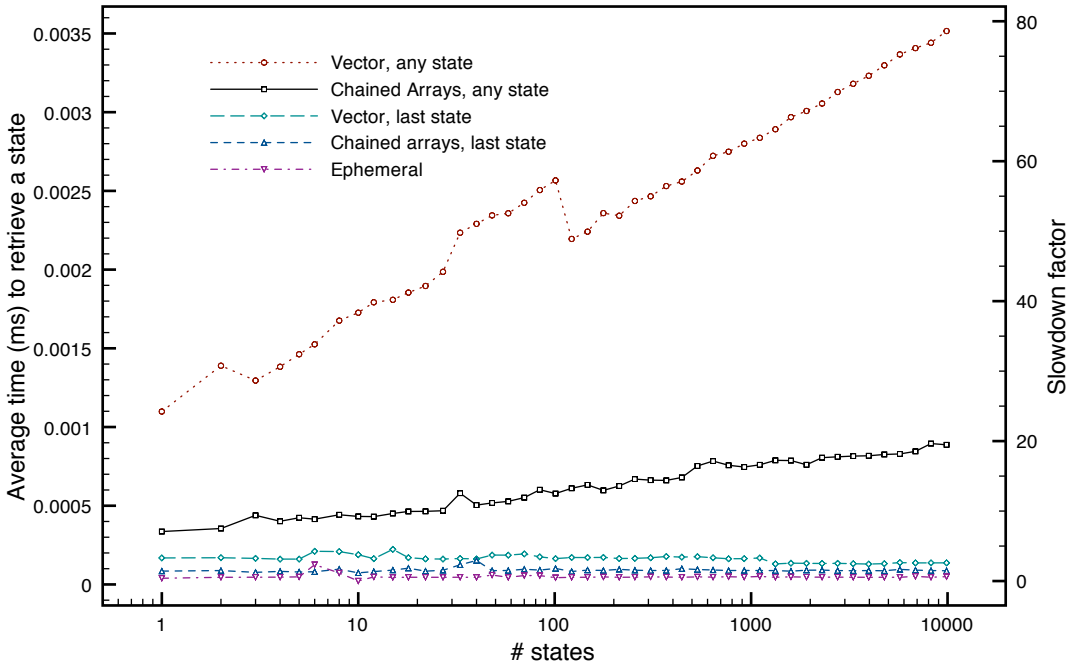


Figure 6.6: Average time per query vs. number of saved states.

last state. For both data structures the query time is independent of the number of saved states (because there is a direct pointer to the last state). The chained arrays are 1.5 time faster than vectors.

As a second test we take the time t to query all n saved states and we report the average time t/n. For both data structures, the curves are logarithmic as expected. Our structure is about 4 times faster than vectors.

Using chained arrays, asking the last state of a selected field is only two times slower than an ephemeral query. Asking old state has a slowdown factor between 7 and 19 (between 1 and 10^4 states).

6.3 Cost of Expressiveness Benchmarks

Previous tests show the time efficiency in the minimal configuration of HistOOry, i.e. the accesses to the field are instrumented manually as follows:

```
MyCounter>>getCounter
    ^counter isStates
        ifTrue: [ counter valueForActiveSnapshot ]
        ifFalse: [ counter ]

MyCounter>>setCounter: anInteger
    counter isStates
        ifTrue: [ counter forActiveSnapshotSetValue: anInteger ]
        ifFalse: [ counter := anInteger ].
    ^anInteger
```

We call this implementation *If/else*. We now analyze the time with more or less access points of configuration to express how the versioning must be integrated in the application. We experiment the following implementations:

- The implementation *Always Selected*: if a field of all instances of a class will never be ephemeral but always selected, we can remove the condition by instrumenting all field accesses as follows:

```
MyCounter>>getCounter
    ^myField valueForActiveSnapshot

MyCounter>>setCounter: anInteger
```

```
    myField forActiveSnapshotSetValue: anInteger.
    ^anInteger
```

- The implementation *Process*: we attach an active snapshot to each process and all instrumented field accesses are redirected to the current process, as defined in the previous chapter.

```
MyCounter>>getCounter
    ^Process activeProcess valueOf: myField

MyCounter>>setCounter: anInteger
    Process activeProcess  replace: myField byNewValue: anInteger in: #myField
    ofObject: self.
    ^anInteger
```

- The implementation *Process + MOP*: we attach a MOP to each process and all instrumented field accesses are redirected to the current process, that redirects itself to the MOP. This allows yet more access points for configuration as explained in previous chapter.

We calculate the time to store a new value in a field in each kind of implementation and we report it in Figure 6.7. We first test with a non selected field. The slowdown time of the implementations *if/else* and *Process* (with MOP or not) versus the ephemeral time is respectively about 500ns and 1000ns. We then test with a selected field. Without surprise, the curves are constant: each indirection adds constant time to the ephemeral store. The additional times vary from between 6000ns and 10000ns.

We also experiment with the query time in the same conditions. The additional times are comparable to the previous test. At each indirection constant time is added.

As expected, adding some points of configuration consumes time. The tradeoff between the efficiency and the expressiveness must be fixed when the object versioning mechanisms are implementation. However we notice that the implementations *if/else* and *Always selected* are very close in term of required time. We therefore prefer to use at least the implementation *if/else*.

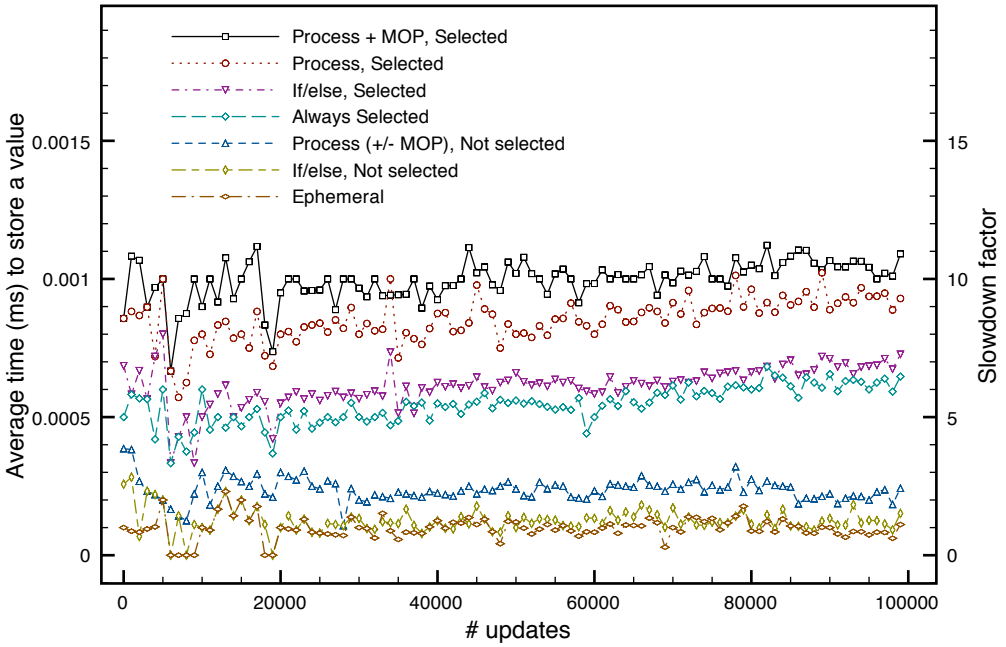


Figure 6.7: Cost of indirections and MOP for a store operation.

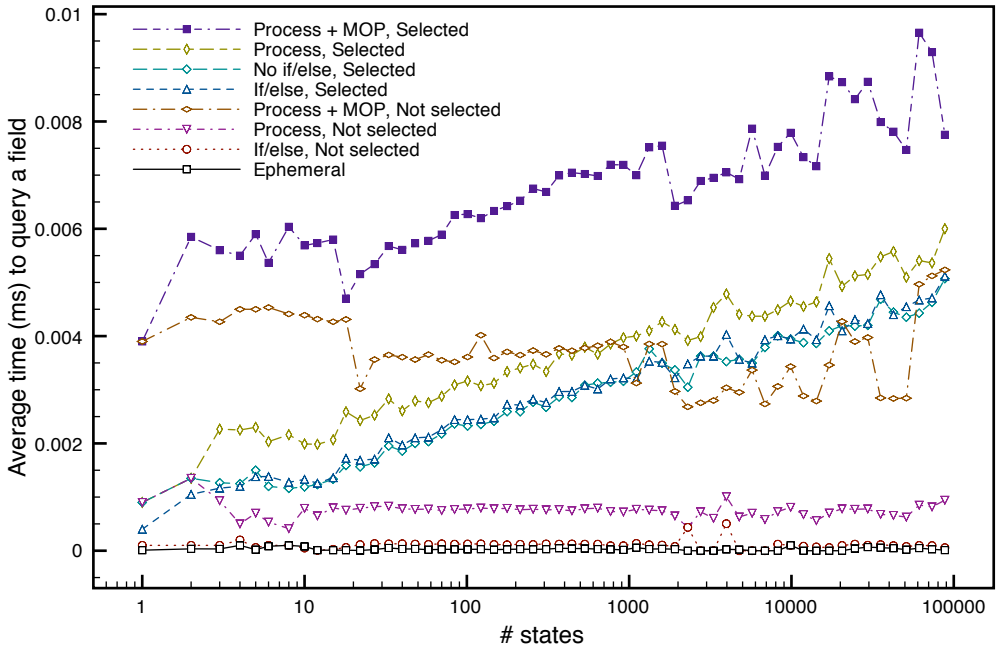


Figure 6.8: Cost of indirections and MOP for a query operation.

6.4 Instrumentation Time Impact Benchmark

We analyze now the slowdown introduced by aspects and bytecode transformation compared to the manual instrumentation.

6.4.1 Smalltalk Bytecode Manipulation

The manipulation of bytecodes allows fine grained transformations. In Smalltalk, our tool transforms any ephemeral bytecodes into versioned bytecodes where each field access is instrumented. Unlike aspects, the produced bytecodes are exactly the same than those obtained after manual instrumentation, as shown in Section 5.2.3. Therefore, the bytecode instrumentation does not introduce an extra slowdown factor during the program execution versus a manual instrumentation. The slowdowns presented in previous section 6.2.1 are therefore the same for bytecode instrumentation.

We think comparable performances can be achieved by manipulating the Java bytecodes [Tanter *et al.* , 2002].

6.4.2 Java Aspects

In Section 6.2.2 we presented slowdown factors for synthetic tests where the code is instrumented manually where field states are stored in data structures accessible directly. We now analyze the cost of the usage of aspects to instrument transparently ephemeral code. In this section we show how the aspects, that forces to store the field states data structures in a dictionary, will increase the slowdown factors of synthetic tests.

