QA SYSTEMS
The Software Quality Company

White Paper

Testing Techniques for Ada 95

The Ada language is widely accepted as the language of choice for the implementation of safety related systems and as a result, much effort has been put into the identification of successful techniques for its testing. In this paper we discuss the impact of the Ada standard upon the testability of safety related systems, and describe techniques which can be utilised to improve the likelihood of achieving testing success.

© Copyright QA Systems GmbH 2020

Contents

QA Systems' fundamental goals are to accelerate and improve software development. Operating on a global scale, QA Systems has over 350 blue-chip customers, spanning a range of industries, including aerospace & defence, automotive, healthcare and railways. The company supplies and supports its own tools, in addition to carefully selected products from strategic business partners, for static or dynamic testing, requirements engineering, architectural analysis and software metrics.

Copyright Notice

Subject to any existing rights of other parties, QA Systems GmbH is the owner of the copyright of this document. No part of this document may be copied, reproduced, stored in a retrieval system, disclosed to a third party or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the prior written permission of QA Systems GmbH.

© Copyright QA Systems GmbH 2016

1 Introduction

The Ada language is chosen for the implementation of safety related systems because it encourages good software engineering practices. Unfortunately, some of these practices make efficient testing difficult. As a consequence, many techniques have been developed which, when applied at the design stage, can improve the testability of Ada software. These techniques remain useful when testing Ada 95 due to the similarity between Ada 95 and the previous Ada standard. In this paper we discuss the impact of Ada 95 upon the testability of safety related systems, and describe techniques which can be utilised to improve the likelihood of achieving testing success. In particular, we concentrate on unit and small scale integration testing, as these are the areas likely to be affected by the use of the new language features. Throughout this paper, our aim is to emphasise **design for testability** as the primary means of achieving testing success.

This paper focuses its attention on three features of the Ada language: Protected Objects, Hierarchical Libraries and Controlled Types. We begin by considering the effect that the hierarchical library has on the testing process and find that it provides the single most significant increase in testability of all Ada 95 language features. We then consider protected objects because we expect them to be one of the first features of Ada 95 to be widely accepted in safety related applications. Finally we consider the significance of using controlled types and discuss the problems that they cause during testing.

© Copyright QA Systems GmbH 2020

2 Testing Techniques

In this section, we discuss each of our chosen Ada 95 language features in turn. We begin our discussions with a brief précis of each feature which is intended as a refresher for readers who are already familiar with Ada 95. Those with no Ada 95 experience should supplement this with a good Ada 95 textbook.

2.1 Hierarchical Libraries

There are several drawbacks to the package model of Ada 83, which become apparent when programming large or complex systems. For example, it is often desirable to use multiple library packages to model a complex abstraction, either for ease of implementation, or so that different aspects of the user interface can be encapsulated separately. In Ada 83, the only way to achieve this implementation is to put the details of the abstraction into the visible part of a package, thereby breaking the encapsulation. The alternative 'safe' approach results in a single monolithic package which is more difficult to use and to maintain.

A more common problem, which almost all Ada engineers have encountered at some time, is the inability to extend a package without recompiling all of its clients. This results in long and unnecessary periods of recompilation.

In Ada 95, these problems have been solved by the introduction of a hierarchical structure to the package model. In this revised model, child units can be declared which have visibility of the private part of their parent package. The simple example below is based upon that in Barnes [5] and illustrates the typical use of a child unit:

```
package Complex is
    type Complex is private;
    function "+"(X, Y : Complex) return Complex;
    function Real(C : Complex) return Float;
    function Imaginary(C : Complex) return Float;
    ...
private
    ...  -- Private type implementing Complex
end Complex;
package Complex.Polar is
    function R(C : Complex) return Float;
    function Theta(C : Complex) return Float;
    ...
end Complex.Polar;
```

In the example *Complex.Polar* is a child package of *Complex*, and as a result the operations *R* and *Theta* can be implemented efficiently using knowledge of the internal representation of the *Complex* type.

