

The Addition of Persistence to Ada95 and its Consequences

Michael Oudshoorn¹ and Stephen Crawley²

¹ Department of Computer Science

University of Adelaide, SA 5005, Australia

² Defence Science and Technology Organisation

PO Box 1500, Salisbury, SA 5108, Australia

Abstract. Research into persistent programming languages and systems in recent years has shown that the technology is useful for developing complex software in many problem domains. This paper explores the issues and consequences of adding persistence to Ada95. The persistence extensions support transparent migration of objects between a program's address space and a persistent store in a way that preserves both type safety and encapsulation of abstract data types.

Keywords and Phrases: Kinds of systems: persistent systems; Ada95; object-oriented technology.

1 Introduction

Ada has recently undergone an intensive and major review and update. The result of this is the new programming language Ada95[23] which is, in the main, backward compatible with Ada83[36]. Two of the principle changes to Ada were the addition of new constructs to change it from an object-based language to a fully fledged object-oriented language and the provision of annexes defining additional language capabilities for specific user communities.

Other recent developments in programming language technology were not incorporated into Ada95 in keeping with the requirement to minimise disruption. These include:

- hybrid type systems[25] and dynamic binding,
- reflection[26], and
- persistence[1].

While some of these technologies are immature and are yet to gain wider recognition, it is reasonable to anticipate changes before the next Ada language revision is due.

This paper examines one of these emerging technologies, namely persistence, and considers how it might be supported in Ada. We explore two possible revisions of the language and call them *persistent Ada95a* and *persistent Ada95b* to distinguish them from each other and from Ada83 and Ada95. The term *persistent Ada95* is used when discussing both variants of persistent Ada.

Our aim in undertaking this persistent extension is to follow, as far as practical and possible, the same revision guidelines adhered to by the Ada95 revision team[37, 38] and to restrict any suggested modifications to the Ada95 semantics to those that are absolutely essential for a persistent version of Ada. In this paper we examine the various forms of persistence which are available and the merits of each. We then focus on the addition of the most appropriate form of

persistence to Ada95 such that the core language is not disturbed; in particular we intend to preserve type safety and encapsulation of abstract data types (ADTs).

The intent of this research is to introduce persistent features to the language Ada95 without disrupting the core language unnecessarily. Interaction with the specialist annexes, such as the real-time annex, or the distributed annex, is not considered. In some cases, such as the use of real-time and persistence annexes, the mixture of the provided features is incompatible and undesirable.

This paper concerns itself mainly with language design issues. A more detailed discussion of the implementation issues can be found in [9, 10].

2 Persistence

Persistence is a general term for mechanisms that save values from a program's execution space so that they can be used in a later execution; i.e. by making the values "persist" from one execution to the next. In its broadest sense, the term covers conventional and object-oriented databases, execution checkpointing and data structure pickling as well as more sophisticated transparent mechanisms. The most sophisticated form of persistence is known as **orthogonal persistence**.

The idea of persistent programming appeared in 1980. Atkinson and Morrison took an existing simple programming language (S-Algol[28]) and extended it to support persistence. The result was PS-Algol[3], the first persistent programming language.

Atkinson *et al*[1] proposed the following two principles for languages that support persistent programming:

"The Principle of Persistence Independence: The persistence of a data object is independent of how the program manipulates that data object, and conversely a fragment of program is expressed independently of the persistence of data it manipulates. For example, it should be possible to call a procedure with its parameters sometimes objects with long term persistence, and at other times only transient."

and

"The Principle of Persistent Data Completeness: In line with the principle of persistence independence, all objects should be allowed the full range of persistence."

A persistence mechanism that follows the two principles above is said to support **orthogonal persistence**. To quote Dearle[13]:

"All data may be persistent and that data may be manipulated in a uniform manner regardless of the length of time it persists. In other words, the right for data to survive for a long (or short) time is independent of the data type."

In other words, in a system that supports orthogonal persistence, there is no conceptual discontinuity between the type and value spaces of an executing program and the persistent store.

There are many possible schemes for providing persistence support for applications systems. (For a comprehensive survey, the reader is referred to [2].) If we restrict ourselves to mechanisms that make program data structures persist (as distinct from saving the information in a different form), we can identify

four common approaches to persistence³. These are checkpointing, persistence by property, persistence by copying, and persistence by reachability.

In a **checkpointing** mechanism, the state of an executing program is saved to non-volatile storage. At some later time, the state can be restored, and the program can resume execution. Checkpointing mechanisms typically do not satisfy the Principle of Persistence Independence, because the persistent data can only be used in the context of a single long-running program execution. In most situations, this is sufficient to make checkpointing inappropriate.

With **persistence by property**, the persistence of data values is determined by some immutable property of the values or their types. In some cases, the decision is made at compile time; e.g. “all objects of a given type will persist”. In others, the decision is dynamic; e.g. “all objects created using a given storage allocator will persist”. As a general rule, persistence by property mechanisms do not satisfy the definition of orthogonal persistence.

With **persistence by copying**, a data structure is made to persist by traversing it and writing a copy to non-volatile storage with the structural information converted to a “flat” form. The saved data structure can be restored by reading the “flat” form of the data into memory and reconstructing the pointers. Persistence by copying schemes are often fully automatic (i.e. requiring no application code), and can preserve sharing and cycles within a single data structure. This approach is also known as data flattening or pickling.

The two main problems with persistence by copying mechanisms are that:

- they do not guarantee to preserve object identity in restored data structures, and
- they can be inefficient when applied to large data structures.

To illustrate the object identity problem, suppose that a program creates two records (*A* and *B*) and that both contain a pointer to another object (*C*). When *A* is saved, the flat form will contain a copy of both *A* and *C*. Similarly, saving *B* gives a flat form containing copies of *B* and *C*. If a later execution restores *A* and *B* and by unflattening the two flat forms, the resulting data structures will include two separate copies of *C*. In other words, the persistence by copying mechanism has failed to preserve the object identity of *C*, and therefore fails to meet the Principle of Persistence Independence.

With **persistence by reachability**, an object’s lifetime depends on whether or not it could be visible to a future program execution. The persistence system allows an application to designate one or more objects as **persistent roots** and provides it with the means to locate the persistent roots in future executions. If an object is “reachable” from a persistent root (e.g. by following pointers), it is therefore potentially to a future program execution, and it is therefore made persistent. The mechanisms for saving and restoring objects are typically transparent to the application program. Possibly the earliest example of persistence by reachability is described in [4].

