The Mjølner BETA System
Persistence in BETA
Reference Manual

Mjølner Informatics Report
MIA 91-20(1.3)
August 1996

Copyright © 1990-96 Mjølner Informatics ApS.
All rights reserved.
No part of this document may be copied or distributed without the prior written permission of Mjølner Informatics

Table of contents

1 Introduction 1
2 Basic Definitions 1
2.1 Objects and Patterns 2
2.2 Betaenv 2
2.3 Fragments and Compiled Code 3
3 The Persistentstore Pattern 4
3.1 Basic Operations 4
3.2 Restrictions 4
3.3 Example 5
4 Advanced Features 7
4.1 Lazy object fetch 7
4.2 References between different persistent stores 8
4.2.1 Where are objects saved? 8
4.2.2 Following references between persistent stores 9
4.3 Limiting reachability based persistence 9
4.3.1 Special objects 9
4.3.2 Runtime types 9
4.3.3 Combining runtime types and special objects 10
4.4 Files used for storing objects 10
5 Known Bugs and Inconveniences 10
5.1 Garbage collection 10
5.1.1 In-memory garbage collection 10
5.1.2 Secondary storage garbage collection 11
5.2 Persistent store identification and cross store references 11
6 Interface Description 12
Bibliography 21
Index 23

1 Introduction

Object persistence

In the Mjølner BETA System objects generated by a BETA program execution may be saved on secondary storage and restored in another BETA program execution. Usually this property of a programming language or system is referred to as *object persistence*. Persistence in the Mjølner BETA System is based on a reachability model, meaning that default behaviour when saving an object on secondary storage is to save everything reachable from the object in question. A discussion of different persistence issues and some of the ideas behind persistence in the Mjølner BETA System may be found in [Agesen 89].

Recognizing that saving the full transitive object closure may at times be too coarse grained, the Mjølner BETA System allows the programmer to control what part of the reachable object graph is actually saved along. This allows pointers from persistent objects to non-persistent objects despite the reachability based persistence model.

Lazy object fetch

Finally, in order to delay object fetch from secondary storage until an object is actually needed, *lazy object fetch* is supported. When using lazy fetch, only the persistence root and a few more objects are initially fetched from secondary storage. Fetch of other reachable objects is deferred until their state is needed. At that time, the object is transparently fetched from secondary storage.

2 Basic Definitions

Persistent object

A *persistent object* is an object that is saved on secondary storage during a program execution and thus survives the program execution in which it was created. A persistent object may be read by another program execution. Any BETA object can in principle be persistent. In the current implementation, the execution state (i.e. component stacks) is not saved. Furthermore, for certain types of objects it may not be meaningful to make them persistent. This is e.g. the case with user interface objects generated by libraries such as `xtenv`, `bifrost` and `macenv`. `Xtenv` objects may e.g. have partial state information about windows, widgets, etc., but this information will not be sufficient to restore the screen.

The transitive closure

By default, when an object is made persistent, all objects that can be reached through references are also made persistent. This includes statically enclosing objects[1]. The set of objects that can be reached from an object in this way is called *the transitive closure* of the object.

Persistent store

Persistent objects are saved in a *persistent store*, which is a collection of persistent objects. A persistent store has a name. In the current implementation, the name of a persistent store is the name of a file system directory containing the files making up the persistent store. Several persistent stores may exist and references between objects in different persistent stores are supported. A persistent store is itself a BETA object with a number of attributes.

Persistent root

An object may be pointed out to become a *persistent root* by means of the `put` operation on the persistent store. A persistent root must be given a logical name in the form of a text string. On checkpoint time, all objects reachable from persistent roots are saved in the persistent store.

---

[1] Objects that are instances of a nested pattern P.PP depends on, i.e. have a reference to, some instance of the enclosing pattern P.

2.1 Objects and Patterns

An object generated as an instance of a pattern is only meaningful as an instance of that pattern. Consider the following example:

```
P: (# ... #);
R: ^P;

&P[] -> R[]; .... (* Save R *)
```

The object referenced by `R` is an instance of `P`. When this object is later read by a program, it must be interpreted by that program execution as an instance of `P`.

It is not enough that the program reading the object has a declaration of a pattern `P` which has the same structure as the pattern `P` which was used to generate the object. It must be the very same pattern.

betaenv

In order for this to work, it is necessary to give a new interpretation to the notion of `betaenv`. This new interpretation is described in the next section. Before doing this we will shortly discuss what it means for objects to be instances of the same pattern.

Consider the following object descriptor:

```
(# T: (#  c: @ integer;
          A: (# b: @integer do c->b #);
          X: @ A;
          Y: @ A
       #);
   V: @ T;
   W: @ T
#)
```

Enclosing object: origin

The outermost object has two internal objects `V` and `W` which are instances of the pattern `T`. Each of `V` and `W` has internal attributes `c`, `A`, `X` and `Y`. The attributes of `V` are different from the attributes of `W`. This should be obvious for `c`, `X` and `Y`, since they occupy different storage locations in `V` and `W` respectively. The pattern attribute `A` of `V` is also different from the pattern attribute `A` of `W`. The reason is that an instance of `V.A` is enclosed by `V` and may therefore refer to attributes of `V` -- `V.A` is said to have *origin* in `V`. The pattern `W.A` is an attribute of `W` and may refer to attributes of `W` -- `W.A` has origin in `W`. An instance of `V.A` is therefore NOT an instance of `W.A`.

The objects `V.X` and `V.Y` are thus instances of the same pattern `V.A`. Similarly the objects `W.X` and `W.Y` are instances of the same pattern `W.A`.