As the name implies it is not only packages which can be child units, in fact a child unit can also be a subprogram, generic or generic instantiation.

© Copyright QA Systems GmbH 2020

2.1.1 Using Hierarchical Library Units to Increase Testability

A problem which is present to some degree at all levels of testing is how to gain access to the information hidden inside a package. The traditional way of obtaining this access is to make use of the test point technique. In the simplest form of this technique, a procedure, known as the test point, is inserted into the package under test. This procedure serves no purpose in the normal implementation of the package and is usually given a null implementation. During testing, the null implementation is replaced with test code which consequently has access to the information hidden inside the package. Unfortunately, this approach suffers from two drawbacks: it is intrusive, and it is difficult to use in all but the simplest of configurations.

Hierarchical libraries provide the Ada 95 tester with an alternative to the test point approach. Unlike test points, the use of a child unit is completely non-intrusive and is extremely flexible. The example below illustrates how a child subprogram can be used to perform an isolation test of one operation of a simple stack abstraction:

```
package Integer_Stack is
    type Stack(Size : Natural) is limited private;
    procedure Push(S : access Stack;
                   E : in Integer);
    function  Pop (S : access Stack) return Integer;


private
    type Stack_Array is array (Natural range <>) of Integer;
    type Stack(Size : Natural) is record
        Depth : Natural := 0;
        Data  : Stack_Array(1 .. Size);
    end record;
end Integer_Stack;
-- Test implemented as a child subprogram
procedure Integer_Stack.Test is
    S : aliased Stack(10);  -- Declare a 10 element stack.
begin
    Push(S'access, 20);     -- Push is under test.
    Check(S.Depth,   1);    -- Checking operations provided by
    Check(S.Data(1), 20);   -- a test harness verify that the
                            -- Push operation was successful.
end Integer_Stack.Test;
```

Under many circumstances, child units are a suitable replacement for test points. However, we cannot eliminate test points entirely. Although child units give access to the private part of a package, the content of the package body remains hidden. In situations where there is information in the body that is required during testing, we must again rely upon the test point technique.

© Copyright QA Systems GmbH 2020

Careful consideration of the requirements of testing during the design phase makes it possible to minimise the need for test points by simply reorganising the implementation of an abstraction. The simplest technique is to move declarations from the body of a package into the private part of its specification, but this technique should be treated with caution for several reasons:

- Declarations that are promoted to the private part of a package are visible to both test and non-test children alike. This is often considered an advantage because it is difficult to know in advance whether an abstraction will need to be extended in the future, and if so which aspects of the implementation will be needed.
- A compile time dependency is introduced between the implementation of an abstraction and its clients. The independence of implementation and interface is one of the great strengths of the Ada package model and the introduction of new dependencies is something which we try to minimise.
- Many object-oriented mappings assign special significance to the private part of a package and consequently there may be conflicting constraints upon the placement of some declarations.

The addition of extra compile-time dependencies may be acceptable for a very stable abstraction, but as a general purpose technique this approach is inadequate. Fortunately there is a powerful alternative which provides full visibility to the implementation of an abstraction while actively reducing compile time dependencies. This technique relies upon the use of private child units which are discussed below.

While public child units are used to extend the interface of an abstraction, private child units can be used to decompose the implementation of an abstraction without exposing any additional functionality to its clients. To enable this, the visibility of a private child unit differs from that of a public child in two ways. A private child unit is totally private to its parent package. It is only accessible to its siblings, and even then only to their bodies; and the visible part of a private child has access to its parent's private part. The first rule is required to prevent a private child exporting any private information from its parent by, for example, using a rename.