Instrumented Store

To analyze the cost of aspects on instrumented field store, we perform a number of updates on a selected field. We separate the time for each step of the versioning of an update operation (see Figure 6.9):

Original Java The time to perform one update in the native Java program;

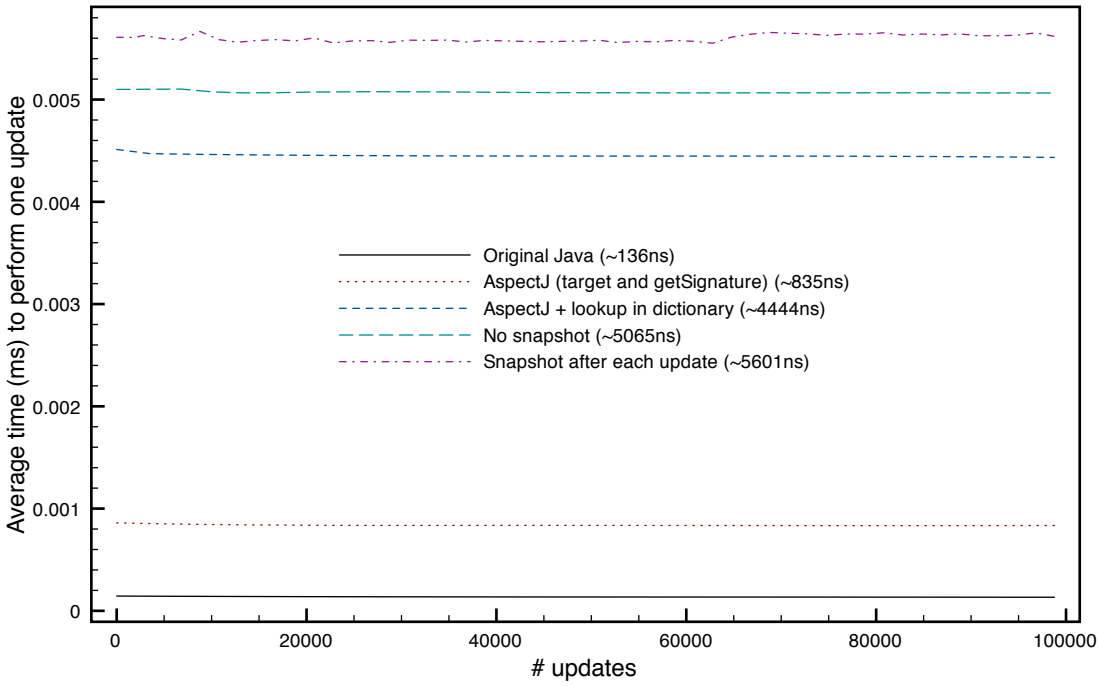


Figure 6.9: Number of updates vs. average time per update.

AspectJ (target and getSignature) The aspect adds new code after each change. It takes extra time to retrieve the target object from the change and extract the name of the affected field from the signature. We measure an overhead of a factor of about 6;

AspectJ + lookup in dictionary As explained in Section 5.2.2.2 a dictionary is used to map the name of the target variable to the states data structure. The measured overhead is of a factor of about 32;

No Snapshot Adding the mechanism described in section 5.2.2.3, without taking a snapshot (all changes update the value of the last state associated to the field). We measure an overhead of a factor of 37;

Snapshot after each update Same as the previous test but taking a snapshot after each update. The measured overhead is now a factor of 41.

Notice that the cost of AspectJ (to retrieve the affected field name and the target object) followed by the search in the dictionary induce an overhead of 32 compared to the average time to perform a change on an ephemeral field in Java. Saving the state in the structure takes only between 600ns and 1200ns, i.e., only 4 to 9 times slower that the original code. If AspectJ would provide a mechanism to put the states directly in the fields, much better results should be achievable.

Instrumented Query

We now analyze the impact of the instrumentation by aspects on a read access to a field that was just updated (only the read time is observed). Here the writability of the active snapshot is important: if the active snapshot s is read-only we are looking for the value of a field in the last saved state before or at version number $s.versionNumber$. Otherwise the actual value of the field is returned (no lookup in dictionary is then performed). We decompose the operation:

Original Java The time of a read in the original Java. Does not differ much from an update ;

AspectJ (Active snapshot is writable) The active snapshot is writable. The aspect returns the actual value contained in the field. We measure an overhead of a factor of about 5.5;

AspectJ (Active snapshot is read-only: getSignature) The active snapshot is read-only, states of this field must be consulted. As a first step we report the search of the name of relevant field, using the signature of the read operation given by AspectJ. We measure an overhead of a factor of about 7.7;

AspectJ + lookup in dictionary After the previous operation the dictionary is consulted to retrieve the states data structure associated to the target variable. The measured overhead is of a factor of about 35;

No Snapshot The entire mechanism is activated, without taking a snapshot after the updates. We measure an overhead of a factor of 43;

Snapshot after each update, Active snapshot on first snasphot The same previous test but taking a snapshot after each change. The active snapshot is the first snapshot of the system: at each read a search must be performed to find the first state in the associated states structure of the target attribute. The curve is logarithmic as expected.

The general observations made in our previous tests are confirmed here: the total performance is dominated by the three first phases.

Two important remarks can be made. First, a drawback of our implementation is that a lookup in dictionary must be done for each operation on an attribute (update or read via the global snapshot). The time of an update followed by read (with an read-only active snapshot) is therefore the sum of their individual time. We could not find a better method considering the features of Java and AspectJ. Second, in order to interpret the large overhead of our system, the following must be taken into consideration :

The JIT compiler is disabled With the compiler enabled the analysis can be done less precisely but we notice that the performance optimizations performed by the compiler reduce considerably this overhead ;

Applications We will see in next section that in real applications, the versioned operations can be mixed with a large number of regular operations, making the overhead acceptable.

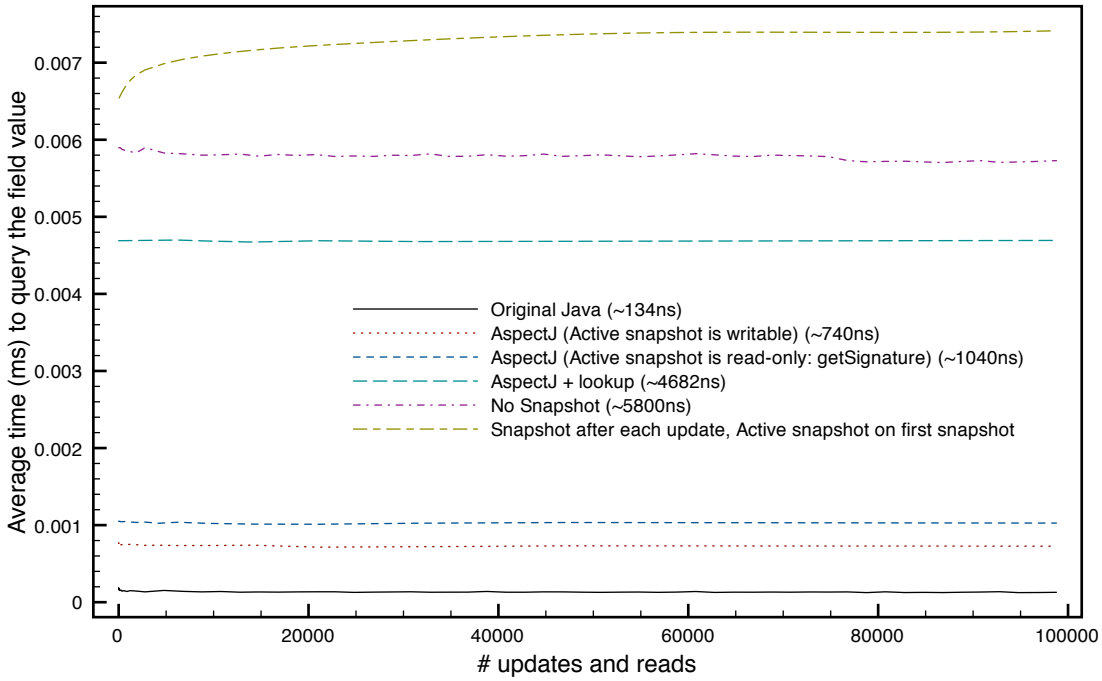


Figure 6.10: Number of updates+reads vs. average time per read.

6.5 Size Efficiency Benchmarks

We presented synthetic benchmarks on time in previous section. In this section we analyze the effective size required by HistOOry.

6.5.1 Smalltalk

To show the size required by HistOOry for each kind of versioning, 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 (or not) a snapshot, and repeat this. This shows us how the space required to store all of these states evolves.

Figure 6.11 shows the size taken after each update of the field. We start the test with an ephemeral field: the size of the object is obviously constant (20 bytes). The size of one state is 186 bytes for linear versioning (a factor of 9.3), 200 bytes for backtracking versioning (a factor of 10) and 1096 bytes for branching versioning (a factor of 54.8).

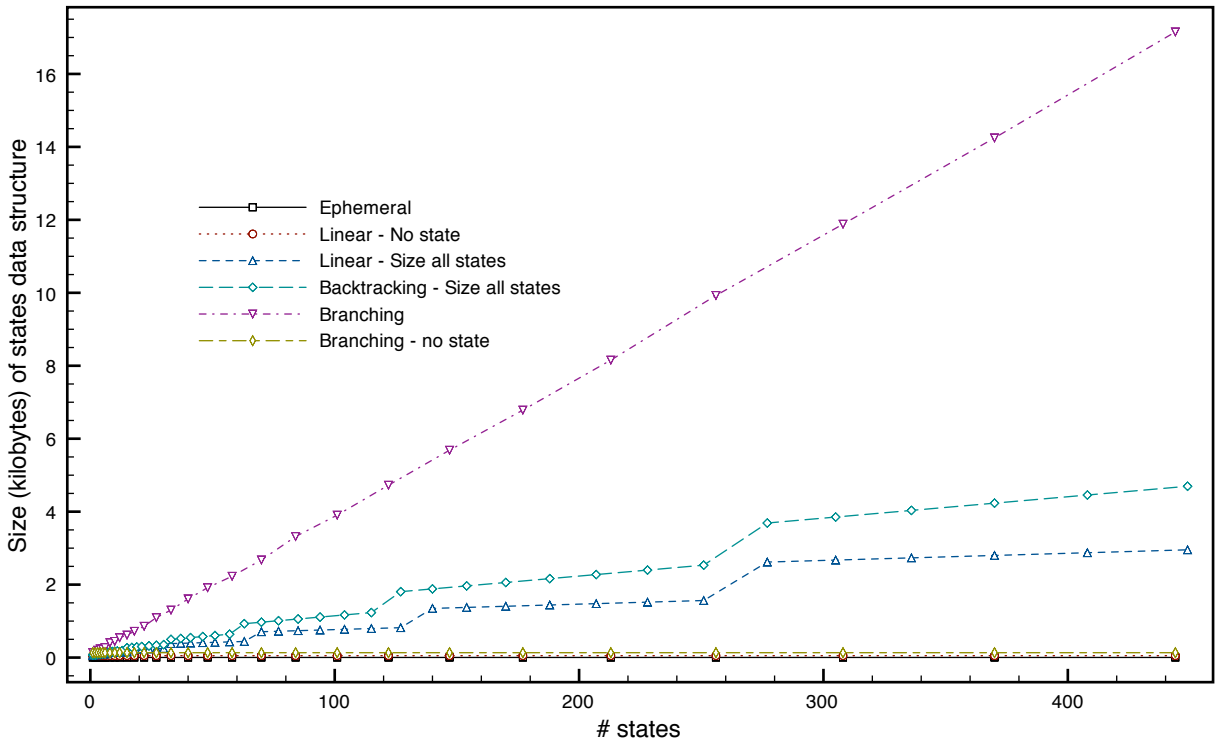


Figure 6.11: Numbers of updates vs. total size of a field.