Persistence by reachability is the mechanism most commonly used in persistent programming language; for example PS-Algol[1, 3] and Napier88[29]. These languages have a relatively simple semantic model for the process of saving persistent data. Persistent data is held in a non-volatile database known as a **persistent store**. At certain points during a program’s execution, the persistent store is **stabilised**. This finds all objects that are root reachable, and saves

³ This simple taxonomy conceals the fact that some real persistence mechanisms are a hybrid of the approaches listed.

a “snapshot” of their current state to the store in a single atomic operation.

Stabilisation occurs either when the program execution succeeds (i.e. when it completes without error) or when the program explicitly requests it. If a program execution fails, the persistent store reverts to the state at the last successful stabilisation point. The onus is on the programmer to ensure that the snapshot taken at each stabilisation point represents a logically consistent state for the persistent database⁴

The simple model of stabilisation above is often adequate, but it presents difficulties in some situations; e.g.

- when two or more independent program executions need to simultaneously use the same persistent store,
- when a multi-threaded program can only reach a globally consistent state by shutting down operation, and
- when a program needs to stabilise very frequently.

While the issues of stabilisation semantics are of fundamental importance to the future of orthogonal persistence, further discussion is beyond the scope of this paper.

Many persistent programming languages also provide a naming system for the persistent store to allow the programmer to organise and browse the stored values in a type-safe fashion. The naming system is typically implemented by providing a directory-like object as a built-in or library data type, with the convention that the persistent root is an object of this type. Such naming systems are not essential, but they clearly form an important part of existing persistent programming systems.

3 Persistence in Ada

The idea of supporting persistence in Ada is not a new one. Proposed and actual examples include Green’s proposed extensions to Ada[21], APPL/A[34], PGRAPHITE[40], the work of Millan and Mulatero[27] and Intermetrics’ proposed ODMG Ada95 binding[31].

Our analysis of the Ada requirements and standards documents [22, 35, 36, 37, 38] suggests that the Ada designers had four principle high-level design goals for the language:

- program reliability and maintenance,
- support for programming in the large,
- support for a wide range of application domains, and
- program execution efficiency.

In this paper, we focus on orthogonal persistence, which we believe most closely matches the four principle design goals for Ada. In particular, orthogonal persistence offers the highest level of reliability and maintainability of all forms of persistence, and supports the broadest range of non real-time applications. Orthogonal persistence is also well suited to software prototyping[20, 32], and to building systems where the ability to evolve is a major requirement.

⁴ In Napier88, the runtime system can initiate an involuntary stabilisation at any time. The problem of logical consistency of the resulting snapshots is finessed by including a checkpoint of the program’s execution state in the snapshot. However, this approach is not without problems.

Adding support for a new programming paradigm to a mature programming language is never a trivial matter. The remainder of this section discusses the main linguistic issues in supporting orthogonal persistence in Ada.

3.1 Type Checking and Type Equivalence

Orthogonal persistence requires that both values and types can have indefinite lifetimes. Furthermore, if a persistent application (program + data) is to evolve, structural equivalence and dynamic type checking are desirable, and indeed necessary, when binding to an object from the persistent store. To this end, existing persistent programming languages use structural type equivalence, so that type compatibility across different executions is a natural extension of compatibility within an execution.

The Ada type system uses name equivalence rather than structural equivalence. Furthermore, the language goes to some length to define the lifetime of a type, and use this as a basis for reclaiming objects. This causes difficulties for persistence in Ada.

```
procedure main is
  type my_type is
    record
      a: integer;
      b: real;
    end record;
  my_rec: my_type;
begin
  if first time the program is executed
  then
    my_rec := my_type(1, 2.0);
    create binding for my_rec with type my_type;
  else
    rebind to my_rec with type my_type;
  end if;
end main;
```

Fig. 1. Violation of type compatibility rules.

If we follow the current Ada type compatibility rules strictly, the example program in Figure 1 cannot work. Each time the program is executed, the declaration of “my_type” is elaborated to give a new type. When the program is executed for the second time, the instance of “my_type” will be different from the instance that was previously stored, and hence the attempt to rebind must fail.

It is clear that we cannot use strict name equivalence for persistent data. There are four main options for type equivalence of open (not private) data types in persistent Ada95:

- Use structural equivalence when binding to persistent data and name equivalence in other situations. This option is compatible with current Ada code, though it provides a “backdoor” that allows programs to circumvent Ada’s name equivalence rule.
- Require persistent types to be declared differently from ordinary types and use structural equivalence for them. This clearly violates the definition of orthogonal persistence. (In effect, this is the approach used in PGRAPHITE[40])

and by Millan *et al*[27].)

- Use a hybrid of structural equivalence and name equivalence in which two types are equivalent *iff* they have the same structure and are defined in analogous lexical contexts. This gives the desired effect in nearly all cases, but it is difficult to specify and to understand.
- Use structural equivalence throughout. This is the cleanest solution, but it is incompatible with Ada83 and Ada95.

Note that the above discussion does not cover compatibility of private types. This is addressed below.

3.2 Package State

Persistent programming languages such as PS-Algol[3] and Napier88[29] do not support any form of static variables. There are no global variables (in the conventional sense), and there is no explicit module construct to directly support visible or hidden variables.

In fact, components of a PS-Algol or Napier88 program can share variables in two ways (in addition to using normal argument passing). First, two program components can independently find any object reachable from the persistent root, and can agree to use such an object like a shared global variable. Second, a pair of functions declared in the same closure can communicate via variables declared in the closure. Since functions are first-class objects in both of these languages, the shared variables are effectively equivalent to hidden static variables. In both cases, the shared variables will live as long as required,

By contrast, an Ada package can declare static variables and constants, either making them visible in the package interface or hiding them in the package body. These variables and constants, along with the execution state of any embedded tasks, constitute the state of the package.

In standard Ada, the lifetime of a package's state is bounded by the execution of the program. By contrast, in persistent Ada, a package's state may need to have a lifetime longer than a program execution.

Figure 2 shows a “name_table” package that stores name strings in the “tab” array declared in the body, and issues “key” values that are actually indexes for the array. Suppose that some program uses this package to record some names, and saves the resulting “key” values in the persistent store. If a later program execution is to use the saved “key” values successfully, the package state including contents of “tab” must also persist.

One way to achieve persistence for package state is to require the programmer to designate the state variables and constants that should be saved between executions. This could be done either by labelling them using pragmas or by including explicit state saving and loading code in the package body. This approach has the following problems:

- It does not deal with state that logically belongs with a package but is actually contained in subsidiary modules or nested tasks.
- The state of dynamically instantiated generic packages is difficult to handle.
- The lifetime required for a package's state may depend on the reachability of the objects it manages; e.g. the “key” values above. This cannot be determined by the package.