Consider an alternative implementation of the simple stack shown above. In this example the test requires access to the package body data to verify that the *Push* operation was successful.

```
package Integer_Stack is
    procedure Push(E : in Integer);
    function  Pop return Integer;
end Integer_Stack;
package body Integer_Stack is
    type Stack_Type is array
          (Natural range 1 .. 100) of Integer;
    Stack_Data  : Stack_Type;
    Stack_Depth : Natural := 0;
    procedure Push(E : in Integer) is separate;
    function  Pop return Integer is separate;
end Integer_Stack;
```

One way of gaining visibility of the internals of this stack is to move the hidden data and type declarations into the private part of the package. However if the stack is changed, for example to increase its size, all of its clients must be recompiled. A better solution is to move the declarations into a private child package as shown below. This introduces no extra compilation dependencies yet renders the stack completely open to inspection during testing.

© Copyright QA Systems GmbH 2020

```
private package Integer_Stack.Impl is
    type Stack_Type is array
          (Natural range 1 .. 100) of Integer;
    Stack_Data  : Stack_Type;
    Stack_Depth : Natural;
end Integer_Stack.Impl;
with Integer_Stack.Impl;
package body Integer_Stack is
    procedure Push(E : in Integer) is separate;
    function  Pop return Integer is separate;
end Integer_Stack;
```

In this scenario the test for the stack abstraction is implemented as a public child of *Integer_Stack*, giving it unrestricted access to the declarations in the specification of the private child:

```
with Integer_Stack.Impl;
use  Integer_Stack.Impl;
procedure Integer_Stack.Test is
begin
    Push(10);
    Check(Stack_Data(1), 10);
    Check(Stack_Depth, 1);
end Integer_Stack.Test;
```

Abstractions that are designed in this way, using a private child package to declare package body data, are not only easier to test but are also more easily extended.

Besides their usefulness as a replacement for test points, hierarchical library units can also improve the testability of a system in other ways. In particular, there are many operations which are specific to testing that would benefit from increased visibility of an abstraction. A good candidate for this type of operation is a “check” procedure which can be used during testing to verify that an object has a particular value. Increased visibility of the internals of an abstraction allows this procedure to take advantage of the full view of a type to give improved diagnostics when the check fails. The great benefit of using child units to implement these utilities is that they can be made available during testing and removed when no longer needed, without affecting any other packages.

The example below illustrates the use of a child package to provide testing-specific operations for the *Complex* type. The *Is_Valid* operation can be used to ascertain that a *Complex* object satisfies its invariants, and may be implemented using the new Ada 95 *Valid* attribute.

```
package Complex.Test_Utilities is
    procedure Check(X, Y : in Complex);
    function  Is_Valid(X : Complex) return Boolean;
end Complex.Test_Utilities;
```

Careful consideration must be given before adding testing utilities to a package because they introduce an unwanted dependency between the test and the implementation details of the abstraction. This becomes a problem if the utilities are added on a per-test basis because any change to the representation of the abstraction requires them all to be updated. To avoid this, the utilities should be considered part of the design of the abstraction, and a few general purpose operations should be added rather than many test specific ones. As a general rule the dependencies of a test should be a subset of the dependencies of the software under test. This ensures that the test will only need modification when the software under test changes.

© Copyright QA Systems GmbH 2020

2.1.2 The Impact of the Hierarchical Library Upon Testing

We have shown that hierarchical library units are a valuable tool when implementing unit and integration level tests, because they allow us an unrestricted view of the private part of a package. Private child units also provide us with a simple idiom for structuring the body of a package so that declarations that are needed during testing are readily available.

In this section we consider the testing issues that arise when hierarchical library units are used, not for testing per se, but to improve the properties of a design.

As mentioned previously public child units can be used to partition different aspects of an abstraction, or to extend an existing abstraction with new functionality. We expect the latter to become common in Ada 95 due to the increased use of object-oriented techniques and the emphasis that they place upon programming by extension. In both of these cases the techniques that we have described in the previous section are sufficient to gain access to all of the information that is needed during testing.

