Evolution Persistence For Objects

Frédéric Pluquet*
Université Libre de Bruxelles
Bruxelles, Belgique
fpluquet@ulb.ac.be

Roel Wuyts†
Université Libre de Bruxelles
Bruxelles, Belgique
roel.wuyts@ulb.ac.be

ABSTRACT

In literature the word "persistence" has different meanings. It is either used to indicate the storage of the state of objects, or in the context of versioning of objects. Firstly this paper establish the separation of these two concepts, and explain the goals of both. The second kind of persistence appears very interesting to us. In the rest of the paper we try to find all types of applications that, written in a language that would include a persistent system, benefits of the new features. For the moment the idea of a new kind of debugger is promising to exploit all resources of this mechanism. We finish this article with some ideas for the continuation of this research.

1. INTRODUCTION

The notion of persistence is used in several places in existing literature, and seems to be a well understood notion. However, when investigating in some detail we see that different semantics are attached to this word in different contexts. One usage of the word persistence is used to describe the storing of objects, and we therefore dubbed it *storable persistence*. The second usage of the word deals with the ability to deal with versioning of objects, and we named it *versioning persistence*. Before the rest of the paper looks in detail at versioning persistence, and what we can do with it, we first have a more detailed look at these two forms of persistence.

- **storable persistence.** The first usage of persistence indicates *the possibility to store objects and to reload them afterwards*, and is the most frequent occurrence. This kind of persistence is used to save the state of objects (i.e. all values of attributes of objects at a given instant) in a physical medium (files, databases, ...). The majority of programs that use storable persistence somehow serialises the objects to some representation (proprietary formats, XML descriptions, database storage, ...). The goal is to make sure that the data in the objects is not lost. Typical is also that the data to be stored is big, and cannot be contained in memory. For example, an insurance company will keep all the data of their clients stored in some databases, and applications that need data will load it from this database, modify it, and write it back. Java Data Objects (JDO) [1] and Smalltalk Images are examples of applications using storable persistence.
- **versioning persistence.** The second usage of persistence indicates *an ability to revert to previous versions of objects*, where a *version* of an object is simply a saved state of an object, possibly including metadata like timestamps or even version numbers. This means that versions of objects can be saved, and that versions can be retrieved easily later on. Versioning persistence is used in a variety of areas (computational geometry, text and file editing, ...) [2]. The research in this area is, up until now, primarily done in the field of algorithmic.

The goals of this article are twofold. The first one is the separation of the two concepts of persistence that we identified. The second one is to study versioning persistence in some more detail, and introducing possible applications.

2. STORABLE PERSISTENCE

If we search for the word "persistence" in literature, nine out of ten times we will find documents that talk about storing objects in a physical system. The purpose of storable persistence in an object oriented context is very simple : save and load objects when necessary.

This simple principle gives rise to very hard problems regarding *efficient* reading and writing of objects. Therefore lots of different approaches exist for this problem: either using proprietary file formats, using different database systems (relational, relational object oriented, object oriented or temporal object oriented database, too name a few), creating optimized data structures to store data efficiently in databases, or using techniques such as *the object faulting*[3, 4, 5] or *an automatic prefetching in queries to database*[6] to minimize loading of objects from a store . Describing all of these techniques is beyond the scope of this paper.

The applications using storable persistence can be divided

*http://www.ulb.ac.be/di/fpluquet/
†http://homepages.ulb.ac.be/~rowuyts/

into two kinds :

1. The first kind adds this persistence to keep ephemeral objects into a backup system. A copy of live objects on physical support is there to restore the system, or a part of the system, after a crash. The data manipulated by these applications are often very important and a loss of data could mean a big loss of revenue.

2. The second kind uses storable persistence to store data that is too big too fit in main memory into a secondary memory. In this case, swaps between the secondary and main memory are necessary.

3. VERSIONING PERSISTENCE

The purpose of versioning persistence is simply to keep versions of objects, such that particular versions of objects can be manipulated on demand.

Several data structures were developed to deal with versioning persistence. These data structures constrain what can be done with older versions of objects to optimize space and/or execution times. There are three well known techniques in existence today: partial, total and confluent persistence [2, 7]. Note that describing the algorithms for these different kinds of versioning persistence is out of the scope of this paper.

- **Partial persistence** An object is partially persistent if all versions can be accessed but only the newest version can be modified (i.e. create a new version from the newest version) [2].

- **Total persistence** Unlike partial persistence, total persistence allows one to read and modify *any* version of an object [2]. Modifying an old version A, which already had a newer version B, creates a new version C. The version B is still accessible, and therefore the version A will then have two versions following it (B and C). Figure 5 shows an object with two concurrent versions.

  We define *branches of a version* to be the different versions emerged from one version. We furthermore define *concurrent versions* to be the versions contained in different branches emerged from a same version. It is important to understand that in a totally persistent context zero, one or more versions can exist after each version.