We take snapshots after each update and we report the total size of the object (including our data structures). For linear and backtracking versioning, the size evolves linearly and there is no significant difference between both curves. Note how the size grows in steps: every jump corresponds to the creation of a new array where the actual states are stored because the last array is full. For the branching versioning the size evolves also linearly but the local tree takes more space than the chained arrays.

To analyze these results, we take the previous sizes and we divide them by the total size of the data they contain: if a field takes s bits in memory to store n integer states, we calculate $s/(n*8)$ to obtain the multiplicative factor to pay by our data structures to store all these integers (each integer takes 8 bytes in memory). Figure 6.12 shows these results. The curves show that after 30 states, the multiplicative factor becomes constant: the size is amortized by the number of saved states. These factors are about 7 for linear versioning,

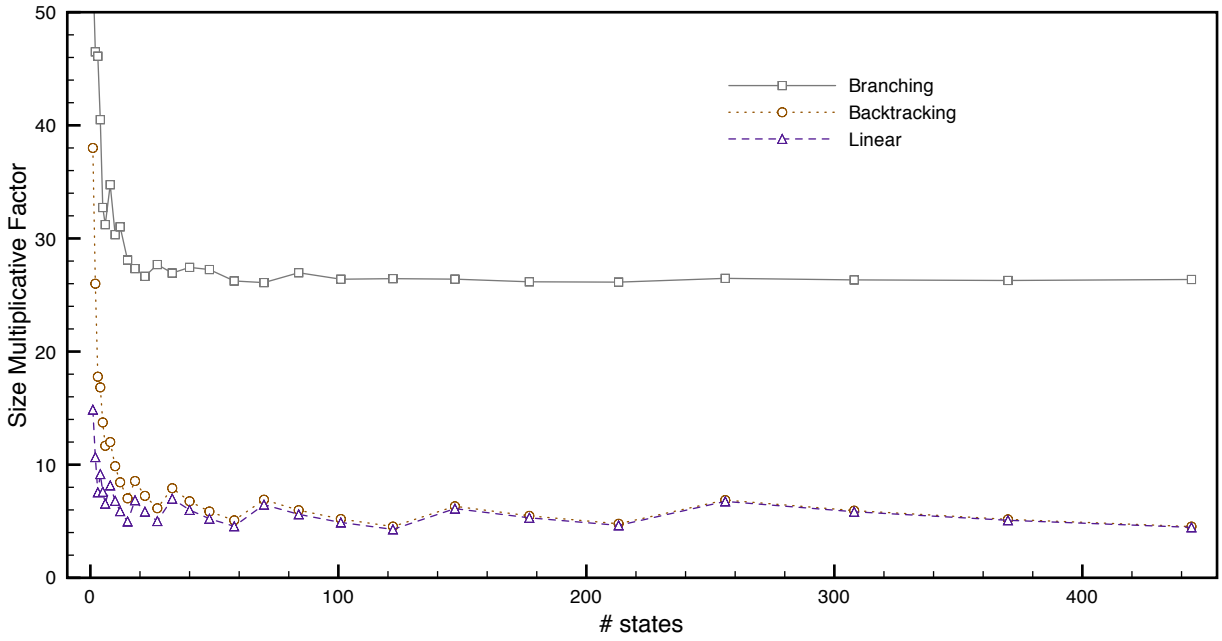


Figure 6.12: Numbers of updates vs. multiplicative factor of size.

10 for backtracking versioning and 40 for branching versioning. Therefore to store 10^5 states of integers (that takes $10^5 * 8$ bytes, i.e. 0.76Mo, to be stored), we need about 5.3 megabytes in linear, backtracking versioning and about 30.5 megabytes in branching versioning.

6.5.2 Java

Now we analyze the space in memory of our Java implementation.

As first test we take a simple class composed by 1, 2, 3 or 4 `Integer` fields. The original size is 8 bytes + 16 bytes per field (4 for the pointer and 12 for the Integer object). When transforming this class to a persistent one the aspect adds a field containing an optimized `Hashtable` instance and some useful informations for AspectJ. The size grows to 50 + 140 bytes per field, an overhead of a factor of about 8.

Figure 6.13 shows the total sizes of objects with, respectively, 1, 2, 3 and 4 fields after updates (of all fields), each one followed by a snapshot. The total size grows linearly according vertical steps due to instantiation of a new array in states structure at each

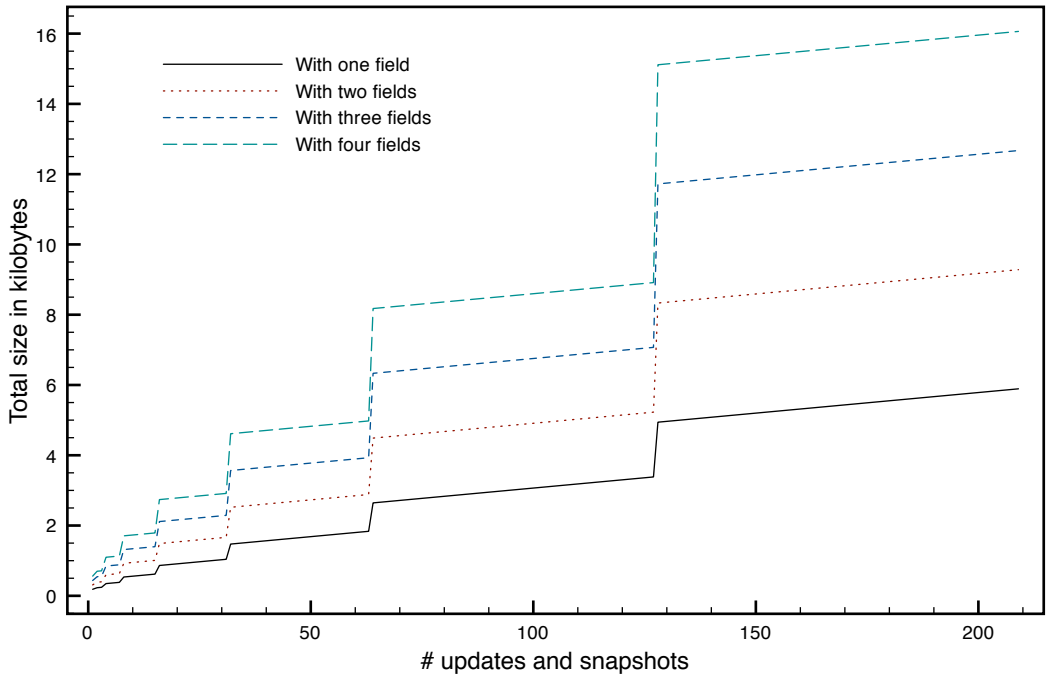


Figure 6.13: Sizes for object with 1, 2, 3 and 4 fields: number of update followed by snapshot vs. the size of the object

power of 2. The steps of the stair graphs are not horizontal because at each change a new state is created and added to the states structure.

6.6 Application Benchmarks

We now analyze the efficiency of our real applications: versioned random treaps, stateful tracer, post-conditions and planar point location solution.

6.6.1 Random Treap

We now test the performance of versioned treaps in Smalltalk and Java.

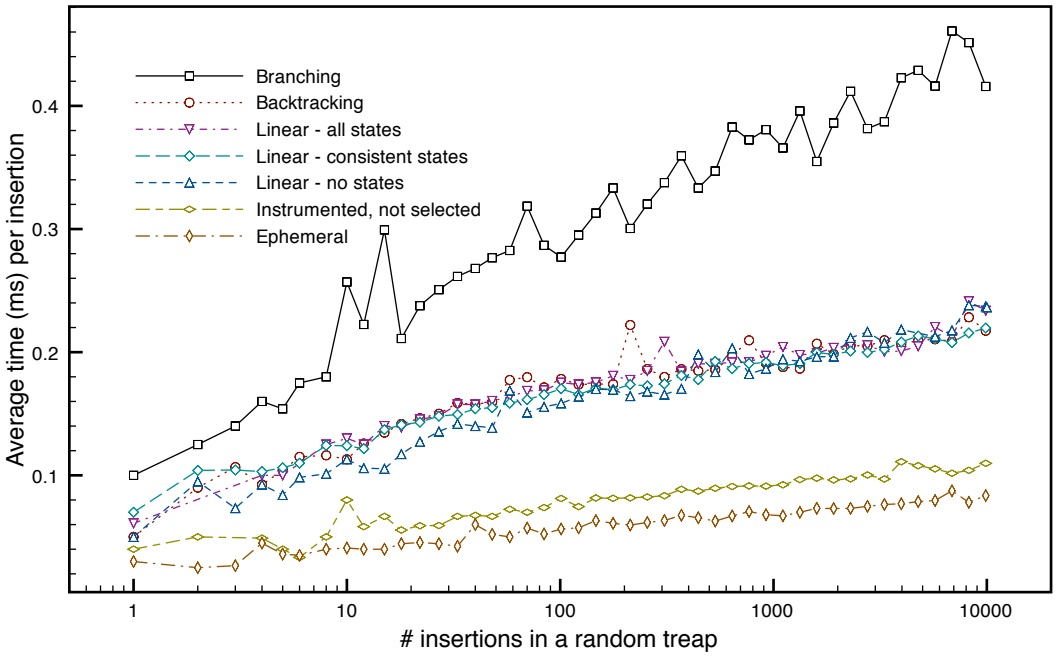


Figure 6.14: Execution times for saving states in a random treap.

6.6.1.1 Smalltalk

For the Smalltalk implementation, the access operations are instrumented by bytecode transformation using process-attached active snapshots. Figure 6.14 shows the average execution times per insertion in a random treap, in the following cases:

1. The treap is not instrumented.

2. The treap is instrumented but none of its fields selected. The slowdown factor is about 1.2;

3. Linear versioning: all fields of the treap are selected but no snapshot is taken. The slowdown factor is about 2.7.

4. Linear versioning: all fields are selected, and a snapshot is taken after each insertion. The slowdown factor is also about 2.7

5. Linear versioning: all fields are selected and snapshots is taken after every change (including the internal rebalancing happening in the treap for example). The slowdown factor is always about 2.7

6. Backtracking versioning: all fields is selected, and a snapshot taken after each insertion. The slowdown factor is about 2.7

7. Branching versioning: all fields are selected, and a snapshot is taken after each insertion. The slowdown factor is about logarithmic (from 4 to 5.3 on the graph).