The example below illustrates the typical use of a child package to segregate operations that are only of interest to a limited number of clients:

```
package Message is              -- The core message abstraction
    type Message is private;
    procedure Send_Message(M : in Message);
    ...                         -- Other operations
private
    type Message is ...;
end Message;
package Message.Tracking is -- Operations to track a message
    procedure Find_Message(M : in Message);
    ...                      -- Other operations
private
    ...
end Message.Tracking;
```

There are several ways in which this pair of packages can be tested. The first alternative is to write a combined test for both sets of operations, using a child package *of Message.Tracking* to gain visibility of both private parts. The obvious problem with this approach is that we lose the independence between the core abstraction and the tracking extensions; although we can use the message abstraction on its own we cannot test it without the tracking extensions. A better approach is to implement a stand-alone test for the core abstraction and then to test the tracking operations separately.

During testing it is important to remember that the operations of a child package have privileged access to the internals of their parent package. As a result it is possible for a child operation to break some of the existing operations of the parent package which were considered adequately tested. This means that an adequate test for the child package, as identified by Harrold [10], must re-test all of the operations of the core abstraction which may have broken. Without careful analysis it is difficult to tell which operations will need to be re-tested in the light of the new package.

© Copyright QA Systems GmbH 2020

Private child units will often be used to structure the implementation of a complex abstraction as a sub-system of packages. For example the *Message* abstraction shown above may be implemented using a queue of incoming and outgoing messages. The fact that the abstraction is implemented using a queue is of no significance to its clients, and so a private child package is used to hide the details. The private descendants used in implementing a sub-system should be subject to the same tests as other units in the system, and in practice private child units introduce no extra difficulty for the test engineer. The queue package from the *Message* sub-system is shown below, together with a child subprogram used to test it:

```
private package Message.Queue is
    procedure Push(M : in Message);
    function  Pop return Message;
    ...
end Message.Queue;
procedure Message.Queue.Test is
begin
    Push(...);
    Check(...);
end Message.Queue.Test;
```

Note that this child subprogram can be made the main subprogram of the test even though it is a private descendant of the *Message* package.

There is at least one other motivation for the implementation of a design using a set of hierarchical library units and that is to avoid polluting the library name-space. The structure of the Ada 95 predefined library hierarchy is in part motivated by this. The use of hierarchical library units for this purpose does not introduce any new testing considerations.

2.1.3 Recommendations

- Use a public child unit instead of a test point to gain visibility of the internals of an abstraction.
- Declare data and types that are local to the implementation of an abstraction in a private child instead of the package body. This makes the declarations available during testing, and also to extensions.
- At design time use a child package to implement utilities that will be useful during testing, e.g. check functions.
- Perform an integration test of the operations of a child unit with those of its parent. This ensures that the child unit has not abused its privileged visibility and detrimentally affected the operations of its parent.
- Avoid extending an abstraction using a child package when the visibility of a normal client will suffice. This will avoid unnecessary re-testing of the parent abstraction.

© Copyright QA Systems GmbH 2020

2.2 Protected Objects

Protected objects were added to Ada 95 to provide a passive and data oriented means of synchronising access to shared data. These objects can be considered similar to packages or tasks in that they are divided into a distinct specification and body. The specification of the protected object encapsulates the data that is being shared, and defines the interface used to access the data. The body of the protected object provides its implementation.

The interface to a protected object is defined by its protected operations, either procedures, functions or entries, which the language guarantees will have exclusive access to the object's data.

The example below uses the familiar concept of a quantity, or counting semaphore to illustrate the typical use of protected objects. For a detailed discussion of the implementation of semaphores and other building blocks using protected objects see [1].

```
protected Semaphore is
    procedure Signal;  -- V operation
    entry Wait;        -- P operation
    function Available return Boolean;
private
    Number_Allocated : Integer := 0;
end Semaphore;
```

Protected procedures and entries have read and write access to the shared data and consequently require exclusive access to the object. Protected functions have read-only access and therefore more than one function can be active simultaneously.