- **Confluent persistence** Confluent persistence is like total persistence, but adds one more feature : the capacity to merge two concurrent versions of an object. A merge of two versions results in a single new version. This type of persistence was introduced by Amos Fiat and Haim Kaplan [7].

We like to stress that although they have different goals and usage, storable and versioning persistence are not incompatible. They can work together : storable persistence can offer means of storage to store the big number of versions needed to be kept by versioning persistence.

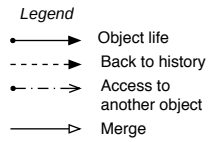

Figure 1: Legend of symbols used in graphs

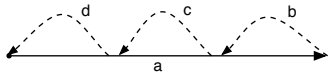

Figure 2: Introspective Persistence: querying past versions of an object

4. CATEGORIZING VERSIONING PERSISTENCE USAGES

Versioning persistence allows one to retrieve and manipulate previous versions of objects. As described in the previous section, different algorithms were devised, with partial and total persistence offering less functionality than confluent persistence. From the rest of the paper we assume to be using versioning persistence with a confluent versioning system. This choice is motivated by the applications we see for this type of persistence, which we detail in the next section.

This section categorises the basic ways versioning persistence can be used, either for introspection, or for reversion.

Note that throughout this section we illustrate these different forms of persistence with a number of figures. Time on all of these figures flows from left to right. The legend is described in Figure 1.

4.1 Introspection

Using versioning persistence for introspection allows a particular version of an object to query previous versions of this object. We see two kinds of these queries that can be useful: queries on the past of a single object and queries on the past of a number of objects.

The basic query of introspective persistence is the following : *Which are the previous values for a given variable of a given object ?* It is illustrated in Figure 2.

When considering multiple objects, we can relate the state of different objects and pose a number of different queries:

1. A first query relates the value of a variable of one object with the value of a variable of a second object at the same time (see Figure 3). In the same vain it can be interesting to know the state of an object before the last change of an another object.

2. A second question is similar to the basic query, but in case of an object with concurrent versions. The question stills the same, but an interesting case is the following : extreme versions of an object, i.e. the last versions of each branches, can be inspected by another ob-

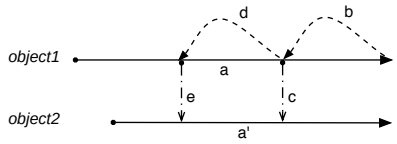

Figure 3: Comparing objects at different versions

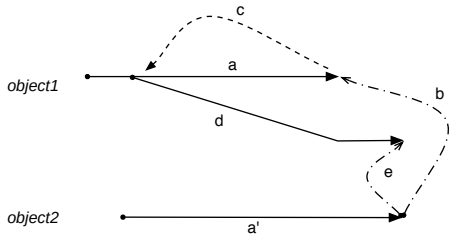

Figure 4: An object compares two concurrent versions of an other object

ject. This object can extract information about these two versions, for example to calculate metrics (see the illustration in Figure 4).

Note that we have not explicitly included information about time in the queries. This could be included in the same way as including versions, and we could then pose queries lie: *What was the state of this variable ten minutes ago?* or *How many minutes have passed between the last change of state of this variable?*.

4.2 Reversion

Using versioning persistence for reversion allows one to revert to a previous version of an object, and start modifying it (hence branching versions) like in Fig. 5, or to merge different versions (Fig. 6).

5. VERSIONING PERSISTENCE AT WORK

This section investigates a number applications we could built if we would have an object oriented language that supported versioning persistence. It therefore motivates our research for such language, and indicates the systems we would like to built to validate versioning persistence.

It is very important to remark that the applications should be able to use versioning persistence without implementing it : the proposed layer will be fully-tested and optimised.

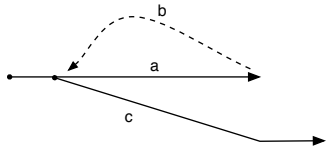

Figure 5: Working on a past state and creating a new history of the same object

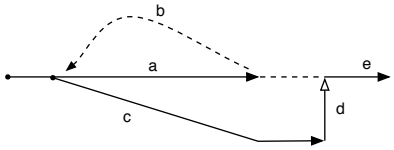

Figure 6: Two versions of an object are merged to give a new version

Developers can therefore work without worrying about persistence and include it when needed during development in an easy and sure way.

A first type of applications that can benefit from versioning persistence are applications where the history of objects is used as an add-on of the main application to improve its functionality. For example, text editors (including all integrated development environments (IDEs) and so on) use the history of text editions to offer the *Undo* functionality. This kind of application could be built without the *Undo* aspect, which could be added later on by falling back to the versioning persistence.

We can even go one step further, and have applications that can revert to the history of an object, and branch of new modifications of this previous state. This is typically not possible in most current applications: most of the time, an application that allows one to undo changes or go backwards in time, results in the old branch being replaced by the newest one. As with the first type of applications, using versioning persistence could add the possibility of working with graphs of objects to applications that currently have no or limited undo facilities.

