

Persistence Closure for the *Gobo* Eiffel compiler

Wolfgang Jansen
wjansen@soft.cs.uni-potsdam.de

January 29, 2015

1 Introduction

The distribution contains an addendum to the *Gobo* compiler for scanning the persistence closure of an object and to treat all touched data in some manner. The main application is storing the persistence closure to a file and retrieving it from a file in the sense of class **STORABLE** described in [2] and [1]. As it turned out, much more was necessary than just to implement one class: additional classes and an own run time environment. Below follows a short description of all the new things. (On first reading you may skip over to the usage part at page 11.)

The implementation essentially follows two ideas. First, storing the persistence closure of an object is just like a **deep_clone** where the copy is written to a file instead to main memory. Second, the internal object structure is accessed in the sense of class **INTERNAL** described in the books mentioned.

Unfortunately, class **INTERNAL** was (and still is) implemented incompletely, similarly for the newer class **TYPE**: creation of **SPECIAL** objects of a non-basic expanded item type and of **agent** types is not supported what may be needed when retrieving a stored object. As it turned out, it was advantageous to implement a couple of auxiliary classes and to implement the persistence closure on top of them instead relying on **INTERNAL**. These classes provide type information (e.g. how many attributes a class has, their names and types). That information is known during compile time and has to be written to the *C* code in order to make it available during run time. That means, the compiler had to be changed such that it writes the needed information to *C* code.

The *C* code added this way consists essentially of a collection of type, attribute, and routine descriptions. Sure, compared to *Gobo*'s general coding scheme for deep traversal the collections request one or two additional indirections. When traversing the persistence closure the indirections are not the time consuming process, but searching objects in the list of already traversed objects is; and not to forget the time for writing or reading. Thus, the chosen approach may be acceptable.

The change of the compiler is essentially an addition, too, such that

- the integrity of the compiler is not affected;

- the generated *C* code is (up to the addition) the same as before;
- the performance of the running program is not affected.

The disadvantage of the approach is that for each new compiler release the changes have to be adapted. The adaption followed the development of *Gobo* 3.9. In the following the main properties (seen from the user's point of view) of the implementation will be described.

The manual describes first (Secs. 2 through 4) the properties of the new classes, then how to use it (Sec. 6).

2 Basic properties

2.1 Basic and independent store

Both variants, basic and independent store, described in [2] have been implemented (there was also a third variant called general store, this is now obsolete).

Basic store mode is aimed at retrieving the persistence closure by the storing system itself (during the same session or a later one). Here, systems are considered identical if their root classes and their *Eiffel* compilation times are equal. The *C* compilation times may differ, the systems may even run on different platforms (so, the implemented basic store mode is more general than what is described in [1]). The file contents is more compact and writing/reading is faster than the independent store mode.

If stored in independent store mode by one *Eiffel* system the persistence closure can be retrieved by another one. Additionally to the data written in basic store mode, this mode writes type descriptions onto the file to ensure correct data association during retrieval. Retrieval works if the types of all stored objects are “alive” in the restoring system. This means, object creation for each type must somewhere occur in the class texts of the restoring system and must not have been eliminated as dead code. Similarly, agents must be “alive” too: for any agent object read from the store file there must be an agent clause in the restoring system referring to the same routine with the same set of open arguments. If types or agents are missing then the corresponding data will be skipped when read. This may often be the appropriate behaviour. Finally, an object of the expected retrieved object's type must somewhere assigned to the variable that will also be assigned by retrieved object. This is necessary to ensure that set of possible types of the variable includes the expected type. (This is not necessary if the system contains a store command for an object of this type, in particular, in case of basic store: the storing routine does the needed assignments.)

A final remark. The requirements on types to be “alive” and on the typeset of a specific variable are necessary for compilation only to ensure that the types etc. are not removed as dead code. It is not necessary that objects of those types have already been created or the assignment as been done when retrieval starts (this means that the object creation or assignment may be dead semantically).

