Journal of Object Technology

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©JOT, 2007

Vol. 6, No. 4, May-June 2007

Persistent Objects and Capabilities in Timor

J. Leslie Keedy, Klaus Espenlaub, Christian Heinlein and Gisela Menger, University of Ulm, Germany

Abstract

The paper describes how the idea of persistent objects is integrated into the Timor programming language. The strategy adopted allows types to be instantiated at two levels: as "files", i.e. objects accessible at the operating system level, and as local objects within files, which resemble objects found in conventional object oriented programs. File objects (with associated methods) can be instantiated and manipulated via *capabilities*, which are accessible both internally and via the operating system. Local objects are accessible via references, which are not visible at the operating system level.

1 INTRODUCTION

Object oriented languages typically provide access to persistent data via standard library routines (e.g. as in C++), sometimes in conjunction with special features (such as `Serializable` in Java). Irrespective of the technique used, the basic assumption is that information generated and used in programs is by default temporary, and special action has to be taken to make it persist independently of program execution.

This approach reflects the assumption of conventional operating systems, which distinguish between a computational virtual memory (in which data are temporary) and a file system (which holds persistent data). This dichotomy in turn reflects a problem with addressing in computer systems. In the 1960s the designers of Multics recognized the disadvantages of this approach, emphasizing the overheads associated with transferring information between the two kinds of memory, and proposed the concept of *direct addressability* of data [4, 6]. Unfortunately their attempt to achieve this proved complicated and inefficient and was therefore not taken over into the Unix system, unlike many other Multics features.

A failure to support direct addressability has at least three kinds of negative consequences. First, programmers must expend considerable additional effort when they write programs which work with persistent data. Second, this additional code and the unnecessary transfer of information between the two kinds of memory both create

Cite this article as follows: J. Leslie Keedy, Klaus Espenlaub, Christian Heinlein and Gisela Menger: "Persistent Objects and Capabilities in Timor", in Journal of Object Technology, vol. 6, no. 4, May-June 2007, pp. 103.- 123 http://www.jot.fm/issues/issue_2007_05/article3

undesirable run-time overheads. Third, programming languages which provide separate facilities for handling temporary and persistent data are unnecessarily complicated by the duplication. In this paper we describe how the designers of the Timor programming language[1] have attempted to eliminate or seriously reduce these problems.

The basis of the Timor approach is to reverse the assumption of most other languages, i.e. it is assumed in Timor that information generated and used in programs is by default persistent. This corresponds in principle to the approach adopted by the persistent programming community, which has its origins in the key paper on *orthogonal persistence* published by Atkinson et al. [2].

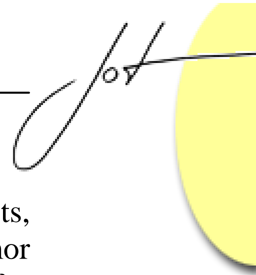
The orthogonal persistence approach eliminates the need for languages to provide separate sets of constructs for handling temporary and persistent data. It also eliminates additional programming effort at the application level for converting information between temporary and persistent formats. However, this does not necessarily imply that the work involved in the flattening of data is entirely eliminated, since in some approaches the latter simply becomes the responsibility of support software (e.g. a run-time library). As later sections of this paper will show, the Timor approach eliminates this also by using direct addressing techniques. And these techniques in turn virtually eliminate the run-time overheads incurred in conventional systems.

Many programmers who develop Timor software will scarcely need to be concerned about issues raised by persistence. As the paper will show, writing a program which does not produce or use persistent data will not require a knowledge of persistence concepts. Developing generally useful software components – the facilitation of which is one of the main aims of the language – will usually also not need any special knowledge of persistence. Creating a new persistent "file" will require only a bare minimum of additional knowledge, and using an existing file requires no knowledge of persistence.

Advocates of persistent programming might be surprised that we use the term *file*. This reflects our view that persistent objects need some organization, and in this sense the traditional concept of a file has been successfully used over many years. Users of conventional file and database systems should have no substantial difficulties in adapting to the Timor concept, provided that they combine the concept with the object oriented way of thinking. A Timor file can be used in similar situations to files in conventional systems. The fundamental difference is that it is defined in OO terms, i.e. it is an object which has its own methods, and these can be invoked just like the methods of any conventional OO object.

File objects are important for naming, for interfacing with operating systems, etc. But they are also very relevant for the issue of garbage collection, especially in the context of distributed objects. Timor allows files to be distributed freely (e.g. across the Internet) and so it is important that garbage collection can take place on individual entities: files are ideal for this purpose, as each file serves as a separate root of persistence.

[1] www.timor-programming.org



In section 2 we briefly describe relevant features of Timor (types, values, objects, references, etc.) as if the language were not persistent. Then in section 3 the Timor concept of persistent objects, and the related concept of capabilities, as "references" for persistent objects, is introduced. Section 4 discusses the relationship between capabilities (for file objects) and references (for local objects). Section 5 provides an overview of the planned implementation. The paper concludes with a brief description of related work (section 6) and a general conclusion (section 7).

Space restrictions prevent a full discussion of the related themes of process organisation (including the issue of persistent processes) and the distribution of files over remote computers. These issues are addressed in a companion paper, scheduled to appear in the next issue of the Journal of Object Technology. In the present paper we can merely hint that Timor supports persistent processes according to the in-process (procedure oriented) model [20, 22] and a distribution concept that encompasses the idea that persistent objects can be directly accessed over wide area networks.

2 THE BASIC TIMOR OBJECT MODEL

In contrast with the conventional OO class construct, Timor distinguishes between type definitions and their implementations [11-13]. A Timor type definition (introduced by the keyword `type`) can have several implementations (introduced by the keyword `impl`) and it is possible that different instances of the same type can have different implementations, even within a single program.