2.2 Betaenv

`Betaenv` is a fragment defining a pattern that currently encloses all BETA code being compiled and executed by the Mjølner BETA System. This means that each program execution creates a new `betaenv` object. Patterns described in different programs will thus never be identical, since they will always directly or indirectly be attributes of the `betaenv` instance created by the program execution. Cf. the above discussion.

Only one betaenv object

To overcome this problem, the persistentstore treats all `betaenv` instances as if they were actually the same object, although in practice a new instance is created in each program execution. For example, consider the following library fragment:

```
mylib.bet

ORIGIN 'betaenv'
---LIB:attributes---
  A: (# ... #);
  B: (# ... #);
```

If `mylib.bet` is included by two or more different programs, then the pattern `A` will logically be the same pattern in both programs, since it is an attribute of the same `betaenv` object in all the corresponding program executions. The same is of course true for `B` and any other pattern declared in `mylib.bet`

Patterns used for generating persistent objects should normally be defined in the `lib:attributes` library slot as in `mylib` above.

However, by using the support for *special objects* as described in a later section, it is possible to obtain the same treatment of other patterns as just described for `betaenv`, i.e. treating instances of the pattern in different program executions as logically the same object. This also allows patterns used for generating persistent objects to be declared in attribute slots different from `lib:attributes`.

2.3 Fragments and Compiled Code

A BETA *pattern* is declared in some *fragment* of the BETA fragment system. The fragment in turn is part of a *fragment group*, corresponding to a BETA source file. For a description of the fragment system, see [MIA 90-2]. To identify the pattern from which a persistent object was created, the object has a reference to the fragment in which the pattern is declared. The fragment is unique in the sense that it is the version used for generating the code that instantiated the object. In order to load an object into the memory of some program execution, the program loading the object must be compiled from and linked with the same version of the fragment from which the object was originally created. Note that this does not prevent exchange of objects between different platforms, since it is the fragments that must correspond, not the compiled code[2].

The current implementation does NOT check if the fragments used for creating a persistent object has been changed since the object was created. It is currently the responsibility of the user to keep track of this.

It should be obvious that changes to a fragment may cause inconsistencies with previously generated persistent objects. Neither is it allowed to change other fragments within the same fragment group.

The fragment used for generating a persistent object is currently identified using the name *without path* of the fragment group in which the pattern is declared. This means that a BETA program using persistence cannot contain two source files with the same name. If this restriction is violated, the program will stop with an error message as soon as the first persistent store is opened.

Multiple equally named BETA source files are not allowed when using persistence and distribution libraries.

The above problems related to the unique identification of patterns will be avoided in future versions of the persistent store.

3 The Persistentstore Pattern

The Mjølner BETA System includes a library defining a persistent store, which keeps track of a directory of references to persistent objects. Some of these objects, the *persistent roots,* may have a logical name. A BETA program using persistent objects must include the persistent store library which is contained in the file

```
~beta/persistentstore/v1.5/persistentstore
```

This file is shown in section 7.

---

[2] The current version of the persistent store does **not** support object exchange between little and big endian machines.

3.1 Basic Operations

The basic operations of the persistent store are as follows:

`create`, `openRead`, `openWrite` are used for creating and opening a persistent store in order to access its contents. The name parameter is interpreted as a pathname relative to the current working directory of the process. When opening a persistent store for reading, it is not possible to update the contents of the persistent store, although the objects fetched may be changed in memory.

`put` points out an object to become a persistent root. This does not affect the contents of the persistent store, but registers the object is be saved in future `checkpoint` operations. At the same time the object is given a textual name to be used in `get` operations.

`get` retrieves an object identified by its textual name from secondary storage. If the object is already in memory, a pointer to the in-memory version is returned without changing the state of the object. Usually the full transitive closure of the object is read from secondary storage at once. This default, however, may be changed by using *lazy fetch.*

`checkpoint` saves the state of persistent objects on secondary storage. The transitive closures of all persistent roots in process memory are traversed, and all the objects saved. Checkpoint has no effect on stores opened by `openRead`.

`close` closes the persistent store. By default `close` performs a `checkpoint` operation before closing the files making up the persistent store. In most uses of the persistent store, it is therefore not necessary to call `checkpoint` explicitly.

Other operations on persistent stores are supported. These operations are described in section 4 on advanced features. In addition to the `persistentstore` operations, the `deletePersistentStore` pattern is available for deleting a persistent store.

A simple example using the `persistentstore` pattern is shown in section 3.3.

3.2 Restrictions

At most one program at a time should open a given persistent store in order to avoid problems with concurrent access. A future version will support a limited form of concurrency control. However, detailed concurrency control is out of the scope of this persistence library. Instead concurrency control is supported by the distributed, object oriented database being developed.

As already mentioned, application programs exploiting object persistence should not include multiple source files with the same name.

Component objects, i.e. objects with their own execution stack, are not allowed in the transitive closure of persistent roots. If components are met during a checkpoint operation, the program will terminate with an error message.