A second approach is to allow the programmer to designate particular packages so that they persist between executions. The first time any program linked

```

package name_table is
  type name is string(1..20);
  function lookup (key: integer) return name;
  function insert (value: name) return key;
end name_table;

package body name_table is
  tab: array(1..100) of name;
  function lookup (key: integer) return name is ...;
  function insert (value: name) return key is ...;
begin
  tab := ...;
end name_table;

```

Fig. 2. State held in an encapsulated array.

to a designated persistent package is executed, the package is elaborated and the state variables and constants are saved to the persistent store. In subsequent executions of the program and any others that use the package, its state is restored by automatically binding the state variables to the corresponding persistent objects. Unfortunately, most of the problems with the previous approach also apply here.

A third approach is to adapt the semantic model for packages so that the collection of variables, constants and other information that constitutes a package's state form a notional **package value**. Package values are not normal Ada values; i.e. a program cannot denote their types, assign them, compare them or pass them as arguments. A program can only create them (by elaborating the package), make them root reachable, and statically or dynamically bind to them.

This last approach is used in both the persistent Ada95a and persistent Ada95b designs, albeit in a modified form in the latter case.

3.3 Maintaining Data Abstraction

Ada's private type mechanism makes type-safe persistence considerably more complex, especially if a program may be edited and recompiled between executions. The complexity arises out of the need to maintain the data encapsulation across executions for ADT's implemented using packages and private types.

The first problem arises from the fact that package state can form an integral part of an ADT. If a package has state (i.e. a notional package value) then the chances are that each elaboration of the package has a distinct **logical identity**. In other words, it is possible for a client to distinguish one elaboration from another by operations on its interface. In such a case, it is likely to be incorrect to create a private value with one instance of a package and use it in with a second.

For example, suppose that the "name_table" package in Figure 2 above is modified so that "key" values have a private type. If this package is elaborated twice, it would be incorrect to use a "key" value created by the "insert" function from one elaboration as an argument to the "lookup" function from the second. In general, any case where a private value is used in the context of the wrong package value is (strictly speaking) a violation of Ada package encapsulation. In many cases (such as this one) it also breaks data abstraction for the ADT's implemented by the package.

A second case where package encapsulation can be violated is when a package is modified and the program is rebuilt between creating a persistent value and using it. The problems that can arise include the following:

- If the public part of a package specification is changed, the package interface signature for a saved package elaboration may not match the signature expected by the program trying to use it. For example, required operations, variables or constants may be missing or have different types to what the program expects.
- Changes to a package specification or body may alter the full types of private types, so that saved private values are incompatible with the package's implementation.
- Changes to a package may alter the effective type of the notional package value, so that the package implementation cannot use previously saved package values.
- Changes to a package body may alter the package's invariants, so that it is no longer able to work with a previously saved package value.

In each case, allowing the use of a saved package value or private value with the modified package is technically a violation of package encapsulation and could lead to erroneous executions or incorrect behaviour that does not conform to the intended ADT semantics. Similar problems may arise when changes are made to other compilation units used by a package.

Any design for persistent Ada95 has to strike a balance between maintaining the data abstraction boundaries implicit in the programmer's use of packages, and allowing the programmer the flexibility to maintain and evolve the persistent system.

3.4 Handling Task Values and Function Pointer Values

Orthogonal persistence requires that task values and (in Ada95) access to subprogram values can persist. This presents some problems:

- The lifetime and accessibility rules for tasks and subprogram pointers in Ada83 and Ada95 limit their usefulness in a persistent context. However, removing these restrictions requires compiler technology that will allocate frames from the heap (if required), and automatically reclaim heap resident frames. While this kind of compiler technology is common in other programming languages, the idea of automatic garbage collection is not acceptable to many Ada users⁵.
- For a program to make use of a persistent task or subprogram, it must have access to the corresponding object code. Either the object code must be saved in the persistent store, or some other mechanism is needed to ensure that the correct version of the code is available.

⁵ For example, some people were adamant that the Ada95 language should not require support for automatic garbage collection since it is supposedly incompatible with real-time programming. Whether or not that is true, the wider Ada user community would have been better served by requiring automatic garbage collection support in core Ada95 and allowing real-time programs to disable garbage collection using a pragma.

3.5 Tagged records

A major innovation in Ada95 is the tagged record type provided to support object-oriented programming. Declaring a record type as `TAGGED` allows other record types to be defined as extensions of the tagged type. A tagged type declaration defines an actual type and a class-wide type. The latter is really a family of types consisting of the type and all extensions of the type. Class-wide types allow abstract operations to be dispatched to specific functions depending on the tag value of an argument or result.

In a non-persistent Ada95 program, the type hierarchy for a class-wide type is known at compile time. In persistent Ada95, we would expect the type hierarchy to evolve over time as the tagged type is extended in different ways. This presents us with problems, both in managing tag values, and in operation dispatching.

One problem is that the tag values for persistent tagged record types need to be consistent across the different program images and through time in the face of extensions to the tagged type hierarchy. The standard Ada95 language was designed so that tag values can be table indexes, and operation dispatching can be a simple table indirection. It is not obvious that we can be this efficient when dispatching operations based on persistent tag values.

A second problem is that a persistent Ada95 program might bind to a tagged record that has given tag, but not have the specific functions needed for that tag. We have identified a number of possible solutions to this problem:

- Do not allow objects to persist with a class-wide type. If programs always bind to a persistent tagged records using their actual types, the compiler can guarantee that the necessary operation code is present.
- Use a type equivalence rule for binding that ensures that the program can handle all possible tag values that the persistent data may contain. For example, the type family for the class-wide type in the program could be required to be a superset of the family in the program that made the type persistent.
- Extend the persistent binding checks to locate and check the tag information to ensure that all tags in the persistent value being bound can be dispatched in the program.
- Change the semantics of dispatching so that `CONSTRAINT_ERROR` is raised if the required operation is not available.

Of the possible solutions above, the second seems the most promising. Note that the approach suggested in [7] of treating records with unrecognised tags as members of a known superclass is not directly applicable to persistent Ada95, since it would break encapsulation of ADT's defined using tagged records.

3.6 Storage Management

Another important new feature in Ada95 is the `CONTROLLED` type mechanism for reclaiming dynamically allocated objects. While this is a vast improvement over Ada83, it does not address the storage reclamation problem as fully as automatic garbage collection. In particular, reclamation based on object finalisation cannot recover cyclic garbage without some extra help from the application to identify and break the cycles.