2.2 Version independence

The independent store mode supports version independence (i.e. version differences between stored and retrieved types) as follows:

1. **no** change of class names;
2. **no** change of attribute names and of those routine names associated with an agent;
3. attributes present in a stored type but absent in the restoring version of the type are ignored (they are not longer needed);
4. attributes absent in a stored type but present in the restoring version of the type are set to their default values;
5. change of an attribute's typeset: the value in the store file is assigned to the attribute if an assignment attempt would do so;
6. change of `INTEGER_*` attributes to other `INTEGER_*` types, of `NATURAL_*` attributes to other `NATURAL_*` types (high order bits are cut off, i.e. a modulo operation according to the target type is done), as well as of `REAL_*` attributes to the other `REAL_*` type (the change from `REAL_64` to `REAL_32` may cause loss of accuracy);
7. change of a type from expanded to not expanded status or vice versa;
8. change of the inheritance structure has no effect directly but may induce cases 3 or 4.

If the restoring system is identical (in the sense of basic store, see Sec. 2.1) to the storing system then none of these changes can occur and any consistence checks or data adaption will be skipped, and the retrieval is nearly as fast as retrieving a persistence closure stored by basic store mode (this makes basic store less important). On the other hand, if both systems are different then the retrieval takes remarkable more time.

2.3 Hardware needs and independence

On lowest level the implementation is based on the following assumptions about the hardware:

1. the memory is byte organised, i.e. if `p` is a `POINTER` addressing a certain memory location then `p+1` addresses the next byte;
2. negative integers are represented in two's complement;
3. floating point data follow the *IEEE754* standard.

To my knowledge, almost all contemporary computers satisfy these requirements.

The persistence closure is written in binary format. To make the format portable a specific coding scheme is used for elementary data.

1. `BOOLEANS` as one byte;
2. `CHARACTER_8s` as one byte each (common *ASCII* format);
3. `CHARACTER_32s` as the corresponding `INTEGER_32` code;
4. `INTEGER_*s` and `NATURAL_*s` are written in 7-bit chunks (obtained by arithmetic shift, starting at low order bits) accompanied with a continuation bit until all bits have been written, negative `INTEGER_*s` get also a sign bit in the last byte;
5. `REAL_*s` are stored in two parts, first the exponent as an `INTEGER`, second the mantissa in 7-bit chunks (obtained by arithmetic shift, starting at high order bits);
6. `POINTERS` are not stored (c.f. Sec. 3.1).

2.4 Error handling

In case that an error arises the following happens.

1. Exceptions raised by the underlying file system are forwarded to the caller of `STORABLE`, `PC_SERIALIZER` `PC_DESERIALIZER`.
2. When during retrieval incorrect contents is read (e.g. a negative type number) then retrieving stops and a developer exception is raised. The same happens if a basically stored persistence closure is retrieved by a different system.
3. If during retrieval an object of a “non-alive” type is read then the type name is added to `PC_DESERIALIZER`’s list of `missing_types` (similarly for missing agents) and retrieving continues by skipping over the object data. Further, if an `INTEGER_*` value is to be converted to a smaller type and the value is too large then `has_large_integer` is set to true. Anyway, exceptions are not raised.

Reading the persistence closure to the end when situation 3 arises has the advantage that following (and hopefully correct) file contents can be read in.

3 Restrictions

3.1 What is not stored?

The following categories of data are not stored:

- pointer values (precisely, they are stored as `NULL` pointers): it is assumed that they denote entities of external code whose structure is unknown in Eiffel, moreover, their values will probably be invalid during retrieval time.

3.2 Other restrictions

- The binary format of the store file is not compatible to that of *ISE*, but work is in progress to provide a conversation tool.

