Making FACE™ Units of Conformance Fully Portable: Coding Guidance for Ada

The Open Group FACE™ Army TIM Paper by:

Benjamin M. Brosgol
AdaCore

September, 2021

Table of Contents

Executive Summary[1]

Source code portability of airborne software is the keystone of the Future Airborne Capability Environment (FACE™) approach and is realized through well-defined application program interfaces (APIs) and widely used industry standards such as IDL, POSIX, and ARINC-653. However, full portability of a Unit of Conformance (UoC) requires more than usage of common APIs and standards. It entails adhering to programming language-specific restrictions to ensure that the UoC has an equivalent effect when ported to a new platform. This paper, geared to UoC developers and project managers, offers guidance for Ada and its formally analyzable SPARK subset. It shows how to deal with vendor-specific or optional language features, how to avoid implementation dependencies, and how to manage target platform dependencies. By adopting the recommended stylistic conventions (most of which can be enforced by static analysis tools) developers of airborne software can use Ada or SPARK to achieve full portability for their FACE UoCs while also realizing the assurance benefits (extensive static checks that catch errors early) that these languages provide.

[1] This paper is an expanded and updated version of an article by the author that was published in March 2021 in *Military Embedded Systems* [1]

Introduction

The FACE approach to reducing life cycle costs is based on reusing software components across different platforms and airborne systems, and the FACE Technical Standard addresses this issue through a reference architecture and data model, well-defined interfaces, and widely used underlying industry standards (such as IDL, POSIX, and ARINC-653). Conformance with FACE requirements is a necessary condition for reuse and demonstrates an important aspect of software portability, but full source code portability means more than using a common set of interfaces. In order for a Unit of Conformance (UoC) in the Portable Components Segment (PCS) to be fully portable, it should have equivalent behavior across different platforms and/or compiler implementations. However, each of the programming languages called out in the FACE Technical Standard – C, C++, Ada, and Java – has features whose effect may depend on the compiler or run-time implementation, and hence may exhibit different behavior on different platforms. Writing a fully portable PCS UoC in any of these languages involves understanding and avoiding potential implementation dependencies. Where full portability is not possible, for example if there are intrinsic target dependencies, the software structure should encapsulate such dependencies.

Portability, or what will be called *functional portability* below to distinguish it from portability in the sense of FACE conformance, has been a goal of programming language design since the earliest days: abstracting away the properties of the underlying hardware and providing a high-level problem-oriented notation that is then compiled into processor-specific instructions. Ideally, functional portability means that a source program can be compiled and run on one platform and then, possibly with a different vendor's compiler toolset, the same program can be successfully compiled and run on either the same platform or a different one and have an equivalent[2] effect. In practice, however, several impediments may arise.

- The usage of language features that are either non-standard (i.e., unique to a specific compiler vendor), or else are standard but optional and not implemented by all compilers;
- The usage of standard language features with imprecisely defined semantics (behavior that depends on the implementation and in some cases may be completely undefined); and
- Dependence on characteristics of the target platform.

A non-portability may manifest itself in several ways. If a program uses a vendor-specific feature, then it may simply fail to compile on a different implementation. On the other hand, if it uses a standard feature whose effect is not precisely defined, then different compilers (or even the same compiler at different times) may generate executables that produce non-equivalent results. A non-portability may also arise from the use of different options for the same compiler on the same source code. For example, an aggressive level of

[2] "Equivalent" informally means that the program has the same external effects except for those resulting from permissible timing differences. A real-time program has a limited concept of which timing differences are permissible – i.e., some of its timing constraints are essential -- since missing a deadline may mean that the program fails to meet its requirements.

optimization may remove "busy wait" code that is present for timing purposes, and disabling run-time checks can result in a hard crash instead of a handled error.

These issues have been a fact of life since the earliest days of high-order languages. Indeed, the proliferation of incompatible and vendor-specific dialects for the languages used within the Department of Defense (DoD) in the 1970s (TACPOL, CMS-2, JOVIAL) was one of the factors that motivated the original development of Ada. Ada has strong advantages to FACE UoC developers in terms of software engineering support and program reliability, and source code functional portability has been a primary goal of the language throughout its history, but even with Ada there are features that have implementation dependencies. To maximize functional portability, an Ada UoC developer should follow a number of stylistic conventions.