There are three alternatives for dynamic storage management in a persistent Ada95 design:

- Integrate persistence with Ada95 `CONTROLLED` types and `UNCHECKED_DEALLOCATE` and rely on the programmer to use finalisation for all storage reclamation. The biggest problem with this approach is that storage leaks in the persistent store will make it necessary to reinitialise the store, and rebuild the persistent data structures from scratch. This largely negates the benefits of using orthogonal persistence.
- Provide automatic garbage collection for the persistent store, but rely on the programmer for storage reclamation in the heap. It remains to be seen how difficult this will be for the application programmer.
- Require that persistent Ada95 implementations provide full automatic garbage collection of both the persistent store and the program’s local heap.

While Ada95 is designed to facilitate portable programs, it also provides constructs for non-portable, hardware specific programming tasks. Some of these constructs are problematical in a persistent Ada95. For example, it makes little sense for an object allocated at a fixed memory address to persist.

The main problem with the implementation dependent constructs in Ada is the potential they have for corrupting persistent data. For example, careless use of `UNCHECKED_ACCESS` and `UNCHECKED_CONVERSION` could damage persistent values that have been paged into the program’s address space. Furthermore, any code which leaves dangling or uninitialised pointers in root reachable objects could disrupt stabilisation, and cause junk to be written to the persistent store.

In the absence of an automatic garbage collector, storage management in persistent Ada95 will place a considerable burden on the programmer. Given the potential for destruction of persistent data, we believe that the arguments for “safe” programming and garbage collection are even stronger in persistent Ada95 than in conventional Ada.

4 Two Designs for Persistent Ada95

As mentioned in the introduction, we have tackled the problem of supporting persistence in Ada95 in two ways, giving rise to the two designs *persistent Ada95a* and *persistent Ada95b* respectively. Both designs provide transparent, reachability-based persistence following the model of stabilisation described in Section 2, and both assume a naming scheme along the lines suggested there.

The *persistent Ada95a* approach is to provide full support for orthogonal persistence at the cost of some changes to the core Ada95 language. The key points of the persistent Ada95a design are:

- The type system uses structural equivalence rather than name equivalence in all situations involving open (non-opaque) types.
- All Ada types (including tasks and access to subprogram types) are first-class types, and their values may persist depending on reachability.
- The type equivalence rules used when binding to persistent values are essentially the same as those used for static type checking.
- Persistent private values can only be used in a later execution if the package value for the package instance that created them also persists. Package values persist if the program explicitly makes them root reachable.

This places the following requirements on the implementation technology for persistent Ada95a:

- In some circumstances, frames for packages, functions, procedures and tasks must be allocated from the heap, and must to be able to persist.

- The correct versions of the object code modules for persistent packages, functions, procedures and tasks must be guaranteed to be available.
- Reclamation of dynamically allocated storage must be fully automatic. Unchecked deallocation, conversion and other low level operations that would interfere with garbage collection must be disallowed.

The resulting design is incompatible with Ada95. Furthermore, the implementation technology required is in advance of current generation Ada83 and Ada95 compilers. Thus, persistent Ada95a should be viewed more as a potential precursor to Ada0X than as an extension to Ada95.

The *persistent Ada95b* approach is to support persistence in a way that is compatible with the Ada95 standard, and that can be implemented using current generation Ada compiler technology. We achieve this by not striving for fully orthogonal persistence, and by relying on the good sense of the programmer in some areas rather than trying to enforce safety. The main points on which the design falls short of the ideal of orthogonal persistence are:

- Task and function pointer types, and any types constructed using them are not persistent.
- Different type compatibility rules apply for normal type checking and for binding to the persistent store.
- A programmer is expected to decide whether or not package instances should persist, and to explicitly bind to persistent package instances.
- The programmer is allowed to say that a given change to a package does not make it incompatible with pre-existing persistent values.

In each case, the short-fall is a consequence of the goal of compatibility with Ada95 and the secondary goal of avoiding “advanced” features that would make acceptance less likely.

We note that the addition of orthogonal persistence to mainstream languages is of considerable interest to the persistent programming community. Furthermore, the exercise of trying to add orthogonal persistence to Ada95 while maintaining compatibility with the base language has revealed some interesting things both about Ada and similar languages, and about persistence.

5 Persistent Ada95a: Modifying the Core Language

This section describes the *persistent Ada95a* design which supports persistence with modifications to the core Ada95 language. Key points for the design are:

- full orthogonal persistence should be supported, and
- the integrity of persistent data is paramount.

5.1 Type Checking

Having considered the various possibilities (see section 3.1), we have concluded that it is most appropriate to use structural type equivalence throughout persistent Ada95a for type checking open (non-private) types. This best matches the requirement for orthogonality of persistence, and gives a degree of flexibility to evolve programs and data involving open types.

The only drawback of using structural equivalence is that Ada subtypes would no longer be effective for expressing semantic distinctions between types with the same logical representations. There are possible solutions to this.

5.2 Packages and Private Types

We believe that there are only two viable options for supporting persistence of package state and private types in persistent Ada95a. The first option is to use the approach given in Section 3.2 of treating the result of a package elaboration as a notional package value. New dynamic binding constructs (see Section 5.3) allow a program to bind to existing persistent package values and to make new ones persist.

Package values and private values are tied together by making the identity of the parent package value a component of either the private type or of the private value itself. Dynamic type checks are performed at the appropriate point as follows:

- If package value identity is part of the private type, it only needs to be checked when a program binds to a persistent instance of the private type.
- If the package identity is part of the private value, it must be checked whenever a package body tries to view a private value via its full type.

While the latter involves more dynamic type checks, it offers more flexibility when private types are used in larger persistent data structures.

Thus, the type equivalence rule for private types is a form of name equivalence that incorporates package instance identity. In addition, substantive changes to a package specification or body must make the package type incompatible with previously saved package values and private values. While this may be a major impediment to system evolution, it is the only way that the language can guarantee that package encapsulation is maintained across program executions.

The second option is to undertake a more radical redesign of Ada's support for data abstraction. One way to look at the above is that we have gone part way towards turning package instances into objects. We could go a step further, replacing Ada packages and private types with an existentially quantified type construct along the lines described by Cardelli in [6]. Existentially quantified types provide a complete separation of types from their implementations, to the extent that a client of a type can transparently use multiple implementations while preserving type encapsulation. Languages that implement existentially quantified types include Russell[16], Napier88[29] and Quest[5].

Whichever option we adopt, the persistent Ada95a design is open to the criticism that it supports two distinct mechanisms for defining objects; i.e. package values / abstract types and Ada95 tagged records.