3.3 Example

The fragment `TextHashTable.bet` defines a pattern `TextHashTable` whose instances are to be made persistent.

```
TextHashTable.bet:

ORIGIN '~beta/basiclib/v1.5/betaenv'
---lib:attributes---
TextHashTable: hashTable
  (# element::< Text;
     hashfunction::<
       (# do e.scanAll (# do ch*26+value -> value #)#);
#);
```

The fragment `fooprod.bet` describes a program that creates a new persistent store and saves some persistent objects:

```
fooprod.bet:

ORIGIN '~beta/basiclib/v1.5/betaenv';
INCLUDE '~beta/persistentstore/v1.5/persistentstore';
INCLUDE 'TextHashTable';

---program:descriptor---
(# PS: @persistentstore;
   H: ^TextHashTable;
do (* Create the persistent store *)
   'myStore' -> PS.create;

   (* Create a table of objects. *)
   &TextHashTable[] -> H[];
   H.init;
   'first' -> H.insert;
   'second' -> H.insert;
   'third' -> H.insert;

   (* Make the table a persistent root. *)
   (H[],'TextTable') -> PS.put;

   (* Checkpoint and close the store. *)
   PS.close
#)
```

The fragment `foocons.bet` describes a program that makes use of some persistent objects:

```
foocons.bet:

ORIGIN '~beta/basiclib/v1.5/betaenv';
INCLUDE '~beta/persistentstore/v1.5/persistentstore';
INCLUDE 'TextHashTable';

---program:descriptor---
(# PS: @persistentstore;
   H: ^TextHashTable;
   T: ^Text;
   do 'myStore'->PS.openWrite;
      ('TextTable', TextHashTable##) -> PS.get -> H[];
      'fourth' -> H.insert;
      H.scan  (# do current[] -> putLine #);
      PS.close
#)
```

Other example usages of the persistent store may be found in the directory `~beta/demo/r4.0/persistentstore`. These demo programs, part of the Mjølner BETA System release 4.0, are listed below:

Demo programs

`showregister.bet` is an example of how to save a simple register in a persistent store. The register is built in a simple interaction with the user and finally saved. Later runs of `showregister` may read the saved register and perform

simple queries. Finally the persistent store containing the register may be deleted.

`hashdemo.bet` builds a simple hashtable of text strings. For each run of the program, an extra element is inserted into the table. If the persistent store does not already exist, it is created, and a new hashtable instance is made a persistent root. If the store already exists, the table is read, an extra element inserted, and the table scanned before the persistentstore is closed, implicitly implying a checkpoint operation.

`structdemo.bet` is similar to `hashdemo.bet`, but illustrates the possibility of saving pattern variables in a persistent store. Pattern variables cannot become persistent roots, but as demonstrated by `structdemo.bet`, they are allowed in the transitive closure of a persistent root.

`largeWrite.bet` and `largeRead.bet` illustrate the use of lazy fetch, i.e. the ability to delay object fetch from secondary storage until the objects are actually needed. `largeWrite` saves a large hashtable in a persistent store. `largeRead` in turn retrieves the hashtable from the store and then scans the table, enforcing all objects to be fetched lazily.

`special.bet` is an example of how to limit the part of the transitive closure of persistents roots saved along during checkpoint operations. By registering the `program` pattern as a *special object*, even objects with origin[3] in the `program` object can be made persistent roots. Furthermore, by registering the `IntegerObject` pattern as a *runtime type,* references to all instances of `IntegerObject` are saved as `NONE` references. Runtime types and special objects are described in detail in the next section.

`crossstore.bet` illustrates the handling of references between objects in different persistent stores. The same element is put into two different hashtables that in turn are saved in two different persistent stores. When one table is then fetched from its persistent store, it becomes necessary to open the persistentstore containing the shared element. The example shows how this must be taken care of by the programmer using the persistent store. The shared element is modified through the second table. On second run of the crossstore executable this modification is made visible through a scan of the first table. Details on references between different persistent stores are described in the following section.

4 Advanced Features

4.1 Lazy object fetch

When fetching an object from a persistent store using the `get` operation, the default is to eagerly fetch all objects in the transitive closure of the persistent root specified. However, since this may involve a huge number of objects not really needed by the current program execution, the persistent store offers the possibility to fetch the transitive closure lazily as the program goes along following references from the persistent root.

allowLazyFetch

By further binding the `persistentstore.allowLazyFetch` virtual to `trueObject`, the default fetch strategy is changed to lazy fetch. Alternatively the fetch strategy may be set on a per `get` basis by further binding the `get.allowLazyFetch` virtual to `trueObject`.

[3] Instances of patterns nested in the `program` pattern.

In short, lazy fetch works as follows. Using the persistent store `get` operation, the object graph reachable from the persistent root is always fetched in a breadth-first manner, whether or not lazy fetch is applied. In the case of lazy fetch, instead of fetching the full object graph, only a limited number of objects are fetched from secondary storage and instantiated in the current process. The objects fetched are the `persistentstore.maxFetchOnDanglerHit`first objects met during the breadth-first traversal. The default number of objects fetched may be changed by further binding the `maxFetchOnDanglerHit` virtual.

So, what about the objects not fetched? Since these objects are not instantiated, it is impossible to setup usual in-memory references. Instead socalled *dangling references* are used. Simply stated, a dangling reference is a negative number uniquely identifying a persistent object to the current process. If a dangling reference is ever followed[4], the same mechanism that checks for `NONE` references will trap to the persistent store kernel in order to transparently fetch the object needed from secondary storage. Also in this case the objects fetched are the `maxFetchOnDanglerHit` first objects met during a breadth-first traversal of the object graph rooted in the object needed. All dangling references in the process referencing newly fetched objects are replaced by genuine in-memory references. A more detailed description of the implementation of lazy object fetch may be found in [Brandt 94].

> **Note that the `--noCheckNone` (or `-s 14 0`)compiler switch suppressing the generation of runtime checks for `NONE` references cannot be used in programs using lazy object fetch!**

In addition to `maxFetchOnDanglerHit` and `allowLazyFetch`, the `persistentstore.OnDanglerHit` and `persistentstore.AfterDanglerHit` virtuals are used in conjunction with lazy object fetch. `OnDanglerHit` is called when a dangling reference has been hit, but before the object is actually fetched from secondary storage. `AfterDanglerHit` is called when the object has been fetched, giving the object as parameter. When `AfterDanglerHit` returns, the program continues whatever it was doing when the dangling reference was hit. The purpose of these virtuals is to offer informative callbacks that may be used for example in interactive programs where lazy object fetch may otherwise result in inexplicable delays.