A second kind of applications are those that need to compare versions of objects. We can find in this category all file servers (like CVS, SVN, ...) where the goal is to save different versions of a same object and to offer the possibility to compare two versions of the same object, merge branches, etc. This is a straightforward of versioning persistence.

A third example are new generations of debuggers. A debugger based on versioning persistence can compare states of an object obtained by sending the same chain of messages on this object but with different parameters. The user can then exploit information given by the debugger, inspecting other states of this object, to take decisions about the application of a message. We have already implemented a first and very promising version of a debugger that provides such functionality. It uses a logic programming language to query execution trace information (as published in [8]), but it could benefit from proper versioning persistence instead of the ad hoc deep copying of objects that is used now.

A fourth application is found after the reading of an article written by Mark Johnson [9] : the versioning of objects to maintain serialization compatibility with JavaBeans. The goal of this system is to maintain compatibility between objects and versions of a program : objects written for version x must be interpreted in a version y, older than x. The proposed solution is to modify the responsible class of the

serialization of objects to accept "compatible" changes (like adding fields, changing the privacy of a field, ...). With a versioning persistent language, another solution is have as many versions of a *reader document* class than versions of the application. When the objects must be read the first step is to determine the version used to write this. The second step is to use the compatible version the *reader document* class to correctly read the object.

Last but not least we plan to use versioning persistence in our research on logic meta programming. The logic meta programming language we use, Soul, is a logic programming language living in symbiosis with its host language, Smalltalk. More specifically it allows to execute object oriented code during logic inference [10]. What is currently not very well handled is the mismatch between side-effects in the object oriented code and backtracking. When using versioning persistence, we could map backtracking during logic execution to reverting to previous versions of objects in the object oriented part.

6. CONCLUSION AND FUTURE WORK

This paper introduces storage and versioning persistence, two kinds of persistence that are currently amalgamated in literature. It then looks in more detail at versioning persistence, differentiating between introspective usage (that allows queries on the past states of objects) and reversion usage (that allows to revert to a previous version of an object and create branches). Language support (or at least a proper framework) can be used in a number of different applications. We enumerate some very straightforward applications (undo functionality and CVS-like repositories), and some more novel ones (advanced debuggers and new logic meta programming languages).

The future work is obviously to validate the ideas put forward by this paper. Therefore we are working on an implementation of versioning persistence in an object oriented language. The goals of this implementation are multiple : experiment with the cost of a memory-intensive mechanism like persistence in an object oriented language, study the possibilities of queries, and implement several techniques to store versions of objects. Let's get to work!

7. REFERENCES

[1] David Jordan and Craig Russell. *Java Data Objects*. O'Reilly Media, Inc., 2003.

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

[3] Antony L. Hosking and J. Eliot B. Moss. Towards compile-time optimizations for persistence. In Alan Dearle, Gail M. Shaw, and Stanley B. Zdonik, editors, *Implementing Persistent Object Bases, Principles and Practice, Proceedings of the Fourth International Workshop on Persistent Objects, 23-27 September 1990, Martha's Vineyard, MA, USA*, pages 17–27. Morgan Kaufmann, 1990.

[4] Antony L. Hosking, Eric W. Brown, and J. Eliot B. Moss. Update logging for persistent programming languages: A comparative performance evaluation. In Rakesh Agrawal, Seán Baker, and David A. Bell, editors, *19th International Conference on Very Large Data Bases, August 24-27, 1993, Dublin, Ireland, Proceedings*, pages 429–440. Morgan Kaufmann, 1993.

[5] J. Eliot, B. Moss, and A. L. Hosking. Expressing object residency optimizations using pointer type annotations. In M. Atkinson, D. Maier, and V. Benzaken, editors, *Persistent Object Systems*, pages 3–15. Springer, Berlin, Heidelberg, 1994.

[6] Ali Ibrahim and William R. Cook. Automatic prefetching by traversal profiling in object persistence architectures. In *Proceedings 20th European Conference on Object-Oriented Programming (ECOOP 06)*, pages ??–??, 2006.

[7] Amos Fiat and Haim Kaplan. Making data structures confluently persistent. In *J. Algorithms*, pages 16–58, 2003.

[8] Stéphane Ducasse, Tudor Gîrba, and Roel Wuyts. Object-oriented legacy system trace-based logic testing. In *Proceedings 10th European Conference on Software Maintenance and Reengineering (CSMR 2006)*, pages ??–?? IEEE Computer Society Press, 2006.

[9] Mark Johnson. It's in the contract ! object versions for javabeans, March 1998. http://www.javaworld.com/javaworld/jw-03-1998/jw-03-beans.html.

[10] Kris Gybels, Roel Wuyts, Stéphane Ducasse, and Maja D'Hondt. Inter-language reflection - a conceptual model and its implementation. *Journal of Computer Languages, Systems and Structures*, 32(2-3):109–124, jul 2006. http://prog.vub.ac.be/Publications/2005/vub-prog-tr-05-13.pdf.