2.2.1 Testing Protected Objects

The Ada 95 Style Guide [6] recommends that the time spent inside protected operations is kept short so that the protected object is locked for as little time as possible. If this guideline is followed, the amount of code involved in the implementation of the protected object will be small. As a result the overhead of isolation testing each protected operation is unacceptably high, and we therefore recommend that the protected object as a whole is considered for isolation testing. Further motivation for testing at this level is the difficulty of isolating protected operations from each other without changing the protected object. The usual means of obtaining this independence is to use separate compilation, but for reasons of efficiency Ada 95 does not allow protected operations to be declared separate.

A problem encountered when testing protected objects, as with any software that encapsulates state, is that there will often be areas of functionality that are difficult to exercise in a test environment. In order to make this functionality testable it is useful to have access to the shared data inside the protected object so that we can manipulate it directly. This can be realised by adding operations to the object's interface which access each protected data item. This allows the protected object to be conveniently set into any state that is required during testing.

It is interesting to note that neither of the techniques described in section 2.1 can be used to improve the testability of protected objects: the child unit approach is ruled out because protected objects cannot have children; and the test point technique is not practical because of the lack of separate compilation.

© Copyright QA Systems GmbH 2020

The example below shows the semaphore example modified for testing:

```
protected Semaphore is
    -- Normal Interface
    procedure Signal;
    entry Wait;
    function  Available return Boolean;
    -- Testing Interface
    procedure Test_Set_Number(N : in Integer);
    function  Test_Get_Number return Integer;
private
    Number_Allocated : Integer := 0;
end Semaphore;
```

The problem with adding testing operations to the protected object is that there is nothing to distinguish them from the object's normal interface. This distinction is important because the testing interface must not be used by any normal clients of the object. One solution to this problem is to encapsulate the protected object inside the private part of a package. This allows the normal interface to be exported from the package while the testing interface remains hidden inside the private part

The example below shows the semaphore example modified to add a hidden testing interface:

```
package Semaphore is
    procedure Signal;
    procedure Wait;
    function Available return Boolean;
private
    protected Semaphore is
        -- Normal Interface
        procedure Signal;
        entry Wait;
        function Available return Boolean;
        -- Testing Interface
        procedure Test_Set_Number(N : in Integer);
        function  Test_Get_Number return Integer;
    private
        Number_Allocated : Integer := 0;
    end Semaphore;
end Semaphore;
```

To utilise the testing interface all tests are written inside a child unit of the *Semaphore* package, from where they have visibility to both the normal and testing interfaces:

```
    ...
    -- Test code at this point has access to both the testing
    -- and normal object interfaces
    ...
end Semaphore.Test;
```


© Copyright QA Systems GmbH 2020

The disadvantage of this structure is that all children of the *Semaphore* package have access to the testing operations of the protected object. An alternative encapsulation strategy is to declare the protected object inside the body of a package and use the test point technique [4] to give test code visibility to the testing operations. Testing is performed by replacing the normally null implementation of the test point with test code which can then freely utilise the testing operations.

```
package body Semaphore is
    protected Semaphore is
        -- Normal Interface
        procedure Signal;
        entry Wait;
        function Available return Boolean;
        -- Testing Interface
        procedure Test_Set_Number(N : in Integer);
        function  Test_Get_Number return Integer;
    private
        Number_Allocated : Integer := 0;
    end Semaphore;
    protected body Semaphore is separate;
    procedure Signal is separate;
    procedure Wait is separate;
    function Available return Boolean is separate;
    -- Testing Interface
    procedure Test_Point is separate;
end Semaphore;
```

It is important to note that there are circumstances in which a protected entry must be called directly, not via an interface subprogram. A good example of this is a protected entry which is the target of a timed or conditional entry call, or the trigger of an asynchronous transfer of control.