It should be noted that there are no semantic differences whatsoever between lazy and eager object fetch. The practical difference lies in different efficiency/memory usage trade-offs.

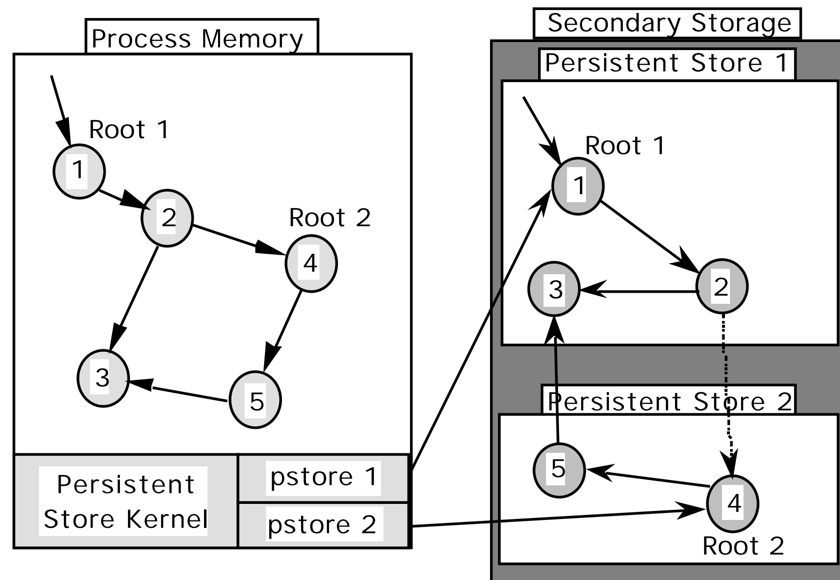
The demo programs `largeWrite.bet` and `largeRead.bet` together illustrates the use of lazy fetch.

4.2 References between different persistent stores

References between objects saved in different persistent stores are allowed, but beware that the support for such references is still somewhat experimental and requires special care in order to avoid problems. This section describes how to exploit cross store references. The `crossstore.bet` demo program is an example usage of cross store references. The drawing below illustrates references between objects saved in different persistent stores. In memory, there is no difference between cross store references and other object references. However, on secondary storage these references cross boundaries between persistent object stores.

---

[4] *Followed* here means "accessing the state of the object referred". Usual reference assignment on dangling references are not different from ordinary reference assignments.



4.2.1 Where are objects saved?

Checkpointing

When performing a checkpoint on a persistent store, either explicitly by calling the `checkpoint` operation or implicitly by closing the store, the object graphs rooted in the persistent roots of the store are traversed and the objects met saved to the persistent store. However, if an object is met that has already been saved in another persistent store, only the identification of that object is saved, and the graph traversal is not continued in that particular direction. Thus, a checkpoint operation on a persistent store only saves objects already belonging to that store, or objects that do not yet belong to any persistent store at all. The result is that a persistent object is saved in the store that first sees the object during a checkpoint operation.

The description above of course demands that we are able to recognize objects that “belong to” other persistent stores. However, when a persistent store is closed, objects that belonged to the store before the `close` operation will, after the `close` operation, semantically turn into copies of the objects belonging to the store. Thus, in order to maintain references between objects in different persistent stores, it is necessary to make explicit checkpoint operations and avoid the implicit checkpoint done by the `close`. The latter is done by further binding the `doCheckpoint` virtual:

```
PS.close (# doCheckpoint:: (# do false -> value #)#);
```

When exploiting cross store references, avoid the implicit checkpoint operation by furtherbinding the `doCheckpoint` virtual on `close` operations. Instead explicit checkpoint operations should be executed on all stores before closing any store.

4.2.2 Following references between persistent stores

If a persistent root is fetched whose transitive closure contains references to objects in other persistent stores, these persistent stores must be open in order to fetch the objects referenced. However, if the store referred is not already open, the persistent store containing the reference is not able to open the store automatically since it has no idea whether it should be opened for reading or writing. Instead it calls the `persistentstore.openpstore` virtual with the full pathname of the store needed, and expects the further binding to open and return the store.

4.3 Limiting reachability based persistence

4.3.1 Special objects

For the primary intended usage, a special object is an object that is thought of as a single logical object that is always present in program executions using some persistent store. Support for special objects may thus be thought of as generic support for pointing out patterns that to some extend have only a single instance shared between all program executions using the pattern.

The state of special objects is never saved persistently. However, references to these objects should be saved so that they may be setup correctly when saved objects referencing a special object are re-instantiated in another process. Typical examples or special objects are application framework objects that are known to be present in the program executions exchanging persistent objects, but should not be saved themselves. Examples of application framework objects are instances of `XtEnv`, `systemenv` and `shellenv`. These application framework objects, of which there should be at most one in each program execution, are not to be saved persistently, but instances of patterns nested inside the application frameworks should be allowed to persist. As already mentioned, `betaenv` is always treated as a special object.

Special objects are registered once in the lifetime of a persistent store by supplying name and type of the object to the `persistentstore.registerSpecialObject` method. The type is saved in the persistent store in order to be used for type checking when registering special object instances as described below.

