Breaking Through the V & V Bottle Neck

Martin Croxford, James Sutton

Publication notes

Presented at Ada in Europe, 3rd October 1995.

Published by Springer-Verlag in “Lecture Notes in Computer Science”, Volue 1031, 1996.

© Springer-Verlag

Copyright © 2003 Praxis Critical Systems Limited. All rights reserved

Presented at "Ada in Europe 1995"
Published by Springer-Verlag in
"Lecture Notes in Computer Science" Volume 1031, 1996

Breaking Through the V and V Bottleneck

Martin Croxford † and James Sutton ‡

† Praxis Critical Systems, Bath, England
‡ Lockheed Aeronautical Systems Company, Marietta, Ga, USA

Abstract. With conventional methods of performing verification and validation - heavily reliant on testing performed late in the software production process - the late detection of errors adds substantially to project costs and delays in delivery, and introduces significant risks. This paper presents a method of software development aimed at "correctness by construction", which greatly attenuates these problems. The process described here has been applied successfully to the development of avionic software for the new C-130J ("Hercules") aircraft.

1 Introduction

In industrial software development generally, but especially for the larger and more complex systems, verification and validation (V & V) costs form an unacceptably large part of total software development cost - a figure of 50% is commonly quoted. Furthermore, this is the area where significant risks arise, or are uncovered. We would like to "design out" a large proportion of the V & V costs and risks.

Practical V & V approaches are currently all of a retrospective or "after the fact" nature. This aspect of V & V has several disadvantages:

- In an increasingly competitive aeronautical industry, the demands for greater functionality and rapid delivery of software make it imperative to "get it right first time". Retrospective verification comes too late to support this goal.

- Not only is it necessary nowadays to design and deliver advanced avionic software products to the market-place very rapidly, but the scale of aeronautical enterprises requires the elimination of risk (of delay for instance) associated with such developments. We cannot wait until system test "to see if it works".

- The achievement of test coverage, such as the Modified Condition/Modified Decision (MC/DC) test coverage required to comply with RTCA DO-178B [7] to Level A (for safety-critical systems) involves a great deal of work. The need to repeat such testing several times, after finding and rectifying errors discovered in the testing process, contributes largely to project costs. We need much earlier detection of defects.

- Retrospective V & V is very sensitive to requirements changes - which occur in every software development project. Each change demands software modifications, and if the software has already undergone V & V then, at the very least, V & V must be repeated on the changed program units. However, sound defensive practise and requirements of regulatory agencies can impose repetition of V & V on all the software in the configuration item containing the modifications. Consequently, small changes to requirements usually lead to massive repetitions of V & V.

- In an era of great safety-consciousness, not only must avionic products be safe, but they must be *demonstrably* safe. It is much easier to achieve the required level of software integrity, with convincing supporting evidence of compliance with a standard such as RTCA DO-178B to Level A through a rigorous construction process than through after-the-fact justification.

The key to breaking through the V & V bottleneck, then, is to strive for *correctness through construction,* by revision of the software development process. Lockheed Aeronautical Systems Company (LASC) and Praxis Critical Systems evolved independently to this approach based on similar philosophical foundations. LASC had focused its attention primarily on the software specification and design (and their corresponding verification) portions of the lifecycle, while Praxis had developed technology for the design and coding (and corresponding verification) portions of the lifecycle. The marriage of the two has provided comprehensive application of the correctness by construction principle throughout the software lifecycle. The first usage of this combination is on the successful development of a new avionics system for the C-130J Hercules II aircraft.

This approach, described below, has not involved replacing the conventional approach to programming by a strictly formal refinement process, but rather it builds on existing strengths, extending the constructive role of the programming language and process; its implementation is both practical and economic, using methods, techniques and tools that are available today. It will be seen, furthermore, that the measures to achieve correctness through construction naturally emphasise modularity and portability, safety and robustness, ease of extension and verifiability. These are essential characteristics of software "building-blocks", to be applicable to evolution over the life of a single product, including requirements changes during its initial development, or over a range of related products.

2 The Engineering Context

The successful use of the technology to be described here has depended not on a single choice of an appropriate method or tool, but on a number of decisions - relating to engineering context, hardware and software architecture, the form of expression of requirements, design rules, use of Ada, and formal methods - which in combination eliminate the "paradigm shifts" and changes in notation between successive

development stages that usually militate against traceability and make testing so expensive. It will therefore be helpful first to set the scene, with a very brief description of the engineering context.

The C-130 Hercules, a tactical transport aircraft with both military and civilian uses, has been in production since 1955. LASC is transforming the Hercules' avionics, propulsion, flight station and certain airframe systems to create the Hercules II airlifter or the C-130J. The Operational Flight Program (OFP) software coordinates and controls the many individual systems, and operates and diagnoses the integrated aircraft as a whole. There is a primary OFP in each of two Mission Computers (MCs), and a backup OFP in each of two Bus Interface Units (BIUs). The MC and BIU OFPs also synchronise and transfer data between the various avionics hardware devices. Examples include data about electronic circuit boards, full authority digital engine control, radar and the fuel system. This data transfer is organised by the two MCs and by several hardware buses, most conforming to MIL-STD-1553.

