Persistent Programming with Speedos and Timor

J.L.Keedy
keedy@jlkeedy.net
formerly Professor/Honorary Professor at the University of Newcastle, NSW and Monash University, Melbourne
the Technical University of Darmstadt, the University of Bremen and the University of Ulm in Germany

Abstract

Speedos (an acronym for ***S****ecure* ***P****ersistent* ***E****xecution* ***E****nvironment for* ***D****istributed* ***O****perating* ***S****ystems) is an unconventional design for a very secure capability-based operating system, which supports an automatically persistent virtual memory. Timor (an acronym for* ***T****ypes,* ***I****mplementations and* ***mor****e) is a programming language designed to support Speedos (and other) systems.*

The paper first introduces the concept of persistence and its background in computer science. It then describes the main features of the Speedos architecture and in particular its virtual memory mechanism, its process model, its protection mechanism and its technique for logging users in and out. This is followed by a short description of Timor, which is an unconventional object-oriented programming language designed inter alia to support the Speedos architecture.

1 Introduction: Background of Persistence in Computer Science

Conventional programming languages usually provide features for manipulating *temporary* data structures which are generally straightforward and convenient for programmers to use. These include structures such as arrays, records and linked lists. However, these convenient programming constructs cannot normally be directly used for accessing *persistent* information, which is held in the file system. This reflects the fact that information in the file system cannot be directly addressed by user programs. Instead the programmer accesses the latter via a special file interface provided by the programming language, which is then transformed into the operating system's file access interface routines.

1.1 Programming Languages

There are several disadvantages in having one style of interface for file data and another for temporary data. First, the temporary data structures in a program are not stored in compatible formats with the persistent data structures. Second, storing temporary data structures into conventional files is often not a straightforward task, not only because of the different formats, but also because pointers consisting of addresses in the virtual memory cannot simply be copied into the file system and later reused, because the underlying main memory or virtual memory addresses may be different at a later time. This problem is further complicated by the fact that files are commonly used concurrently by several application processes.

This theme was taken up in the early 1980s by M.P. Atkinson and his colleagues at the University of Glasgow together with R. Morrison and his group at the University of St. Andrews. In order to avoid having two different approaches for programming temporary and persistent data, they developed a programming technique known as *persistent programming*, based on the use of *orthogonal persistence* [1]. They argued inter alia that the same data structuring mechanisms should be used to program temporary data structures in the computational memory and to program persistent data structures. To demonstrate this idea they developed the persistent programming language PS-Algol [2][1] and later a persistent programming language called Napier [3].

[1] see also https://en.wikipedia.org/wiki/PS-algol.

The persistent programming groups set about demonstrating the feasibility of persistent programming by implementing "persistent object stores" for PS-Algol and Napier above conventional hardware, using the basic facilities of conventional file systems. Such a software-oriented approach, which inevitable has a high performance overhead because it had to be implemented in a conventional virtual memory environment, was forced upon them by a lack of appropriate hardware.

1.2 Database Management

The management of large bulk data files has become a specialized activity, known as database management, and this has resulted in the development of special database languages which have tended to use quite different data models from those underlying the design of programming languages. Hence these too have quite different interfaces from the programming language data structures. While such database systems tend to provide much more powerful facilities than basic file systems they add yet another layer of complex software which adds to the inefficiency of data accesses. Thus although in the final analysis the application manipulates its persistent data – like its temporary data structures – in the virtual memory, it may have to do this indirectly via database routines which themselves may call file system routines.

1.3 Further Duplications

The sharp division in most systems between a computational virtual memory and a file and/or database system gives rise to at least two further areas of duplication and unnecessary complication: synchronisation and protection. With regard to synchronisation, the CPU normally provides simple and efficient mechanisms, but above this the file system provides further synchronisation mechanisms, and then on top of that there are often additional database mechanisms to achieve synchronisation. This is necessary because the CPU instructions cannot act directly on synchronisation variables in the file or database system, since the latter cannot be directly addressed in the virtual memory.

And perhaps most significantly from our current perspective, the conventional virtual memory organisation leads to a multiplicity of protection mechanisms. This is inevitable if data in the file and database systems cannot be directly addressed. This additional complexity is more likely to assist security breaches than to hinder them.

What all of these points clearly indicate is that enormous benefits could be gained if it were possible to address both non-persistent (computational) and persistent (file and database system) data structures in the virtual memory in a uniform manner. How then can such a directly addressable file system be implemented?

1.4 Architecture and Hardware

1.4.1 Multics

In the mid-1960s, when mainframe computer systems carried out their work in a batch processing mode and personal computers had not yet been invented, computer architecture researchers at MIT in Cambridge, Massachusetts, developed a significant research system called Multics [4, 5]. Its aim was to demonstrate ideas relevant for time-sharing, i.e. for computer systems where individual users sit at terminals and interact directly with the (shared) central computer.

Among the many revolutionary design ideas which appeared in Multics was an idea called "direct addressability" by Multics designers. What they aimed to achieve with this idea was to allow *all the information in a system* to be directly addressable in the virtual memory, including information held in the file system. The fundamental advantage which they saw was that it avoids much copying of information between the file system and the computational (virtual) memory. The lack of success of this idea – in my view the most significant of all the

Multics ideas – in the last six decades is due not to a fault in the basic idea, but in the way it had to be implemented on the hardware available at that time. The Multics idea is discussed in chapter 12 of the book "Making Computers Secure" (volume 1) which can be downloaded on the Speedos website (https://www.speedos-security.org/).