The overall curves remain similar: they still show that the time 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 about 2.7, which is much better than 7 from the synthetic example. The explanation of these surprisingly low overheads compared to the synthetic benchmarks is the following. When an insertion is performed in a versioned treap the operations are either ephemeral (e.g. comparisons or assignments of temporary variables) or a read in present (the active snapshot is writable) or a versioned operation. Versioned operations are a minority and do not increase too much the total time. The same observations are be applied to the search operations.

Our second benchmark (Figure 6.15) gives the average time to browse each element of an ephemeral and a versioned random treap by browsing recursively each node. This operation on an ephemeral random treap takes linear time $O(n)$, as expected. If the treap is instrumented but not versioned, the execution time is 2 times slower.

For versioned treaps several cases are considered for each kind of versioning:

1. the active snapshot is writable;

2. the active snapshot is at middle of snapshots (if there are k insertions with snapshots, the active snapshot is the $(k/2)$th snapshot generated).

The slowdown factor is about 16 for linear versioning and 36 for backtracking versioning (regardless of the active snapshot). For branching versioning, if the active snapshot is writable, a logarithmic slowdown, varying between 50 and 80, is observed. If the active snapshot is at the middle of snapshots, the logarithmic slowdown factor is between 13 and 41.

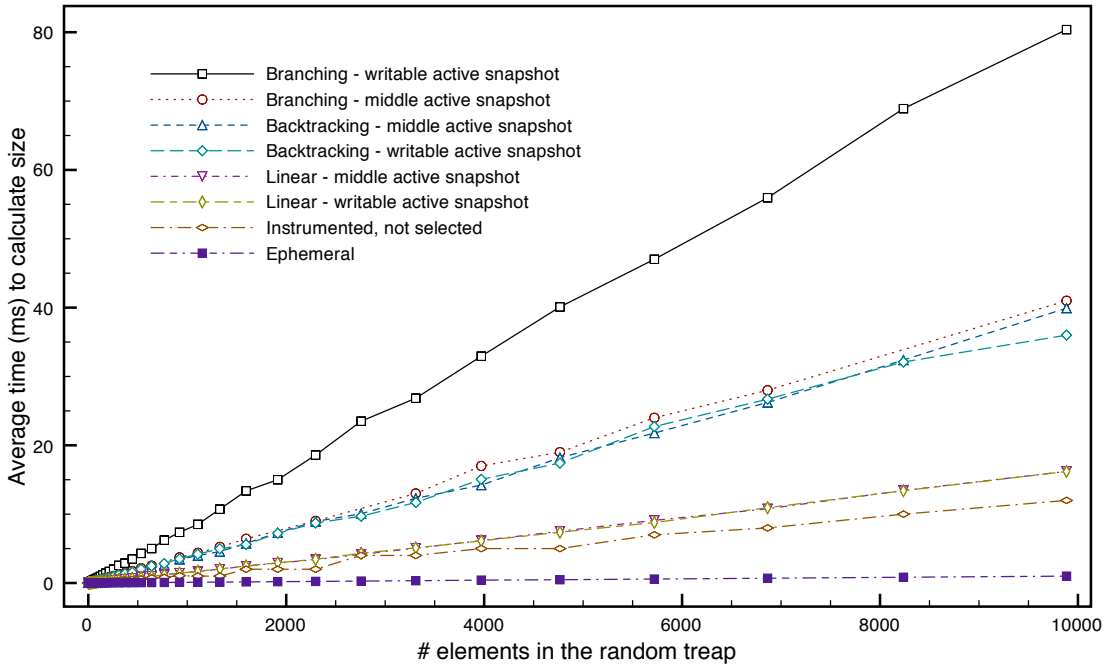


Figure 6.15: Browse each element of a random treap.

Finally we show the sizes required by versioned treap for each kind of versioning (Figure 6.16). The linear and backtracking versioning grow 6 times the size of the ephemeral structure while a factor of 16 is observed for branching versioning.

6.6.1.2 Java

Our first benchmark (see Figure 6.17) shows the average time per insertion in a treap vs. the number of elements in the treap. We insert n elements in a treap, we take the total time t and we calculate the average t/n. The same test is performed on ephemeral and versioned random treaps (without snapshot and with snapshot after each insertion). An overhead of roughly 2 is observed for versioned ones. Notice that taking snapshots after each insertion does not increase the time by insertion considerably, due to the fact that the lookup in the dictionary takes more time than updating the last state or adding a new state in the states structure.

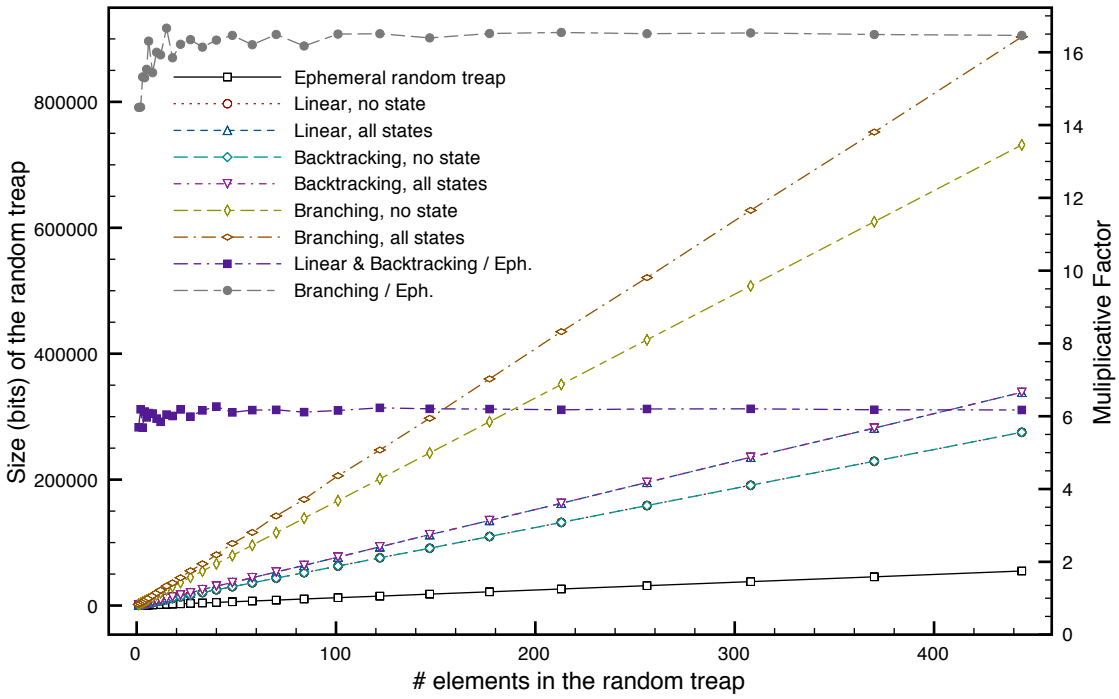


Figure 6.16: Numbers of update vs. total size of a field.

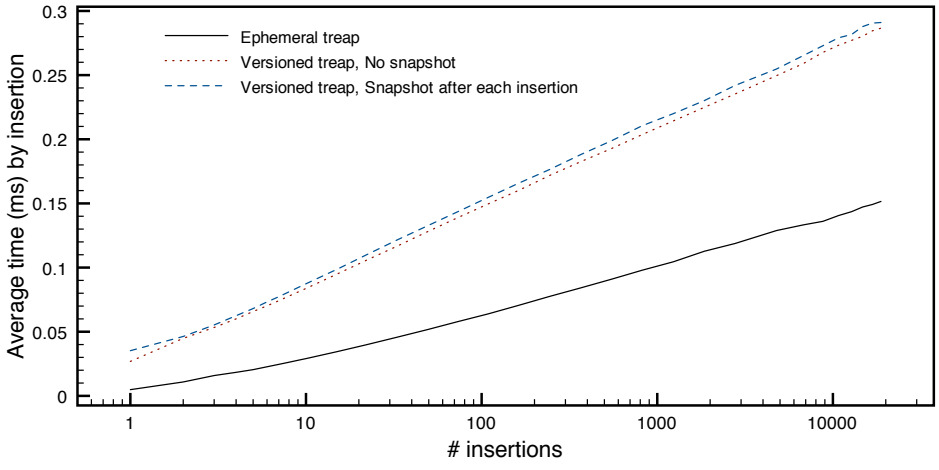


Figure 6.17: Insertion in ephemeral and versioned treaps: number of insertions vs. average time per insertion.

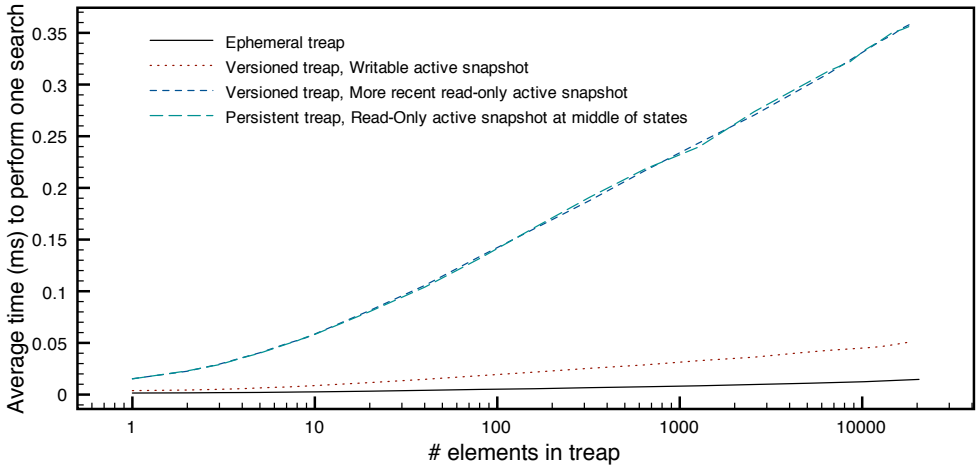


Figure 6.18: Search in ephemeral and versioned treaps: number of elements in treap vs. average time per search.

Our second benchmark (Figure 6.18) gives the average time for searching in ephemeral and versioned random treaps. As a first result experiments indicate that search in an ephemeral random treap takes time $O(\lg n)$. For versioned treaps several cases of the active snapshot are considered:

1. writable: the overhead is about 3.6;

2. active snapshot on the more recently read-only snapshot (the before last saved value is reached in the states structure): the overhead is about 25;

3. active snapshot at middle of states (if there are k insertions with snapshots, the active snapshot is the $(k/2)$th snapshot generated): the overhead is also about 25.

Note that the theoretical expected search time is $O(\lg n * \lg \lg n)$: the expected number of states in a treap node is no more than the logarithm of the size of its subtree. However in our tests, the dictionary lookup dominates the running time, explaining the roughly logarithmic curve.

