



On the Principled Design of Object-Oriented Programming Languages for High-Integrity Systems

Roderick Chapman, Janet Barnes, and Brian Dobbing

Publication notes

Submitted as a position paper to the 2nd NASA/FAA Object Oriented Technology in Aviation Workshop.

On the Principled Design of Object Oriented Programming Languages for High Integrity Systems

Janet Barnes, Brian Dobbing, Rod Chapman
Praxis Critical Systems Limited.

rod.chapman@praxis-cs.co.uk

(c) 2002 Praxis Critical Systems Limited

"Design principles? Hmm...yes...you should definitely have some!"
Guy L Steele Jr., ACM SIGPLAN PLDI conference, 1994

Introduction

Systems for which failure can cause loss of life, injury, environmental damage, or financial loss are known as high integrity systems. These are all systems for which the cost of failure is not tolerable or affordable.

As high integrity systems become more prevalent in our every day lives the number of people involved in the production of these systems has increased, bringing popular technologies from the mainstream software community to the world of high integrity systems. Object-Oriented Technology (OOT) is seen by many as the current “silver bullet” of software development; it is popular in the software community at large and benefits from a wide range of tool support. In looking to embrace this popular technology within the high integrity sector it is crucial that we ensure that the underlying design principles for high integrity systems are not compromised.

A key aspect in certification of a high integrity system is the ability to validate the correctness of the system before it enters into service. A common approach to such validation is to use extensive dynamic testing after the initial development is complete, to the point whereby sufficient confidence has been gained in the correct operation of the system. Unfortunately such techniques can be expensive as they find faults late in the development lifecycle, and the number of tests required to gain the necessary assurance that all requirements have been met in a representative sample of operational contexts, and that full code coverage has been achieved, is considerable. For the most critical systems, it has been argued that testing alone can never be adequate to develop the level of confidence required[12].

An alternative approach is to develop the software in a language that can be analysed statically using tool support, allowing the software to be rigorously validated as the design and code is produced. This approach is particularly beneficial when applied during the design stage, when interface specifications are being constructed, since the ensuing implementation can be verified to comply with the design at the point of construction.

The SPARK language is designed primarily for the comprehensive application of static analysis. The depth of the analyses achievable ranges from data-flow and information-flow analysis, through proof of absence of run-time exceptions, to formal verification of correct functionality of the code against a specification. This paper looks at the underlying design principles of SPARK, a language with its roots in the high integrity domain, and considers the aspects of OOT that have been adopted without compromising those principles.

The need for static analysis

Static analysis is recognised as a valuable mechanism for verifying software. It is mandated for safety critical applications that are certified to the UK Defence Standard 00-55 [5]. Industrial experience shows that the use of static analysis during development eliminates classes of errors that can be hard to find during testing[11]. Moreover, these errors can be eliminated by the developer before the code has been compiled or entered into the configuration management system, saving the cost of repeated code review and testing which results from faults that are discovered during dynamic testing.

Static analysis as a technology has a fundamental advantage over dynamic testing. If a program property is shown to hold using static analysis, then the property is guaranteed for all scenarios. Testing, on the other hand, may demonstrate the presence of an error, but the correct execution of a test only indicates that the program behaves correctly for the specific set of inputs provided by the test, and within the specific context that the test harness sets up. For all but the simplest systems, exhaustive testing of all possible combinations of input values and program contexts is infeasible. Typically, test cases are devised to represent broad classes of inputs, so that tests can be created that use a representative value from each possible input class. However complex program state contexts are usually only creatable during integration and system testing, when it may be very difficult to simulate all possible operational states. Further, the impact of correcting errors that are found only at this stage of the lifecycle is generally massive in comparison to those found during development.

There are many methods of static analysis. By using combinations of these methods, a variety of properties can be guaranteed of a program. The following list of forms of analysis is drawn from a study of a variety of standards that is presented in an ISO Technical Report [3].

Control Flow

Control flow analysis ensures that code is well structured, according to some criteria such as flow-graph reducibility.