5.3 Persistent Binding

In Ada83 and Ada95, the WITH statement statically binds a compilation unit to the external units that it depends on. We have shown that package elaborations need to be made persistent, and that static binding to package elaborations is insufficient. Persistent Ada95a therefore needs to support dynamic binding to persistent package instances as well as to persistent data structures.

Binding to persistent values can be provided by a builtin generic BIND function with an interface as in Figure 3. A call to the BIND function establishes a binding with the persistent store, by returning the persistent value associated with a given name in the persistent store's name space. The BIND function would raise exceptions such as NAME_ERROR and TYPE_ERROR if the named object did not exist or had the wrong type.

```

generic
  type obj_ptr <>;
function bind (obj_name: in string) return obj_ptr;

```

Fig. 3. The generic function.

An example of the use of the bind function is shown in Figure 4. The body of the “appl” procedure can access the state object and anything reachable from it using normal Ada statements. In addition to BIND, functions are required to insert new name/value pairs into the persistent name space, update them and remove them.

```

with bindings;

procedure appl is
  type state_type is ...;
  type state_ptr is access state_type;
  function bind_state is new bind(state_ptr);
  p: constant state_ptr
    = bind_state("/users/steve/my_state");
begin
  ...
end appl;

```

Fig. 4. Using the bind function.

An alternative way to support binding is to define new Ada constructs that perform the same function. An Ada binding construct would be more readable, and would allow us to support bindings to L-values as well as to R-values. In Figure 5 for instance, “p” is declared as a variable, so that an assignment to “p” in the body would update a root reachable cell. Napier88 is an example of an existing persistent programming language that supports persistent L-value binding[12].

```

declare
  p: state_ptr binds to "/users/steve/my_state";
begin
  ...
end;

```

Fig. 5. Ada syntax extensions for object binding.

Binding to persistent package instances in persistent Ada95b requires syntax extensions⁶ to the Ada WITH statement as shown in Figure 6. When this code is compiled, the compiler finds the package specification for “my_pack” in the Ada library, and statically checks the usage of imported types and operations. The generated object code for “appl” would include a package interface signature for “my_pack”.

When the WITH DYNAMIC statement is elaborated, the type of the named persistent package value is checked to ensure that it conforms to the interface signature expected by the compilation unit. Each time the “appl” procedure is called, it tries to bind “xyz” to a persistent object. In this case, the type checker

⁶ Given that packages cannot be used as generic parameters, it is not feasible to support dynamic package binding using builtin functions.

must ensure that the package instance bound to “my_pack” in the WITH clause is the same one that created the private value that is bound to “xyz”. While the name string used in this example is a constant, it could equally have been a runtime computed value.

```
with dynamic my_pack from
  "/users/steve/my_pack_elaboration_1"
procedure appl is
  xyz: my_pack.some_private_type binds
    to "/users/steve/thingy";
begin
  ...
  my_pack.some_op(xyz, 42);
  ...
end appl;
```

Fig. 6. Ada syntax extensions for persistent package binding

When a persistent package instance makes use of subsidiary compilation units to hold its state or perform operations, these units must also persist. This happens naturally, since any state in the subsidiary units will be reachable via the package’s closure. In the case where the package uses WITH DYNAMIC to bind to the subsidiary unit, the associations established in that binding will persist.

Finally, we considered the possibility of making the persistence of package instances implicit by including a hidden link from each private value to the package value that created it. This would eliminate the need to explicitly make package values root reachable, and to explicitly bind to them. However, we were not convinced that we could make this scheme work.

5.4 Tasks

The Ada language supports both task types, and tasks as (almost) first-class values. It is therefore natural, and desirable, to make tasks persist in persistent Ada95a. (Indeed, if we do not allow persistent tasks, we would not be able to make package elaborations that contain tasks persistent.)

If we assume that tasks are truly first-class (i.e. if we remove the Ada restriction that prevents a task value from being passed outside of its declaration scope), the mechanics of making a task persist do not present any new problems. Roughly speaking, a task is saved by saving the task descriptor, dynamic and lexical contexts and links to the requisite executable code and recording the saved task in a global list in the store. When the persistent store is reactivated, the saved state of the tasks is restored, and the task descriptors are reconstructed and added to the relevant queues.

The linguistic difficulty with persistence of Ada tasks is in specifying the lifetime of a task. The visibility of a task to the current and future executions not simply determined by its reachability. A non-reachable task can affect an execution by updating data structures shared with other tasks, by attempting rendezvous or by interacting with the external environment. Furthermore, it is generally impossible to determine whether or not a task could be visible current and future executions.

This leaves us with two alternatives for the lifetime of persistent tasks. The first alternative is to make all non-terminated tasks persist indefinitely. This is

conceptually clean, but practice the cost of maintaining an arbitrary number of persistent tasks make this unworkable. The second alternative is to *define* that the lifetime of a task is determined by its reachability, and specify that any tasks that are found to be non-reachable will be aborted.

5.5 Subprogram Pointer Values

If we assume that access values to subprogram types are first-class values, there are no new linguistic problems involved in making them persistent. Saving a subprogram value is simply a matter of saving the frames that form its static context along with a reference to the relevant executable code.

5.6 Storage Management and Related Issues

The persistent Ada95a design requires automatic garbage collectors for both the persistent store and the program's local heap. This is in line with the design requirement that the integrity of persistent data is paramount. Furthermore, a garbage collected heap is a prerequisite for implementing first-class subprogram pointers and tasks.

A number of restrictions are made on the use of implementation dependent and UNCHECKED operations so that the garbage collector and the persistence mechanisms can operate safely. Unchecked conversions involving pointer types are forbidden, as are other type system loopholes that can be used to make bogus pointer values. Unchecked deallocations simply reset the pointer to NIL without attempting to deallocate the object. Finally, the compiler must ensure that all pointers are initialised to safe values.

6 Persistent Ada95b: the Pragma Based Approach

The second design for a persistent Ada95, which we call *persistent Ada95b*, is based on the following premises:

- The design makes no changes to the core Ada95 language. Instead, the persistence extensions are restricted to those appropriate to an Ada95 annex.
- The design does not support fully orthogonal persistence.
- The design can not guarantee the integrity of persistent data.
- The programmer is allowed some flexibility in maintaining ADT encapsulation for persistent data.
- The design should be implementable using current generation Ada95 compiler technology; i.e. heap garbage collection is not an absolute requirement.