This paper summarizes Ada functional portability guidance – how to avoid the impediments listed above – with a focus on features allowed by the Security and Safety-capability sets of the FACE Technical Standard, Edition 3.1 [2]. The guidance covers both Ada 95 [3] and Ada 2012 [4] and, where applicable, shows how the SPARK [5] Ada subset[3] can be used to mitigate potential non-portabilities. (Throughout this paper the language name "Ada" refers to both Ada 95 and Ada 2012 unless indicated otherwise.) This guidance is not an exhaustive list; additional information on Ada functional portability may be found in the Software Productivity Consortium's *Ada 95 Quality and Style* [6], and a complete set of implementation-defined characteristics is listed in Annex M of the Ada reference manual [3][4].

[3] The SPARK programming language is an extensive subset of Ada 2012, designed to support formal methods-based source code static analysis. Using the SPARK language and its associated toolset, the developer can verify a range of assurance properties with mathematical rigor; examples include correct information flow, absence of run-time errors, key integrity constraints, and even functional correctness with respect to formally specified requirements. Evidence from these verification activities can reduce the effort in demonstrating that software meets the objectives in assurance standards such as DO-178C [9]. Since SPARK is an Ada subset, a SPARK program can be compiled by any Ada compiler that implements the Ada 2012 standard. On a historical note, the initial version of SPARK was developed in the 1980s at the University of Southampton in the UK as a successor to the Pascal-based SPADE project, and the name "SPARK" originally stood for *SPADE Ada Kernel*. ("SPADE" is itself an acronym for *Southampton Program Analysis and Development Environment*). The SPARK language is currently maintained by AdaCore and Capgemini Engineering (formerly Altran).

Guidance for Vendor-Specific or Optional Language Features

Language extensions

In the interest of functional portability, the DoD's certification policy for Ada compilers has included a "no supersets" directive from the outset. That policy, however, has always recognized the utility of vendor-specific functionality as long as no new syntax is introduced, and thus allows certain kinds of language extensions:

- Implementation-defined libraries
- Implementation-defined pragmas
- Implementation-defined attributes[4]
- Implementation-defined arguments to pragma `Restrictions`
- Implementation-defined aspects (Ada 2012)

The FACE Safety-Extended and Safety-Base & Security capability sets impose a few restrictions in this area – for example, implementation-defined pragmas are not allowed for data structures composed from FACE interfaces – but do not otherwise restrict such language extensions. Indeed, further restrictions are not likely needed if a FACE UoC ported to a new platform will be compiled using the same vendor's technology. However, and this is the case for the UoC on both the original platform and the new one, the UoC developer will need to verify that the implementation of any subprogram from an implementation-defined library that is called from the UoC satisfies the restrictions in the relevant capability set. Evidence supplied by the library vendor can assist in this process.

If porting to a different compilation system is anticipated, then the use of implementation-defined language extensions should be minimized. Documentation supplied by the vendor, such as the GNAT Reference Manual [7], can be consulted to identify the relevant features. Ada 2012 has explicit support for enforcing the absence of implementation-defined extensions, through arguments to `pragma Restrictions` [3, §13.12.1]; for example, `No_Implementation_Pragmas` and `No_Implementation_Units`. In practice, vendors can also mitigate the issue by supporting implementation-defined features provided by other vendors, thereby easing the transition path.

Optional features

Another impediment to functional portability is to use a feature defined in the language standard and supported by the implementation on the original platform but not by the implementation on the new platform.

[4] An *attribute* is a property of a program entity denoted by the syntax *`Entity'AttributeName`* (e.g., `X'Length` is the number of elements in a one-dimensional array `X`).

The certification policy for Ada addresses this issue as well (the "no subsets" directive). Every Ada compiler has to implement the full language. Nevertheless, the revision process that led to the design of Ada 95 recognized that particular domains have specialized (and sometimes conflicting) requirements: for example, aborting a task at the earliest possible point is critical in a real-time program but might be impractical or infeasible on some mainframe operating systems. This realization led to the development of the Specialized Needs Annexes in the Ada 95 language standard, which are optional with respect to compiler certification. The certification policy for Ada 95 and later versions of the standard requires a compiler to implement the full "core" language, including the predefined environment (standard library) and the inter-language interfacing facilities, whereas the Specialized Needs Annexes (Systems Programming, Real-Time Systems, Distributed Systems, Numerics, Information Systems, Safety and Security) are optional.