1.4.2 Monads

In the late 1970s and early 1980s my team at Monash University in Melbourne, Australia tackled the problem of providing a uniform mechanism for managing the persistence problem in a radically different way. Whereas Multics accepted the existence of and need for a file system, Monads completely rejected this concept and chose to view all the memory which holds information as a very large virtual memory which can be directly addressed by programs. This implied that virtual addresses must be much larger than 32 bits, which was the standard at that time. Consequently my PhD students David Abramson and John Rosenberg designed and built a system known as the Monads-PC [6], which had 60 bit virtual addresses. This system proved the concept of a directly addressable large virtual memory which functioned without an additional file system. Several Monads-PC computers were actually built and successfully used at several universities in Australia and in Germany for research and teaching purposes. The Monads website (https://www.monads-security.org/) contains extensive diagrams, text and references which explain both the hardware and the software for the Monads-PC including the Persistent Virtual Memory concept, and also the idea of orthogonal paging and segmentation, which allows both paging and segments of any size (both smaller and larger than the single page size).

1.4.3 Later Attempts to support Persistence

Around the time that 64-bit computers were introduced there was a spate of attempts to take advantage of this situation, as is described in [7]. As often happens, at least two of these, both Australian, were overlooked in papers emanating from US researchers. The first of these was a paper co-authored by David Abramson and myself [8]. The second Australian paper, which first appeared as a report from the University of New South Wales [9], was quite interesting, in that it used 3 bits in virtual addresses to identify which of 8 computers in a local area network the relevant data was to be found. However this illustrates the weakness of all the proposals of that generation, i.e. it was not extendible for use over the internet. That is one of the issues which Speedos addresses.

2 The Speedos Architecture

Speedos[2] is a design for a capability-based operating system [10] which inter alia automatically supports persistent programming via an unconventional virtual memory organisation and a rigorous use of the in-process design method.

2.1 The Speedos Virtual Memory Organisation

Persistence (the property via which data remains after the power has been turned off) is achieved at the operating system level in Speedos via its unconventional virtual memory organisation. Figure 1 shows how most conventional computers organise virtual memory.

The Monads systems (see section 1.4.2 above), a forerunner of Speedos, successfully implemented a different and conceptually much simpler solution in the early 1980s, in the form of "Persistent Virtual Memory" (see Figure 2).

[2] For an overview of the system architecture see https://www.speedos-security.org/. From this website various books and papers providing further information about Speedos can be downloaded free of charge.

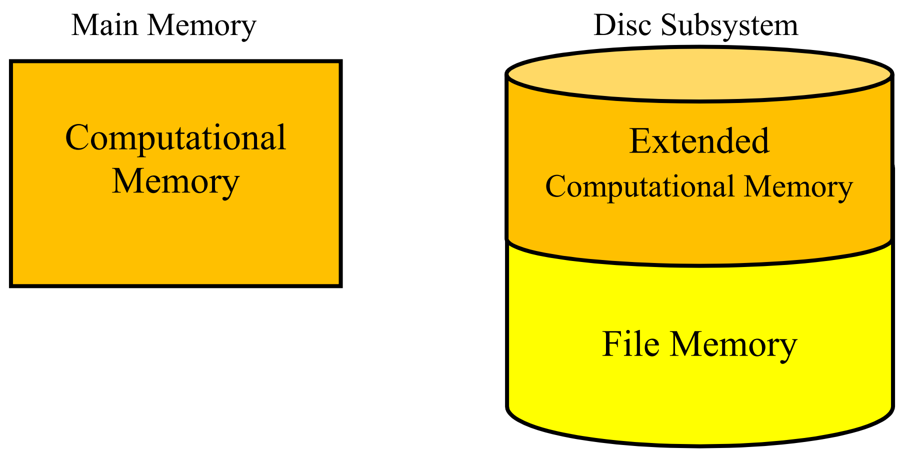

Figure 1: Conventional Virtual Memory Organisation

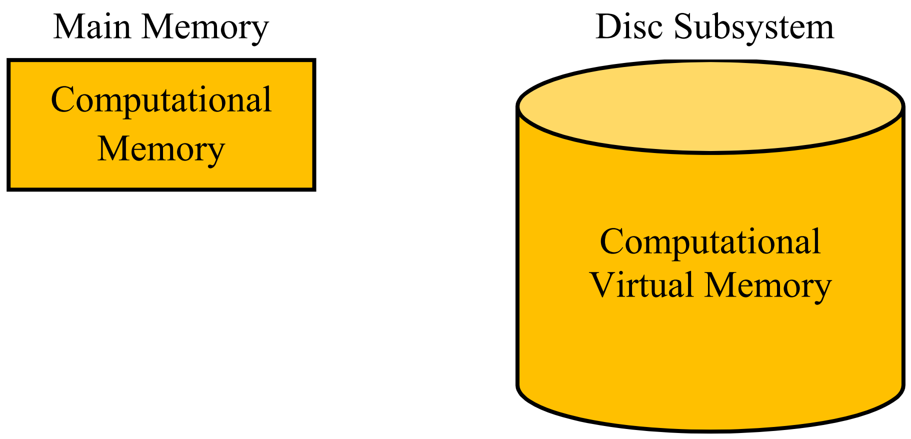

Figure 2: Persistent Virtual Memory

This solution, which eliminates entirely the idea of a separate file system, allows programs directly to address the entire file system via a single addressing mechanism. This mechanism solves many operating system problems (e.g. that of the Multics designers) and allows programmers to use the persistent programming paradigm proposed by Atkinson and Morrison [1].

This view of virtual memory implies that virtual addresses must be very much larger than conventional virtual addresses. In the Speedos solution these are 256 bits long and provide a worldwide addressing mechanism, as is shown in Figure 3.