Types and Implementations

Timor types are defined in terms of makers (constructors) and instance methods. Each instance method must be characterised either by the keyword `op` (designating an *operation*, i.e. an instance method which may change the state of its instance) or by the keyword `enq` (designating an *enquiry*, i.e. an instance method which cannot change the state of its instance).

Types are defined strictly in accordance with the information hiding principle [21]. For programming convenience method pairs which set and get a value or reference can be defined in type definitions as *abstract variables*. However (unless an optimising compiler determines otherwise) an abstract variable is implemented as a pair of methods [13]. Here is a type definition containing abstract variables:

```
type Person {
instance:
 String name, address;  // abstract variables
 Date dateOfBirth;
 Person* spouse;        // an abstract reference
}
```

The abstract variable `name` is a shorthand definition for the instance methods:

```
final op String name(String name);
final enq String name();
```

and the abstract reference spouse corresponds to the methods:

```
final op Person* spouse(Person* spouse);
final enq Person* spouse();
```

The other abstract variables have analogous definitions. Programmers can provide explicit implementations of the methods representing abstract values; otherwise the compiler provides an automatic implementation [13].

Timor does not directly support static methods or binary instance methods [5]. The latter cannot be defined because instance methods cannot have parameters or return values of their own type. The only exception – which does not create the problems associated with binary methods - is the implicitly defined set method of an abstract reference, as illustrated above. This mechanism provides the ability, for example, to define linked lists.

An *implementation* normally contains code which implements a specific type. Different implementations of a type can be used for different instances in the same program. In addition to the sections contained in a type definition an implementation usually includes a state section, in which the state variables of instances are defined.

Objects and Values

Timor distinguishes between *objects*, which can be regarded as independent, dynamically created entities in a system, and *values*, which "belong to" or characterise individual objects. For example in the above definition a Person instance is characterised by values such as a name, a date of birth, an address, etc.

A type can be instantiated either as a value or as an object[2]. In both cases a maker can be used and this returns a *value*. Hence the value returned by a maker can be directly assigned to a value variable, e.g.

```
Person aPersonValue = Person.init();
```

But if a new dynamic object is needed, the value returned by the maker can be converted by the new operator into an object, e.g.

```
Person* aPersonReference = new Person.init();
```

In this example the new operator creates a new object which is initialised as a copy of the value[3] produced by the maker, and returns a reference to it. The assignment operator copies this to the reference variable aPersonReference.

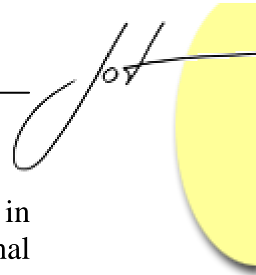
The new operator need not be associated with the invocation of a maker. Any value can in principle serve as an operand, e.g.

```
Person* aPersonReference = new aPersonValue;
```

In this case a new object is created whose value is a copy of aPersonValue.

[2] Basic types (e.g. int, boolean) are first class types, in contrast with Java and many other programming languages. Hence it is possible to instantiate these, like other types, as objects or values.

[3] new in fact makes a copy of the value, but the compiler optimises away redundant copying.



In accordance with the information hiding principle there is no operator (such as `&` in C) which allows a reference to be generated for a value. Hence the methods of internal values of an object cannot be called by clients of the object, but only by the object's own instance methods. There is however a dereferencing operator which returns the value of an object. For example, given an `int` object created as follows:

```
int* intRef = new int.init(3);
```

its value can be obtained for example as follows:

```
int i = *intRef;
```

The value of a user defined type can also be obtained using this operator:

```
Person p = *aPersonRef;
```

References and Access Restriction

References are logical entities which are intended to express relationships between objects, e.g. `Person* spouse`. The syntax chosen for references (the type name followed by an asterisk symbol `*`) resembles pointer syntax in C, but semantically there are significant differences between a C pointer and a Timor reference. While a Timor reference can be regarded as a pointer from one object to another, this expresses a logical concept, not a physical memory relationship. Hence there are no operators which allow references to be modified in the C style (e.g. by adding integer values). Furthermore references are themselves not regarded as objects, so that references to references are not possible. Many references can refer to the same object. Since Timor supports subtype polymorphism, a reference for an object of a subtype can be assigned to a supertype reference variable (and similarly a subtype value can be assigned to a supertype value variable).

A reference variable can be restricted such that it can only be used to invoke a subset of an object's methods. The restriction defines the subset of the methods which are permitted. These can be defined as *views* [11]. Some standard views are predefined. For example `all` defines all the instance methods of the type of a specific reference. This may not appear to be a restriction, but in fact it is useful. For example the reference variable

```
Person[:all:]* p;
```

is not quite equivalent to

```
Person* p;
```

because downcasts are not permitted on restricted references. Other standard views include `enq` and `op`, which respectively define all the enquiries (reader methods) and all the operations (writer methods) of a type. Individual views can be defined (retrospectively where appropriate) which list particular methods of specific types.

The effect of restricting references is similar to that of reducing the access rights in capabilities at the operating system level [19]. In this case the access rights are in effect the rights to invoke particular methods. Access rights can never be increased, since a more restricted reference cannot be assigned to a reference variable which is less

restrictive. This is also why downcasts are not permitted on restricted references. For a more detailed description of restricted references see [17].

Deleting Objects

Like C++ Timor allows objects to be explicitly deleted, using an operator `delete`, which takes a reference as its parameter. If an attempt is made to access a deleted object, an exception is raised. In contrast with C++, the condition that an object has been deleted is reliably detected by the run-time system (see section 5).

Because values associated with an object are private to that object, these can be deleted with the object to which they belong. Similarly when a new value is assigned to a value variable, the old value (however complex), can also immediately be deleted. These possibilities reduce the need for garbage collection, but do not eliminate it entirely.