Data Flow

Data flow analysis ensures that there is no path through the program that would result in access to a variable that does not have a defined value.

Information Flow or "Data Coupling"

Information flow analysis is concerned with the dependencies between inputs and outputs within the code. It checks the specified dependencies against the implemented dependencies to ensure consistency. To be effective, information flow analysis needs to be performed with knowledge of the system requirements. It can be a powerful tool for demonstrating properties such as non-interference between critical and non-critical data.

Symbolic Execution

Symbolic execution generates a model of the function of the software in terms of parallel assignments of expressions to outputs for each possible path through the code. This can be used to verify the code without the need for a formal specification.

Formal Code Verification

Formal code verification is the process of proving the code is correct against a formal specification of its requirements. Program units are verified separately. Each operation is specified in terms of the preconditions that need to be satisfied for the operation to be callable, and the post conditions that hold following a successful call to the operation. The verification process demonstrates that, given the pre-conditions, execution of the operation always gives rise to the post-conditions. The level of proof depends on the information provided in the formal specification. This can vary depending on the aspects of the code that need to be verified; this can vary from the proof of a single invariant right up to full functional behaviour[13].

Proof of absence of run time errors is a special form of formal code verification. This does not require the provision of a formal specification of the program. Instead, formal code verification techniques are used to demonstrate that at every point in the code where a run-time error may occur, (for instance numeric overflow), the pre-conditions on execution of that code and the current set of data values in the expression guarantee that the run-time error will not occur. This is a very valuable property to be able to demonstrate, especially in systems where the occurrence of an unexpected run-time exception is generally unrecoverable, and the overhead of dynamic defensive mechanisms for preventing all such faults is unacceptable.

Language Design for Static Analysis

When designing a language that allows static analysis to be performed, clearly the language must support most if not all of the above static analysis techniques. In addition, in order to apply these techniques in practice both efficiently and early in the software development life-cycle, the following properties are required.

Tool checkable

All rules of the language must be tool checkable in a reasonable time. If some of the language rules cannot be checked by a tool then there will be no clear guarantee that the rules are satisfied by a program - this might mean that the program cannot be shown to exhibit the desired properties. For example, there are a few rules of MISRA-C, the C

subset defined for use by the automotive industry [6], that are very difficult to validate using efficient analysis algorithms.

Individual components amenable to analysis

It should be possible to perform analysis on small and possibly incomplete components of a system so that static analysis can be performed incrementally as the system is developed. If a program can only be statically analysed once it is fully developed then this is too late in the lifecycle to obtain the major benefits of the techniques, such as the correct construction of the design and implementation to prevent errors at source. Addition of new subsystems should not be able to "break" the analysis of other, finished subsystems.

The SPARK Language

History

SPARK was originally defined in 1988 by Bernard Carré and Trevor Jennings [1]. It is a mechanically verifiable language that was designed to be amenable to static analysis. There were a number of original goals of the language - these were achieved by careful choice of the language subset and the use of annotations to provide the additional information necessary to improve language security. Over the years the SPARK language has grown, initially with the transition from Ada83 to Ada95 as the language upon which it is based, and latterly to extend the language features that can be offered without compromising the original language goals. The latest extensions support statically analysable object-oriented features of Ada95[9].

Language goals

The original goal of the SPARK language was to develop a language that supports the static analysis techniques listed above and that allows practical exploitation of these techniques within software development for industrial high integrity systems. The pillars on which the language was built are as follows.

Logical Soundness

The language shall be unambiguous. All language features that allow the construction of programs of non-determinable meaning are eliminated. As a result, the behaviour of a SPARK program is wholly determined from the source code, with no implementation dependencies (such as the evaluation order of expressions or parameter passing mechanism).

Simplicity of Formal Language Definition

The language semantics shall be conducive to the production of a formal language definition. The formal semantics of SPARK was produced in 1994 [2].

Expressive Power

The language shall be suitable for the construction of industrial high integrity software. Whilst the exclusion of non-deterministic constructs was essential, it was important not to oversimplify the language to the point that it was not suitable for industrial use. SPARK

retains all the important features of Ada for constructing well-engineered code for real-time systems.