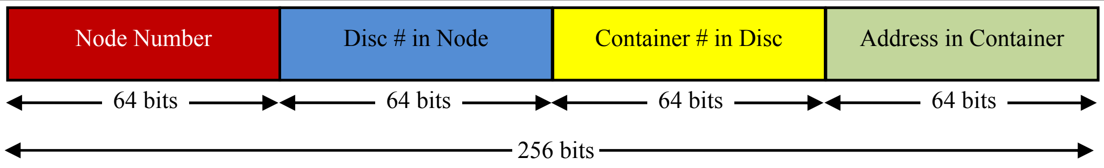

Figure 3: A Speedos Full Virtual address

A virtual address advises the Speedos system at which node in the internet the address was first assigned, the disk number within node defines the disc on which the addressed object was initially created, the container (the physical file) describes where it was created and the address within container indicates the offset in container of the byte or word which is addressed. All of these are used as aids to finding a file and its contents. But these are not fixed in cement. To discover how Speedos virtual addresses are actually managed (including movements between nodes, etc.) and how such long virtual addresses can be efficiently translated into main memory addresses, volume 2 of the book "Making Computers Secure" [10] should be consulted. This can be downloaded from the Speedos website at https://www.speedos-security.org/.

2.2 Capability Based Protection

Obviously the objects in Speedos cannot simply be arbitrarily addressed, and addresses cannot simply be manufactured as arbitrary 256 bit entities. Hence it is important to have a protection mechanism which prevents this. Protection in Speedos is based on a two-level capability mechanism.

2.2.1 Segment Capabilities

The first level, which functions at the level of the actual code execution, allows segments in the same module to be protected from each other. The mechanism ensures that a segment can only access other segments if it can access a protected pointer to them and even then the mechanism determines whether they can be accessed in read-only or read-write mode. How this functions is described in detail in the paper "S-RISC – Adding Security to RISC Systems", which can be downloaded from the Speedos website. The paper also describes an enhancement to RISC systems which would allow otherwise conventional RISC computers to support capability-based addressing.

2.2.2 Module Capabilities

More significantly from the viewpoint of this paper, there is a second level capability mechanism which ensures that modules in the virtual memory can only access other modules if they have been granted permission to do so. There are no conventional files and there is no conventional files system. Instead *all* modules in the virtual memory are based on the information-hiding principle [11, 12, 13], as is illustrated in Figure 4.

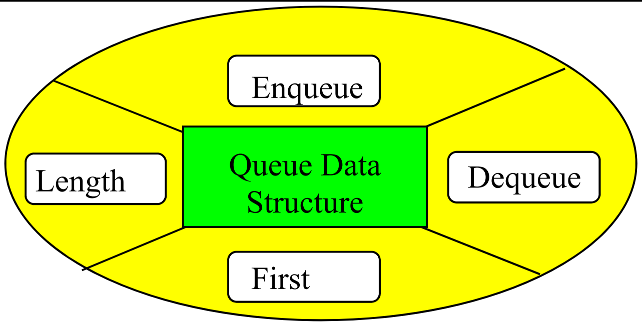

Figure 4: A Simple Information Hiding Module – A FIFO Queue

Not all information-hiding modules need to conform exactly to this scheme. For example the persistent data structure(s) may be absent, with the result that *programs* can be defined which might have only a single entry point or might have multiple entry points, in which case a compendium of board games can be programmed, for example.

Such modules are the only free-standing units in a system, i.e. the entire persistent memory is populated only by information-hiding modules. They communicate with each other via *inter-module calls*.

In order to make a call from one module to another the calling module must present the kernel with a valid *module capability* (see Figure 5) which identifies both the module to be called and the number of the entry point which it wishes to call. The access rights consist primarily of a bit list indicating which entry points can be called via this capability. If the kernel is satisfied with the module capability, it then makes an in-process call to the called module, on the calling user's process stack. Eventually, possibly following further inter-module calls and returns (also executed by the kernel), the called module will be reactivated at the instruction following the inter-module call.

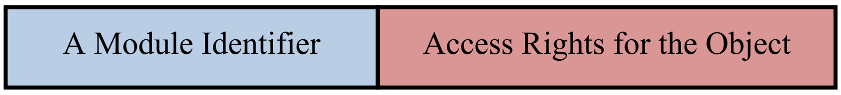

Figure 5: A Module Capability

2.3 Semantic Files and Semantic Access Rights

Conventional file systems often provide a simple procedural interface with routines for example to insert and delete a record for a particular file. In this sense they can be viewed as providing a very minimal support for the information hiding principle, which at least hides details such as the physical location of the file on disc but may also provide a basic structuring and accessing mechanism (e.g. indexed sequential).

This means that the entire semantics associated with the information in the file must be programmed in the programs which use the file. But this leads to the same disadvantages which led Parnas to propose the information hiding principle. Each program using the file – and in a banking system that will be a considerable number – e.g. a program for managers, one for tellers, one for accountants, one for calculating interest, etc. – must know the layout of an account record and the significance of its fields. So, changing something about the record layout will probably involve changing several programs. It also leads to duplicated software in the individual programs, where they have to provide the same or similar.

The Speedos alternative allows a much more flexible approach to information hiding, with corresponding software engineering and protection benefits. Consider for example a bank accounts file in a conventional environment. The file system will provide operations that allow individual bank account records to be inserted, viewed, modified, deleted, etc. (see Figure 6). But conventional file systems take no account of the fact that the records are bank account records. From the viewpoint of the file system they could just as easily be car registration records, or club membership records, etc.