4 Advanced features

4.1 Use of `default_create`

During retrieval from an independently stored file new objects of reference type are created by means of the type's version of `default_create` provided that this is a creation procedure of the type (otherwise, the standard method described in [2] is used). Any attributes occurring in the store file then overwrite the default setting, while the default setting is preserved for attributes not occurring in the store file and for `POINTER` values. This way, objects may be more correct when retrieved from independent store and if the class versions differ or if `default_create` was able to set appropriate `POINTER` values.

This method of object creation is more elaborated than the one described in [2]: `default_create` was not yet part of *Eiffel* when this book had been written and all attributes of new objects were preset to their type default value. Using the more elaborated method seems to be in the spirit of *Eiffel* since one source of errors is now excluded.

4.2 Identification of once values

The retrieved persistence closure of an object is self-contained in the sense that there are no references from within the closure to objects outside. A special question remained open so far: how are the objects in the closure related to global data such as constants and values of once functions? In short, those values are not stored. But storing can be controlled such a way that each stored object of reference type gets an indication whether it is identical to a once value. If so then the class and function names defining the once values are stored, too.

The effect during retrieval is as follows. When an object indicated this way has been retrieved and the associated once function of the restoring system has not yet been initialised then it gets initialised by the object. This means that retrieving the object is considered as the first call to the once function. On the other hand, if the once function has already been initialised then the association does not take place, i.e. once value and the retrieved object are really two objects. The identification helps to make the retrieved objects consistent with global settings of the restoring system.

The identification process does not work for expanded objects occurring in the persistence closure since they are copies of once values and cannot be identified with the once value itself (in contrast, once values of reference types can be identified by their references). Similarly, the values of basic expanded types cannot be identified with constant attribute definitions. In particular, the last category includes the values of `unique INTEGERS`. In contrast to explicitly

defined constants or explicitly assigned once values, their values may change between system versions in a manner not controllable by programmers (e.g. a 99 in one version may become a 123 in the next). Thus, a retrieved `INTEGER` value whose meaning is to denote one out of a range of unique integers may not belong to the restoring system's actual range of these unique integers. This is expressed by the following warning.

Warning

If the stored persistence closure is to be retrieved by a different system then it should be avoided to store objects whose attributes depend on `unique INTEGERs`.

4.3 Other classes for persistence closure

Class `STORABLE` as it is described in [2] has two disadvantages. First, retrieving an object does not follow the query/command separation principle:

```
my_obj ?= storable_obj.retrieved
```

has a side-effect (namely moving the file's read position). Second, there is an asymmetry between storing and retrieving. When storing an `STORABLE` object, say `storable_obj`, the object stores itself. When retrieving by `storable_obj` as target then this is a device to retrieve something else.

To overcome these drawbacks and also to add a bit more functionality, two more classes have been implemented: `PC.SERIALIZER` and `PC.DESERIALIZER`, and `STORABLE` is merely a wrapper class.

Class `PC.SERIALIZER` lets you write a comment onto the file. The comment may be used for data description if the file name is not sufficient. After retrieval the comment is available in feature `comment` of class `PC.DESERIALIZER`. Moreover, this class offers some features, e.g. `missing.types`, which describe possible failures of the retrieval.

Storing and retrieving is done as follows:

```
some_storing_procedure (some_object: detachable ANY)
  local
    f: RAW_FILE
    s: PC_SERIALIZER
    dd: STRING
  do
    dd := "data description"
    create f.make_open_write ("some_file")
    create s
    s.put (some_object, f, dd)          -- independent store
    -- or
    s.put Basically (some_object, f, dd) -- basic store
    f.close
  end
```

```

some_retrieving_function: detachable ANY
  local
    f: RAW_FILE
    d: PC_DESERIALIZER;
  do
    create f.make_open_read ("some_file")
    create d
    d.read
    Result := d.top_object
    f.close
  end

```

4.4 More data handling

Besides storing and retrieving in binary format, more treatments of the persistence closure are possible. To this end, an effective descendent class of `PC_ABSTRACT_TARGET` is to be developed whose procedures define the treatment of elementary data while the traversal through the persistence closure is organised by class `PC_SERIALIZER`. Several effecting classes come with the distribution, and the following are integrated into the `PC_SERIALIZER` class:

1. printing in a human readable format (i.e. “deep printing” of the object) by `put_text`, this may be of interest when testing a program;
2. storing in *XML* format by `put_xml` (experimental, retrieval not yet implemented);
3. writing to memory by `put_memory`, this way, the behaviour of `deep_twin` is mimicked.