Security

The language shall be secure. For a language to be secure it must be possible to ensure that any program conforms to all of its language rules statically (without needing to apply language rules to the dynamic execution of the program). All the rules of SPARK are machine-checkable in polynomial time, and are designed to be decidable - i.e. the language is designed so that the question "Is this program legal SPARK?" always has a simple "Yes/No" answer. This contrasts starkly with the more common heuristic-based forms of static analyses performed on whole (i.e. unsubsetted) languages, where "Don't know" answers can prevail.

Verifiability

The language shall support formal verification of the correctness of programs using rigorous mathematical analysis. For this to be feasible for large applications it must be possible to verify fragments of the code. The SPARK annotations introduce sufficient contextual information to allow verification of code sections with reference only to the specifications of the packages that are used. This allows effective analysis of large systems in an incremental manner.

Bounded Space and Time Requirements

The language shall imply statically-bounded space and time constraints. In high integrity applications, it is essential that the necessary memory requirements of the program should not exceed the available resources. SPARK programs are inherently bounded in space as there is no recursion or dynamic memory allocation. All constraints on a SPARK program are static, which leads to deterministic execution times.

Further goals

The fundamental goals above were supplemented with further goals that relate to the practicality of deploying SPARK in industrial high integrity systems:

Correspondence with Ada

All SPARK programs shall be legal Ada programs. There are major benefits from sharing technology and general resources with an existing standard language. Moreover the logical soundness property means that a SPARK program will always execute in the same way regardless of the compiler implementation choices allowed by freedoms in the Ada language standard.

Verifiability of Compiled Code

The object code for a SPARK program should implement the same semantics as the source code. Since all the verification is performed on the source code, it is important that the correspondence between the source code and the object code is accurate, otherwise the results acquired through analysis of the source code cannot be applied with any confidence to the object code. Clearly this relies on the compiler being free from errors in its translation of SPARK constructs. Although SPARK cannot guarantee this, there is some evidence that the simplifications inherent in the Ada subset defined in

SPARK result in only well-validated areas of the compiler being exercised. Further, the generated object code tends to be relatively easy to relate back to the source code.

Minimal Run-Time Library Requirements

The language shall imply minimal use of a run-time library. The full Ada language supports multiple complex features, such as concurrency and exception handling, and hence needs large run-time library support. Since the run-time library becomes part of the deployed system, it needs to be demonstrably correct. SPARK has been deployed in a system that requires no run-time library support whatsoever [7]. Many Ada compiler vendors provide small run-time libraries that have supporting evidence for certification purposes. SPARK is compatible with all of these COTS products.

Object-Oriented Technology in SPARK

The concept of a *class* as being a specification of a generic state definition plus a set of operations on that state, and the concept of an *object* as being an instance of a class, are at the very heart of OOT. The popularity of this approach is based on its suitability to model the real world much more directly than before, and it has spawned a plethora of design tools and methodologies. The other main advantage of OOT is the potential for extension and reuse. If classes are at the correct level of abstraction then it is highly likely that extensions to a system can be incorporated by introducing further subclasses without disturbing the existing implementation.

The implementation of objects in most OOT languages is highly dynamic. An object is generally explicitly constructed, which results in the implicit runtime invocation of dynamic storage allocation followed by a *constructor* operation to perform the initialisation. The object is accessed indirectly, and the accessor variable is weakly typed (“*polymorphic*”) which allows it to point to other objects of different subclasses within the same class hierarchy at runtime. The specific subclass of the referenced object can be used to control at runtime which specific version of an operation within a class hierarchy is invoked (“*dynamic dispatching*”). The destruction of an object may also be implicit and occurs when there are no remaining access paths to it during the dynamic execution of the program. As part of destruction, there may also be implicit invocation of a *finalization* operation. The detection of the point at which an object may be destroyed is generally the responsibility of a runtime *garbage collector*.