In practice this optionality has not been an issue, since the annexes of relevance to most Ada users – Systems Programming and Real-Time Systems – are supported by the vendors in the Ada ecosystem. Moreover, the FACE Safety and Security capability sets prohibit the Distributed Systems, Numerics and Information Systems Annexes, so their optionality is not relevant to functional portability. Nevertheless, the Systems Programming and Real-Time Annexes raise a few issues that might affect FACE UoC developers:

- Some of the services defined in these annexes and permitted by the FACE Safety and Security capability sets are intrinsically system dependent (for example, interrupt handling) and thus will require revision on porting to a different execution environment. Designing the application to encapsulate such dependencies will ease the porting effort.
- The FACE Safety and Security capability sets significantly restrict the functionality supplied by these annexes. The UoC developer will need to demonstrate (through link-time tests, static analysis, and/or code review/inspection) that prohibited features in these annexes are not used. The Ada compiler itself can check for many of these, if the UoC includes[5] `pragma Profile(Ravenscar)`, `pragma Restrictions(No_IO)`, and `pragma Restrictions(No_Dependence => …)` for prohibited packages. A tool such as AdaCore's GNATcheck coding standard enforcer can provide a more complete check on the absence of prohibited features.

[5] `pragma Profile(Ravenscar)` was introduced in Ada 2005 and is available in Ada 2012. Its Ada 95 equivalent is a set of pragmas as specified in [3, §D.13].

Guidance for Features with Implementation-Dependent Semantics

Functional portability requires well defined semantics, so that the same program when compiled for different target platforms (or by different compilers on the same platform) will have an equivalent effect on each. In practice, however, there is sometimes a tradeoff between precisely defined semantics and efficient run-time performance. Since efficiency is typically a critical requirement for programmers, language standards[6] contain features whose effect may vary across different implementations. This section identifies several areas in Ada where FACE UoC developers may encounter such features, with guidance on how to mitigate the potential non-portability.

Order of evaluation in expressions

To facilitate optimization, Ada does not specify the order of evaluation of the terms comprising an arithmetic expression, but in some cases the result can be affected by the order that is chosen. As an example, suppose a function `F(X)` increments a global variable by `X` and then returns the resulting value. The expression `F(1)-F(1)` computes as `-1` if the `F(1)` on the left is evaluated first, and as `+1` if the `F(1)` on the right is evaluated first[7].

One way to mitigate this issue is to ensure that arithmetic expressions with multiple terms do not invoke functions with side effects, either directly or indirectly. Rather than writing the expression as `F(1)-F(1)` the programmer can express the logic deterministically as one of the following:

```
declare
   N1 : Integer := F(1);
   N2 : Integer := F(1);
begin
… N1-N2…
end;
```

```
declare
   N1 : Integer := F(1);
   N2 : Integer := F(1);
begin
… N2-N1…
end;
```

The potential non-portability of expression evaluation is eliminated completely in the SPARK Ada subset, since functions in SPARK are not allowed to have side effects and thus cannot assign to non-local variables. The value of an expression is the same, regardless of the compiler's choice of evaluation order.

[6] Java is often cited as a language with precise semantics, but the thread facility leaves many facets of the scheduling policy implementation dependent.

[7] This example assumes that the sum will not overflow, and that the global variable has been initialized.

Parameter passing

Formal parameters are defined in Ada by specifying the direction of data flow:

- "**`in`**", from the caller to the called subprogram[8];
- "**`out`**", from the called subprogram back to the caller when the subprogram returns; or
- "**`in out`**", from the caller to the called subprogram, and then from the called subprogram back to the caller when the subprogram returns.

For each parameter to a subprogram, the compiler chooses whether the parameter is passed *by copy* or *by reference*. For certain classes of types -- in particular, scalar types and access types ("pointers") – the semantics of parameter passing is *by copy*. For other classes of types (including tagged types, which are used in object-oriented programming) the semantics of parameter passing is *by reference*. But for types that do not fall into these categories, the compiler can choose either strategy, and it typically uses the type's object size as the determining factor: if the size of each object is smaller than some threshold value, then *by copy* is used, otherwise it will be *by reference*. For example, the compiler will typically choose *by copy* for a record (analogous to a C struct) that contains two Boolean values; and *by reference* for a record containing an array of several hundred integers.