6.1 Persistent Types

The basis of persistent Ada95 is the notion that types can have the property of **persistency**; i.e. the property that values of the type may potentially persist beyond a program execution. We follow the precedent of [30] and [39] and classify types as either **persistent types** or **non-persistent types** according to whether or not they have this property⁷. Note that the term **persistent type**

⁷ This terminology is somewhat misleading in that it seems to suggest that the types themselves are persistent objects. While this may in fact be true, this is not the intended meaning.

does not mean that all objects of that type *will* persist, but rather that objects of that type *may* persist depending on their reachability.

In persistent Ada95b, most types are persistent types by default. The exceptions to this are as follows:

- Ada95 task types and access to subprogram (function pointer) types are non-persistent.
- Ada allows a program to declare access (pointer) types for user defined storage pools (heaps). These types are non-persistent.
- Types defined with representation attributes or representation clauses are non-persistent.
- Private types declared in non-persistent packages are non-persistent. (Package persistency is discussed in a later section.)
- Types composed or derived from non-persistent types are also non-persistent.

Note that some of these restrictions could be removed in an implementation of persistent Ada95b that supported garbage collection, persistent code objects and first-class function values.

6.2 Storage Management

In keeping with our stated design aims, the persistent Ada95b design requires that the persistent store be garbage collected, but it does not require heap garbage collection. It remains to be seen how easy it will be for a programmer to manage the heap in this sort of environment.

Some restrictions must apply to the creation and use of access values in order to avoid potential damage to the persistent store:

- Root reachable objects must not contain uninitialised pointer values or pointers to objects that have been deallocated.
- Root reachable objects must not contain pointers that are not heap resident; e.g. pointers obtained using the `ACCESS` attribute.

In practice, it will be difficult to check these restrictions in a non-garbage collected implementation.

6.3 Persistent Binding

Persistent Ada95b provides pragmas for associating an Ada identifier with a value (or updatable cell) in the persistent store's name space. If an object declaration is followed by a `BIND` pragma, its meaning is altered from a declaration of a local object to a binding to a persistent object.

```
declare
  my_ident : integer := 0;
  pragma Bind(my_ident, "my_counter");
begin
  my_ident := my_ident + 1;
end;
```

Fig. 7. A persistent counter using L-value binding.

In Figure 7, the program establishes an L-value binding between the identifier “my_ident” and a cell in the persistent store with the name “my_counter” and type integer. If there is no cell named “my_counter” in the persistent name space, one will be created and initialised to the value given (zero). Operations

on “my_ident” in the BEGIN – END block will read and update the persistent variable in the same way as a normal variable. The only difference is that, updates made to “my_ident” will be saved to non-volatile storage if and when the persistent store is next stabilised.

The syntax for the persistent binding pragmas is given in Figure 8. The “<name>” is an Ada object identifier declared in the preceding object declaration, the “<name_expr>” is a String valued expression whose value is a name in the persistent namespace, and the optional “<space_expr>” is an expression which gives an alternative start point for the namespace lookup. The difference between the BIND and REBIND pragmas is that the former will insert a name into the persistent namespace if necessary, while the later expects the name to be present already. The initial value expression in an Ada object declaration is required when there is a BIND pragma and forbidden when there is a REBIND pragma.

```
pragma Bind(<name>, <name_expr>
           [, <space_expr>]);
pragma Rebind(<name>, <name_expr>
             [, <space_expr>]);
```

Fig. 8. The Bind and Rebind pragmas.

Establishing the persistent binding specified by a BIND pragma involves the following steps:

- 1) The persistent store’s namespace is searched for the name.
- 2) If the name cannot be found, the initial value expression is evaluated, and a new constant or variable is added to the persistent namespace with a name, type and initial value as given.
- 3) If the name is found, the initial value expression is not evaluated. Instead, the type of the existing persistent variable or constant is compared with the object declaration, and an exception is raised if the types are not binding compatible (see below).
- 4) The constancy of the name is checked, and if the program is trying to perform a variable binding for a persistent constant, an exception is raised⁸.
- 5) A binding is then established between the declared identifier and the persistent value or cell. This binding remains valid for the lifetime of the Ada identifier.

In the case of the Rebind pragma, step 2 above is replaced by the following:

- 2’) If the name cannot be found, an exception is raised.

We follow the example of Napier88[12] in supporting L-value bindings (bindings to cells) in persistent Ada95b as well as R-value bindings (bindings to values). This allows us to define persistent variables in a natural way. Without L-value binding we would need to simulate persistent variables using access values. Figure 9 shows how this would be done for the persistent counter example in Figure 7.

⁸ An L-value binding to a persistent R-value could be treated as an ordinary variable declaration where the variable is initialised to the value of the persistent constant. However, this would violate the “no surprises” principle since assignments to the Ada identifier would not persist, when it would appear to the programmer that they should.

```

declare
  type iptr is access integer;
  my_ident : constant iptr := new(0);
  pragma Bind(my_ident, "my_counter");
begin
  my_ident.all := my_ident.all + 1;
end;

```

Fig. 9. A persistent counter using R-value binding.

6.4 Type Equivalence

When a program binds to an object in a persistent store, the runtime system performs a dynamic type check to ensure that the binding is type-safe, and that it does not violate ADT encapsulation. It is clear that type checking of bindings cannot use the standard Ada95 type equivalence rules. Section 3.1 identified 4 possible type equivalence rules that could be used for persistent binding of open (not private) types. While the hybrid structural and name equivalence rule could be used, we have decided to use pure structural equivalence for persistent binding in persistent Ada95b because it is more consistent and easier to understand. Note that persistent binding is only allowed for persistent types: use of a BIND pragma with a declaration that has a non-persistent type should give a compile time error.

The type equivalence rules for persistent binding also need to ensure that data abstraction for private types that are used to model abstract types. These design rules are explained in the following section.

6.5 Packages

As we described earlier, maintaining data encapsulation of packages and private types across executions is a challenge for any persistent Ada95 dialect. The key problems are preserving package state between executions, ensuring that private types are not used with the wrong package instance, and dealing with evolution of a package's specification or implementation.

In persistent Ada95b, we do not attempt to make persistence of packages and private values fully orthogonal. Instead, we classify each package as one of **stateful**, **stateless** or **non-persistent** using pragmas and a heuristic default package persistency rule. This classification determines not only the way that package state is handled, but also the persistency of private types and the binding type compatibility rules used for them.

Stateless packages are those for which the state of a package instance does not need to be saved between runs. Persistent objects with a private type are assumed to be independent of the package instance that created them. A persistent binding involving a private type defined by a stateless package simply requires structural equivalence of the full type.