4.5 Class `PC_ACTIONABLE`

Class `PC_ACTIONABLE` is the implementation of class `ACTIONABLE` described in [2]¹, it has been developed to control the storing and retrieving of objects of the descendant classes of `PC_ACTIONABLE`.

For example, an attribute of a `*FILE*` type may logically represent an open file after retrieval since it was open during store time, but it is not opened physically since connection needs an explicit call to the operating system. To make the object consistent, procedure `post_retrieve` should be rewritten to open the file physically or, the other way around, `pre_store` and `post_store` may be rewritten to close the file before storing and to re-open it afterwards.

The typical implementation of the procedures of `PC_ACTIONABLE` looks like the following.

¹It was planned to implement class `ACTIONABLE` directly and to put it into cluster `$GOBO/library/free_elks/kernel`. But then client classes could not be compiled by the *ISE* compiler that does not provide the class in its own `free_elks` cluster.

```

pre_store
do
    preserve          -- save contents of 'Current'
    attr1 := Void     -- prevent storing of 'attr1'
    attr2.modify      -- store a modification of 'attr2'
end

post_store
do
    attr2.revert      -- revert modification of 'attr2'
    restore          -- reset contents of 'Current'
end

post_retrieve
do
    create attr1.make  -- new value of 'attr1'
    attr2.revert      -- revert modification of 'attr2'
end

```

Procedure `preserve` is a feature of `PC_ACTIONABLE` and performs a `standard_twin` of `Current`, the resulting object is held outside the persistence closure. Conversely, procedure `restore` of `PC_ACTIONABLE` resets `Current` by a `standard_copy` from that object. Modifications of attributes of attributes, ... (such as `attr2.modify` in the example) have to be reverted explicitly (here, by `attr2.revert`); `modify` and `revert` are user supplied procedures.

Warning

Procedures `post_store` and `post_retrieve` may rely on the attributes of `Current` but they should not depend on attributes of attributes, ... in case of cyclic object dependencies since procedures `post_store` respectively `post_retrieve` may have not been called for subattributes, i.e. these may still be invalid. (There is no problem if `Current` and the attributes involved do not belong to a cycle of object dependencies.)

In case of the example this means that `attr2.revert` may be dangerous.

4.6 A tool to analyse store files

Class `PC_TOOL` provides an interactive tool to analyse an existing store file. It is meant to be used as root class and file `tool.xace` comes with it. The tool holds the file name and the object descriptions of a loaded store file which then can be analysed immediately or which are analysed by re-reading the file. The objects are represented by internally assigned consecutive numbers, types are represented by their compiler generated numbers.

The tool has a command line user interface which provides the commands given in table 1 on page 9. Parameters are to be written as follows:

Command	Params	Meaning
load	<i>g</i>	Load data from file <i>f</i> (ext: .gs)
info		Show header info of the loaded file
size		Print type and object counts, memory size
types		Print the list of types and type ids
fields	<i>tid</i>	Print field names and types of type <i>tid</i>
objects	<i>tid</i>	Print object ids having type <i>tid</i>
name	<i>id</i>	Print the qualified name of object <i>id</i>
verbose	<i>id</i>	Like name <i>id</i> including type names
rename	<i>f</i>	Rename class or attribute names according to dictionary in file <i>f</i>
extract	<i>id f</i>	Write persistence closure of object <i>id</i> to file <i>f</i> (ext: .gs)
data	[<i>f</i>]	Write the data in readable format to file <i>f</i> (ext: .txt , default: standard output) with a leading dollar sign for object ids
xml	<i>f</i>	Write data in <i>XML</i> format to file <i>f</i> (ext: .xml)
cc	<i>f</i>	Write data as <i>C</i> code to file <i>f</i> (ext: .c , .h)
help		Print this command table
quit		Quit the tool