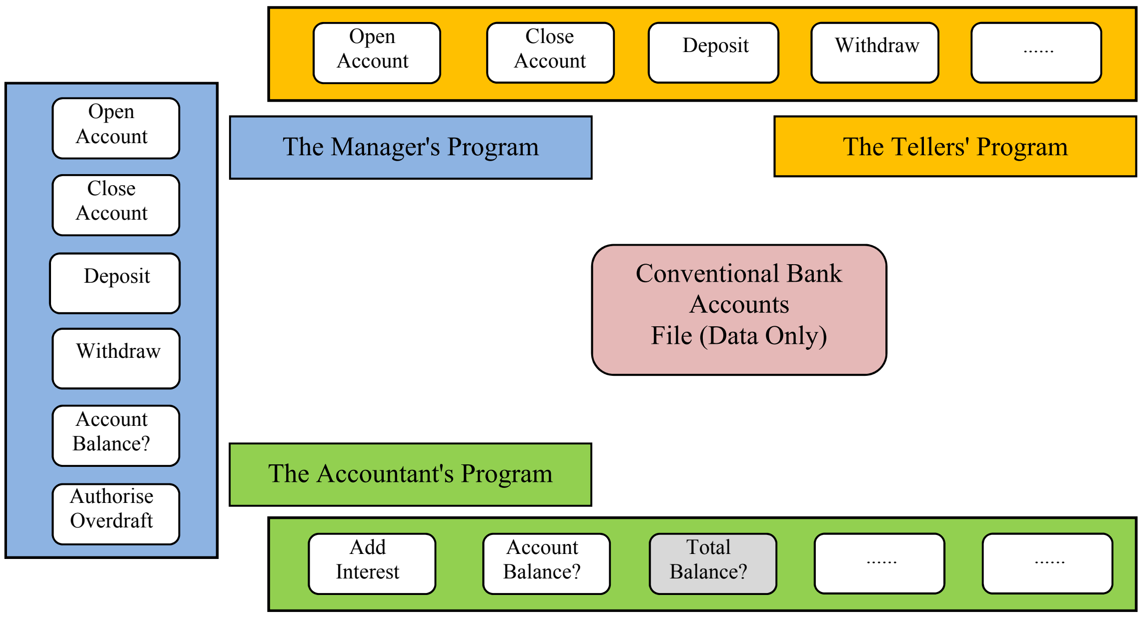

Figure 6: A Conventional View of a Bank Accounts File

In Speedos all of this can easily be avoided by providing a semantically appropriate interface to an information hiding file, e.g. a method for adding interest to an account, one to authorise an overdraft, etc. (see Figure 7).

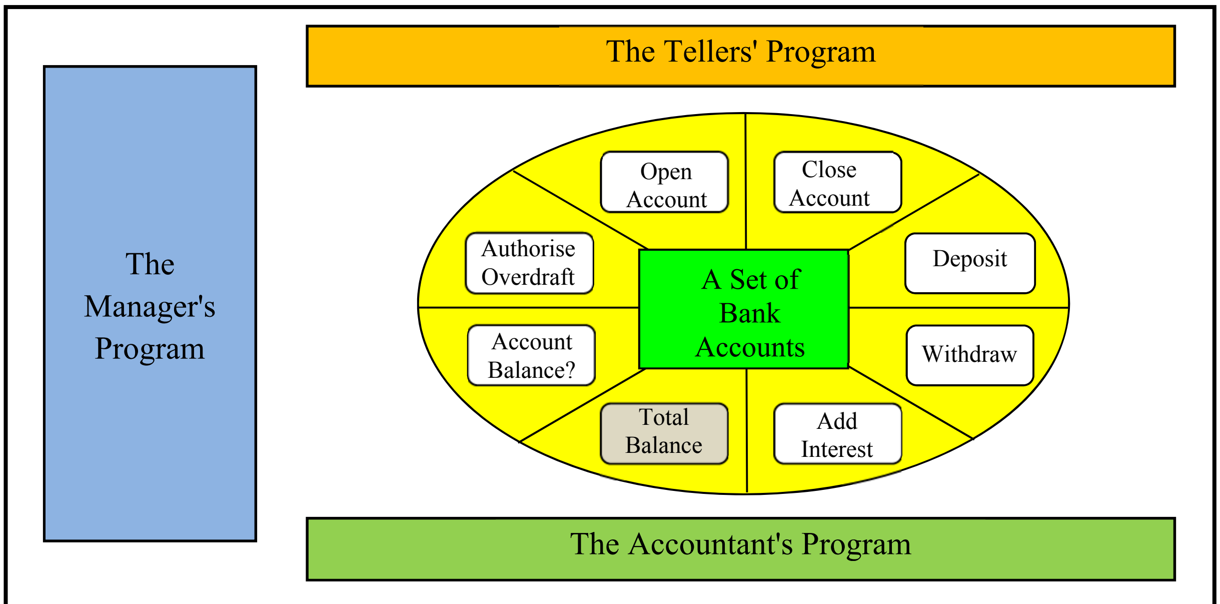

Figure 7: The Information Hiding Solution

Notice also that this arrangement allows the designer of a system to provide internal protection within a file, in that different agents can be given appropriate (but potentially different) access rights to the same file, as Figure 8 illustrates.

	Teller	Branch Manager	H.O. Accountant	H.O. Auditor
Open Account	√	√	x	x
Close Account	√	√	x	x
Deposit	√	√	x	x
Withdraw	√	√	x	x
Transfer	√	√	√	x
Add Interest	x	x	√	x
Authorise Overdraft	x	√	x	x
Customer Number	√	√	x	√
Overdraft Limit	√	√	√	√
Current Balance	√	√	√	√

A tick indicates that the subject at the head of the column may carry out the operation in the corresponding row.

Figure 8: Access Rights expressed as Semantic Operations

On this basis different agents within an organisation (here a bank) can be given different module capabilities with access rights appropriate for their role in the organisation. For example a Head Office Accountant might have a module capability as shown in Figure 9.

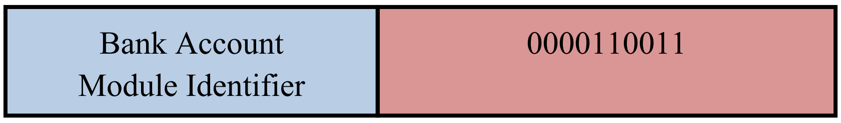

Figure 9: A Module Capability for the Head Office Accountant

2.4 Qualifiers

Qualifiers are a new and unusual kind of module. In addition to having normal methods a qualifier can have "bracket" methods. These can qualify the behaviour of a different object (the "target" object) with which the qualifier is associated. They may have automatically activated methods (called "call-in" methods) which "catch" a call to a method of the target instance. They may also have "call-out" methods which are triggered by a call from a qualified object to some other object (the call-out object). Although they can be used to implement features such as synchronisation and protection, qualifiers have no special role with respect to persistence and so are not further described here. They are explained in detail in volume 1 of the book "Making Computers Secure" and an implementation is described in volume 2 of the same book. These can both be downloaded from the Speedos website[3]. A description of qualifiers, with examples explaining how they can be used to protect and confine the information held in other modules, appears in [14], which can also be downloaded at the Speedos website.