Parameter Passing

The Timor distinction between references and values has determined the nature of the parameter passing mechanism. Each parameter is passed either *as a value* (i.e. a copy of a value is passed/returned) or *as a reference to an object*, in which case a copy of a reference (which can be restricted) is passed/returned. The latter is *not* equivalent to a parameter passed by reference in the conventional sense, since references cannot be generated for the internal values declared within an object.

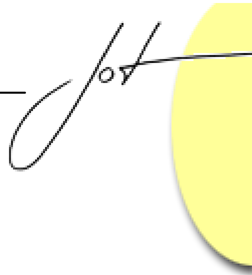
Programs

The conventional notion of a program, i.e. an algorithm which is executed as a result of a command to the operating system, and which on completion leaves behind no state (except in so far as the programmer explicitly organises this by storing information in files), has never fitted well with the OO paradigm. Languages such as Java and C++ rely on special conventions such as "main" to allow them to be interfaced to the operating system. Timor, in contrast, has no special program concept.

A Timor program is defined simply as a normal instance method of some type. As for any Timor type, its state can be instantiated as a value or as an object, and the corresponding instance method is invoked on this in the usual way. This also implies that programs can be invoked not only at the OS level but also from within Timor programs.

A conventional program normally leaves no state behind. At the programming language level his can easily be simulated in that the state is not assigned to a variable. In the sequel the term "program" should be interpreted in this sense. Later in the paper it will be explained how programs operate at the OS level. We now explain how a "program" can be executed at the programming language level.

Because a type definition can include multiple instance methods, types can be defined which in effect are collections of programs, e.g.



```
type GamesCompendium {
instance:
 op void chess();
 op void draughts();
}
```

The chess program can be executed within some other program or object with an expression such as:

```
GamesCompendium.init().chess();
```

In this case a new "value" of the type `GamesCompendium` is created for the duration of the program execution. It is also possible to execute a program as an "object", e.g.

```
(new GamesCompendium.init()).chess();
```

Here a new state is instantiated as an object and then the instance method corresponding to a program is invoked on this. Because the object is not assigned to a variable, the state of the program no longer exists after the instance method execution ends. The persistence concept which is developed in the following sections will be used to explain how an extension of this idea allows an instance method of any type to be invoked at the OS level as a program.

Other Timor Types

Timor's rich type system allows types to be defined in terms of the inheritance paradigm [11, 12, 15] and/or as "adjectival" types [14, 16]. But since the methods of such types (even the bracket methods of qualifying types) can be regarded as instance methods and makers, they need not be discussed in detail here.

3 PERSISTENT OBJECTS

The basic Timor concepts described in the previous section correspond loosely to concepts found in other object oriented languages, even if the details are often significantly different. In contrast the Timor view of persistence differs radically from that found in conventional object oriented languages. It has more in common with the programming language style known as "orthogonal persistence" [2] which was first presented in terms of the language PS-algol.

The starting point for understanding persistence in Timor is that *everything is persistent* unless and until it is explicitly or implicitly deleted, or it becomes unreachable. Hence there are no mechanisms for writing data to and/or reading data from conventional files (although special types can be developed to achieve this, for example if an application needs to interface with an existing file system).

Objects and Files

If data items can persist independently of the programs which use them, and if the same data entity can be used by different programs, it is clear that particular items of persistent data must be separately identifiable, in order that programs can indicate which data items they need. There must also be a protection mechanism to ensure that only authorised users and programs can gain access to persistent data items. These are features conventionally provided in the file systems of operating systems. A conventional file (hereafter called an *OS file*) is a named data container which can be protected. OO languages generally use OS files as containers into which objects can be stored after they have been flattened, i.e. converted into an "appropriate" format.

In contrast Timor implements the notion that a *file is a persistent object* which might be, but is not necessarily implemented using an OS file. Like any other Timor object, it has a state which can be accessed only via the methods defined for its type. Because any kind of Timor type can be instantiated as a file object, all kinds of structural units (e.g. programs, databases, subroutine libraries, etc.) can all be instantiated as files.

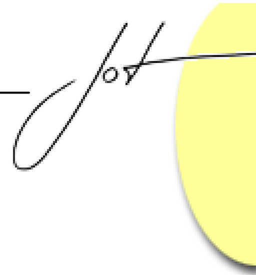
Capabilities

Like OS files, Timor file objects (hereafter referred to as "Timor files", or simply as "files") need to be separately identifiable and need to be protected. For this purpose Timor supports a special kind of reference, known as a *capability*. A Timor capability has similar properties to those described in section 2 for references, except that it is visible not only at the programming language level, but also at the OS level.

Capabilities in capability based operating systems consist primarily of a unique identifier of the referenced object, together with a set of access rights. As we shall see in the later discussion of implementations of Timor persistence (section 5), this structure also forms the basis for Timor capabilities. The unique identifier within a capability must provide a sufficient basis for locating a referenced file object, and in the context of remote method invocations in the Internet this implies that such unique identifiers must be very large.

The set of access rights within a capability defines the subset of the file's methods which can be invoked by the holder of that capability. Different capabilities for the same file can have different sets of access rights. Restrictions on access rights can be defined at the Timor programming level using the same restriction mechanism as exists for normal references (see section 2).

Syntactically capabilities resemble references. They are distinguished from references by the use of a double asterisk. The relationship between capabilities and references, and the reasons for distinguishing between them, are discussed in section 4.