Table 1: Interactive commands of class PC_TOOL

tid manifest integer denoting a type id (type ids are available via command **types**);

id manifest integer denoting an object id;

f filename given as valid *Eiffel* identifier or an *Eiffel* simple manifest **STRING** (i.e. enclosed in quotation marks), in case of an *Eiffel* identifier an appropriate file extension will be added, moreover, the files are located in the current working directory since other path components are not added.

Command **rename** might need more explanation. It reads from file *f* pairs of old and new names, one pair on each non-empty line (and ignoring *Eiffel* comments). Precisely, a pair may be constructed as follows:

```
pair ::= old_class_name new_class_name
        | old_class_name"."old_attribute_name new_attribute_name
        | "."old_feature_name new_attribute_name
```

where in the last case the *old_class_name* is taken from the most recent pair containing an *old_class_name*. The role of the command is to convert a store file written by an old system version to the corresponding class and feature names of a new system version. It is wise after renaming to save the new settings by command

```
extract 1 new_file_name
```

Commands `load`, `help`, and `quit` are always available, the others only if a store file is loaded that was written in independent store mode. The file to be loaded may also be given as command line parameter. After loading ident numbers and type information of all stored objects are known but not the object contents, so commands `extract`, `data`, `xml`, and `cc` need to re-read the file.

A final hint: the command names may be abbreviated to as many characters to make the command unique (in most cases, just one character). In case of a non-unique abbreviation, the commands take precedence in the order of the command table.

5 Benchmarks

To demonstrate the efficiency of the classes the persistence closure of an artificial system has been analysed consisting of many highly interconnected objects. The following tables show the size of different store modes as well as the time consumption on a specific platform. It is remarkable that the binary store files are more compact than the storage in memory (here: a 32-bit architecture, experience says that for a 64-bit architecture the figure is to be multiplied by 1.5), moreover, the rate of further compression is about 2.5 (the compression rate of other binary files such as executable binaries is often about 4, i.e. they are less compact).

	Indep.	Basic	XML	Text	Memory	<i>ISE</i>
plain	54 894	54 893	904 801	265 569	63 334	126 260
7z	17 256	17 400	48 309	21 460	<i>n.a.</i>	52 901

Table 2: Size in kBytes for different store and compression modes (“Text” means the human readable format, “plain” means no compression). For comparison, the occupied memory of the persistence closure and the same system basically stored by the *ISE* generated system are shown, too. The persistence closure consists of about 4 800 000 reference objects in 17 types.

	Indep.	Basic	<i>ISE</i>
store	3.19	3.19	1.58
retrieve (same system)	2.43	2.42	3.58
retrieve (other system)	2.81	<i>n.a.</i>	<i>n.a.</i>
to memory	<i>n.a.</i>	2.79	<i>n.a.</i>
deep_twin	<i>n.a.</i>	$> 12 \times 60 \times 60$	6.99

Table 3: Time consumption in seconds for different store modes, the top object is the same as in Tab. 2. For comparison, time consumption of `deep_twin` and the runtime of the *ISE* generated system is shown, too.

The larger retrieval time by another system (here, merely a recompiled system) than by the storing system is caused by the effort to determine whether

the retrieved object's dynamic type belongs to the typeset of the attribute to be set and, if so, to determine the right place of the retrieved object within its enclosing object (that place is a priori known when retrieving by the storing system). The runtime of a call to `deep_twin` is added for comparison to storing to memory (both routines have the same functionality). Function `deep_twin` takes so many time since references to already copied objects are stored in an auxiliary array to be searched for repeatedly occurring objects whereas the `PC_*` classes use hash tables for this purpose.