The potential functional portability issue for such a type is that in some situations the effect of the program may depend on the compiler's choice. This can occur in cases of aliasing (a global variable is both passed as a parameter to, and assigned from, the called subprogram) or exception handling (a formal "**`out`**" or "**`in out`**" parameter is assigned from the called subprogram but an exception is propagated out of the subprogram before the subprogram returns).

These issues may arise in the context of the FACE Safety and Security capability set restrictions but can be mitigated in several ways.

For a type that may be implemented with either *by copy* or *by reference* parameter passing, the first issue can be avoided by ensuring that a global variable is not passed as a parameter to a subprogram that can assign to the variable. This can be detected by code review/inspection or static analysis and in fact is automatically detected by the SPARK toolset (which prohibits such aliasing) if the developer adheres to the SPARK Ada subset.

The second issue can be avoided by appropriate programming style: deferring any assignment to the formal parameter until after it can be assured that exception propagation will not occur. This can be arranged by declaring a local variable to hold the value that will be ultimately assigned to the formal parameter, updating this variable according to the subprogram logic, and then assigning it back to the formal parameter just before the return. The potential non-portability can be completely avoided with SPARK without the need for such a local variable, since the proof tools can demonstrate the absence of run-time exceptions.

[8] A *subprogram* in Ada corresponds to a function in C or C++ or a method in Java. A subprogram in Ada is either a *function* (which returns a value to the caller) or a *procedure* (which is invoked as a statement and is analogous to a void-returning function in C).

References to uninitialized variables

The Ada language does not require that all variables be initialized. Such a rule would be problematic, since in some cases there might not be a sensible initial value, and the program logic might require initialization to be supplied by an external input at system startup. Also, during the initial language design the analysis required to detect potential references to uninitialized variables was considered outside the scope of a compiler; a sound analysis that ensured no false negatives would generate too many false alarms. There was also the question of what would be required in terms of initialization enforcement for array elements.

That raises the possibility of a reference to a variable before it has been initialized. In the absence of a guaranteed value, the Ada semantics leave the effect of such a reference undefined. Ensuring that variables are assigned before being referenced is outside the scope of the restrictions in the FACE Safety and Security capability sets, and thus needs to be enforced through other means.

Several Ada language features can help:

- Some types do require a default initial value; in particular when an access value (pointer) is declared, it will be set to the special value **null** unless the declaration provides an explicit initialization. This avoids the error of using an uninitialized value as a pointer; an attempt to dereference the **null** value raises an exception.
- The programmer can define default initial values for record fields. Declaring a variable of such a type will create an object with well-defined values for these fields.
- In Ada 2012 any scalar type can define a default initial value.

Data flow analysis technology has made considerable progress since Ada's inception, and references to uninitialized variables for other types are detected in many instances by the Ada compiler, especially at higher optimization levels where sophisticated flow analysis is used. Static analysis tools such as AdaCore's CodePeer can also address this issue. And as with all the other potential non-portabilities discussed in this section, references to uninitialized variables are completely prevented in SPARK since they will be detected by the SPARK proof tools.

Concurrency

Ada has a powerful and high-level concurrency model, but in the interest of supporting a wide range of target processors and operating systems the language allows a number of scheduling policy decisions to be determined by the implementation. One example is the choice of which queue should be served when a server task providing multiple services (with callers queued on several) is ready. This nondeterminism is mitigated by the Ravenscar profile [8] [3, §D.13], a simple, deterministic, and efficient subset of the Ada tasking features. Both the FACE Safety-Extended and Safety-Base & Security capability sets restrict the Ada tasking facility to the Ravenscar subset and thus avoid the functional portability issues of the full tasking model.

The Ravenscar subset is supported by SPARK, and thus a SPARK program will avoid the concurrency nondeterminism of the full Ada tasking model.

An Ada UoC developer can also achieve portable concurrent programming through Ada bindings to the ARINC 653 APEX or POSIX pthreads APIs permitted by the targeted profile, instead of using Ada's Ravenscar features.