The MC and BIU OFPs also process and transfer information needed for the flight station hardware to interface with the pilot, copilot, navigator and auxiliary crew members. This information is interdependent with the avionics, propulsion and airframe systems data that are also conveyed by the communications buses and handled by the MCs. Examples of flight station devices include head-up displays, radio control panels, and caution/warning/alert annunciators. The MC and BIU software amounts to some 200K lines of source code.

Without entering into details of system architecture, it will be clear that the MC and BIU software performs a great multiplicity of functions (to transfer information to and from a bus, originating and terminating at many different kinds of devices), but that these functions are mutually independent, and simple to describe in a precise manner. (Of course the *implementations* of these functions may use common architectural components, as is explained below.) Exploitation of the "separability" of the many functions of the OFPs contributed greatly to the simplicity of the expression of requirements, their implementation and V and V [4].

3 Correctness by Construction

The essence of a constructive approach is the "factorisation" or decomposition of a large system development into a number of small steps, each constructing a "product" according to its own rules of "well-formation" which can be enforced by a tool (in the way, for instance, that a compiler imposes the syntactic and static-semantic rules of a programming language). The well-formation rules must in themselves guarantee a certain consistency between the input and output of each step (e.g. of data flow or information flow), and facilitate complete verification of functionality. Decomposition of this kind is the key to containment of system complexity, allowing us to reason about fragments of implementation in terms of fragments of specification.

Of course, to reason about a program in terms of its constituent parts, we must be able to think of these in abstract terms, i.e. in terms of *specifications* of the functions they are intended to perform rather than their implementation details. And if these specifications are to be manipulated mechanically, they must be formulated rigorously, in a well-defined notation, as *formal specifications.* However, if formal specifications are produced for program constituents, we can perform formal verification of their code, mechanically, as this code is produced. (By formal verification we mean here mathematical and semi-automated verification of internal consistency of the software in terms of absence of data and information flow errors, as well as verification of the correct implementation of requirements.) We note the following advantages of this constructive approach:

- Formal verification of the code of a program component can be performed as soon as this is written, and in particular, before compilation and testing. We achieve early warning of many kinds of programming errors, and the expensive testing processes are usually performed once only, as *confirmation* of correctness.

- Both coding and formal verification of a program unit can be performed in isolation, using only specifications of the software components used by the unit under construction. Thus we can verify some parts of a system, while others are incomplete.

- Effects on specifications and code of program components, of changes in requirements specification, is contained to the minimum possible.

4 Requirements Engineering

The software requirements for the MC and BIU are specified using an extension of the Software Productivity Consortium's (SPC) "CoRE" (Consortium Requirements Engineering), a formal requirements modelling method based on the work of David Parnas [8, 2]. Here, the software requirements are described in a tabular form, specifying input-output relationships mathematically. This method of description is, like most, better with some kinds of system than others. CoRE is particularly well-suited for systems which have many inputs, many outputs, and relatively simple transfer functions between them. This category includes the OFPs being developed for the C-130J. An extension of CoRE by LASC represents the CoRE data by Yourdon Data-Flow Diagramming, includes the notion of "domain generics", and uses a CASE tool to maintain the CoRE data dictionary and to perform automatic well-formation checking of the CoRE model [4].

An example of a CoRE requirement is given in Figure 1. The table defines a relation. The heading of the rightmost column would normally be the name of a specific abstracted output (the relation's "dependent variable"); the names of all other columns would be the names of specific abstracted inputs (the "independent variables"). The cells below the headings in all but the rightmost column define subranges of the

independent variables. The cells in the rightmost column define the function used to derive the output given the combination of subranges in the cells to its left in its row. This kind of tabulation makes it possible to positively verify coverage of the entire ranges of the inputs used to derive the output (and all the combinations of subranges). It also highlights the importance of boundary values at the "lines" between the rows.

In practice, this specification would be implemented by an Ada procedure or function subprogram. On completion of execution of this subprogram, the conditions to be satisfied, as specified by the CoRE table, can be expressed as a post-condition in first-order logic, as shown in Figure 2. (The notational conventions used here are those of SPARK, which we discuss below.)

abstracted input #1 ("i1")	abstracted input #2 ("i2")	abstracted input #3 ("i3")	abstracted output
"x"	"x"	subrange 3.1	f1(i1, i2, i3)
"x"	subrange 2.1	subrange 3.2	f2(i1, i2)
subrange 1.1	subrange 2.2	subrange 3.2	f3(i3)
subrange 1.2	subrange 2.2	subrange 3.2	f4(i1, i3)