For comparison, the performance of persistence management of the same object by the *ISE* compiler has been tested. The tables show in columns *ISE* the results for the *ISE* system relying on class `STORABLE`. Only the basic store mode and the memory copy have been tested. The size of the *ISE* generated files is in much larger than that of the *Gobo* generated files (why is mysterious) while the time consumption of is less specific: storing/retrieving times of the *Gobo* implementation are between *ISE*'s pure *C* implementation, and the runtime when storing to memory is comparable ("to memory" of *Gobo* is to be compared to `deep_twin` of *ISE*).

6 Usage

Because the handling of the persistence closure is written in Eiffel, also its classes have to be known during compilation: clusters

```
{GOBO}/library/introspection
{GOBO}/library/introspection/gec
{GOBO}/library/persistence
{GOBO}/library/persistence/gec
{GOBO}/library/kernel/basic
{GOBO}/library/time
```

have to be added to the cluster part of the ACE file (or their equivalents to the XACE file). The clusters have to be added even in the case that the new classes are not used explicitly, e.g. if class `STORABLE` is used. If the persistence tool described in Sec. 4.6 is to be installed then also cluster

```
{GOBO}/library/persistence/tool
```

is to be added. This cluster contains also the appropriate XACE file.

If the system will be compiled by means of `geant` and if `system.xace` contains directly or indirectly (i.e. by recursively mounting other XACE files) the tags

```
<mount location="$GOBO/library/library.xace"/>
<mount location="$GOBO/library/kernel.xace"/>
```

then nothing is to be done: the latter XACE files provide all clusters needed.

References

- [1] MEYER, B.: *Reusable Software: The Base Object-Oriented Libraries*. Prentice Hall, 1994.
- [2] MEYER, B.: *Object-Oriented Software Construction*. Prentice Hall, 1997.

A File format

After a header part containing information about the creator, the store mode etc., the main part of the store file follows: an object's persistence closure. Its structure will be described in this appendix.

The general syntax of the persistence closure stored in binary files is given by the following Backus-Naur form.

Closure of an object:

- (1) $Closure ::= Object_ident \{ Type_or_data \ Object_ident \}^*$
- (2) $Type_or_data ::= Announcement \mid Data$
- (3) $Announcement ::= Type \ [Count] \ [Once]$
- (4) $Data ::= \{ Field \}^{Count}$
- (5) $Field ::= Reference \mid String \mid Basic \mid Data$
- (6) $Reference ::= Object_ident \ [Announcement \ [Data]]$
- (7) $Once ::= Class \ [Routine_name]$

Closure of a type:

- (8) $Type ::= Type_ident \ [Flags \ Type_def]$
- (9) $Type_def ::= Basic_type \mid Reference_type \mid$
- (10) $Special_type \mid Tuple_type \mid Agent_type$
- (11) $Basic_type ::= Class$
- (12) $Reference_type ::= Class \ Generics \ Field_defs$
- (13) $Special_type ::= Item_type$
- (14) $Tuple_type ::= Generics$
- (15) $Agent_type ::= Routine_name \ Open_closed \ Base_type$
- (16) $Field_defs$
- (17) $Class ::= Class_ident \ [Class_name]$
- (18) $Generics ::= Count \ \{ Type \}^{Count}$
- (19) $Field_defs ::= Count \ \{ Type \ [Field_name] \}^{Count}$

Elementary data:

- (18) $Class_name ::= String$
- (19) $Routine_name ::= String$
- (20) $Open_closed ::= String$
- (21) $Object_ident ::= Natural$
- (22) $Type_ident ::= Natural$
- (23) $Class_ident ::= Natural$
- (24) $Count ::= Natural$

Some remarks seem to be in order, in particular, to explain how alternatives are resolved, how repetition counts are obtained, and when optional parts are present.