Elaboration order

An Ada program typically consists of a main subprogram together with the modules ("packages") that the main subprogram depends on, directly or indirectly. Program execution entails first executing any run-time code in the various dependent packages (for example to initialize global data) – a step known as "package elaboration" – and then invoking the main subprogram. The order in which the packages are elaborated is partially constrained by language semantics but in general is also implementation dependent, and different orders can yield different results, including the possibility of a run-time error known as "access before elaboration". (Implementation dependence is intrinsic to the language semantics, since any attempt to completely specify the elaboration order would end up prohibiting useful cases such as interdependent packages.)

Several techniques are available to avoid the potential non-portability:

- Add appropriate pragmas to fully constrain the elaboration order (or to partially constrain the order but where different choices all have the same effect).
- Avoid elaboration-time code in the dependent packages; move such code into procedures and invoke the procedures explicitly at the start of the main subprogram. Since no code is being run at elaboration time, the order of elaboration does not matter.

Elaboration order nondeterminism can also be avoided by using SPARK, since the SPARK restrictions prevent "access before elaboration" errors and prohibit constructs (such as one package making an elaboration-time assignment to a global variable in another package) that could make a program's effect sensitive to the chosen elaboration order.

Guidance for Target Dependencies

`System.*` package hierarchy and representation clauses

Although low-level programming involves accessing target-specific characteristics, the Ada standard helps to mitigate the non-portability through standard language features. The package `System` declares a type `Address` and associated operations, and the child packages `System.Storage_Elements` and `System.Address_To_Access_Conversions` offer standard facilities for dealing with "raw storage" and for treating a pointer as a physical address or *vice versa*. Representation clauses allow the program to define low-level properties of program entities, such as the layout of a record or the address of a variable. These features are permitted by the FACE Safety and Security capability sets. Although their usage is platform specific, encapsulating such code in the bodies of packages will localize and help minimize the adaptation needed when porting the code to a new target platform.

Numeric type representation

The predefined numeric types in Ada (`Integer`, `Float`, etc.) have implementation-defined ranges / precisions. Thus an `Integer` may be 16 bits on some target processors and 32 bits on others. This can cause functional portability issues if the programmer implicitly assumes that an `Integer` always has some minimum range; an arithmetic expression may overflow when the code is ported to a platform where `Integer` has a narrower range. (In Ada the overflow will raise an exception, which is arguably preferable to having the operation implicitly "wrap around" to a negative value.)

The potential non-portability can be avoided by declaring custom numeric types instead of using the predefined types. For example, a type representing values that are known to be in a specific range, say -1,000,000 through +1,000,000 inclusive, can be declared explicitly:

```
type Sensor_Range is range -1_000_000 .. 1_000_000;
```

The compiler will map this type to a twos-complement machine integer representation that covers at least the specified range. On a 16-bit machine this would correspond to the `Long_Integer` type; on a 32-bit machine it would be `Integer`. A program using the `Sensor_Range` type can be ported from a 32-bit machine to a 16-bit machine without changing the source code. The explicit declaration of a new type has the added benefit of protecting against data mismatches; for example an attempt to assign either an `Integer` or a `Long_Integer` to a `Sensor_Range` variable (or *vice versa*) without an explicit conversion would be compile-time error.

Conclusions

UoC full portability is key to realizing the benefits that the FACE approach offers, but it entails more than adhering to the interfaces allowed in the FACE standard and obeying the restrictions defined for the relevant capability set and profile. Writing fully portable code requires not only FACE conformance but also functional portability. That means following appropriate usage patterns, especially for features whose semantics are not completely defined by the language standard.

Ada is, in general, a language with strong support for functional portability, and over the years system modernizations have ported large Ada programs to new hardware and new compiler implementations with minimal problems. Nonetheless, functional portability does not come automatically, it must be planned for, and a variety of stylistic guidelines can help. A critical point is to be aware of features whose effect may be implementation dependent, and to use programming idioms that avoid the nondeterminism. This is especially important for applications that need to adhere to one of the FACE Safety and Security capability sets / profiles. Such applications have strong assurance requirements, which are difficult to demonstrate if the code uses language features that are not precisely defined. The SPARK subset of Ada is particularly relevant here, since the SPARK language restrictions ensure deterministic semantics.

In brief, adopting appropriate stylistic conventions for Ada (most of which can be enforced by static analysis tools) or using SPARK can help developers achieve full portability for their FACE UoCs while also realizing the assurance benefits that these languages bring.

References

(Please note that the links below are valid at the time of writing but cannot be guaranteed for the future.)