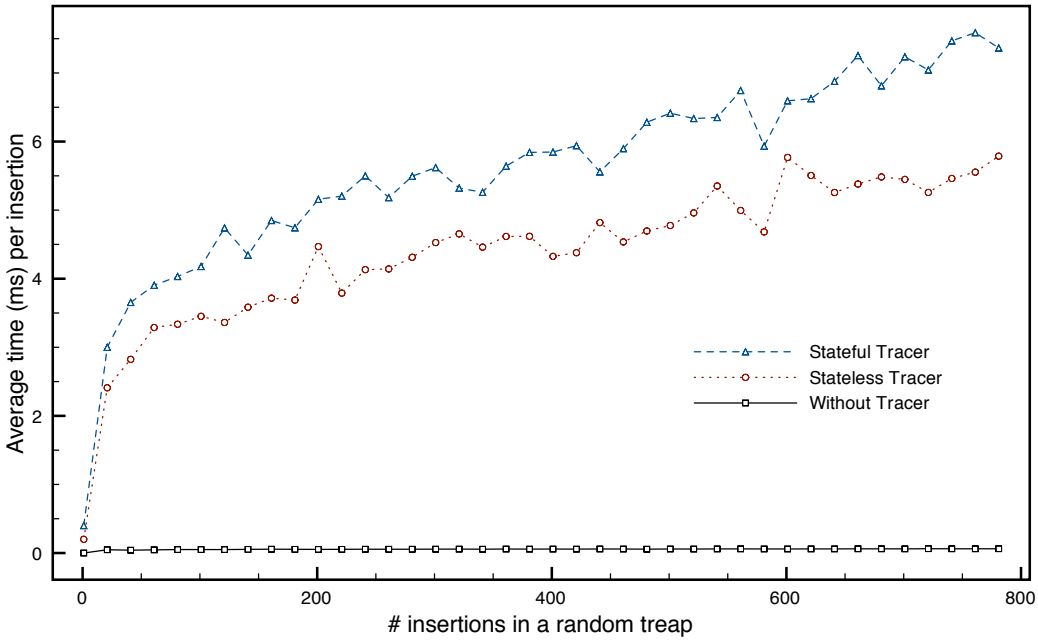


Figure 6.19: Execution times for adding elements in a treap without tracing, with a stateless tracer, and with a stateful tracer.

6.6.2 Capturing Stateful Execution Traces

This section shows the performance of our Smalltalk 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 measured 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 did the benchmarks when not tracing at all, for a stateless tracer that does not keep any state at all and for a stateful tracer that uses HistOOry as described in Section 6.1.1.

Figure 6.19 shows the results. Note how transforming a stateless tracer into a stateful one 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.

6.6.3 Postconditions

In Section 6.1.2 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.

swap:with: This is a method on class `SequenceableCollection` that swaps the place of elements on the indices given as argument. For our benchmark we created collections of different sizes (ranging in size from 1 to 800 elements). We added either simple integers or array objects of 100 elements pointing to `nil`). We then perform 10000 swaps at random indices and take the total time. Dividing this total time by 10000 gives us the average execution time per swap.

We have performed 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 Section 6.1.2. and the method where we added 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)])
```

Figure 6.20 shows the results. First of all, it shows that an implementation that uses copies has an execution time that grows linearly with the size of the collection (and very fast, depending on the size of the data structure). The implementations based on HistOOry have a constant cost that does not depend neither on the number of elements in the collection nor on the kind of the elements (integers or arrays). The reason is that 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 before executing the normal body of the method.

What is interesting, however, is that the copying approach is faster than the HistOOry

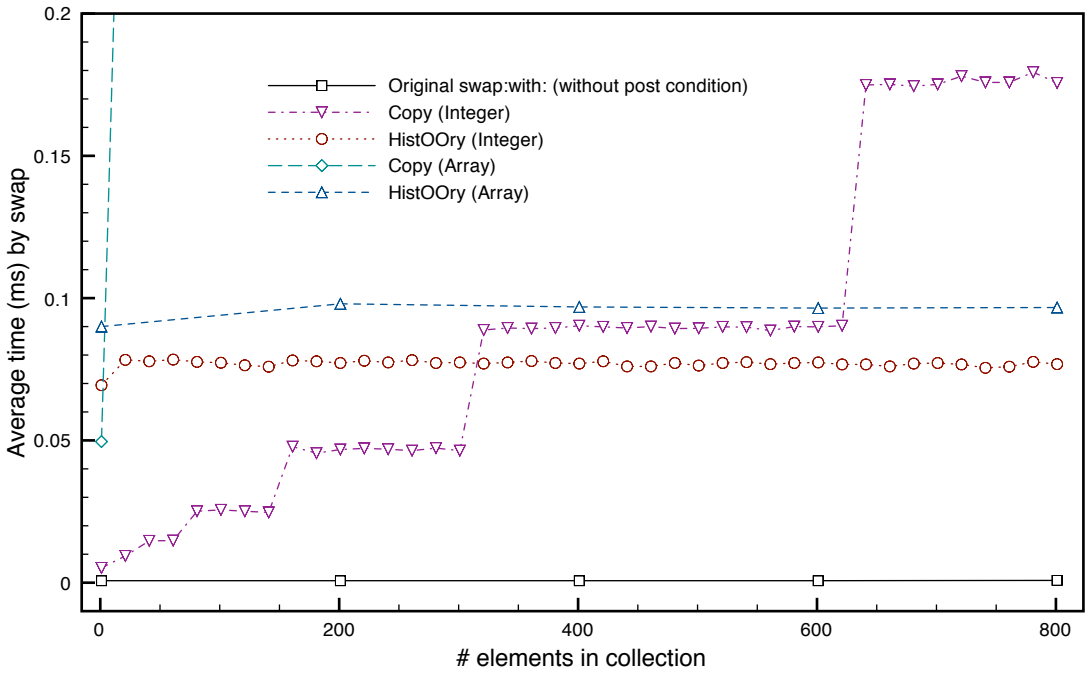


Figure 6.20: Postconditions (swap:with:): Number of elements in collection vs. time per swap.

based approach for smaller collection sizes. The reason is simple: 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 cost is higher than the simple copy operation for small collections.

addAll: This is a method on class `OrderedCollection` that adds all elements in the argument collection to the receiver collection. We compared two different scenarios. In the first 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 compared the original implementation with an implementation that has postconditions

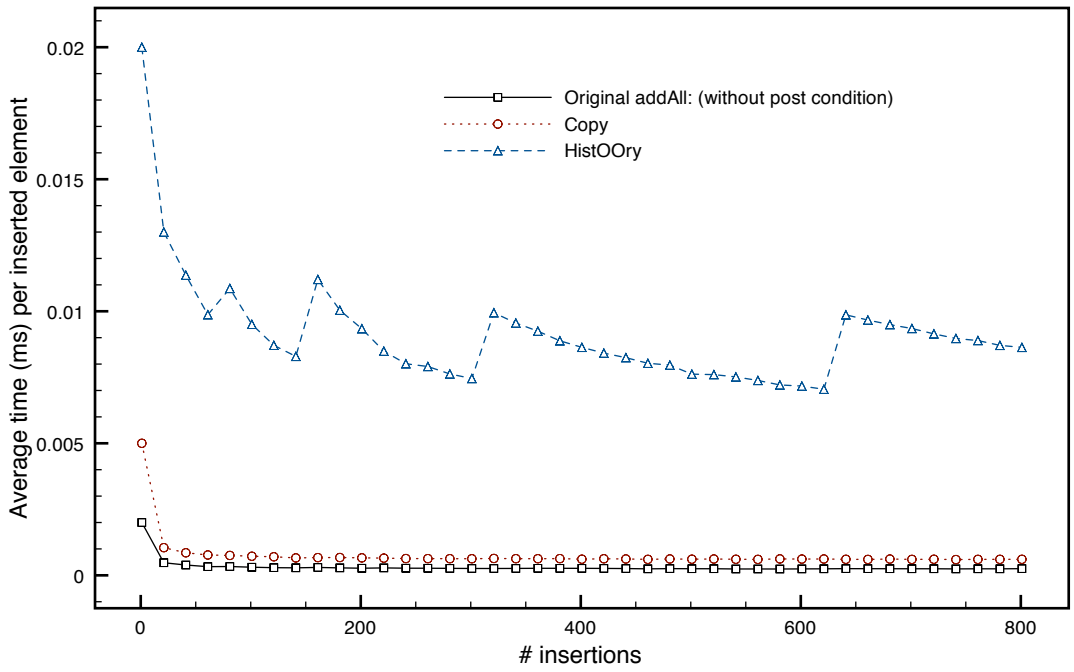


Figure 6.21: Numbers of elements in collection to add vs. time (ms) per insertion.

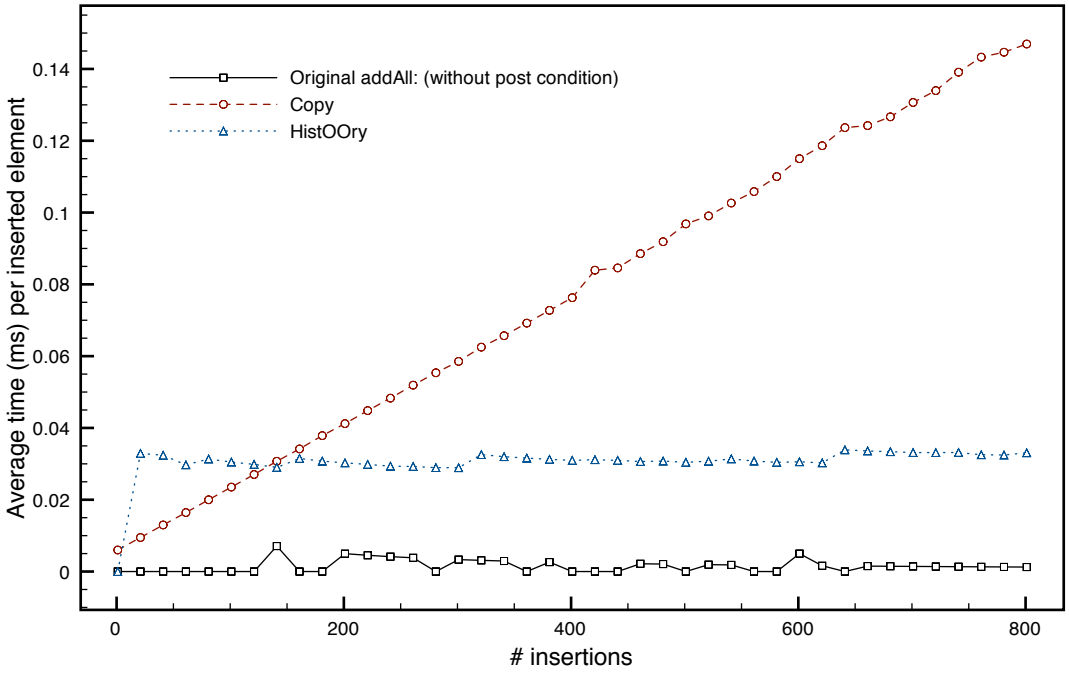


Figure 6.22: Number of elements added one by one vs. time (ms) per insertion.

based on HistOOry as shown in Section 6.1.2 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
```

The time results to add the larger collections are shown in Figure 6.21 while the time results to add collections of size 1 are shown in Figure 6.22. The results are similar to the previous experiment: we again see linear execution time for the implementations based on copying and bounded execution time for the HistOOry-based implementations.