1. *Object_idents* are **NATURALS** to identify the stored objects, e.g. consecutively increasing (implemented) or the file position (not implemented). The *Object_idents* are distributed into three exclusive categories: **unknown**, **announced**, and **ready**. The category may (and in most cases will) change during the *Object_ident*'s life time as follows:

unknown \rightarrow **announced** \rightarrow **ready**.

The **Void** reference is considered to be a **ready** “object” and gets the *Object_ident* 0.

A *Type_ident* denotes a type of the storing system and is related to the type's internal ident there. In case of basic store, all *Type_idents* are **ready** from the beginning, otherwise, they are **unknown**, then their category varies during as follows:

unknown \rightarrow **ready**.

2. The axiom (i.e. the starting element) of the syntax is rule (1): *Closure*. The sequence stops as soon as a **ready** *Object_ident* is encountered. This serves as some sort of end-of-file marker and denotes the object whose persistence closure the file contains. The other *Object_idents* are associated to the following *Type_or_data* entry. The ordering of objects within the file is, in general, not prescribed.
3. Not surprisingly, an *Announcement* is selected in rule (2) if the *Ident* is **unknown**, otherwise *Data*.
4. Reading an *Announcement* changes the associated *Object_ident*'s category: from **unknown** to **announced**. *Count* is present in case of a **SPECIAL** type meaning the associated object's field count. The *Announcement* contains enough information to allocate memory for the object represented by the *Object_ident*, and it is guaranteed that the *Announcement* of an object is written before the *Data*. The *Once* part is present if requested by the storing procedure (c.f. Sec. ??).

5. Reading *Data* (4) fills the fields of the object denoted by its *Object_ident* and changes the *Object_ident*'s category from **announced** to **ready**. *Count* denotes the number of fields of an object: the number of elements if it is a **SPECIAL** object otherwise, the number of attributes. In any case, *Count* is given by the *Anouncement* associated to the same *Object_ident*. *Fields* are arranged in the order prescribed the object's *Type*.
6. The alternatives in rule (5) are resolved by the *Field*'s type: *Reference* if it is a reference type, *Basic* if it is a basic expanded type, and *Data* (with the *Object_ident* of **Void**) if it is another expanded type. In the latter case, the *Field*'s type controls the *Data*.
7. A *Reference* is represented by its object's *Object_ident*. If that *Object_ident* is **unknown** then the *Type* follows embedded into an *Announcement* (that turns the *Object_ident* into **announced**). For the *Data* entry in rule (6) entry see the discussion on different variants below.
8. If the object announced is actually the value of a once function then *Once* (7) contains the description of its defining class and its *Routine_name*. Otherwise, *Class* reduces to *Class_ident* = 0 and *Routine_name* is absent.
9. The *Type* (8) includes *Flags* and *Type_def* iff its *Type_ident* is **unknown**, this is changed to **ready**. The *Flags* are a couple of bits containing information to distinguish in rule (9) between the different kinds of *Type_defs*. The *Type_def* contains information to identify the type uniquely:
 - (a) A *Basic_type* (10) is a basic expanded type; it is identified solely by its *Class*.
 - (b) A *Reference_type* (11) denotes a type different from any **SPECIAL** type, **TUPLE** type or agent description, respectively. It is identified by its *Class* and by its actual generic parameters.
 - (c) A *Special_type* (12) is identified solely by its item type.
 - (d) A *Tuple_type* (13) is identified by its actual generic parameters (tuple labels do not play any role).
 - (e) An *Agent_type* (14), i.e. the type of an object created by an **agent** expression, is identified by the target type and the name (within that type) of the routine to call, and how open and closed operands are mixed. To this end, *Open_closed* is a string composed of question marks for the open operands and underscores for the closed operands.
10. Like *Type*, the *Class* (15) includes the *Class_name* when the *Class* is encountered for the first time.
11. An *Field_defs* block (17) is present in rule (3) if attribute descriptions are needed. In particular, attribute descriptions of *Special_types* and *Tuple_types* are automatically generated according to the item and generic types, respectively, so they do not occur in the file. *Field_defs* (17) of *Reference_types* are identified by their name whereas those of

Agent_types (i.e. the descriptions of closed operands and the `last_result`, if any) are identified by position. In the latter case, the *Field_name* is not present.