2.5 In-Process Design

There are two basic models for decomposing an operating system into processes. The simplest involves having a separate process for carrying out each operating system activity. We refer to this kind of design as *out-of-process*, because operating system services are provided for an application *out of* the application's process, in a separate system process. The technique is sometimes called *message-oriented*, because the application process must pass its parameters as a message from one process to another.

In the second model operating system services are provided in a process belonging to the *application*. We refer to this kind of design as *in-process*. It is sometimes also called *procedure-oriented*, because the operating system routine is implemented on the stack of the user requesting the operating system service as a procedure call.

These two models are described in more detail in [15], where it is shown that the in-process model has many advantages and is therefore used in the Speedos design. The kernel achieves this by implementing in-process inter-module calls

The rigorous application of the in-process model in Speedos also allows the virtual memory to be shared over the Internet [15] via *remote* inter-module calls.

2.6 Persistent Processes and Secure Logging In

Persistence based on our solution has further advantages. Because the entire content of the virtual memory is persistent, not only files and programs but also *processes* and their threads are persistent. This means that the current state of a process is also preserved automatically when users log out. Thus when he wishes to log in again, he can in principle simply resume his work in the same state that he had left it. Consequently there is no reason for the system automatically to delete his process on logout and to start a new process for him when he logs in again. That is clearly more efficient and more convenient for users.

But the idea of persistent processes brings a further advantage. It opens up the way for adding greater security to the login mechanism. If the final action which a user takes before logging out is to call a "logout" module (which he must do anyway to warn the process

[3] see https://www.speedos-security.org/

scheduler that his process should be temporarily deactivated) he can do this from a logout module of his own choice. Such a module (which is owned by the user, who can also determine what it does) can contain arbitrary checks devised by the user to check his own identity. This need not be a simple password, it can for example be a dynamic password, a cognitive password and/or whether the person attempting to log in has to conform to some required actions[4]. The kernel's part in the login and logout mechanism is trivial. In the case logging in of simply advises the process scheduler that the user is active. This then activates the user's process in the latter's logout module, which then validates the user or, if the checks fail, informs the process scheduler to deactivate the process again. Notice also that there is no central file which can be hacked to obtain login information.

That concludes our simplified description of those parts of the Speedos architecture which are relevant to the theme of persistence. Much more information can be obtained at https://www.speedos-security.org/, including a two volume book on Speedos which can be downloaded free of charge.

3 Persistence in Timor

Timor is an unconventional object-oriented programming language designed inter alia to support programs written for the Speedos architecture. A description of the language appears in the book "TIMOR-An Object- and Component Oriented Language" which can be downloaded free of charge at the Timor website[5]. The website also provides a list of published papers describing various aspects of Timor, which varies in the following ways from conventional object-oriented programming languages. It

- replaces OO classes with a type definition that can potentially have a number of different implementations, each with a single constructor which can have implementation-oriented parameters that can differ in different implementations of the same type;
- supports inheritance in the case of subtype hierarchies which derive from a common abstract ancestor, where the subtypes primarily vary the behaviour of their supertypes rather than add new methods (although new methods can also be added), e.g. as in the case of a collection hierarchy;
- adds the concept of views, which are incomplete types (with implementations), that can be usefully incorporated into different type definitions;
- supports diamond inheritance, and multiple and repeated inheritance with separate types, using a technique known as parts inheritance;
- replaces subclassing by a flexible new implementation technique based on re-use variables;
- introduces a new kind of component, known as a qualifying type, which contains bracket methods that allow instance methods of other objects to be "qualified" in a modular way, e.g. to protect or synchronise them, thus supporting the separation of concerns;
- provides uniform support for distribution and persistence in the form of persistent objects and persistent processes;
- introduces an unusual way of handling makers (the Timor name for application-oriented constructors), binary methods, and class (static) variables and methods, in a new kind of type, known as a co-type, which can be automatically adjusted covariantly to reflect a subtype hierarchy;
- supports genericity in forms which reflect the unusual features of Timor, adding function parameters which allow programmers considerable flexibility, for example by allowing a programmer to redefine what is meant by such issues as equality.

[4] Login Security checking is discussed in Making Computers Secure, volume 1, chapters 4 and 15, and in volume 2, chapter 22 which can be downloaded from the Speedos Website https://www.speedos-security.org/

[5] https://www.timor-programming.org/

3.1 Support for Collections of Objects

Of particular significance for a persistent system is that Timor has strong support for collections of objects, since it must provide an alternative for conventional file systems and database systems, which are usually primarily concerned with accessing large numbers of "records". Timor fills this gap by supporting a library of collection types with various implementation possibilities, known as the Timor Collection Library (TCL). The library, which is based on the doctoral thesis of my former assistant Dr. Gisela Menger [16], has as its starting point an abstract type *Collection*, which defines the methods shared by all its subtypes.