Stateful packages are those whose package state must persist. A persistent private value created by a given package instance can only be used if the state of the package instance persists. When a persistent binding involves a private type declared by a stateful package, the type equivalence rule ensures that persistent objects belonging to the private type can only be used in conjunction with the package instance that created them.

Non-persistent packages are those which have private state that cannot be made to persist; e.g. those with package level data structures with non-persistent types, or with embedded tasks. Private types declared by a non-persistent package are also non-persistent.

The persistency of a package can be determined from the package specification. The default rules are as follows:

- If the specification contains any state declarations with non-persistent types, the package defaults to non-persistent.
- If the specification contains state declarations that all have persistent types, the package defaults to stateful.
- If the specification contains no state declarations at all, the package defaults to stateless.

The default rules can be overridden by including a `PACKAGE_PERSISTENCY` pragma (see Figure 10) in the package specification.

```
pragma Package_Persistence
    ([stateful | stateless | nonpersistent]);
pragma Package_Bind(<package>, <name_expr>
    [, <space_expr>]);
pragma Package_Rebind(<package>, <name_expr>
    [, <space_expr>]);
```

Fig. 10. The Package Persistency and Binding pragmas.

A package body must be consistent with the persistency of its specification. If the declaration of a private type is deferred to the package body, the corresponding full type must be consistent with the persistency implied by the interface. Furthermore, if a package is stateful, its body must not declare any hidden state variables or constants with non-persistent type, and it must not include any embedded tasks, since both of these would make it impossible to save the state of a package instance.

The scheme for making instances of stateful packages persist and for using them are analogous to those used to make objects persist. Persistent Ada95a defines two pragmas `PACKAGE_BIND` and `PACKAGE_REBIND` (see Figure 10) that alter the meaning of a `WITH` statement or a package declaration to a persistent binding. The two forms are illustrated in the Figure 11.

```
with Table;
pragma Package_Rebind(Table, "my_table");
procedure Example is
    package Table2 is new Table;
    pragma Package_Bind(Table, "my_table2");
    ...
end Example;
```

Fig. 11. Use of the package binding pragmas.

The process of making a persistent package binding is similar to a binding to a persistent object. An attempt is made to locate the package instance named by the pragma in the persistent namespace. If the lookup fails, a package instance is elaborated in the normal way, and is inserted into the persistent namespace. If the lookup succeeds, the existing persistent instance is used.

When a binding is made to a persistent package instance, the runtime system checks that the package code in the executing program is compatible with the

persistent instance. The check will ensure that the program uses the correct types for the package state, and that the package code linked into the program image is **semantically compatible** with the package instance.

Semantic compatibility is determined by comparing semantic keys from the package code and the package instance. Each time a package is changed and recompiled, it is (by default) given a new semantic key. Normally (i.e. if the programmer does nothing), packages with different keys are semantically incompatible. However, the code library system associated with the persistent store will allow the programmer to assert that given pairs of keys are fully or upwards compatible, subject to the constraints of representational type compatibility.

Finally, a `TYPE_PERSISTENCY` pragma is provided to give the programmer finer control over the type persistency and binding compatibility rules. The programmer may specify that a given open type uses private type binding rules, that objects of a given private type are independent of a stateful package instance, or even that a given type is non-persistent.

7 Conclusions

In this paper we have presented two designs for persistent programming extensions to Ada95. We have shown that while full orthogonal persistence is possible, the resulting design is not directly compatible with Ada95. The alternative design which is fully Ada95 compatible does not support fully orthogonal persistent, and requires more care and attention from the application programmer.

The persistent Ada95 extensions will not be suitable for use in a hard real-time applications. The unpredictable nature of persistence is incompatible with the requirements for real-time programming, and it is likely to continue to be so for the foreseeable future. However, this is not a good argument for not supporting persistence as an Ada95 Annex. A real-time programmer should use a compiler that can “turn off” all support for persistence.

Assuming that code objects can persist, a persistent Ada environment can support incremental binding[14] and linguistic reflection using callable compilers[11, 33]. Incremental binding allows the programmer to build flexible, integrated tools from shared libraries of persistent software and data components; e.g. [15]. The incremental binding model is well suited to prototyping[8], and to building software systems with a strong requirement for evolution. Linguistic reflection enhances support for prototyping by allowing interactive programming, data visualisation[18] and application generator tools[17] to be built. It also supports advanced software construction techniques such as hyper-programming[19, 24], and is useful for building applications that operate at the meta-program or meta-data level.

We have identified a problem common to most examples where persistence support has been added to an existing programming language; namely that the persistence mechanism breaks the language’s data encapsulation schemes. In persistent Ada95, we address the problem by preserving the state of package instances, ensuring that private objects are used with the correct package instances, and ensuring that the package code used is representationally and semantically compatible (in the sense described above).

Evolution of systems with persistent data is a difficult problem in general, and for persistent programming systems in particular. In the persistent Ada95b

design, we provide ad-hoc support for evolution of packages, at the risk of programmer directed violation of data abstraction. Other possible alternatives include the use of reification for schema evolution, and the use of “projection” for evolution of ADT interfaces.

It is clear that adding orthogonal persistence to Ada is consistent with three of the four main Ada design principles. The only area of doubt is the fourth principle: that the language should support efficient execution. While current implementations of persistence are not very efficient, we expect this to improve as a result of current research.

At a more detailed level, adding persistence to Ada requires some changes that are contrary to the original Steelman requirements. In particular:

- Steelman specifically requires name equivalence of types, but pure name equivalence is impractical in a persistent language.
- Steelman requires (or is interpreted as requiring) static type checking, but practical persistent languages require that the binding of applications to persistent values uses dynamic type checking.
- Steelman requires that automatic garbage collection be an optional feature, but in a persistent Ada, automatic garbage collection is essential for at least the persistent store.

In each case, we believe that the Steelman requirements are largely a reflection of the state of the art in programming language design and implementation in the mid 1970’s.