In addition to the initial registration, an instance of the special object must be supplied by each process using the persistent store by calling the `persistentstore.registerSpecialInstance` method when the persistent store has been opened, but before any `get` operations are made. The instance given to the `registerSpecialInstance` method must be a subtype[5] of the type given to the `registerSpecialObject` operation.

The demo program `special.bet` contains an example usage of special objects.

4.3.2 Runtime types

Runtime types are patterns whose instances are used at runtime, but should not persist across program executions. As mentioned in section 2, an example of this is interface objects such as windows. Another example is objects used for caching purposes at runtime and referenced from persistent objects although the cache objects themselves should not be saved across program executions. By registering the pattern `p` as a runtime type, instances of `p` are not saved during checkpoint operations even though they are found in the transitive closure of a persistent root. Instead references to these objects are saved as `NONE` references.

Runtime types are registered by calling `persistentstore.registerRuntimeType`. As runtime types registered using `registerRuntimeType` are not saved persistently in the store, `registerRuntimeType` must be called for each runtime type in each session using the persistent store in question. If needed, it is of course possible to save a table of runtime types in a persistent store. The demo program `structdemo.bet` is an example of how a table of pattern variables may be saved in a persistent store.

The demo program `special.bet` contains an example usage of runtime types.

4.3.3 Combining runtime types and special objects

Since references to special object instances are treated differently than references to instances of runtime types, it is a contradiction to register the same pattern as a special object and as a runtime type in the same persistent store. Doing so will result in a runtime error.

---

[5] The subtype relation is reflexive, i.e. any pattern is a subtype of itself.

Furthermore, since it is not allowed to save an object without the knowledge that all its origins will be available when the object is to be reinstantiated, instances of runtime types should not be origins of objects saved. If an instance of a runtime type is needed as origin for some other object to be saved, the runtime type instance is saved anyway, disregarding the fact that it is an instance of a runtime type.

Different persistent stores used in the same program execution may have different sets of special objects and runtime types registered.

4.4 Files used for storing objects

The name parameter to the `create`, `openRead` and `openWrite` operations in the `persistentstore` pattern is interpreted as a directory name relative to the current directory of the process. When creating a new persistent store, this directory is created along with the files `locg`, `oinx` and `data`. Thus, for a persistent store created by:

```
'myStore' -> PS.open
```

the directory `myStore` and the files

```
...myStore/locg, myStore/oinx, myStore/data
```

are created. For deleting the files making up a persistent store, the `deletePersistentStore` pattern is available.

5 Known Bugs and Inconveniences

5.1 Garbage collection and persistence

With respect to garbage collection and persistence, there are two separate issues to consider, namely the usual in-memory garbage collection and garbage collection of the persistent store on secondary storage. These are considered in turn below.

5.1.1 In-memory garbage collection

The persistent store kernel keeps track of persistent objects loaded into the current process by maintaining a table of references to these objects. This table is shared by all persistent stores in a program execution. As long as a persistent store is open, no objects from that store can thus become garbage, since they are at least referenced from the internal object table. Currently the only way to delete objects from the internal table is to close the store. Thus, to allow in-memory garbage collection of persistent objects, the persistent store in which these objects are saved must be closed.

A side-effect of deleting objects from the object table is is course that the persistence kernel no longer knows that these objects are persistent, and thus semantically these objects turn into in-memory copies of the real persistent objects, now only available on secondary storage.

5.1.2 Secondary storage garbage collection

Currently there is no built-in support for garbage collection of persistent stores. Thus, once saved in a store, an object stays there until the store is deleted, even though the object may no longer be reachable from any persistent root.

However, for small persistent stores whose objects fit into virtual memory of the computer at once, and that are not referenced from other persistent stores, it is possible to perform a simple garbage collection using the basic operations of the `persistentstore` pattern. This is illustrated by the `PersistentGC.bet` demo fragment. `PersistentGC` simply reads the transitive closures of all persistence roots

into memory, deletes the store, and then saves the persistence roots in a new store with the same name as the old store.

Note, however, that the fragments used to generate the objects saved in the store must be linked with the executable performing the collection.

The demo program `gc.bet` illustrates how to first delete a number of elements from the persistent table generated by `largeWrite.bet`, and then perform a garbage collection on the store, using `PersistentGC`.

5.2 Persistent store identification and cross store references

Object ID

The persistent store identifies objects using a two-part object id, each part being a 32 bit integer. The first part identifies the persistent store in which the object is saved, and the second part is a unique identification of the object within that store. Currently the persistent store id is simply the system time (in seconds) when the store was created. A persistent store containing references to other persistent store thus maintains a mapping from these creation times to the full pathname of the persistent stores, in order to be able to call the `openpstore` virtual with the correct pathname.

Unfortunately this identification is not entirely unique. The persistent store kernel ensures that no two stores created by the same process gets the same creationtime, but there is currently no way to ensure that different processes do not create persistent stores with the same creation time. A process simultaneously opening two persistent stores with the same creation time will therefore in the best case receive wrong `alreadyOpen` exceptions, and in the worst case wrong in-memory object graphs may be created.

In future versions of the persistent store, this problem will be solved by using an alternative identification scheme.