This model severely conflicts with the highly static model that is required in high integrity systems to meet the goals of time- and memory-boundedness assurance, and of determinism in data transformation due to code operation. The challenge for SPARK was to define a static model that allows the existing analysis verification techniques to be employed, whilst retaining sufficient elements of OOT to enable the maximum number of the benefits to be realised.

The key aspects of OOT are summarised below, together with the approach that is adopted in the SPARK language in each case.

Constructors and Finalizers

The model of object creation and destruction in SPARK is explicit. A SPARK object is created simply via its declaration, and its lifetime is statically determined by the scope in which it is declared. The declaration defines its type (and class) statically. Strong typing is a key feature of the SPARK language that prevents objects of incompatible types (classes) being inadvertently interchanged. In addition, static typing ensures that the object attributes are fully known and hence the full range of static analysis techniques may be applied.

This model avoids the need to create objects in dynamically-allocated memory, such as a *heap*. This avoids all the associated non-determinism regarding heap fragmentation, exhaustion and compaction, unbounded heap allocation times, and the overheads and unpredictability inherent in the use of a garbage collector. This is essential in order to prove statically the time and memory constraints of the system.

The object may be initialised either as part of its declaration, or via an explicit call to an initialisation function. In SPARK, there is an annotation that distinguishes these two cases, thereby reducing the risk that initialisation is inadvertently omitted. In addition, data flow analysis techniques ensure that the object is initialised prior to first use.

Object finalization is also explicit in SPARK. A finalizer operation must be called explicitly prior to the exit of the object's scope if some finalization action is required. Note that storage reclamation for the object is automatically achieved via the scope-based nature of the object declaration.

Abstraction

Abstraction involves the identification of the abstract features of a class - these are the key features that define each object of the class. SPARK supports abstraction via its abstract data types that define both the abstract data fields for members of the class and the abstract operations for the class.

Encapsulation

Encapsulation involves the hiding of implementation detail from clients of the class. SPARK supports encapsulation via packages. The package specification provides the visible interface for the class. This interface includes the abstract view of the data that is to be made visible to clients, the signature of all visible operations (methods) of the class, and annotations that define the abstract semantics of each operation in terms of its effect on the class data and the global state of the program.

The package body encapsulates the entire implementation of the class, which is hidden from clients of the class. SPARK language rules verify that the implementation conforms to the annotations that define its visible semantics.

Modularity

SPARK is a highly modular language. The package construct allows a program to be partitioned into logically separate components that include linkage information to define the inter-relationships and dependencies.

Hierarchy and Inheritance

In OOT the concepts of hierarchy and inheritance are closely coupled. In order to create a hierarchy of related classes, a subclass can be created that is based on a more abstract superclass. The subclass automatically inherits all the visible attributes and operations of the superclass and in addition, the subclass may override certain inherited operations with specialist implementations, and may also add further attributes and operations that are appropriate to itself.

The creation of hierarchies and the application of single-level inheritance are both supported within SPARK via its derived types and child packages. It should be noted that SPARK does not support multiple inheritance, where a subclass may have more than one superclasses of equal status. The reason for this restriction is based on experience of the problems that full multiple inheritance have demonstrated in earlier OO languages, both in terms of ambiguity of semantics and of implementation complexity. In contrast, a simpler form of multiple inheritance based on adding the concept of implementing one or more *interfaces* to the single inheritance model, as is found for example in Java™, offers a solution that avoids the ambiguities and complexity of the full multiple inheritance model. This simpler form is to be added to the next revision of the Ada language standard, at which point it may also be adopted within SPARK.

A further characteristic of the SPARK inheritance model is that it enforces statically the Liskov/Wing substitution principle [4]. This principle essentially requires that where a subclass operation overrides an operation in the superclass then the subclass operation will conform to all pre- and post-conditions of the superclass operation. In this context, conformance implies that the subclass operation requires at most the pre-conditions of the superclass operation, and promises at least the post-conditions of the superclass operation. This principle is essential in providing the guarantee that a new extension to a class hierarchy does not violate any established contracts.