Instantiating Files and Local Objects

A file can be created from within a Timor program by invoking the `create` operator. This functions analogously to the `new` operator, returning a capability for a new file object. To create a file of type `Person` (see section 2) the programmer might write:

```
Person** p = create Person.init();
```

However, this would result in an overhead equivalent to the creation of an OS file. Consequently it makes more sense, as in conventional systems, to bundle together a number of person objects into a single file. In Timor terms this could be achieved simply by using a collection type, e.g.

```
List<:Person:>** personList = create List<:Person:>.init();
```

but it would be equally possible, and perhaps preferable, to define a new type which provides suitable semantic functions, e.g.

```
type PersonDatabase {
instance:
 op PersonId newPerson(String name; Date dob)
                               throws PersonAlreadyExists;
 // the type PersonId is discussed in section 4.
 enq String personName(PersonId p)
                                throws PersonDoesNotExist;
 op void changeAddress(PersonId p, String address)
                                throws PersonDoesNotExist;
 op void marry(PersonId p1, p2)  throws PersonDoesNotExist;
 ...
 enq int personCount();
 enq PersonId oldestPerson();
}
```

The final methods in this example illustrate that it is often useful not only to collect similar objects together for efficiency reasons but also to provide further methods which treat the collection as a useful single entity for summarising or comparison purposes. Similarly the `marry` method illustrates how a method which might in other OO languages have been implemented as a binary method (for the type `Person`) can naturally be implemented as a normal instance method of a client type of `Person`.

Given such a type this could be instantiated as a file, as follows:

```
PersonDatabase** pdb = create PersonDatabase.init();
```

But it could equally well be instantiated as a normal local object, e.g.

```
PersonDatabase* pdb = new PersonDatabase.init();
```

Programs and Files

As was described earlier, a "program" is simply a normal instance method of some type. Executing a program therefore requires an instance of the type in which the instance method is defined, and the activation of the corresponding method on this instance. To activate a program in the conventional sense, i.e. at the OS level, the user instantiates the

type in which his program is defined *as a file object* and uses the OS to invoke the appropriate method. On completion of the program's execution its file can (but need not) be deleted[4].

Persistence of Local Objects

Local objects are always created within some file object, which may be either an object intended to persist independently of individual program activations (e.g. a `Person-Database` file object) or an object created especially for executing a program. Hence the persistence of a local object (in so far as it is not explicitly deleted) coincides with the persistence of the file object in which is created. In this sense local objects are by default persistent and no special steps need be taken by a programmer to make them persistent.

Deleting Files

Temporary files created within a Timor program can be automatically deleted by the compiler (e.g. when the capability is not assigned to a capability variable), or they can be deleted explicitly. To delete a data file explicitly from within the code of a Timor program, the `delete` operator is used in association with a file capability (which must have a delete access right set). When a file is deleted all its local objects are implicitly deleted.

Qualifiers for Files

Just as dynamic qualifiers can be instantiated as and applied to local objects to provide features such as monitoring, and especially protection [14], so also qualifying types can be instantiated as file objects and can be similarly used to monitor and protect Timor files.

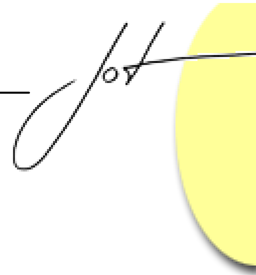
As for local objects, the effect of qualifying a file object is that the method invocations to and from it can be intercepted and examined in bracket methods, which may, but need not, permit the call to proceed. Dynamic qualifiers applied at the file level treat a file object as a single entity. Thus call-in bracket methods defined at the file level are always applied directly to the instance methods of the file object itself as they are invoked, while call-out bracket methods are applied whenever either the file object itself or any of its local objects invokes methods of other file objects [18].

4 CAPABILITIES AND REFERENCES

A detailed discussion of how capabilities and references are related at the language level has been postponed until this point in order that the implications of a decision can be clearly evaluated. At least the following possibilities arise.

---

[4] The individual steps of file instantiation, method invocation and file deletion can of course be hidden behind a macro or graphic interface of the command interpreter.



- Capabilities are the same as references.
- Capabilities are subtypes (or supertypes) of references.
- Capabilities and references are indirectly related, via a common supertype.
- There is no direct relationship between capabilities and references.

Linguistic arguments undoubtedly favour the first solution. The language would (apparently) be simpler and more regular, programmers would not have to distinguish between two separate but related concepts, and local objects could be used interchangeably with files.

Why Capabilities are not simply References

Despite the apparent attractiveness of treating capabilities and references completely uniformly, there are practical arguments which suggest that this is not feasible. Here are two.

First, it must be possible to locate a file simply by providing a capability for it. How this can be implemented will be discussed later; it is sufficient to note here that this aim is only achievable if capabilities contain fully unique identifiers for files, which must (e.g. in the context of the Internet) be very large numbers. It would be enormously inefficient to expect every reference for every local object to hold such numbers.

Second, garbage collection of the local objects within a file object must take place on the basis of locating references for these objects (as in conventional OO programs). If no distinction were made between capabilities and references, garbage collection would become a world-wide activity (and would unrealistically assume that all references are available on line when garbage collection occurs).

These two points are sufficiently convincing, in our view, to make the distinction between capabilities and local references important, leading to a two level object system as we have outlined in earlier sections of this paper.

Why Methods of Local Objects cannot be Invoked from Other Files

Given a distinction between references and capabilities, the question arises whether references to local objects can be allowed to escape from a file, i.e. can they be passed as parameters to methods of other files and stored in these files. This would not necessarily produce a garbage collection problem, since the garbage collecting of a file could be defined as the removal of local objects which are not reachable via local references within the file. (Ignoring external references to local objects would be equivalent to the practice in the Internet that the validity of bookmarks for local objects at remote sites is not guaranteed.)

However, if local references were allowed to escape, then they would have to be modified (assuming that they were normally represented as local addresses) to indicate that they should not be interpreted as local to their new environment.

The question would then arise whether such references to local objects of foreign files could be used directly to invoke the methods of these remote local objects. The risk

that the objects might have been garbage collected is not a convincing counterargument, because in a system which allows explicit deletion of objects, a programmer must always expect the possibility that an object which he invokes does not exist. However, there are two other arguments which have convinced us that this possibility is not viable.