6 Interface Description

```
ORIGIN '~beta/basiclib/v1.5/betaenv';
(*
 * $RCSfile: persistentstore.bet,v $ $Revision: 1.7 $ $Date:
1996/06/12 13:49:04 $
 *
 * COPYRIGHT
 *       Copyright Mjolner Informatics, 1992-94
 *       All rights reserved.
 *)
BODY 'private/persistentstoreBody';

---- lib: attributes ----

persistentstore:
  (#
     (* OPENREAD
      * ========
      *
      * Opens THIS(persistentstore) for reading, i.e. it
      * is not allowed to write objects to the store.
      * Checkpoint operations will be ignored.
      *
      * The name parameter is the name of the directory
      * containing the store and is interpreted as a path
      * relative to the current directory of the process.
      *)

     openRead:
       (# alreadyOpen:< PSexception
            (#
            do 'persistentstore.openRead: "'
                 -> msg.putText; fullName[] -> msg.putText;
               '" already open' -> msg.putText;
               INNER
            #);
          notFound:< PSexception
            (#
            do 'persistentstore.openRead: "'
                 -> msg.putText; fullName[] -> msg.putText;
               '" not found' -> msg.putText;
               INNER
            #);
          accessError:< PSexception
            (#
            do 'persistentstore.openRead: No access to "'
                 -> msg.putText; fullName[] -> msg.putText;
               '" ' -> msg.putText;
               INNER
            #);
          name: ^Text;
       enter name[]
       do ...;
       #);

     (* OPENWRITE
      * =========
      *
      * Opens THIS(persistentstore) for writing, i.e. it
      * is allowed to update the objects saved in the
```

```
 * store.
 *
 * The name parameter is the name of the directory
 * containing the store and is interpreted as a path
 * relative to the current directory of the process.
 *)

openWrite:
  (# alreadyOpen:< PSexception
       (#
       do 'persistentstore.openWrite: "'
            -> msg.putText; fullName[] -> msg.putText;
          '" already open' -> msg.putText;
          INNER
       #);
     notFound:< PSexception
       (#
       do 'persistentstore.openWrite: "'
            -> msg.putText; fullName[] -> msg.putText;
          '" not found' -> msg.putText;
          INNER
       #);
     accessError:< PSexception
       (#
       do 'persistentstore.openWrite: No access to"'
            -> msg.putText;  fullName[] -> msg.putText;
          '"' -> msg.putText;
          INNER
       #);
     name: ^Text;
  enter name[]
  do ...;
  #);

(* CREATE
 * ======
 *
 * Creates a new persistentstore. The name entered
 * is interpreted as a path relative to the current
 * directory of the process. The new store is opened
 * with write permission. *)

create:
  (# alreadyOpen:< PSexception
       (#
       do 'persistentstore.create: "'
            -> msg.putText; fullName[] -> msg.putText;
          '" already open' -> msg.putText;
          INNER
       #);
     exists:< PSexception
       (* The old store is deleted if exists returns.
        *)
       (#
       do 'persistentstore.create: "'
            -> msg.putText; fullName[] -> msg.putText;
          '" already exists' -> msg.putText;
          INNER
       #);
     creationError:< PSexception
       (#
       do 'persistentstore.create: Failed creating "'
            -> msg.putText; fullName[] -> msg.putText;
          '"' -> msg.putText;
          INNER
```

```
      #);
    name: ^Text;
  enter name[]
  do ...
  #);

(* CHECKPOINT
 * ==========
 *
 * Saves the state of all objects in the transitive
 * closure of objects made persistent roots or
 * fetched from this persistentStore since open.
 * Checkpoint has no effect if the store was opened
 * by openRead. *)

checkpoint: (#...#);

(* CLOSE
 * =====
 *
 * Performs a checkpoint, unless the doCheckpoint
 * virtual returns false, and then closes
 * THIS(persistentStore). Objects fetched from
 * THIS(persistentStore) will now turn into copies
 * of the objects saved in the store. This means that
 * the fact that the objects originally came from
 * this store, is forgotten.
 *
 * If allowLazyFetch is TRUE, dangling references
 * may exist to objects not yet fetched from this
 * persistentstore. If this is the case, the
 * danglersExists virtual is called. Default action
 * is to kill the process, but other possibilities
 * are to either fetch the missing objects before
 * closing, or simply ignore the warning and close
 * the store anyway. In the latter case, usage of
 * the objects mfetched could be fatal when trying
 * to access an object that was newer fetched from
 * the store. Ignoring should thus only be done if
 * you are not going to access the object (copies)
 * fetched during the "transaction" about to end. *)

close:
  (# danglersExists:<
      (# todo : @Integer;
         kill:   (# exit 0 #);
         (* Kill the process. Default action. *)
         fetch:  (# exit 1 #);
         (* Fetch the missing objects.        *)
         ignore: (# exit 2 #);
         (* Ignore the dangling references.   *)
      do kill -> todo ; INNER
      exit todo #);
    doCheckpoint:< BooleanValue
      (# do true -> value; INNER #);
  do ...
  #);

(* GET
 * ===
 *
 * Reads the persistent root named "name" into memory
 * and returns a reference. If the object is already
 * in memory, a reference to the in-memory object is
 * returned. In that case, the state of the object
```

```
 * is left untouched. *)

get:
  (# quaError:< Exception
       (#
       do 'persistentstore.get: Qua error getting "'
            -> msg.putText; name[] -> msg.putText;
          '"' -> msg.putText;
          INNER
       #);
     notFound:< Exception
       (#
       do 'persistentstore.get: "' -> msg.putText;
          name[] -> msg.putText;
          '" not found' -> msg.putText;
          INNER
       #);
     allowLazyFetch:< BooleanValue
       (#
       do THIS(persistentstore).allowLazyFetch
            -> value;
          INNER
       #);
     name: ^Text; type: ##Object;
     theObject: ^Object;
  enter (name[],type##)
  do ...;
  exit theObject[]
  #);

(* PUT
 * ===
 *
 * Turns obj into a persistent root with textual name
 * "name". The state of obj is not saved until a checkpoint
 * operation is performed. *)

put:
  (# obj: ^Object; name: ^text;
  enter (obj[],name[])
  do ...
  #);

(* SCANROOTNAMES
 * =============
 *
 * Iterates over the names of the persistent roots
 * in this persistent store. *)

scanRootNames:
  (# current: ^Text;
  do ...
  #);

(* CLOSURE CONTROL
 * ===============
 *
 * These attributes are used to limit the part of the
 * object graph saved during the checkpoint operation.
 * On the interface level of a persistentstore, there
 * is currently two ways of limiting the object graph:
 *
 * 1. Special objects.
 *
 *    Special objects are objects whose state is
```

```
 *    NEVER saved persistently. However, references
 *    to these objects should be saved so that they
 *    may be setup correctly when saved objects
 *    referencing special objects are reinstantiated
 *    in another process. Typical examples or special
 *    objects are application framework objects that