Support for the Liskov substitution principle in SPARK is based on explicit pre- and post-condition annotations that may be applied to any operation, allowing formal verification of code properties. The use of pre-conditions is a very powerful tool in defining the conditions under which an operation can be used safely. One of the major risks of inheriting operations is not understanding the pre-conditions associated with the use of the operation, and not abiding with those pre-conditions in the context of a subclass. SPARK protects against this since the preconditions are stated explicitly and more importantly, the SPARK Examiner tool will statically analyse the program to ensure that the operation is only ever invoked within its pre-conditions.

Polymorphism and Dynamic Dispatching

Polymorphism is a key feature of OOT. A polymorphic object may contain a value that belongs to any subclass of the class of the object. The subclass of the value of a polymorphic object may also change dynamically during program execution. This can be very powerful in allowing general operations to be built which are valid for all subclasses of a given superclass, even before all the subclasses have been identified. In addition, this feature can be used in conjunction with operation inheritance hierarchies to implement dynamic dispatching (or “*dynamic binding*”).

However polymorphism and dynamic dispatching undermine certain aspects of static analysis. The actual object that is represented by a polymorphic object is only known at runtime, which interferes with data and information flow analysis. Similarly, the actual operation that is called via dynamic dispatching is only known at runtime, which undermines control flow analysis. These issues apply not only to formal verification by a static analysis tool, but also to verification by code review and by test. There is also an impact on other forms of static verification techniques such as schedulability and response time analysis, model checking and object code coverage analysis.

For these reasons, polymorphism and dynamic dispatching are not directly supported in SPARK. However in recognition that some parts of a high integrity system may be at a lower integrity level that does not require verification using rigorous static analysis techniques, SPARK supports a model whereby polymorphism and dynamic dispatching may be used as a *hidden* implementation of an abstraction whose specification can be expressed in SPARK. In this case, the hidden implementation is not analysed by the SPARK Examiner tool, but the specification is analysed to ensure that the contract between the abstraction and its clients are not broken.

Other Concerns over use of OOT in High Integrity

There are a number of practical concerns over the use of OOT within the development and verification of high integrity systems that cannot be addressed directly by language-level solutions. The two major concerns are as follows.

Obscurity

The creation of large class hierarchies can result in obscurity in determining the provenance of an inherited operation at the point of call. This is particularly the case where full multiple inheritance is permitted, and was a strong driver behind omitting this feature from both SPARK and the Ada language. SPARK addresses this issue by eliminating dynamic dispatching, and so the actual operation to be called is statically determinable, and thus reviewable and verifiable.

In addition, when an operation is inherited by a subclass, it is often the case that the restrictions and implicit semantics on the use of the operation are not apparent to users of the subclass. This is significantly influenced by the physical separation of the inherited operation from the implementation of the subclass. This can lead to tried and tested operations being used outside of the context that is assumed as part of the guarantee contract. SPARK addresses this issue via its pre-condition annotations that can be used to express contextual assumptions explicitly, and by requiring the Liskov substitution principle [4] to apply statically to all class hierarchies. Another language which offers conformance to the Liskov substitution principle is Eiffel [8]. However with Eiffel, non-conformance to this principle is only detected dynamically at runtime.

Dead-code/Deactivated code

OOT developments are very susceptible to generating deactivated code. This is due to the way that classes are constructed, starting with abstract classes and building on these to construct a hierarchy of classes with potentially specialised versions of operations. It is

easy to see that there may be no instances of a superclass in a program and if all the subclasses override a particular method then the superclass method constitutes dead or deactivated code.

The main problem with dead or deactivated code within a high integrity system is obtaining the confidence that it cannot be reached in the execution of the program, and thus cannot have any effect on the runtime behaviour. It is difficult to demonstrate through testing alone that code cannot be reached. It is usual in high integrity systems to impose structural coverage requirements on program testing to guarantee that all code has been exercised - code that cannot be reached by test should be removed or justified by analysis. However systems generated using OOT are likely to have far more dead or deactivated code than systems generated using methodologies traditionally favoured for high integrity systems, and so the degree of manual justification may be substantial. It may be necessary to provide improved tool support to justify the acceptance of dead and deactivated code in programs built through OOT. The SPARK model simplifies this approach by ensuring that the set of operations that are actually called within a program is statically determinable.