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*, 26(4):360–365, 1983.
2. M.P. Atkinson and O.P. Buneman. Types and persistence in database programming languages. *ACM Computing Surveys*, 19, June 1987.
3. M.P. Atkinson, K.J. Chisholm, and W.P. Cockshott. PS-Algol: An Algol with a persistent heap. *ACM SIGPLAN Notices*, 17(7):24–31, July 1981.
4. M.P. Atkinson, K.J. Chisholm, and W.P. Cockshott. Algorithms for a persistent heap. *Software – Practice and Experience*, 13(3):259–272, March 1983.
5. L. Cardelli. Typeful programming. Technical Report SRC Report 45, DEC Systems Research Centre, Palo Alto, 1989.
6. L. Cardelli and P. Wegner. On understanding types, data abstraction and polymorphism. *ACM Computing Surveys*, 17(4):471–523, 1985.
7. C. Charlton, P. Leng, and M. Rivers. Using inheritance to provide schema views in a shared persistent object database. In *Proceedings TOOLS Europe ’94: Technology of Object-Oriented Languages and Systems, Versailles France*, March 1994.
8. S.C. Crawley. The ORBAT demonstrator: Using Napier88 for prototyping defence related software. Technical Report to appear, Defence Science and Technology Organisation, 1996.
9. S.C. Crawley and M.J. Oudshoorn. Orthogonal persistence and Ada. In *Proceedings TRI-Ada’94, Baltimore MD*, pages 298–308. ACM, November 1994.
10. S.C. Crawley and M.J. Oudshoorn. Persistence extensions to Ada95. In C. Mingins, R. Duke, and B. Meyer, editors, *Proceedings of TOOLS PACIFIC ’95, Melbourne Australia*, pages 25–39. Prentice Hall, November 1995.
11. Q.I. Cutts. *Delivering the Benefits of Persistence to System Construction and Execution*. PhD thesis, Department of Computational Science, 1993.
12. A. Dearle. Environments: a flexible binding mechanism to support system evolution. In *Proceedings of the 22nd Hawaii International Conference on Systems Sciences*, volume 2, January 1989.

13. A. Dearle. Use of orthogonal persistence technology in industrial and application oriented research and development. Technical Report PS-26, Department of Computer Science, University of Adelaide, August 1994.
14. A. Dearle, Q.I. Cutts, and R.C.H. Connor. Using persistence to support incremental system construction. *Microprocessors and Microsystems*, 17, 1993.
15. A. Dearle, M.J. Oudshoorn, and K. Wyrwas. An integrated approach to the generation of environments from formal specifications. *Australian Computer Science Communications*, 16:217–228, February 1994.
16. J. Donahue and A. Demers. Data types are values. *ACM Transactions on Programming Languages and Systems*, 7:426–445, July 1985.
17. D. Engelhardt. Tiffany. a persistent user interface management system for Napier88. Technical report, Department of Computer Science, University of Adelaide, 1992. Honours thesis.
18. A.M. Farkas. Aberdeen: A Browser allowing intERactive DEclarations and Expressions in Napier88. Technical report, Department of Computer Science, University of Adelaide, 1991. Honours thesis.
19. A.M. Farkas, A. Dearle, G.N.C. Kirby, Q.I. Cutts, R.C.H. Connor, and R. Morrison. Persistent program construction through browsing and user gesture with some typing. In *Proceedings 5th International Workshop on Persistent Object Systems, San Miniato, Italy*, September 1992.
20. R.P. Gabriel (Ed.). Draft report on requirements for a common prototyping system. *ACM SIGPLAN Notices*, 24:93–165, March 1989.
21. G. Green. Access values pointing to any type. *ACM Ada Letters*, 10:101–109, May-June 1990.
22. J.D. Ichbiah, J.C. Heliard, O. Roubine, J.G.P. Barnes, B. Krieg-Brueckner, and B.A. Wichmann. Rationale for the design of the ADA programming language. *ACM SIGPLAN Notices*, 14, June 1979.
23. ISO. *Ada95 Reference Manual*. International Standard ANSI/ISO/IEC-8652:1995, 1995.
24. G.N.C. Kirby, R.C.H. Connor, Q.I. Cutts, A. Dearle, A. M. Farkas, and R. Morrison. Persistent hyper programs. In *Proceedings 5th International Workshop on Persistent Object Systems, San Miniato, Italy*, September 1992.
25. B.H. Liskov, R. Atkinson, T. Bloom, E. Moss, J.C. Schaffert, R. Scheffler, and A. Synder. *CLU Reference Manual*, volume 114 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin Heidelberg, 1981.
26. P. Maes. Concepts and experiments in computational reflection. In *Proceedings of the International Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 147–155, 1987.
27. T. Millan and F. Mulaturo. Ada persistence through an OODBMS O2. *Ada User Journal*, 16:71–82, June 1995.
28. R. Morrison. S-Algol: a simple Algol. *BCS Computer Bulletin*, 2(31):17–20, 1982.
29. R. Morrison, A. Brown, R.C.H. Connor, and A. Dearle. The Napier88 reference manual. Technical Report PPRR-77-89, University of St Andrews, 1989.
30. J. Richardson, M. Carey, and D. Schuh. The design of the E programming language. *ACM Transactions on Programming Languages and Systems*, 15(3):494–534, July 1993.
31. M.T. Rowley. An OMDG Ada95 binding. Technical report, Intermetrics Inc, September 1995. <http://www.inmet.com/pob.html>.
32. J.T. Schwartz. Prototyping technology in the DARPA strategic software technology program. In *Proceedings of the International Conference on Computer Languages, Miami Beach, Florida*, October 1988.
33. D. Stemple, R.B. Stanton, T. Sheard, P. Philbrow, R. Morrison, G.N.C. Kirby, L. Fergaras, R.L. Cooper, R.C.H. Connor, M.P. Atkinson, and S. Algaic. Type-safe linguistic reflection. Technical Report CS/92/6, University of St Andrews, 1991.

34. S.M. Sutton, D. Heimbigner, and L.J. Osterweil. Language constructs for managing change in process-centered environments. *ACM SIGSOFT Software Engineering Notes*, 15:206–217, December 1990.
35. U.S. Department of Defense. *Requirements for High Order Computer Programming Languages: Steelman*. United States Department of Defence, June 1978.
36. U.S. Department of Defense. *The Programming Language Ada Reference Manual, ANSI/MIL-STD-1815A-1983*. United States Department of Defense, Washington, D.C., 1983.
37. U.S. Department of Defense. *Ada Board's Recommended Ada 9X Strategy*. Office of the Under Secretary for Defence Applications, Washington, D.C., 1988.
38. U.S. Department of Defense. *Ada 9X Requirements*. Office of the Under Secretary for Defence Applications, Washington, D.C., December 1990.
39. B. Walsh, P. Taylor, C. McHugh, M. Riveill, V. Cahill, and R. Balter. The Commandos supported programming languages. Technical Report TCD-CS-93-34, Trinity College, Dublin; Unite mixte BULL-IMAG, Trinity College Dublin, January 1993.
40. J.C. Wilden, A.L. Wolfe, C.D. Fisher, and P.L. Tarr. PGRAPHITE: An experiment in persistent object management. *ACM SIGSOFT Software Engineering News*, 13:130–142, November 1988.