[1] B.Brosgol, "Making Software FACE™ Conformant and Fully Portable: Coding Guidance for Ada," in *Military Embedded Systems*, March 15, 2021
https://militaryembedded.com/avionics/software/making-software-face-conformant-and-fully-portable-coding-guidance-for-ada

[2] The Open Group, *FACE™ Technical Standard, Edition 3.1*.
https://publications.opengroup.org/standards/face/c207

[3] *Ada Reference Manual ISO/IEC 8652:1995(E) with Technical Corrigendum 1, Language and Standard Libraries*
https://www.adaic.org/resources/add_content/standards/95lrm/ARM_HTML/RM-TTL.html

[4] *Ada Reference Manual ISO/IEC 8652:2012(E) with Technical Corrigendum 1, Language and Standard Libraries*
http://www.ada-auth.org/standards/rm12_w_tc1/html/RM-TTL.html

[5] AdaCore and Altran; *SPARK 2014 Reference Manual*;
http://docs.adacore.com/spark2014-docs/html/lrm/

[6] C. Ausnit-Hood, K.A. Johnson, R.G. Pettit IV, S.B. Opdahl (Eds.), *Ada 95 Quality and Style*; Lecture Notes in Computer Science 1344, Springer; 1995.

[7] AdaCore, *GNAT Reference Manual*
http://docs.adacore.com/live/wave/gnat_rm/html/gnat_rm/gnat_rm.html

[8] B. Dobbing and A. Burns; *The Ravenscar Tasking Profile for High Integrity Real-Time Programs*.
http://www.sigada.org/conf/sigada2001/private/SIGAda2001-CDROM/SIGAda1998-Proceedings/dobbing.pdf

[9] RTCA/EUROCAE DO-178C/ED-12C – *Software Considerations in Airborne Systems and Equipment Certification*, December 2011

About the Author

Dr. Benjamin Brosgol is a member of AdaCore's senior technical staff. He has been involved with programming language design and implementation throughout his career, concentrating on languages and software engineering technologies for high-assurance systems. He was a member of the design team for Ada 95, and he also served in the Expert Group for the Real-Time Specification for Java (Java Specification Request JSR-001).

As one of AdaCore's representatives on the FACE Consortium, Ben is currently serving as Vice Chair of the Technical Working Group (TWG) and is an active member of the TWG's Operating Systems Subcommittee as well as the Enterprise Architecture EA-25 subteam. He has written papers and delivered talks at several FACE TIMs, and his articles on technical aspects of the FACE approach have appeared in professional publications for the defense industry.

Ben has presented papers and tutorials on high-assurance software topics at numerous conferences and workshops including ESC (Embedded Systems Conference), ICSE (IEEE/ACM International Conference on Software Engineering), IEEE Security Development (SecDev), STC (Software Technology Conference), and conferences run by the ACM Special Interest Group on Ada (SIGAda) and Ada-Europe.

He holds a BA in Mathematics (with honors) from Amherst College, and MS and PhD degrees in Applied Mathematics from Harvard University.

About The Open Group FACE™ Consortium

The Open Group Future Airborne Capability Environment™ Consortium (the FACE™ Consortium), was formed as a government and industry partnership to define an open avionics environment for all military airborne platform types. Today, it is an aviation-focused professional group made up of industry suppliers, customers, academia, and users. The FACE Consortium provides a vendor-neutral forum for industry and government to work together to develop and consolidate the open standards, best practices, guidance documents, and business strategy necessary for acquisition of affordable software systems that promote innovation and rapid integration of portable capabilities across global defense programs.

Further information on the FACE Consortium is available at www.opengroup.org/face.

About The Open Group

The Open Group is a global consortium that enables the achievement of business objectives through technology standards. Our diverse membership of more than 800 organizations includes customers, systems and solutions suppliers, tools vendors, integrators, academics, and consultants across multiple industries.

The mission of The Open Group is to drive the creation of Boundaryless Information Flow™ achieved by:

- Working with customers to capture, understand, and address current and emerging requirements, establish policies, and share best practices
- Working with suppliers, consortia, and standards bodies to develop consensus and facilitate interoperability, to evolve and integrate specifications and open source technologies
- Offering a comprehensive set of services to enhance the operational efficiency of consortia
- Developing and operating the industry's premier certification service and encouraging procurement of certified products

Further information on The Open Group is available at www.opengroup.org.