 *    are known to be present in the program
 *    executions exchanging persistent objects, but
 *    should not be saved themselves. Examples of
 *    application framework objects are instances of
 *    UIenv, systemEnv and shellEnv. (betaEnv is
 *    always treated as a special object.)
 *
 *    Special objects are registered once in the
 *    lifetime of a persistent store by supplying
 *    name and type of the object to the
 *    "registerSpecialObject" method. The type is
 *    saved in this persistentstore to be used for
 *    type checking when registering special object
 *    instances as described below.
 *
 *    In addition to the initial registration, an
 *    instance of the special object must be supplied
 *    by each process using the persistent store by
 *    calling the "registerSpecialInstance" method
 *    when the persistentstore has been opened, but
 *    before any get operations are made. The instance
 *    given to the registerSpecialInstance method must
 *    be a subtype of the type given to the
 *    registerSpecialObject operation.
 *
 * 2. Runtime types.
 *
 *    Runtime types types of objects that are used at
 *    runtime, but should not persist across program
 *    executions. An example of this is user interface
 *    objects such as windows. Registering a
 *    runtimeType means that instances of subtypes
 *    are not saved, and the corresponding references
 *    saved as NONE references. As runtime types
 *    registered using registerRuntimeType are not
 *    saved persistently in the store,
 *    'registerRuntimeType' must be called for each
 *    runtime type in each session. It is of course
 *    possible to save a table of runtime types in a
 *    persistent store if needed.
 *
 * It is a selfcontradiction to register the type of
 * a special object as a runtime type or vice versa.
 * Doing so results in a runtime error.
 *
 * Furthermore it is necessary to ensure that
 * instances of runtime types are not origins of
 * other objects saved. This is because it is not
 * possible to save an object without the knowledge
 * that all its origins will be available when the
 * object is to be reinstantiated.
 *
 * Although it probably makes no sense, different
 * instances of persistentstore may have different
 * sets of specialObjects and runtimeTypes registered.
 *)

(* REGISTERSPECIALOBJECT
 * ===================== *)
```

```
registerSpecialObject:
  (# alreadyThere:< Exception
       (#
       do 'registerSpecialObject: Special object "'
            -> msg.putText; name[] -> msg.putText;
          '" already exists: ' -> msg.putText;
          INNER;
          false -> continue;
       #);
     name: ^Text; type: ##Object;
  enter (name[],type##)
  do ...
  #);

(* REGISTERSPECIALINSTANCE
 * ====================== *)

registerSpecialInstance:
  (# quaError:< Exception
       (#
       do 'registerSpecialInstance: Qua error on "'
            -> msg.putText; name[] -> msg.putText;
          '" instance' -> msg.putText;
          INNER;
          false -> continue;
       #);
     notFound:< Exception
       (#
       do 'registerSpecialInstance: Special object "'
            -> msg.putText; name[] -> msg.putText;
          '" not registered.' -> msg.putText;
          INNER;
          false -> continue;
       #);
     o: ^Object; name: ^Text;
  enter (o[],name[])
  do ...
  #);

(* REGISTERRUNTIMETYPE
 * =================== *)

registerRuntimeType:
  (# type: ##Object
  enter type##
  do ...
  #);

(* LAZY OPTIONS
 * ============
 *
 * Attributes in this section are concerned with the
 * lazy fetch of persistent objects. If used, object
 * fetch from secondary storage may be delayed until
 * the objects are actually needed. By using a trap
 * mechanism, fetching takes place transparently
 * without applications being aware of it. *)

(* ALLOWLAZYFETCH
 * ==============
 *
 * If not further specified, all objects in the
 * transitive closure of an object requested in a
 * get operation are always fetched at once. This
```

```
 * default may be changed by furtherbinding
 * "allowLazyFetch" and setting "value" to "true".
 * Default may be overridden "per get" by using the
 * allowLazyFetch virtual of get. *)

allowLazyFetch:< BooleanValue;

(* MAXCOUNTONDANGLERHIT
 * ====================
 *
 * When a reference to an object not yet fetched
 * from secondary storage is encountered, some
 * number of objects reachable from the object
 * referred are fetched too. The objects fetched
 * are the 'value' first unfetched objects
 * encountered during a breadth-first traversal of
 * the object graph, using the object referred by
 * the original dangling reference as a root.
 *
 * Object fetch continues until:
 *
 *    1: No more unfetched objects are reachable from
 *       the root object.
 * or 2: maxCountOnDanglerHit objects have been
 *       fetched.
 *)

maxCountOnDanglerHit:< IntegerValue
  (# do 100 -> value; INNER #);

(* ONDANGLERHIT, AFTERDANGLERHIT
 * =============================
 *
 * OnDanglerHit is called when a dangling reference
 * is hit. When the object referred has been fetched,
 * AfterDanglerHit is called with the newly fetched
 * object as parameter. *)

OnDanglerHit:< Object;
AfterDanglerHit:<
  (# theObject: ^Object;
  enter theObject[]
  do INNER
  #);

(* OPENPSTORE
 * ==========
 *
 * If this persistentstore contains references to
 * objects in other persistent stores, it may be
 * necessary to open the stores to be able to follow
 * these references. When this happens, the
 * openpstore virtual is called. If the persistent
 * store named is not opened and returned in ps, the
 * program will terminate. *)

openpstore:<
  (# psname: ^Text;
     ps: ^persistentstore;
  enter psname[]
  do INNER
  exit ps[]
  #);

pspriv: @...;
```

```
  do INNER;
  #);

(* DELETEPERSISTENTSTORE
 * =====================
 *
 * Deletes an existing persistentstore. It is not possible to delete a
 * persistentstore that is open in this program execution. If tried
 * anyway,
 * the "alreadyOpen" exception is raised. In case the process does not
 * have
 * sufficient access priviliges to delete the store, the "accessError"
 * exception is raised. *)

deletePersistentStore:
  (# alreadyOpen:< PSexception
       (#
       do 'persistentstore.delete: "' -> msg.putText;
          fullName[] -> msg.putText;
          '" is currently open' -> msg.putText;
          INNER;
       #);
     accessError:< PSexception
       (#
       do 'persistetstore.delete: Unable to delete "' -> msg.putText;
          fullName[] -> msg.putText; '"' -> msg.putText;
          INNER;
       #);
     notFound:< PSexception
       (#
       do 'persistetstore.delete: "' -> msg.putText;
          fullName[] -> msg.putText; '" not found' -> msg.putText;
          INNER;
       #);
     name: ^Text;
  enter name[]
  do ...
  #);

(* PSEXCEPTION
 * ===========
 *
 * PSexception is used in several exceptional situations where the
 * files
 * making up an persistent store are not accessable. The fullName
 * parameter
 * is the full path of the directory expected to be a persistent
 * store. *)

PSexception: exception
  (# fullName: ^Text;
  enter fullName[]
  do INNER
  #)
```