6.6.4 Planar Point Location

The test for the planar point location was realized as follow. For each n, number of given points in the plane, we generate random points (using the command line program `rbox`[3]) and generate a Delaunay triangulation [Preparata & Shamos, 1985] for these points (using the command line program `qhull`[4]). We run our implementation of the planar point location using as parameters the set of points and the segments generated. We measure the time t to locate n other random points in the plane. Figure 6.23 shows the average time t/n to perform a search in Smalltalk implementation. As expected the curve is nearly logarithmic.

6.7 Conclusions

This chapter validates the expressiveness of our model and the efficiency of our implementation. We propose firstly case studies of three complex applications using HistOOry.

[3]http://www.qhull.org/html/rbox.htm
[4]http://www.qhull.org/html/qh-optq.htm

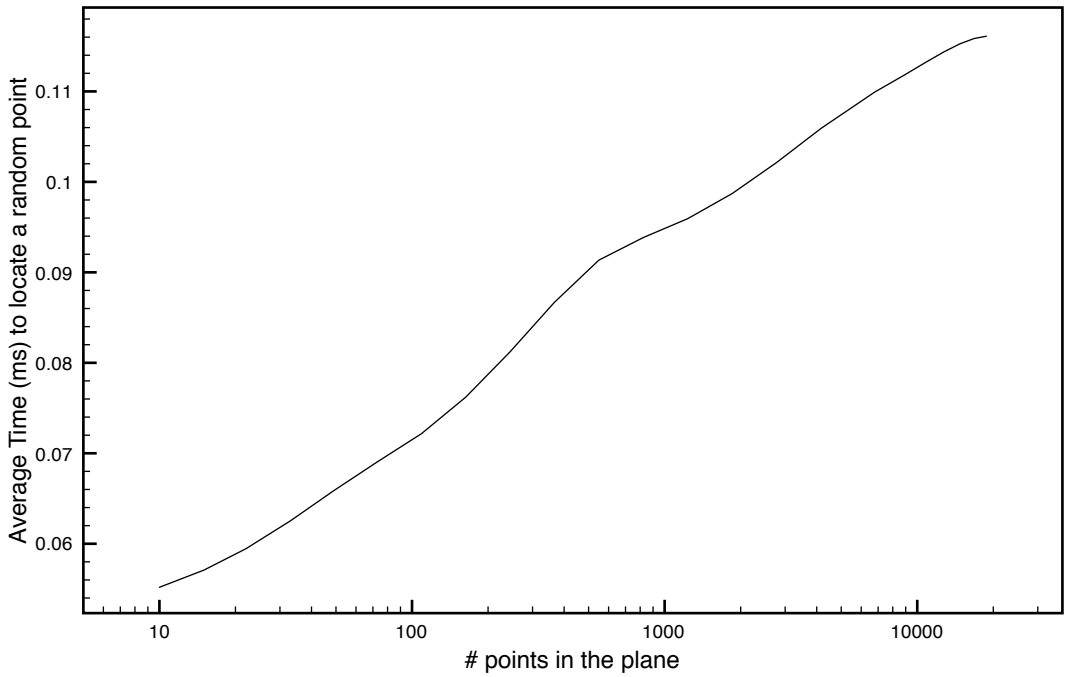


Figure 6.23: Planar Point Location

The case studies show three applications with different goals that use HistOOry. Our fine-grained model and our transparent implementation allows one to specify easily which states of which objects must be kept.

In a second part we provide a fine-grained study of the efficiency of our implementations (Smalltalk and Java). To be the most complete possible, we decompose our study to show the time consumed by every part of our implementation separately. We start by analyzing synthetic tests with a manually instrumented object and we compare the time of each operation to the ephemeral operation. Then we examine the time to add process-attached active snapshot and adds some MOP for more expressiveness. Then we show the cost of automatic instrumentation of field accesses to obtain a transparent integration. After the time efficiency, we study the space used by HistOOry in memory.

We finish this chapter by a study of the applications performance.

Chapter 7

Conclusions

In this dissertation we design an expressive and efficient object versioning system for object-oriented languages. We develop a model that allows the developer to save only what it is necessary. This selection has an impact on execution time and required space: we do not loose time and space to save useless states. We develop efficient data structures to implement this model in a real world. We show how this system can be integrated in an object-oriented language, especially the tools to do so in a transparent way. As validation we implemented our system in Smalltalk and Java and show real applications that use it. We finish by performing benchmarks to evaluate the efficiency of our implementations.

Chapter 2 presents the state of the art of versioning. We explain the existing methods of object versioning in the theoretical literature. These methods are compared and we discuss why we chose the fat node method for our implementations.

Chapters 3 to 6 each study one original contribution. In Chapter 3 we introduce a **fine-grained object versioning model**. This model allows one to save wanted states and forget the unwanted ones. To achieve this, the control is given to the developer, the only one who knows which are the interesting states to keep. The mechanisms provided by the model to keep the interesting states of interesting fields are the *selection* of fields and the *snapshots*. The selection marks the fields as versioned. The developer takes a snapshot each time the current values of selected fields must be kept. Selection allows a fine-grained control of what is really saved by the system, from very precise to all states of all fields. Although our system is fine-grained, manually selecting each field can be cumbersome. We propose an extension of the model to select automatically fields.

Snapshots are the central part of the model. Each snapshot is an object that contains

properties and answers to messages. The snapshots are kept by the developer (e.g. in snapshot sets, that are collections that answer to queries on snapshot relations). A snapshot is a view on the system at the time the snapshot was taken. When old states must be browsed, the corresponding snapshot is activated: all accesses to the selected fields are then executed at snapshot time while ephemeral fields are accessed as usually.

We stress three important features of our model. First, it is designed to be used in any object-oriented language. As example we successfully implemented it in Smalltalk and Java. Second, it is independent to the support used to save states (files, databases or application memory). Third, the model is compatible for the linear, backtracking and branching versioning.

In Chapter 4 we propose **efficient data structures and algorithms** to implement our model in memory only. They are adapted from the *fat node* method of Driscoll et al. [Driscoll *et al.* , 1986]. For each kind of versioning we develop our own data structures, where the tradeoffs between saving time, query time and space are well adapted to the common usage of in-memory object versioning.

Taking a snapshot always takes constant time (worst case for linear and backtracking versioning and amortized for branching versioning). That means that saving the state of the system at a given time is independent of the number of previously taken snapshots. In fact, the state of the system is saved during the fields accesses and not while the snapshot is taken.

When a new value is stored in a selected field, we save the old value if necessary. When the value of a selected field is queried, the value is sought in old states based on the active snapshot. We optimize our states data structure to store new states more effectively. The time to store a new value in a field f is constant in linear versioning, amortized constant in backtracking versioning and logarithmic in the number of saved states for f for branching versioning. The time to get the value of selected field f is logarithmic in the number of saved states for f for all kinds of versioning.

In Chapter 5 we show how to achieve **a well adapted integration of object versioning in object-oriented languages**. We firstly propose a clean API, in which three primitives are enough to use object versioning (`selectObject`,`Snapshot atNow` and `snapshot execute: aBlock`). We explain how aspects and bytecode transformation allow a transparent integration of object versioning such that the developer uses ephemeral and versioned

objects in exactly the same way. We also show how processes and MOPs can be used to improve the expressiveness of the model: that adds some new access points on the path between the field access and the states data structure. These access points can be used to configure object versioning in an even finer way.

Finally we give some implementation indications with respect to the behavior of garbage collectors and reflective methods.

In Chapter 6 we **evaluate our implementations** on both sides. We start by showing the real usage of our model to implement complex applications in Smalltalk. These examples show how easily our system is to build applications with distinct goals. Then we evaluate the efficiency by providing benchmarks on required time and space. We decided to do very fine-grained benchmarks for all substeps of our implementation. We first decompose the time for each basic operation (i.e. select a field, take a snapshot, store a value in a field and query the value of field). We note that the results for Smalltalk and Java are very close. Our experiences show that the worst-case synthetic slowdown factors to store a value that will be kept in a field are about 7 for the linear versioning, 20 for branching versioning and 250 for branching versioning. The query times are logarithmic as expected.

We then analyzed each indirection that adds to the basic operations: the usage of process, the usage of MOP, the usage of aspects and bytecode transformation for a transparent integration. The indirection to processes and MOP add some constant time as expected. We see that bytecode instrumentation allows fine-grained modifications such that the instrumented bytecodes are identical to those of a manual instrumentation. AspectJ and Java introduce two additional slowdowns. On one hand, we do not have a fine control on the output produced by AspectJ: instrumented code takes more time than manual instrumentation. Some more work, certainly related to aspects management, must be executed. On the other hand, Java is statically typed and AspectJ does not allow changing the static type of variables. Our states data structures must therefore be put in a dictionary in which the lookup introduces a big slowdown on execution time.

Because we decompose our benchmark, we hope that this is easier to implement the object versioning by using some tradeoffs between the implemented features (process, MOP and transparency) and their impact in the time.

We also estimate the size required by our data structures. As result we show that only 5.3 megabytes are necessary to save 10^5 integer states with linear and backtracking versioning

and 30.5 megabytes for branching versioning.

We finish with benchmarks on the real applications developed with our system. As example, we show that when we select a random treap (with its bytecodes instrumented), storing a value results in a slowdown factor of about 2.7 in linear versioning, 2.7 in backtracking versioning and 4 to 5.3 in branching versioning (between 1 and 10^4 states). These tests show that the slowdowns found for synthetic benchmarks (where each operation is an update or a query) are minimized in real usage (where each operation is not always related to the versioning).

7.1 One more thing...[1]

This dissertation sounds like the end of our research while in fact it is just the beginning for our successors. There are some related subjects we did not have time to explore.

We did not study merging versioning (also known as confluent persistence) [Driscoll *et al.* , 1994]. This kind of versioning is even more complicated than branching versioning. The difficulty stems from the efficiency of data structure where each version can be created from more than one version.

This dissertation limits itself by saving states in memory only. Obviously when the application is closed, all saved states are forgotten. It could be interesting to make a bridge between our implementation and existing persistent mechanisms to store states on physical support.

As last not least, we designed our data structures in such a way that all selected states of a field are kept while the object that contains the field is not deleted (or garbage collected). But it might be possible that some states are no longer useful because there are no more snapshot that point to it. Indeed the snapshots are managed by the developer. If these snapshots are deleted (or garbage collected), states that point to these snapshots are no longer accessible by the user and they could therefore be deleted. Care must be taken that the states removed are not shared between several snapshots. If this is correctly implemented, this feature could allow for even better query times and smaller sizes in memory.

[1] We hope there is no Apple's patent for this sentence :-)

List of Tables

Bibliography

[Acar *et al.* , 2004] Acar, Umut A., Blelloch, Guy E., Harper, Robert, Vittes, Jorge L., & Woo, Shan Leung Maverick. 2004. Dynamizing static algorithms, with applications to dynamic trees and history independence. *Pages 531–540 of: SODA '04: Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics.

[Agarwal, 1992] Agarwal, Pankaj K. 1992. Ray shooting and other applications of spanning trees with low stabbing number. *SIAM J. Comput.*, **21**(3), 540–570.