The first of these is the information hiding principle. The concept that a file *is* an object (rather than that it merely *contains* objects) implies that it should be possible to change the implementation of this object without clients being affected. This is a notion with far-reaching consequences. It not only implies that the implementations of internal objects might also change, but also that in a more drastic internal reorganisation some objects might not exist at all in their earlier form, being replaced for example by objects of other *types*, with different method definitions. It is for precisely this reason (at a lower level) that the decision was made that Timor should not support an address generating operator (in the style of `&` in C++), which would have made it impossible to guarantee the proper working of programs following the change in an implementation for a local object.

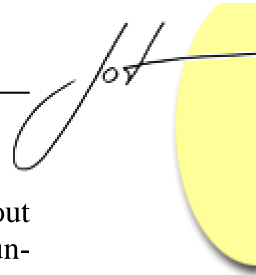
The second argument against allowing the methods of local objects of one file to be invoked from within the context of another file involves the use of Timor's qualifying types. If a file were protected or synchronised by a qualifier (e.g. an ACL qualifier or a reader-writer qualifier [14]), it would be impossible to ensure that the effects of these qualifiers could not be bypassed by going directly to a local object of the supposedly protected or synchronised file.

Reconciling Information Hiding with References and Capabilities

For the reasons discussed in the last section Timor does not allow the local objects of a file to be invoked directly from outside the file to which they belong. The consequence of this rule is that references may not be used as parameters or return values for the methods of inter-file calls. How can this be avoided?

The simplest way would be to distinguish between files and local objects *in type definitions*. With such a distinction the methods of some types ("file types") would be permitted to have only capabilities and values as parameters/return values, while the methods of others ("local types") would be permitted only references and values. This would have the linguistic advantage that static type checking could occur, since the types of parameters/return values for methods of "file types" would have to be declared statically as capabilities or values, but not references. The disadvantage of this solution is that types would in some cases need to be doubly defined, which clearly violates the principles laid down for orthogonal persistence [2].

Another possibility would be to define a capability for any type `T` as a subtype of a reference for `T`, i.e. `T**` is a subtype of `T*`. This has the positive effect that within a particular file context a capability can always be used where a reference is expected. This comes closer to upholding the principles of orthogonal persistence. But it has a severe disadvantage: each inter-object method invocation (including all normal method invocations between local objects) would require a run-time check to ensure that local references are not being passed to or returned from inter-file calls. (The apparent



alternative of allowing references to be passed as parameters/return values without allowing them to be *used* outside the local context does not help, as this also requires run-time checks.)

Defining capabilities as supertypes of references would have the consequence that any reference could be used as a capability, which is clearly inappropriate.

A different possibility is to treat `T**` and `T*` simply as separate types, unrelated except via `Handle`, the common supertype of all references and capabilities. This has the disadvantage that a capability cannot be used as a reference. On the other hand it allows static type checking in that the compiler can examine the methods of a type and determine which of these are suitable as inter-file methods (i.e. which do not have references as parameters/return values). Only the latter can then be permitted to invoke methods of objects which are statically referenced by capabilities. (Since `Handle` has no methods, method invocations must always be separately associated either with capability or reference variables in this solution.)

This solution has at least two advantages: (a) it allows the parameters/return values for the methods of all types to be defined in any combination of capabilities, references and values (though only methods which avoid local references can be treated as public file methods) and (b) static type checking is possible. However, it has the disadvantage that within a particular file context a capability cannot always be used where a reference is expected (as for the subtyping solution). To make this possible, for each Timor type `T` there is a type designated as `T***`, known as a *type handle* for `T`. This is a subtype of `Handle` and a supertype of the types representing the various modes which a `T` type can take (i.e. a value, a reference and a capability), in the example `T`, `T*` and `T**`. For cases where polymorphism is appropriate, the type handle `T***` can be used. But where methods are invoked via a `T***` variable, dynamic type checking is necessary. However, since we anticipate that professional designers will in many cases consciously design types either as files or as local objects (just as they do in current systems) type handles are likely to be used only infrequently. Hence their introduction is an attempt to achieve efficiency and static type checking in most situations, but without ruling out polymorphism where this may be useful.

The effect of this is not that only types with parameters specified as type handles can be instantiated as files – any type can in principle be instantiated as a file. The limitation is that *methods* of a type which is instantiated as a file can only be invoked provided that they have references neither as parameters nor as their return value. Thus for example the instantiation of `Person` as a file (see section 3) is valid, despite the fact that there are methods for setting and getting the abstract reference `spouse`. However, an attempt to call these methods via a type handle could lead to a run-time error.

Referencing Objects of Other Files

For programmers unfamiliar with the idea that local references cannot be used globally (i.e. outside the context of the file to which they are local), the question arises how an object local to one file object might be identified uniquely in another and then later not

only be found but referenced again in the file object to which it belongs. Put simply, how can a method of `PersonDatabase` (section 3) pass out identifiers for its `Person` objects to other programs/file objects and then later not only receive them as parameters to other methods but also obtain local references for them?

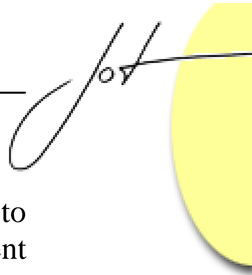
In a more conventional database system this would be achieved by using a unique attribute of the local object (e.g. name and date of birth or some other "key") and then using an index to find the local object (e.g. record). This is in principle a sensible approach and can be applied to Timor files. However, Timor provides globally unique identifiers (over time and space), even for local objects (although these do not appear directly in references). Each file object has a world-wide unique identifier which can be obtained from within the file using the pseudo variable `fileNum`. Similarly each local object within a file has an object identifier which is unique within the file, which can be obtained by using the `identify` operator (in conjunction with a reference for the object). As these identifiers are never reallocated to other files or objects within a file, it is possible for a programmer to define a type (such as `PersonId` in section 3) with values which uniquely identify objects (either directly or indirectly) and which can be passed out of the file as values.