Bibliography

[Agesen 89] Ole Agesen, Svend Frølund, Michael Hoffmann Olsen: *Persistent and Shared Objects in BETA*, Computer Science Department, Aarhus University, DAIMI IR-89, April 1989.

[Madsen 93] O. L. Madsen, B. Møller-Pedersen, K. Nygaard: *Object-Oriented Programming in the BETA Programming Language*, Addison-Wesley, 1993, ISBN 0-201-62430-3

[Brandt 94] Søren Brandt: *Implementing Persistent and Shared Object in BETA. Progress report.* Technical Report, Computer Science Department, Aarhus University, May 1994.

[MIA 90-2] Mjølner Informatics: *The Mjølner BETA System: BETA Compiler Reference Manual* Mjølner Informatics Report MIA 90-2.

Index

The entries in the index with *italic* pagenumbers are the identifiers defined in the public interface of the libraries:

The minor level entries refer to identifiers defined local to the identifier of the major level entry. For those index entries referring to patterns with super- or subpatterns within the library, these patterns are specified in special sections of the minor level index for that identifier.

Entries with plain pagenumbers refer to the text of this manual.

!!!Externally defined
. IntegerValue
. . !subpatterns
. . . maxCountOnDanglerHit *18*
. PSexception
. . !subpatterns
. . . accessError *19*
. . . alreadyOpen *19*
. . . notFound *19*
accessError *19*
AfterDanglerHit *18*
allowLazyFetch *18*
alreadyOpen *19*
application framework objects 9
betaenv 2
checkpoint 1, 4, 8, *14*
close 4, *14*
Component objects 4
component stacks 1
concurrency control 4
create 4, *13*
creationtime 11
cross store references 8
dangling references 7
deletePersistentStore 4, *19*
. accessError *19*
. . !!superpattern
. . . PSexception *19*
. alreadyOpen *19*
. . !!superpattern
. . . PSexception *19*
. name *19*
. notFound *19*
. . !!superpattern
. . . PSexception *19*
exception
. !subpatterns
. . PSexception *19*
fragment 3
fragment group 3
fullName *19*
Garbage collection 10
get 4, *15*
hashdemo.bet 6
largeRead.bet 6
largeWrite.bet 6
lazy object fetch 1, 7
maxCountOnDanglerHit *18*
name *19*
notFound *19*
object id 11
object persistence 1
OnDanglerHit *18*
openpstore *18*
openRead 4, *12*
openWrite 4, *13*
pattern 3
persistent object 1
persistent root 1
persistent store 1
PersistentGC.bet 11
persistentstore 9, *12*
. AfterDanglerHit 7, *18*
. allowLazyFetch 7, *18*
. checkpoint *14*
. close *14*
. create *13*
. get *15*
. maxCountOnDanglerHit *18*
. . !!superpattern
. . . IntegerValue *19*
. maxFetchOnDanglerHit 7
. OnDanglerHit 7, *18*
. openpstore *18*
. openRead *12*
. openWrite *13*
. pspriv *18*
. put *15*
. registerRuntimeType 10, *17*
. registerSpecialInstance 9, *17*
. registerSpecialObject 9, *17*
. scanRootNames *15*

PSexception *19*
. !!superpattern
. . exception *19*
. fullName *19*
pspriv *18*
put 4, *15*
reachability model 1
registerRuntimeType *17*
registerSpecialInstance *17*
registerSpecialObject *17*
runtime type, 6
Runtime types 9
scanRootNames *15*
showregister.bet 6
special objects 6, 9
special.bet 6
structdemo.bet 6
transitive closure 1