[Agarwal *et al.* , 2003] Agarwal, Pankaj K., Har-Peled, Sariel, Sharir, Micha, & Wang, Yusu. 2003. Hausdorff distance under translation for points and balls. *Pages 282–291 of: SCG '03: Proceedings of the nineteenth annual symposium on Computational geometry*. New York, NY, USA: ACM Press.

[Aho & Hopcroft, 1974] Aho, Alfred V., & Hopcroft, John E. 1974. *The Design and Analysis of Computer Algorithms*. 1st edn. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

[Ahuja *et al.* , 1993] Ahuja, Ravindra K., Magnanti, Thomas L., & Orlin, James B. 1993. *Network Flows: Theory, Algorithms, and Applications*. 1 edn. Prentice Hall.

[Alstrup *et al.* , 2001] Alstrup, Stephen, Brodal, Gerth Stlting, Grtz, Inge Li, & Rauhe, Theis. 2001. Time and Space Efficient Multi-Method Dispatching.

[Aronov *et al.* , 2006] Aronov, Boris, Bose, Prosenjit, Demaine, Erik D., Gudmundsson, Joachim, Iacono, John, Langerman, Stefan, & Smid, Michiel H. M. 2006. Data Structures for Halfplane Proximity Queries and Incremental Voronoi Diagrams. *Pages 80–92 of: Proc. of the 7th Latin American Symposium on Theoretical Informatics (LATIN'06)*.

[Arumugam & Thangaraj, 2006] Arumugam, Gurusamy, & Thangaraj, Muthuraman. 2006. An efficient multiversion access control in a Temporal Object Oriented Database. *Journal of Object Technology*, **5**(1), 105–116.

[Atallah & Fox, 1998] Atallah, Mikhail J., & Fox, Susan (eds). 1998. *Algorithms and Theory of Computation Handbook*. 1st edn. Boca Raton, FL, USA: CRC Press, Inc.

[Atkinson & Morrison, 1995] Atkinson, Malcolm, & Morrison, Ronald. 1995. Orthogonally persistent object systems. *The VLDB Journal*, **4**(July), 319–402.

[Aurenhammer & Schwarzkopf, 1991] Aurenhammer, Franz, & Schwarzkopf, Otfried. 1991. A simple on-line randomized incremental algorithm for computing higher order Voronoi diagrams. *Pages 142–151 of: SCG '91: Proceedings of the seventh annual symposium on Computational geometry*. New York, NY, USA: ACM Press.

[Beech & Mahbod, 1988] Beech, David, & Mahbod, Brom. 1988. Generalized Version Control in an Object-Oriented Database. *Pages 14–22 of: Proceedings of the Fourth International Conference on Data Engineering*. Washington, DC, USA: IEEE Computer Society.

[Ben-Amram & Galil, 1992] Ben-Amram, Amir M., & Galil, Zvi. 1992. On pointers versus addresses. *J. ACM*, **39**(July), 617–648.

[Bender *et al.* , 2002] Bender, Michael A., Cole, Richard, Demaine, Erik D., Farach-Colton, Martin, & Zito, Jack. 2002. Two Simplified Algorithms for Maintaining Order in a List. *Pages 152–164 of: Proceedings of the 10th Annual European Symposium on Algorithms*. ESA '02. London, UK, UK: Springer-Verlag.

[Bern, 1988] Bern, M. 1988. Hidden surface removal for rectangles. *Pages 183–192 of: SCG '88: Proceedings of the fourth annual symposium on Computational geometry*. New York, NY, USA: ACM Press.

[Bern *et al.* , 1990] Bern, Marshall, Dobkin, David, Eppstein, David, & Grossman, Robert. 1990. Visibility with a moving point of view. *Pages 107–117 of: SODA '90: Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics.

[Bjornerstedt & Britts, 1988] Bjornerstedt, Anders, & Britts, Stefan. 1988. AVANCE: an object management system. *SIGPLAN Not.*, **23**(January), 206–221.

[Bose *et al.* , 2003] Bose, Prosenjit, van Kreveld, Marc, Maheshwari, Anil, Morin, Pat, & Morrison, Jason. 2003. Translating a regular grid over a point set. *Comput. Geom. Theory Appl.*, **25**(1-2), 21–34.

[Brant *et al.* , 1998] Brant, John, Foote, Brian, Johnson, Ralph, & Roberts, Don. 1998. Wrappers to the Rescue. *Pages 396–417 of: Proceedings of ECOOP'98*.

[Cabello *et al.* , 2002] Cabello, Sergio, Liu, Yuanxin, Mantler, Andrea, & Snoeyink, Jack. 2002. Testing Homotopy for paths in the plane. *Pages 160–169 of: SCG '02: Proceedings of the eighteenth annual symposium on Computational geometry*. New York, NY, USA: ACM Press.

[Chazelle & Guibas, 1986] Chazelle, Bernard, & Guibas, Leonidas J. 1986. Fractional cascading: I. A data structuring technique. *Algorithmica*, **1**, 133–162.

[Cheng & Ng, 1996] Cheng, Siu-Wing, & Ng, Moon-Pun. 1996. Isomorphism testing and display of symmetries in dynamic trees. *Pages 202–211 of: SODA '96: Proceedings of the seventh annual ACM-SIAM symposium on Discrete algorithms*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics.

[Codd, 1970] Codd, E. F. 1970. A relational model of data for large shared data banks. *Commun. ACM*, **13**(June), 377–387.

[Collette *et al.* , 2011] Collette, S., Iacono, J., & Langerman, S. 2011. *Confluent Persistence Revisited*. Tech. rept. arXiv:1104.3045. arXiv, Cornell University. 15 pages.

[Cook & Reckhow, 1972] Cook, Stephen A., & Reckhow, Robert A. 1972. Time-bounded random access machines. *Pages 73–80 of: Proceedings of the fourth annual ACM symposium on Theory of computing*. STOC '72. New York, NY, USA: ACM.

[Cormen *et al.* , 2001] Cormen, Thomas H., Stein, Clifford, Rivest, Ronald L., & Leiserson, Charles E. 2001. *Introduction to Algorithms*. 2nd edn. McGraw-Hill Higher Education.

[Demaine *et al.* , 2004] Demaine, Erik D., Iacono, John, & Langerman, Stefan. 2004. Retroactive data structures. *Pages 281–290 of: SODA '04: Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics.

[Demaine *et al.* , 2008] Demaine, Erik D., Langerman, Stefan, & Price, Eric. 2008. Confluently Persistent Tries for Efficient Version Control. *Pages 160–172 of: Proceedings of the 11th*

Scandinavian workshop on Algorithm Theory. SWAT '08. Berlin, Heidelberg: Springer-Verlag.

[Demeyer *et al.* , 2002] Demeyer, Serge, Ducasse, Stéphane, & Nierstrasz, Oscar. 2002. *Object-Oriented Reengineering Patterns*.

[Denker *et al.* , 2005] Denker, Marcus, Ducasse, Stéphane, & Tanter, Éric. 2005. Runtime bytecode transformation for Smalltalk. *Computer Languages, Systems & Structures*, **32**(2-3), 125–139.

[Dietz & Sleator, 1987] Dietz, Paul, & Sleator, D. 1987. Two algorithms for maintaining order in a list. *STOC '87: Proceedings of the nineteenth annual ACM symposium on Theory of computing*.

[Dietz, 1989] Dietz, Paul F. 1989. Fully Persistent Arrays (Extended Array). *Pages 67–74 of: WADS '89: Proceedings of the Workshop on Algorithms and Data Structures*. London, UK: Springer-Verlag.

[Dobkin & Lipton, 1976] Dobkin, D., & Lipton, R. 1976. Multidimensional Searching Problems. *SIAM Journal of Computing 5*, 181–186.

[Dobkin & Munro, 1980] Dobkin, David P., & Munro, J. Ian. 1980. Efficient Uses of the Past. 200–206.

[Driscoll *et al.* , 1986] Driscoll, James R., Sarnak, Neil, Sleator, Daniel D., & Tarjan, Robert E. 1986. Making Data Structures Persistent. *Journal of Computer and System Sciences*, 86–124.

[Driscoll *et al.* , 1994] Driscoll, James R., Sleator, Daniel D. K., & Tarjan, Robert E. 1994. Fully persistent lists with catenation. *J. ACM*, **41**(September), 943–959.

[Ducasse, 1999] Ducasse, Stéphane. 1999. Evaluating Message Passing Control Techniques in Smalltalk. *Journal of Object-Oriented Programming (JOOP)*, **12**(6), 39–44.

[Ducasse *et al.* , 2006] Ducasse, Stéphane, Gîrba, Tudor, & Wuyts, Roel. 2006. Object-Oriented Legacy System Trace-based Logic Testing. *In: Proceedings 10th European Conference on Software Maintenance and Reengineering (CSMR 2006)*. IEEE Computer Society Press.

[Eckel, 2002] Eckel, Bruce. 2002. *Thinking in Java*. 3rd edn. Prentice Hall Professional Technical Reference.

[Edelsbrunner *et al.* , 2004] Edelsbrunner, Herbert, Harer, John, Mascarenhas, Ajith, & Pascucci, Valerio. 2004. Time-varying reeb graphs for continuous space-time data. *Pages 366–372 of: SCG '04: Proceedings of the twentieth annual symposium on Computational geometry*. New York, NY, USA: ACM Press.

[Edelweiss & Moreira, 2005] Edelweiss, Nina, & Moreira, Álvaro Freitas. 2005. Temporal and versioning model for schema evolution in object-oriented databases. *Data Knowl. Eng.*, **53**(May), 99–128.

[Eppstein, 1994] Eppstein, David. 1994. Clustering for faster network simplex pivots. *Pages 160–166 of: SODA '94: Proceedings of the fifth annual ACM-SIAM symposium on Discrete algorithms*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics.

[Feder & Motwani, 2005] Feder, Tomás, & Motwani, Rajeev. 2005. Finding large cycles in Hamiltonian graphs. *Pages 166–175 of: Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*. SODA '05. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics.

[Fiat & Kaplan, 2003] Fiat, Amos, & Kaplan, Haim. 2003. Making Data Structures Confluently Persistent. *Pages 16–58 of: J. Algorithms*.

[Gabow, 2004] Gabow, Harold N. 2004. Finding paths and cycles of superpolylogarithmic length. *Pages 407–416 of: Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*. STOC '04. New York, NY, USA: ACM.

[Gabow & Nie, 2008] Gabow, Harold N., & Nie, Shuxin. 2008. Finding Long Paths, Cycles and Circuits. *Pages 752–763 of: Proceedings of the 19th International Symposium on Algorithms and Computation*. Berlin, Heidelberg: Springer-Verlag.

[Goodrich & Tamassia, 1991] Goodrich, Michael T., & Tamassia, Roberto. 1991. Dynamic trees and dynamic point location. *Pages 523–533 of: STOC '91: Proceedings of the twenty-third annual ACM symposium on Theory of computing*. New York, NY, USA: ACM Press.