12. *Basic* is one of `Boolean`, `Character_*`, `Integer_*`, `Natural_*`, `Real_*` (recall that `POINTERS` are not stored), and *Natural* is `Natural_32`. Data of these types are written onto the files as described in Sec. ??.

String is a `STRING_8`. `STRING_*` objects are considered atomic, they consist of the number of characters (formatted like any other *Natural*) followed by the sequence of characters. Thereby, `STRING_32` objects are stored as their corresponding UTF-8 form.

There are three special variants (requested by the storing system and coded in the file header), each making the syntax a bit more simple, especially rule (6).

1. The *Announcements* of fields (if of a reference type and the *Object_ident* of the object there is still `unknown`) are written before the *Data* of the enclosing object. This way, only `announced` and `ready Object_ids` occur in rule (6) and this rule becomes:

$$Reference ::= Object_ident$$

This variant ensures that there are as less as possible forward references.

2. In case of an `unknown Object_ident` only the *Announcement* is written in rule (6). This way, only the *Object_ident* of the root object occurs in rule (1) followed by the sequence of *Data* entries. Rule (6) becomes

$$Reference ::= Object_ident \ [Announcement]$$

3. In case of `unknown Object_ids` the object's *Data* follow immediately the *Announcement* in rule (6) dropping the leading *Object_ident* of *Data*. The *Object_ident* becomes `ready` since *Data* have been written. Only the *Data* of the root object occurs in rule (1). Rule (6) becomes

$$Reference ::= Object_ident \ [Announcement \ Data]$$

Variant 3 is more efficient than the others since it generates a slightly shorter file and needs much less auxiliary data to work (but a much larger call stack), this variant is used in `put_memory`. Variant 2 stores the objects in a flat manner with the root object first. All commands of the tool described in Sec. 4.6 can operate on files stored by this variant, so this variant is the default when storing to a file. The first variant is used to store *XML* files to reduce forward references.

Finally, the header is the sequence of the entities given in Tab.4: The manifest is the fixed string `EIFFEL_STORE`, the version has format *major.minor* and is currently 2.6. Both are *ASCII* strings without interior white space but closed by a space. All following entities are coded in the format described in this appendix and in Sec. 2.3 (as soon as the version number is known the file contents

Format	Meaning
String	Manifest
String	Version
STRING	Root class name
INTEGER_64	Compilation time of the storing system
INTEGER_64	Creation time of the running system description
INTEGER_64	Store time of the file (often the same as the creation time)
STRING	Comment (empty string if originally <code>Void</code>)
NATURAL_32	Flags

Table 4: Header entities

can be decoded correctly). The times are given in milliseconds since the epoch in the sense of class `DT_DATE_TIME`. The flags are the bitwise `or` of the store mode (two lowest bits) and some binary values. The store mode is:

- 0 Variant 2 in *FIFO manner*, this is the default;
- 1 Variant 2 in *LIFO manner*;
- 2 Variant 3;
- 3 Variant 1.

The codes and default setting of the other flags are shown in Tab. 5.

Code	Def.	Answers question
0x010	no	Are values of once functions indicated?
0x020	yes	Has <code>pre_store</code> been applied to <code>PC_ACTIONABLE</code> objects?
0x040	no	Are object indents not consecutive?
0x080	no	Are file positions used as object idents?
0x100	no	Are capacities (besides counts) of <code>SPECIAL</code> objects included?

Table 5: Meaning of header flags