Collection Type Name	Duplication Criterion	Ordering Criterion
Bag	Allow duplicates	No ordering
Set	Ignore duplicates	No ordering
Table	Signal duplicates	No ordering
List	Allow duplicates	User ordered
OrderedSet	Ignore duplicates	User ordered
OrderedTable	Signal duplicates	User ordered
SortedList	Allow duplicates	Sorted
SortedSet	Ignore duplicates	Sorted
SortedTable	Signal duplicates	Sorted

Figure 12 The concrete collection types

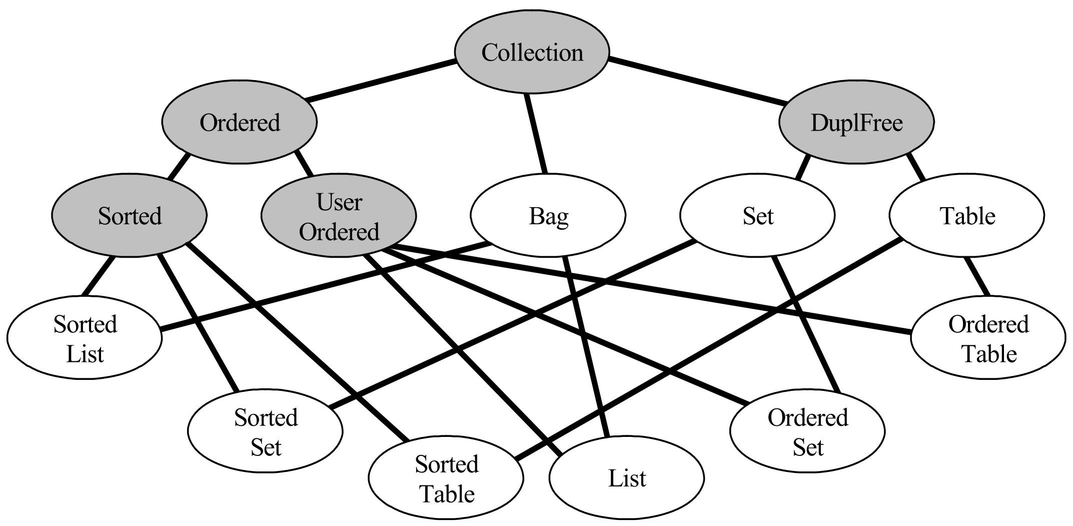

Figure 13: Structure of the TCL Collection Types

The TCL is based on two basic criteria. The first determines whether duplicate items are permitted, are ignored or whether a warning signal is raised when an attempt to insert a duplicate is detected. The second concerns the ordering of items in the collection, i.e. whether there is no ordering, whether the user determines the ordering, or whether they are automatically sorted. This results in a type hierarchy with five abstract types and nine concrete types. The names of the 9 concrete types are shown in Figure 12.

Figure 13 shows all 14 types, including the abstract types (which are shaded). This is designed to allow a maximum of polymorphism. The items in a collection are defined generically.

In derived types the actions of the *insert* method are specified more precisely, depending on the node in question. Thus the *insert* method of the abstract type *UserOrdered* defines that *insert* appends the element at the end of the collection (and adds new methods for inserting at other positions) but without defining its duplication properties further. On the other hand the *insert* method of the concrete type *Bag* is defined without specifying ordering, but indicating that duplicates are accepted (with the effect that the duplicate exception *DuplEx* can be removed from *Bag*'s insert method).

3.2 Implementations and Code Reuse

Whereas in conventional OO programming languages the re-use of code is achieved via subclassing, which is based on inheritance, Timor separates code re-use entirely from inheritance. An implementation of a type is designated as the type name followed by a double colon followed by an implementation name, e.g. `Queue::ArrayImpl` introduces an implementation of the type `Queue`. Each concrete type must have at least one implementation, called `<typename>::Impl`, which in appropriate cases serves as a default. However, this must not necessarily have been explicitly coded. For example, the basic types can have an implicit implementation. Similarly the compiler can automatically provide implementations for "abstract variables"[6] and "records"[7].

To support code re-use in Timor a new concept, called *re-use variables*, is introduced. Like other concrete variables, such variables are included in the state sections of implementations, but unlike most other variables which typically appear in a state section, they may be declared either as types or as implementations. They may not be declared as local variables in individual methods.

Re-use variables are like normal variables in that they form part of the state of the implementation in which they are declared. They are recognisable because their declarations begin with a hat symbol (^), e.g.

```
^Queue myQueue = Queue::ListImpl(); /* Here the re-use
     variable myQueue is declared as a type variable and
     a list implementation constructor initialises it */
```

or

```
^Queue::ArrayImpl myQueueImpl = Queue::ArrayImpl(100);
 /* Here the re-use variable is declared as an implementation
     variable and is initialised by an array implementation
     constructor */
```

or in the case of a typeless implementation simply

```
^::usefulCode
```

In all cases the programmer of the implementation in which the declarations are embedded has access to their interface methods, but in the case of an implementation variable being declared, the programmer can also access its internal state variables and its private instance methods. (If a re-use variable is declared via a type declaration, a maintenance programmer

6 Timor types strictly follow the information-hiding principle by not permitting "raw" data declarations (i.e. fields) to appear in interface definitions. Only methods are permitted. However, to simplify programming, Timor allows some methods to be defined in a type interface as if they were variables. But in reality the compiler treats each such variable declaration as a pair of methods, often known as "setters" and "getters".

7 Often for programming in the small, a type definition may consist entirely of abstract variables; this is often called a "record". In this case the compiler produces a standard implementation of the entire type with the implementation name <typename>::*Impl*. If abstract variables are not explicitly initialised default methods are provided automatically.

can immediately recognise that any implementation of the type can be used.)

A re-use variable may even be another implementation of the type being implemented. On the other hand the type of a re-use variable does not necessarily have any formal relationship with the type being implemented, except that some (or all) of the interface methods may have the same definition.

The important difference between a re-use variable and a normal variable is that the compiler compares the definitions of its interface methods with those of the type being implemented. If some of these have matching signatures, it uses the methods of the re-use variable to implement them, unless the programmer has also declared the same method explicitly in the instance section. In the latter case the explicit method in effect overrides that provided by the re-use variable. Interface methods of a re-use variable which do not match the type definition are ignored. However they can be invoked in the implementation by programmers.

Re-use variables can greatly simplify implementations of the TCL. The first type to be implemented is `List`, and the code of this implementation is declared as a re-use variable in standard implementations of all the other concrete types, with remarkably few modifications. This is illustrated in more detail in chapter 13 of the book "TIMOR-An Object- and Component Oriented Language", which provides an outline of both the type definitions and implementations of the TCL methods. This can be downloaded from either the Speedos website[8] or the Timor website[9].