[Guéhéneuc *et al.* , 2002] Guéhéneuc, Yann-Gaël, Douence, Rémi, & Jussien, Narendra. 2002. No Java without Caffeine: A Tool for Dynamic Analysis of Java Programs. *Pages 117– of:*

Proceedings of the 17th IEEE international conference on Automated software engineering. ASE '02. Washington, DC, USA: IEEE Computer Society.

[Gupta *et al.* , 1994] Gupta, Prosenjit, Janardan, Ravi, & Smid, Michiel. 1994. Efficient algorithms for generalized intersection searching on non-iso-oriented objects. *Pages 369–378 of: SCG '94: Proceedings of the tenth annual symposium on Computational geometry*. New York, NY, USA: ACM Press.

[Hajiyev *et al.* , 2006] Hajiyev, Elnar, Verbaere, Mathieu, & de Moor, Oege. 2006. CodeQuest: Scalable Source Code Queries with Datalog. *Pages 2–28 of: Proceedings of ECOOP '06*.

[Hamou-Lhadj & Lethbridge, 2004] Hamou-Lhadj, Abdelwahab, & Lethbridge, Timothy. 2004. A Survey of Trace Exploration Tools and Techniques. *Pages 42–55 of: Proceedings IBM Centers for Advanced Studies Conferences (CASON 2004)*.

[Hershberger, 2006] Hershberger, John. 2006. Improved output-sensitive snap rounding. *Pages 357–366 of: SCG '06: Proceedings of the twenty-second annual symposium on Computational geometry*. New York, NY, USA: ACM Press.

[Jones & Lins, 1996] Jones, Richard, & Lins, Rafael D. 1996. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons.

[Karger *et al.* , 1993] Karger, David, Motwani, Rajeev, & Ramkumar, G.D.S. 1993. On Approximating the Longest Path in a Graph. *Algorithmica*, **18**, 421–432.

[Khaddaj *et al.* , 2004] Khaddaj, Souheil, Adamu, Abdul, & Morad, Minur. 2004. Object versioning and information management. *Information & Software Technology*, **46**(7), 491–498.

[Kiczales & Hilsdale, 2001] Kiczales, Gregor, & Hilsdale, Erik. 2001. Aspect-oriented programming. *SIGSOFT Softw. Eng. Notes*, **26**(September).

[Kiczales & Rivieres, 1991] Kiczales, Gregor, & Rivieres, Jim Des. 1991. *The Art of the Metaobject Protocol*. Cambridge, MA, USA: MIT Press.

[Kiczales *et al.* , 2001] Kiczales, Gregor, Hilsdale, Erik, Hugunin, Jim, Kersten, Mik, Palm, Jeffrey, & Griswold, William G. 2001. An overview of AspectJ. *Pages 327–353 of: Proceedings of ECOOP'01*.

[Klein, 2005] Klein, Philip N. 2005. Multiple-source shortest paths in planar graphs. *Pages 146–155 of: SODA '05: Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics.

[Koltun, 2001] Koltun, Vladlen. 2001. Segment intersection searching problems in general settings. *Pages 197–206 of: SCG '01: Proceedings of the seventeenth annual symposium on Computational geometry*. New York, NY, USA: ACM Press.

[Lange & Nakamura, 1997] Lange, Danny B., & Nakamura, Yuichi. 1997. Object-Oriented Program Tracing and Visualization. *Computer*, **30**(5), 63–70.

[Li, 1999] Li, Xue. 1999. A Survey of Schema Evolution in Object-Oriented Databases. *Pages 362– of: Proceedings of the 31st International Conference on Technology of Object-Oriented Language and Systems*. TOOLS '99. Washington, DC, USA: IEEE Computer Society.

[Lienhard *et al.* , 2008] Lienhard, Adrian, Gîrba, Tudor, & Nierstrasz, Oscar. 2008. Practical Object-Oriented Back-in-Time Debugging. *Pages 592–615 of: ECOOP '08: Proceedings of the 22nd European conference on Object-Oriented Programming*. Berlin, Heidelberg: Springer-Verlag.

[Liu, 1996] Liu, Zhiqing. 1996. A persistent runtime system using persistent data structures. *Pages 429–436 of: SAC '96: Proceedings of the 1996 ACM symposium on Applied Computing*. New York, NY, USA: ACM Press.

[Mehlhorn *et al.* , 1994] Mehlhorn, K., Sundar, R., & Uhrig, C. 1994. Maintaining dynamic sequences under equality-tests in polylogarithmic time. *Pages 213–222 of: SODA '94: Proceedings of the fifth annual ACM-SIAM symposium on Discrete algorithms*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics.

[Meyer, 1992] Meyer, Bertrand. 1992. Applying Design by Contract. *IEEE Computer (Special Issue on Inheritance & Classification)*, **25**(10), 40–52.

[Miller & Thomas, 1976] Miller, Lance A., & Thomas, John C. 1976. Behavioral Issues in the use of interactive systems. *Pages 193–216 of: Interactive Systems, Proceedings, 6th Informatik Symposium*. London, UK: Springer-Verlag.

[Millstein & Chambers, 1999] Millstein, Todd, & Chambers, Craig. 1999. Modular Statically Typed Multimethods. *Pages 279–303 of: Proceedings of ECOOP '99*.

[Okasaki, 1998] Okasaki, Chris. 1998. *Purely functional data structures*. Cambridge U.K. New York: Cambridge University Press.

[Oussalah & Urtado, 1996] Oussalah, C., & Urtado, C. 1996. Adding semantics for version propagation in OODBs. *Pages 222– of: Proceedings of the 7th International Workshop on Database and Expert Systems Applications*. DEXA '96. Washington, DC, USA: IEEE Computer Society.

[Oussalah & Urtado, 1997] Oussalah, Chabane, & Urtado, Christelle. 1997. Complex Object Versioning. *Pages 259–272 of: Proceedings of the 9th International Conference on Advanced Information Systems Engineering*. London, UK: Springer-Verlag.

[Overmars, 1981] Overmars, Mark H. 1981. *Search In The Past II*. Tech. rept. RUU-CS-81-9.

[Parrish *et al.* , 1998] Parrish, Allen, Dixon, Brandon, Cordes, David, Vrbsky, Susan, & Lusth, John. 1998. Implementing persistent data structures using C++. *Software: Practice and Experience*, **28**(15), 1559–1579.

[Pilato *et al.* , 2008] Pilato, C. Michael, Collins-Sussman, Ben, & Fitzpatrick, Brian W. 2008. *Version Control with Subversion*. 2 edn. O'Reilly Media.

[Pothier *et al.* , 2007] Pothier, Guillaume, Tanter, Éric, & Piquer, José. 2007. Scalable omniscient debugging. *SIGPLAN Not.*, **42**(10), 535–552.

[Preparata & Shamos, 1985] Preparata, Franco P., & Shamos, Michael I. 1985. *Computational geometry: an introduction*. New York, NY, USA: Springer-Verlag New York, Inc.

[Reiss & Renieris, 2000] Reiss, Steven P., & Renieris, Manos. 2000. Generating Java trace data. *Pages 71–77 of: Proceedings of the ACM 2000 conference on Java Grande*. ACM Press.

[Reiss & Renieris, 2001] Reiss, Steven P., & Renieris, Manos. 2001. Encoding Program Executions. *Pages 221–230 of: Proceedings of the 23rd International Conference on Software Engineering*. Toronto, Ontario, Canada: IEEE.

[Reps *et al.* , 1983] Reps, Thomas, Teitelbaum, Tim, & Demers, Alan. 1983. Incremental Context-Dependent Analysis for Language-Based Editors. *ACM Trans. Program. Lang. Syst.*, **5**(July), 449–477.

[Rodríguez *et al.* , 1999] Rodríguez, L., Ogata, H., & Yano, Y. 1999. TVOO: a temporal versioned object-oriented data model. *Inf. Sci.*, **114**(March), 281–300.

[Rumbaugh *et al.* , 1999] Rumbaugh, James, Jacobson, Ivar, & Booch, Grady (eds). 1999. *The Unified Modeling Language reference manual*. Essex, UK, UK: Addison-Wesley Longman Ltd.

[Sarnak & Tarjan, 1986] Sarnak, Neil, & Tarjan, Robert E. 1986. Planar point location using persistent search trees. *Commun. ACM*, **29**(7), 669–679.

[Seidel & Aragon, 1996] Seidel, Raimund, & Aragon, Cecilia R. 1996. Randomized Search Trees. *Algorithmica*, **16**(4/5), 464–497.

[Snodgrass, 1992] Snodgrass, Richard Thomas. 1992. Temporal Databases. *Pages 22–64 of: Proceedings of the International Conference GIS - From Space to Territory: Theories and Methods of Spatio-Temporal Reasoning on Theories and Methods of Spatio-Temporal Reasoning in Geographic Space*. London, UK: Springer-Verlag.

[Tanter *et al.* , 2002] Tanter, Éric, Ségura-Devillechaise, Marc, Noyé, Jacques, & Piquer, José M. 2002. Altering Java Semantics via Bytecode Manipulation. *Pages 283–298 of: Proceedings of the 1st ACM SIGPLAN/SIGSOFT conference on Generative Programming and Component Engineering*. GPCE '02. London, UK: Springer-Verlag.

[Turek *et al.* , 1992] Turek, John, Wolf, Joel L., Pattipati, Krishna R., & Yu, Philip S. 1992. Scheduling parallelizable tasks: putting it all on the shelf. *Pages 225–236 of: SIGMETRICS '92/PERFORMANCE '92: Proceedings of the 1992 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*. New York, NY, USA: ACM Press.

[Vishwanathan, 2000] Vishwanathan, Sundar. 2000. An approximation algorithm for finding a long path in Hamiltonian graphs. *Pages 680–685 of: Proceedings of the eleventh annual ACM-SIAM symposium on Discrete algorithms*. SODA '00. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics.

[Westbrook & Tarjan, 1989] Westbrook, J., & Tarjan, R. E. 1989. Amortized analysis of algorithms for set union with backtracking. *SIAM J. Comput.*, **18**(February), 1–11.

[Willis *et al.* , 2006] Willis, Darren, Pearce, David J., & Noble, James. 2006. Efficient Object Querying for Java. *Pages 28–40 of: Proceedings of ECOOP'06*.

[Won, 1990] Won, Kim. 1990. *Introduction to object-oriented databases*. MIT Press, Cambridge, Mass. :.

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

[Yellin, 1992] Yellin, Daniel M. 1992. Algorithms for subset testing and finding maximal sets. *Pages 386–392 of: SODA '92: Proceedings of the third annual ACM-SIAM symposium on Discrete algorithms*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics.

[Zdonik, 1984] Zdonik, Stanley B. 1984. Object management system concepts. *Pages 13–19 of: Proceedings of the second ACM-SIGOA conference on Office information systems*. COCS '84. New York, NY, USA: ACM.

[Zhang & Li, 2007] Zhang, Zhao, & Li, Hao. 2007. Algorithms for long paths in graphs. *Theor. Comput. Sci.*, **377**(May), 25–34.