A file object can later generate a reference for this identifier (when it is passed back in as a parameter), by maintaining an index which maps unique identifiers onto local references, with individual entries created when identifiers are constructed, before they are passed out. As indicated above, this approach is reminiscent of conventional database systems which find individual records using an indexing mechanism.

Can a file method have a guarantee that global identifiers for its local objects, when passed back to it as parameters, are genuine? The ultimate answer to this question must be negative, as the Timor compiler has no control over the values of types passed into it from programs not written in Timor. Hence if security is an issue, a global identifier of a local object cannot be considered to have the same properties as a capability (or local reference). Before converting a global identifier for a local object into a reference, the receiving file object must therefore in appropriate cases carry out additional checks (just as the acceptance of a bank card must be supplemented by a check of the PIN, for example). A special language mechanism is not defined for achieving this, as it may be appropriate in some applications for programmers to take arbitrary measures to ensure the integrity of their parameters. Of course standard library routines can be implemented to assist with such protective measures, e.g. via a mechanism such as the password capability technique [1]. Standard components can also be developed to map identifiers onto references, and to support the related indexing operations.

5 IMPLEMENTING TIMOR PERSISTENCE

Implementing Timor's persistence concepts in a conventional OS environment such as Unix or Windows is clearly not a straightforward activity. However, its key features (including persistent files, capabilities) map directly onto the SPEEDOS operating



system[5] [7], which is a new operating system being developed as a parallel project to Timor. Hence an implementation of the Timor persistence concepts in an environment controlled by the SPEEDOS system is a relatively straightforward matter, and needs no further discussion.

SPEEDOS Emulator

It is of course important that Timor programs can be executed in environments which run under the control of conventional operating systems. To achieve this we plan that the Timor run-time environment consists of two almost independent parts:

- a conventional run-time system for managing the usual run-time activities needed by any programming language, and
- a SPEEDOS emulator, which emulates those parts of the SPEEDOS kernel necessary for managing persistent files and capabilities and for executing inter-file method calls, as well as providing operating system access to Timor resources.

A SPEEDOS emulator is required on each system which supports Timor. From the viewpoint of implementing the concepts described in this paper, its key features are as follows:

- Memory mapped files are used as containers for file objects, thus avoiding the need to transform data structures between different formats and in effect providing a form of direct addressability.
- The ability to access directory modules, which can be viewed superficially as types that map string names onto capabilities. Such directories can of course be defined and implemented in Timor.

In order to make capabilities created in Timor accessible at the operating system level, such a directory system, using a predefined directory format known to the SPEEDOS emulator, will be used. Facilities will then also be provided by the emulator to access such capabilities in order to execute programs, etc.

Capabilities

To provide effective protection, capabilities must identify files uniquely both over time and over space. To achieve this uniqueness capabilities contain very large numbers (see section 3) which can be interpreted in such a way that the object can be located. (How this works in detail is not relevant to the present paper.)

In addition they must clearly define the file access permitted to the holder of the capability. In contrast with conventional file systems (e.g. in Unix or Windows) the access rights in capabilities are not expressed in terms of basic rights such as the permission to read or write data, but as a set of permissions indicating the right to invoke methods of the file object.

[5] www.speedos-security.org

Organisation of File Containers

Each file container holds a local object list, which locates instance records of its internal objects when references are evaluated. The first entry within the object list (i.e. object 0) locates the instance record of the main object (e.g. the `PersonDatabase` instance). References are not implemented as direct pointers but contain an indirection via the object list. This means that when an object is explicitly deleted, its object list entry can be so marked. If an attempt is made to access a deleted object, this is detected from its entry in the object table and an exception is raised.

The state of a file consists of object 0 and any objects which can be reached directly or indirectly from the object 0 instance record. Other objects in the object list can be deleted and their space reclaimed. Because only one real pointer to an object exists (in the object table), physical relocation of objects is simple.

Garbage Collection

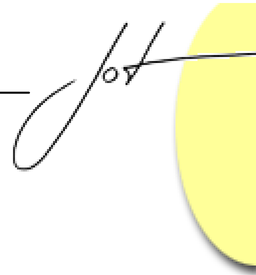
Garbage collection can be considered at two levels. At the level of files (where pointers are capabilities) it would be absurd to use conventional garbage collection techniques. It is clear that a search of all possible Timor programs and files (worldwide in the Internet) to find capabilities for other files, and then delete those files for which no capability exists, would be a fruitless and exorbitantly expensive procedure (even if one could guarantee that all files containing capabilities were actually available on line to search). We therefore adopt the conventional view found in file systems, *viz.* that files are explicitly deleted. No attempt is made to garbage collect on the basis of capabilities.

Garbage collection within individual files is a quite different matter. We have already indicated that each file serves as a separate persistent root, and garbage collection of an individual file, if this is inactive[6], consists of locating all objects reachable from object 0 and deleting the rest.

In general garbage collection activities are considerably reduced in comparison with a language such as Java (a) because objects can be deleted explicitly and (b) because Timor distinguishes between objects and values. If an object is deleted, all its values can be deleted, because by definition there are no references to values. What remains when an object is deleted is a single entry in the object table, which indicates the demise of the object. Otherwise its entire state, and (recursively) the states of all its values, can be deleted and the space explicitly reclaimed.

If a new (complex) value is assigned to a value variable of some object, it is always possible to delete the old value and reclaim the space, since by definition there are no references to value variables. Garbage collection within a file can therefore be confined to locating *objects* (but not values within objects) for which no references exist.

[6] For garbage collection of an active file, the references in the activation records of active threads must also be taken into consideration.