3.3 Generic Types and Implementations

The TCL would not be very useful without being able to define the elements in a collection generically. Timor allows the elements in a collection to be defined generically but these can then be instantiated as actual collection types, e.g. as a set of the type `Person`, written `Set<Person>`. The identifiers of generic types are distinguished from other identifiers in that the first two characters must be capital letters and all further letters must also be capitals, though other symbols are permitted. The definition of a set of generically defined elements might be `Set<ELEM>`.

Timor supports genericity in the sense that a generic definition provides a pattern for a number of types or their implementations. Such patterns are called *templates*. In order to produce actual types and implementations, a template must be *actualised*. A template itself plays no role at run-time, in contrast with the entities actualised from it.

A template consists of a *template header* and a template body. It has a template identifier and it may use both normal and generic identifiers in its definition. The template identifier is not a generic identifier; it must be unique in the same sense that type and implementation identifiers are unique.

A template header consists of the keyword `template` followed by a set of template parameters bracketed by the bracketing pair < and >. A template body is a normal type or implementation definition which includes the template identifier and may contain generic identifiers.

Here is an example of a template for producing types, known as a *type template*:

```
template <ELEM>
// ELEM is a generic identifier for a type parameter
abstract type Collection {
/* Collection is the template type identifier. This example
   is a shortened version of the abstract supertype of
   collections in the Timor Collection Library (TCL) */
```

8 https://www.speedos-security.org/

9 https://www.timor-programming.org/

```
instance:
enq[10] Int size();  // returns current number of elements
op void clear(); // removes all elements in this collection
enq Boolean contains(ELEM e) throws NullEx;
// returns true if e is an element in this collection
op void insert(ELEM e) throws DuplEx, NullEx;
/* a general method to insert elements; a DuplEx exception or
  a NullEx exception may be thrown by appropriate subtypes */
op void remove(ELEM e) throws NullEx, NotFoundEx;
/* removes e (at most once) if this is contained in the
   collection */
}
```

In this example the template identifier is `Collection` and the single template parameter is `<ELEM>`, where `ELEM` is a generic identifier and type indicates what kind of template it is. The keyword **`abstract`** indicates that in this case the type defined in the following template is an abstract type. The template body consists of the remaining lines of the example.

Like normal Timor types, each type template can have more than one implementation. These are defined in implementation templates, which also consist formally of a template header followed by a template body. The identifier of an implementation template is the name of the implementation. Here is an example:

```
template <ELEM>
impl List::Impl{  /* List::Impl is the template id; it is followed by
the code of the generic implementation */
}
```

However, as the template header of an implementation cannot differ from that of its generic type, this can be abbreviated simply to the keyword template, as follows:

```
template impl List::Impl{
...
// code of the implementation
}
```

Actualising a generic template consists of substituting an actual type name for each generic parameter which appears in the template header. A variable of an actual type can be declared as follows.

```
List<Person*> personList;
// a variable for a List of Person references
```

Instances of generic types are initialised by invoking a constructor of one of the implementations of the type. Here is an example showing how a constructor for a `List<Person*>` might be called:

```
List<Person*>::Impl();
```

Further information about templates, including how they can be derived from other templates, can be obtained from the book "TIMOR-An Object- and Component Oriented Language".

We have now shown how the Speedos persistent memory can be populated with capability protected information-hiding file modules which correspond to, and in effect also replace the need for, conventional files systems. Depending on the implementation selected these can produce similar results say to conventional sequential files, to indexed sequential

[10] **enq** and **op** are keywords which introduce instance methods. These are important for synchronisation purpose. An enquiry (**enq**) may not modify the state data, an operation (op) can change the state data.

files, to random access to B-tree files, etc. but also to multi-indexed files, etc. Furthermore, the maximum file size of 2^{42} bytes (or words, depending on the implementation, see Making Computers Secure vol. 2 chapter 23) makes Speedos/Timor suitable for managing big data. Finally both new type definitions and new implements can be added to the TCL.

3.4 Protecting Information in the Persistent Memory

One of the key features of Speedos is to provide secure mechanisms for its users. These include capability based protection (see section 1.3 above) and qualifiers (see section 1.3 above), which can qualify or modify the behaviour of the instances of other types.

3.4.1 Managing Speedos Access Rights

For this purpose Timor introduces a mechanism called *restrictors*, which list the names of methods of a module that can be called when the (information-hiding) file is accessed via the variable in question. The method list is contained in the brackets [: and :], e.g.

```
    TextFile[:insert, remove:]* tf;
```

For more details see chapter 15 of the book TIMOR-An Object- and Component Oriented Language.

3.4.2 Qualifiers

These are types which can modify the behaviour of other types (see section 1.5 above). How they achieve this is illustrated diagrammatically on both the Timor website

(https://www.timor-programming.org/qualifier-based-protection.html)

and on the Speedos website

(https://www.speedos-security.org/qualifier-based-protection.html).

Qualifying types are normal types which also have *bracket methods*. These are methods which can "catch" calls from one module (the client object) to another module (the target object) and can access and modify the parameters being passed between them. They can, for example, invalidate a capability being from the client object to the target. They can also access their own state data and thus for example examine an access control list to determine whether the call is permitted, and if not the can block the call, or make an entry in a log module.

As just described these are *call-in brackets* but qualifier modules can also contain *call-out* brackets which can examine and modify the parameters being passed out of a module, and can from their own state data or via calls to other modules determine whether the call-out may proceed. Such bracket methods can neutralise attempts by hackers to obtain information to which they are not entitled. How such qualifiers function is described in chapter 13 section 19 of Making Computers Secure volume 1 and how they can be implemented is described in chapter 24 of Making Computers Secure volume 2.