2.2.2 Testing Clients of Protected Objects

Protected objects can have a negative impact on the testability of parts of a system that use them unless care is taken to ensure that they can be sufficiently de-coupled from the rest of the system. When the client of a protected object is subject to isolation testing all calls to the protected operations must be simulated. This would normally be achieved by writing “stubs” for the called operations using a suitable test harness [11]. The efficient use of this technique relies upon the simulated operations being declared separate. As mentioned above, Ada 95 does not allow protected operations to be declared separate, so during testing the protected body as a whole must be simulated.

An attractive alternative is available if the protected object is encapsulated inside a package as described above. In this situation it is the interface subprograms of the package which, when declared separate, are simulated, rather than the protected operations themselves.

© Copyright QA Systems GmbH 2020

2.2.3 Recommendations

- Consider testing protected objects as a whole rather than isolation testing each operation.
- Define testing operations for each protected object, to allow direct manipulation of the object's state.
- If testing operations are added, shield them from the user by defining a testing interface.
- Consider using a testing interface to de-couple the protected object from its clients during isolation testing.
- Alternatively, declare the body of the protected object separate to allow it to be simulated.

2.3 Controlled Types

Regardless of how well an abstraction is designed and implemented, if it is not used correctly then it will not work. Probably the most common misuses of an abstraction are incorrect initialisation and inadequate clean up when it object goes out of scope. The Ada language exacerbates this problem by allowing a scope to be left in many different ways, for example by the propagation of an exception, an abort, a return statement, or an asynchronous transfer of control.

In Ada 95 the designer of an abstraction can ensure that these operations happen correctly under all circumstances by using Controlled Types. A controlled type is created by derivation from one of the predefined types in package *Ada.Finalization*, and from this type it inherits either two or three of the following subprograms:

- *Initialize*, which is called to provide default initialisation of a new object;
- *Finalize*, which is called to clean up an object when it is no longer needed;
- *Adjust*, which is called during assignment to modify Ada's default bit-wise copy. Note that adjustment is only available for non-limited types.

The default implementation of these subprograms can be overridden by the derived type, and is automatically called by the Ada runtime system when an object of the type is created, deleted or assigned.

The example below, which is taken from the Ada 95 Rationale [3], illustrates the use of a controlled type for the safe management of a resource:

© Copyright QA Systems GmbH 2020

```
package Handle is
    type Handle(Resource : access My_Resource)
              is limited private;
    ...    -- Operations on the Handle abstraction
private
    type Handle is new Ada.Finalization.Limited_Controlled
            with null record;
    procedure Initialize(H : in out Handle);
    procedure Finalize  (H : in out Handle);
end Handle;
package body Handle is
    procedure Initialize(H : in out Handle) is
    begin
        Lock(H.Resource);
    end Initialize;
    procedure Finalize(H : in out Handle) is
    begin
        Unlock(H.Resource);
    end Finalize;
...
end Handle;
```

In this example the creation of a *Handle* object causes the *Initialize* procedure to be called, which in turn locks the specified resource. When the object goes out of scope, no matter how the scope is left, the *Finalize* procedure guarantees that the resource is unlocked.

2.3.1 The Impact of Controlled Types Upon Testing

It is expected that many abstractions will take advantage of the benefits of controlled types to provide a cleaner, more reliable interface. When a controlled type is used for this purpose it is considered an implementation detail of the abstraction and its use is hidden from the abstraction's clients. This is of course advantageous because the abstraction can then be re-written to add or remove the controlled type without changing the remainder of the system. It is unfortunate then that the use of a controlled type often emerges during testing and must be considered part of the abstraction's interface.