6 RELATED WORK

Most OO languages provide no direct support for persistence and for the protection of persistent information. In contrast Java has basic extensions which assist with persistence but not protection (which is provided in Timor via both capabilities and dynamic qualifiers). We therefore now briefly consider persistence in Java.

In Java, unlike Timor, persistence is not automatic, but must be explicitly organised by the programmer. For this purpose the `Serializable` interface can be used. This removes from the programmer the burden of flattening objects, using a deep copy approach, i.e. each object reachable from the specified object (provided that it also extends the `Serializable` interface) is serialized, and the result can for example be written out to a file and then later read back and the objects reconstructed. Changes made to an object after it has been written out are not reflected in the file, i.e. such objects are *saved* but are not really persistent in the Timor sense. Furthermore, if two objects O_1 and O_2 which share a third object S are separately written out and read back in, the sharing semantic is lost (because S is copied separately with O_1 and O_2). In contrast, the Timor approach to persistence has no difficulties with semantics of shared objects, because flattening never needs to happen.

Major objects in Java programs (equivalent to Timor files) cannot simply be defined as persistent and their methods directly invoked from other (external) objects, as in Timor. However Java provides a remote method invocation (RMI) mechanism which allows Java programmers to achieve an apparently similar effect. Provided that a Java object belongs to a class which extends `Remote`, something like a Timor "file" object can be set up and its methods can be invoked from external objects. But because objects are not naturally persistent, the object must have been activated before it can be invoked, i.e. remote objects, unlike Timor files, are active entities with their own thread(s).

Classes that appear as parameters and return values of the methods of classes which extend `Remote` have to be defined as `Serializable`, because these are always passed as copies (even if this was not the intention of the programmer and even if this changes the semantics of the class when an instance of the class is used locally). In practice this corresponds loosely to Timor's decision not to allow references for local objects to appear on a file interface. But if objects are passed by value in Timor, flattening is of course unnecessary, and the semantics are quite clear and consistent.

Java provides no protection mechanisms for controlling access to major objects. In contrast Timor allows capabilities which establish a client's right to invoke a file (i.e. a remote object). Capabilities can restrict a client's access to a subset of the file's methods. In addition, the mechanism for qualifying objects using bracket methods can be arbitrarily programmed to provide access control lists, password checking and an endless number of other protection checks.

Proposals have been made for adding persistence to Java (cf. [3, 9, 10]), based on the concept of orthogonal persistence first proposed in [2]. With this approach a Java programmer can identify a "root of persistence", such that all objects reachable from this

are automatically persistent. If a program terminates normally the persistent store is automatically updated, but on abnormal termination the transaction associated with the program aborts and the store is not updated.

According to a recent tutorial [24] Orthogonal Persistent Java (OPJ) now supports persistence for remotely invocable objects. The main difference from standard RMI is that remote objects are created once (as a root of persistence). However, a program is needed to make such an object available for remote use (i.e. to export it by binding it to an RMI registry), and its process then waits for incoming calls. Whenever the remote system is restarted this exportation process must be repeated. Such remote objects behave more like Timor files, but their persistence must be actively maintained, and the kind of protection mechanisms supported by Timor are not available. References for internal objects of a remote persistent object are passed by copying (see above for normal Java), but it has been suggested that future work in this area could lead to the idea that such objects are only partially copied [23]. But it is not clear how this could solve the semantic problem mentioned above.

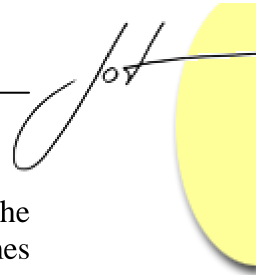
7 CONCLUSION

The paper presents a new concept for supporting persistent objects in programming languages. Unfortunately space restrictions have prevented us from describing some closely related concepts in detail. In addition to its support for persistent objects, Timor also supports persistent processes, following the in-process or procedure-oriented model (cf.[20, 22]). In addition it has a simple concept for managing the distribution of persistent files (and processes) over remote computers. These themes, which affect a number of issues discussed in the present paper (e.g. garbage collection, inter file calls), are discussed in detail in a companion paper, which is scheduled to appear in the next issue of the Journal of Object Technology.

One of the key applications for persistence is database systems, and in this respect Timor provides further relevant features which are indispensable for a modern database system.

Since the idea of transactions was first proposed, this technique has assumed a central role in conventional database system design (cf. [8]). Timor does not support a special transaction technique, because qualifying types [14] can be programmed to provide transaction support on a general purpose basis, allowing different policies to be adopted for individual systems (e.g. with an optimistic or a pessimistic approach) [17].

Conventional statically typed OO languages do not include a mechanism which allows an object to continue to exist in the face of changes to its type, for example to reflect role changes in the real world. For database applications such role changes are of course essential. In Timor they can easily be modelled using attribute types dynamically. The static use of attribute types is described in [16]; in a future paper we will show how it is possible dynamically to add attributes to and remove them from individual objects in a type safe manner.



When support for transactions and for changing object roles is combined with the persistence, distribution and protection concepts described in this paper, it becomes evident that Timor can be viewed as an object oriented database language.

REFERENCES

[1] M. Anderson, R. D. Pose, and C. S. Wallace, "A Password-Capability System," *The Computer Journal*, vol. 29, no. 1, Feb. 1986, pp. 1-8, 1986.

[2] M. P. Atkinson, P. J. Bailey, K. J. Chisholm, P. W. Cockshott, and R. Morrison, "An Approach to Persistent Programming," *The Computer Journal*, vol. 26, no. 4, pp. 360-365, 1983.

[3] M. P. Atkinson, L. Daynès, M. J. Jordan, T. Printezis, and S. Spence, "An Orthogonally Persistent Java," *ACM Sigmod Record*, vol. 25, no. 4, 1996.