3.5 Further Timor Features

Timor has a number of other unusual features which are described in the book "TIMOR-An Object- and Component Oriented Language" which can be downloaded both from the Speedos website (https://www.speedos-security.org/) and from the Timor website (https://www.timor-programming.org/).

4 Conclusion

As Atkinson and Morrison [1] have convincingly demonstrated, the concept of persistence is significant in principle because it greatly simplifies programming, but without adequate hardware support it is difficult to implement. The Monads-PC system [17] was the first (and only) system which has effectively implemented such a hardware system in practice. However, the size of its virtual addresses, although revolutionary in the 1980s, has proved to be far too small for potential use over the Internet. Nevertheless the basic concept underlying the Monads-PC system was sound and Speedos, a successor system, has now been proposed and designed using the same basic concept (see the Speedos website https://www.speedos-security.org/). Speedos uses 256-bit addresses which are unique over all Speedos nodes on the internet, and are structured in such a way that such addresses provide clues to locating the objects and processes to which they refer. Speedos also has a technique which allows such long virtual addresses to be translated into main memory paged addresses as efficiently as normal pages addresses in RISC systems[11]. Currently no actual capability hardware exists, but Keedy has described elsewhere how existing RISC system hardware designs [18] could be easily adapted to function as capability systems, known as S-RISC, while still supporting programs previously developed as conventional RISC programs (after a recompilation). It has become impossible (as a result of the miniaturisation of chips) for normal software researchers to build complete hardware systems, so we must rely on Microsoft or Apple or Intel, etc. to build the hardware required to implement an S-RISC design. This would at last enable software designers to build a really secure system. The costs to industry[12] and to the war effort for Ukraine and Israel of not having really secure systems are immense. Solutions for the difficult problems in building a Speedos operating system are described in volume 2 of "Making Computers Secure" [10].

A different issue arises at the software level. In a genuinely persistent environment there is no place for a conventional file system! The Speedos design envisages that the virtual memory is populated by persistent information-hiding modules. This encourages good software design practice, but raises the question of what happens as a replacement for a file system. The answer given in this paper is twofold.

First, there is the fact that a persistent information-hiding module, as we envisage it, can be organised to implement semantic routines in information-hiding modules. These can reflect the various functions which users want/need in order to carry out their work. They can be protected by capabilities with separate access rights which are relevant to their different roles in an organisation.

Second, the programming of such modules can be carried out using the Timor programming language. This is a flexible way of organising data structures which opens up many possibilities for implementing the data. In particular the Timor Collection Library encourages a maximum of polymorphism while at the same time implementations of these collection types can be defined to provide the equivalents of conventional access methods, such as sequential access, indexed sequential access, random access, B-trees, multi-level indexing or any other access method, e.g. for managing big data.

Funding: This research did not receive any specific grant from funding agencies in the public, commercial, or not-for-profit sectors.

[11] see Making Computers Secure volume 2 chapter 23 section 1.

[12] see https://www.speedos-security.org/the-costs-of-existing-systems.html

References

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

[2] M. P. Atkinson, P. J. Bailey, K. J. Chisholm, W. P. Cockshott and R. Morrison, “PS-Algol: A Language for Persistent Programming,” in *Proceedings of the 10th Australian National Computer Conference*, Melbourne, Australia, 1983.

[3] R. Morrison, A. Brown, R. Carrick, R. Connor, A. Dearle and M. P. Atkinson, “The Napier Type System,” in *Proceedings of the 3rd International Workshop on Persistent Object Systems*, 1989.

[4] F. J. Corbato and V. A. Vyssotsky, “Introduction and Overview of the MULTICS System,” in *Proceedings of the 1965 Fall Joint Computer Conference*, 1965.

[5] E. I. Organick, The Multics System: An Examination of its Structure, Cambridge, Mass.: MIT Press, 1972.

[6] J. Rosenberg and D. A. Abramson, “MONADS-PC: A Capability Based Workstation to Support Software Engineering,” in *Proceedings of the 18th Hawaii International Conference on Systems Sciences*, 1985.

[7] J. Carter, A. Cox, D. Johnson and W. Zwaenepoel, “Distributed operating systems based on a protected global virtual address space,” in *Proceedings Third IEEE Workshop on Workstation Operating Systems*, Key Biscayne J.B. Carter, A.L., Cox, D.B., Johnson, W. Zwaenepoel FL, USA, 1992.

[8] D. A. Abramson and J. L. Keedy, “Implementing a Large Virtual Memory in a Distributed Computing System,” in *Proceedings of the 18th Hawaii International Conference on System Sciences*, 1985.

[9] G. Heiser, K. Elphinstone, S. Russell und G. Hellestrand, „A Distributed Single Address-Space Operating System Supporting Persistence,“ Sydney, march 1993.

[10] J. L. Keedy, Making Computer Secure, Speedos Website, https://www.speedos-security.org/, 2021.

[11] D. L. Parnas, “Information Distribution Aspects of Design Methodology,” in *Proceedings of the 5th World Computer Congress*, 1971.

[12] D. L. Parnas, “A Technique for Module Specification with Examples,” *Communications of the ACM,* vol. 15, no. 5, pp. 330-336, 1972.

[13] 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.

[14] J. L. Keedy, “Protecting and Confining Information with Speedos,” *Speedos Website,* 2024.

[15] J. L. Keedy, „Why Speedos executes Threads exclusively In-Process,“ Speedos Website (https://www.speedos-security.org/), 2024.

[16] G. Menger, Unterstützung für Objektsammlungen in statisch getypten objektorientierten Programmiersprachen, Dr. rer.nat thesis, University of Ulm Germany, 2000.

[17] J. Rosenberg and D. A. Abramson, “MONADS-PC: A Capability Based Workstation to Support Software Engineering,” *Proceedings of the 18th Hawaii International Conference on Systems Sciences,* pp. 515-522, 1985.

[18] J. L. Keedy, “S-RISC: Adding Security to RISC Computers,” SPEEDOS Website (https://www.speedos-security.org/), 2023.