To understand why this happens consider the way in which the simple abstraction shown below, which is built using the *Handle* abstraction, might be tested :

```
package Resource_Wrapper is
    type Resource_Wrapper is limited private;
    procedure Operation1(R : in Resource_Wrapper);
    ...
private
    The_Resource : aliased My_Resource;
    type Resource_Wrapper is record
        H : Handle(The_Resource'access);
        ...
    end record;
end Resource_Wrapper;
```


© Copyright QA Systems GmbH 2020

Assuming that the isolation testing strategy is used to test this abstraction, all calls to subprograms that are not part of the software under test must be simulated. This causes a problem because the controlled operations are implicitly called during the test whenever a *Resource_Wrapper* object is created or deleted. Like the other operations the controlled operations can be simulated, but doing so introduces a dependency between the test and the implementation of the abstraction. For example the test for *Resource_Wrapper* must change if the *Handle* abstraction is rewritten to avoid the use of controlled types. In practice the problem is made worse if the software under test is implemented using several different abstractions, all of which use controlled types. In this situation the test transitively depends upon the implementation of all of the abstractions.

The simulation of controlled operations and the dependency that this introduces can be avoided by using the operation's real implementation during testing. However, the integrity of the test can only be guaranteed if the controlled operations have already been thoroughly tested. A large part of the system may be involved in the verification of the controlled types, particularly when they have a non-trivial implementation, and many of the benefits of isolation testing are lost.

A solution to this problem is to provide a null implementation of the controlled operations during testing. This enables the test to remain independent of the controlled nature of the abstraction because the offending operations need not be simulated. In addition the implementation of each null operation is trivial and therefore does not need to be tested before it is used. It is important to note that this strategy will only work if the encapsulation provided by an abstraction is complete, in other words the software under test must not directly depend upon the actions of the controlled operations. The elided example below helps illustrate this point:

```
type T is new Ada.Finalization.Controlled with record
    X : Integer;
end record;
procedure Initialize(O : in out T) is
begin
    O.X := 1;  -- During testing this is replaced by null.
end Initialize;
function F return Integer is  -- Software under test.
    Object : T;               -- Null Initialize called.
begin
    return Object.X + 1; -- X has not been initialised.
end F;
```

A controlled abstraction, like any other abstraction, must at some point be tested. During testing the controlled operations are considered part of the software under test and are not simulated. However, the subprograms that are called by the controlled operations are simulated and this may cause difficulties.

The number of calls made to controlled operations will often be large; every time a controlled object is created, deleted or assigned a call is made. This places a significant burden upon the tester who must specify the order of these calls. In some situations it may also be difficult to identify the precise order in which the calls occur. This is due to the implementation permissions granted by the ARM for non-limited controlled types. A good example of the use of such permission is in the elimination of temporary objects during the assignment of controlled types.

In practice these problems only occur when a controlled operation results in a call to a subprogram that is outside the software under test. Fortunately most of the common uses of controlled types will not cause this problem.

© Copyright QA Systems GmbH 2020

2.3.2 Recommendations

- Do not simulate calls to controlled operations during testing; simulating these calls introduces a dependency between the test and the use of the controlled type.
- During testing, provide a null implementation of any controlled operations that are not part of the software under test. These operations can then safely be called as part of the test.
- Be aware that difficulties may arise when testing a controlled abstraction. In particular remember that the implementation permissions granted by the ARM may result in platform specific tests.

3 Summary and Conclusions

In this paper we have presented several techniques that can be used to improve the testability of software written in Ada 95. In particular we have focused our attention upon three features of the Ada language: the hierarchical library, protected objects, and controlled types. We have shown that the hierarchical library has the greatest impact upon the testing process because it provides a convenient method of accessing the information hidden inside an abstraction. The lack of visibility of this information was a significant obstacle to achieving testability when using Ada 83. We have also shown that the full benefits of the hierarchical library can only be realised if testing is considered at an early stage in the development process.

We followed our discussion of the hierarchical library by considering some of the difficulties encountered when testing software that uses protected objects; the effects of controlled types were also considered. Again, the techniques that we recommend emphasise that early consideration of the testing process is essential if testing is to be both thorough and efficient.

© Copyright QA Systems GmbH 2020