[4] A. Bensoussan, C. T. Clingen, and E. C. Daley, "The MULTICS Virtual Memory: Concept and Design," *Comm. ACM*, vol. 15, no. 5, pp. 308-318, 1972.

[5] K. B. Bruce, L. Cardelli, G. Castagna, The Hopkins Objects Group, G. T. Leavens, and B. Pierce, "On Binary Methods," *Theory and Practice of Object Systems*, vol. 1, no. 3, pp. 221-242, 1995.

[6] R. C. Daley and J. B. Dennis, "Virtual Memory, Processes and Sharing in MULTICS," *Comm. ACM*, vol. 11, no. 5, pp. 306-312, 1968.

[7] K. Espenlaub, "Design of the *SPEEDOS* Operating System Kernel," PhD Thesis, *Department of Computer Structures*, University of Ulm, 2005.

[8] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*. San Francisco. Morgan Kaufmann, 1993.

[9] M. J. Jordan, "Early Experiences with Persistent Java," The First International Conference on Persistence and Java, Glasgow, 1996.

[10] M. J. Jordan, "A Comparative Study of Persistence Mechanisms for the Java Platform," http://www.sunlabs.com/techrep/2004/smli_tr-2004-136.pdf, 2004.

[11] J. L. Keedy, G. Menger, and C. Heinlein, "Support for Subtyping and Code Re-use in Timor," 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002), Sydney, Australia, 2002, Conferences in Research and Practice in Information Technology, vol. 10, pp. 35-43.

[12] J. L. Keedy, G. Menger, and C. Heinlein, "Inheriting from a Common Abstract Ancestor in Timor," *Journal of Object Technology*, vol. 1, no. 1, pp. 81-106, www.jot.fm/issues/issue_2002_05/article2, 2002.

[13] J. L. Keedy, G. Menger, and C. Heinlein, "Taking Information Hiding Seriously in an Object Oriented Context," Net.ObjectDays, Erfurt, Germany, 2003, pp. 51-65.

[14] J. L. Keedy, K. Espenlaub, G. Menger, and C. Heinlein, "Qualifying Types with Bracket Methods in Timor," *Journal of Object Technology*, vol. 3, no. 1, pp. 101-121, www.jot.fm/issues/issue_2004_01/article1, 2004.

[15] J. L. Keedy, G. Menger, and C. Heinlein, "Inheriting Multiple and Repeated Parts in Timor," *Journal of Object Technology*, vol. 3, no. 10, pp. 99-120, www.jot.fm/issues/issue_2004_11/article1, 2004.

[16] J. L. Keedy, G. Menger, and C. Heinlein, "Diamond Inheritance and Attribute Types in Timor," *Journal of Object Technology*, vol. 3, no. 10, pp. 121-142, www.jot.fm/issues/issue_2004_11/article2, 2004.

[17] J. L. Keedy, K. Espenlaub, C. Heinlein, G. Menger, F. Henskens, and M. Hannaford, "Support for Object Oriented Transactions in Timor," *Journal of Object Technology*, vol 5, no. 2, March-April 2006, pp. 103-124 http://www.jot.fm//issues/issue_2006_03/article1.

[18] J. L. Keedy, K. Espenlaub, G. Menger, and C. Heinlein, "Call-out Bracket Methods in Timor," *Journal of Object Technology*, vol. 5, no. 1, 2006, pp. 51-67, http://www.jot.fm/issues/issue_2006_01/article1.

[19] B. W. Lampson, "Protection," Proc. 5th Princeton Symposium on Information Sciences and Systems, 1971.

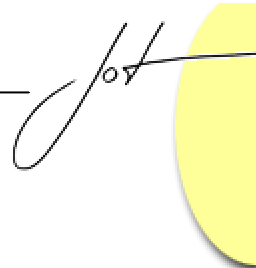
[20] H. C. Lauer and R. M. Needham, "On the Duality of Operating System Structures," *ACM Operating Systems Review*, vol. 13, no. 2, pp. 3-19, 1979.

[21] D. L. Parnas, "On the Criteria to be Used in Decomposing Systems into Modules," *Communications of the ACM*, vol. 15, no. 12, pp. 1053-1058, 1972.

[22] K. Ramamohanarao, "A New Model for Job Management Systems," PhD Thesis, *Department of Computer Science*, Monash University, 1980.

[23] S. Spence, "Distribution Support for PJama", http://www.dcs.gla.ac.uk/~susan/perdis.html, 2004.

[24] Sun MicroSystems, "The OPJ Tutorial", http://research.sun.com/forest/opj.tutorial.tutorial.html, 2000.



About the authors

J. Leslie Keedy retired from the position of Professor and Head, Department of Computer Structures, University of Ulm, Germany in 2005, where he previously led the Timor language design and the Speedos operating system design groups. His email address is keedy@jlkeedy.net. His biography can be visited at http://www.jlkeedy.net/biography_short.php

Klaus Espenlaub completed his Ph.D. in Computer Science at the University of Ulm in 2005. He is currently employed by InnoTek Systemberatung GmbH. His research interests include secure operating systems, protection mechanisms and computer architecture. His email address is klaus@espenlaub.com.

Christian Heinlein has been working as a Scientific Assistant at the University of Ulm, Germany, where he conducted the research project APPLEs, that aims at developing “Advanced Procedural Programming Languages,” which are both conceptually simpler and more flexible than standard object-oriented languages. He can be reached at christian.heinlein@uni-ulm.de. See also www.informatik.uni-ulm.de/rs/mitarbeiter/ch/apples.

Gisela Menger received a Ph.D. in Computer Science from the University of Ulm in 2000. She recently retired from the Department of Computer Structures at the University of Ulm. Her research interests include programming language design and software engineering. Her email address is gisela.menger@uni-ulm.de.