Conclusions

The use of object-oriented technology (OOT) has been shown to be of great value in many market sectors, but the challenges to the use of such technology within high integrity systems that require evidence of rigorous verification and certification are non-trivial.

Many proponents of OOT cite the gains in productivity and re-use that OOT will bring. We have seen little evidence to back up this claim[10], especially in relation to the construction of high-integrity systems. While re-use might apparently improve the productivity of design and coding activities, it may well have a detrimental effect on later activities such as integration test and certification. Our experience with SPARK suggests that actually slowing design and code (a bit) as a route to easing test and certification (a lot) is more useful. Secondly, we only ever measure whole-lifecycle productivity—we find that speed of coding has little to do with real progress.

Despite some original pessimism over OOT, we have indeed managed to incorporate a reasonably useful subset of such facilities in SPARK, without compromising the original design goals of the language. While OO zealots might bemoan the lack of polymorphism and dynamic-binding in SPARK, the simple fact that these features currently defy static analysis is a strong justification for their exclusion from a high-integrity language.

Finally, a brief word on standards. Our experience with the UK's and other European software standards has validated the SPARK approach in many industrial domains, and SPARK meets all the requirements for language design and static analysis laid down by these standards. DO-178B, on the other hand, places great emphasis on testing (and, in particular, coverage analysis) while having comparatively little to say on programming languages and static analysis. This is a somewhat unfortunate—the difficulties of creating critical systems are so great that we should deploy every weapon available to us, rather than relying on just one. Notwithstanding this, SPARK has been used to great effect in the Lockheed C130J and C27J programmes[11], but these remain isolated

examples, perhaps because "178B doesn't say I *have* to do static analysis, so I won't!" We strongly believe that DO-178C would benefit from recognizing the contribution that static analysis and the associated language design issues have to make.

References

1. B. A. Carré and T. J. Jennings, *SPARK – The SPADE Ada Kernel*, Department of Electronics and Computer Science, University of Southampton, 1988.
2. D. W. R. Marsh and I. M. O'Neill, *The Formal Semantics of SPARK*, Program Validation Ltd, Southampton, 1994.
3. ISO/IEC TR 15942, *Guide for the Use of the Ada Programming Language in High Integrity Systems*
4. B. Liskov, J. Wing, *A Behavioural Notion of Subtyping*. ACM Transactions on Programming Languages and Systems, Vol. 16, No 6, November 1994.
5. U.K. Ministry of Defence *00-55 Requirements of Safety Related Software in Defence Equipment*, 1997. www.dstan.mod.uk
6. MISRA, *Development Guidelines for Vehicle Based Software*, The Motor Industry Research Association, 1994.
7. R. Chapman and R. Dewar, *Re-engineering a safety critical system with SPARK95 and GNORT*, Lecture Notes in Computer Science 1622, Reliable Software Technologies, Ada Europe 1999.
8. B. Meyer, *Eiffel : The Language* , ISBN 0-13-247925-7 -- Prentice Hall 1992.
9. J. Barnes, *High Integrity Software: The SPARK Approach to Safety and Security*. Addison Wesley, 2003.
10. G. Goth, *Has Object-Oriented Programming Delivered?* IEEE Software, Volume 19, Number 5, September/October 2002.
11. P. Amey, *Correctness by Construction: Better can also be Cheaper*. CrossTalk journal, March 2002. PDF on www.stsc.hill.af.mil and also www.sparkada.com
12. Butler, Ricky W.; and Finelli, George B, *The Infeasibility of Quantifying the Reliability of Life-Critical Real-Time Software*. IEEE Transactions on Software Engineering, 19(1):3-12, January 1993.
13. S. King, J. Hammond, R. Chapman, A. Pryor, *Is Proof More Cost-Effective Than Testing?* IEEE Transactions on Software Engineering, Vol 26., No. 8, August 2000, pp 675-685.