Figure 1. Example of a CoRE Requirement

```
--# post
--#   (IsInSubrange_3_1(i3)       -> Output = f1(i1, i2, i3)) and
--#  ((IsInSubrange_2_1(i2) and IsInSubrange_3_2(i3))
--#                              -> Output = f2(i1, i2))     and
--#  ((IsInSubrange_1_1(i1) and IsInSubrange_2_2(i2)
--#     and IsInSubrange_3_2(i3)) -> Output = f3(i3))         and
--#  ((IsInSubrange_1_2(i1) and IsInSubrange_2_2(i2)
--#     and IsInSubrange_3_2(i3)) -> Output = f4(i1, i3));
```

Figure 2. Post-condition for Example of a CoRE Requirement

5 Design

Architectural design is carried out using a Domain-Specific Design Language (DSDL), strongly influenced by the Software Productivity Consortium's ADARTS (Ada-based Design Analysis for Real-Time Systems) method [9], in particular its notions of "Class Structuring" which LASC have extended and at the same time "specialised" with rules of coherence tailored to their application domain. These rules are enforced or otherwise checked by use of Teamwork templates, ultimately represented by Ada packages. More detailed design and coding are both performed through instantiation and population of templates, with extensive use of EMACS

scripts to automate the process. The "refinement" of design to the final executable code proceeds through as many as six "levels", at each of which the working material is compilable Ada text, which can be checked mechanically.

In Figure 3 is shown a greatly simplified and abstracted version of the portion of the DSDL for the devices that are attached to the data buses on the C-130J. It is recorded in something similar to Buhr notation, as the CASE tool that was used implemented a variation of Buhr; however, other notations would have been just as suitable. Underlying textual definitions were also developed for the classes (outer boxes) and their methods (inner boxes). The dashed boxes in the illustration are syntactic elements that must be "instantiated" by the detailed designers with the relevant details of each specific device in the device category covered by the DSDL. Thus, the detailed design is simply the set of instantiations of the DSDL.

This example is indicative of the highly-factored nature of the system and software architecture, which is a key element of our approach.

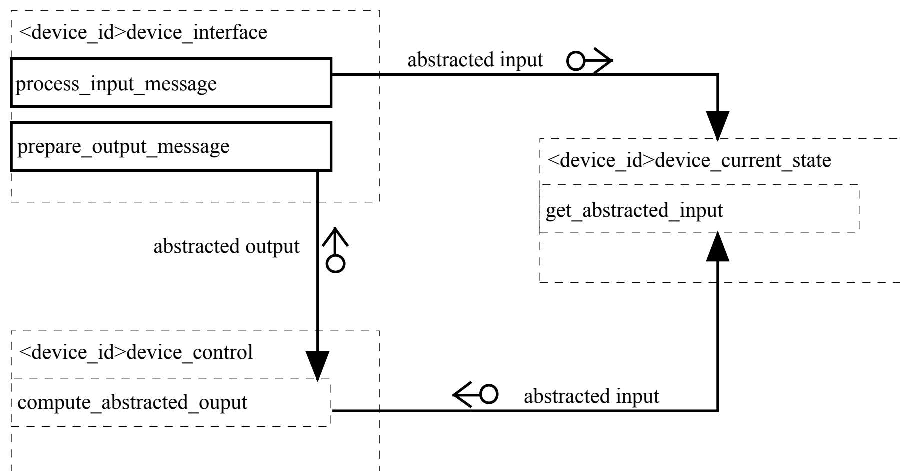

Figure 3. Simplified DSDL for Bus Devices

6 Implementation

The Ada text for all components containing significant system semantics is written using SPARK, a system of annotations (or "formal comments") and restrictions [1] that is applied to Ada to simplify demonstrating program correctness [3, 5]. SPARK was developed by Program Validation Ltd (PVL), now incorporated in Praxis Critical Systems. SPARK is formally defined (in the formal specification language "Z", plus inference rules [6]) and its use is supported by a software tool, the SPARK Examiner, which checks conformance of Ada texts with the rules of SPARK, and performs

different kinds of analyses, described below. SPARK is used in numerous military and civil safety-critical applications, e.g. avionics, railways, and nuclear power.

The method of developing the code, using templates, helps to preserve the correspondence between each "code refinement" and its predecessor, but strong additional checks are performed in the development process, as follows.

Firstly, as soon as executable code of subprogram bodies is produced, the SPARK Examiner performs their data- and information-flow analyses, and compares the results with flow relations given as SPARK annotations, these being derived from the CoRE and design documents. (At present these relations are produced manually, but in future they could be generated directly, within templates.)

Next, the formal specification descriptions (in Parnas tables) in the formal requirements specifications are embedded in SPARK program texts as post-conditions, of the kind shown in Figure 2. These annotations are expressed in a language which is essentially the language of Ada expressions, with a few extensions for instance to represent logical inference, and to describe the effects of updating operations on composite objects.

From this information, and the code, the SPARK Examiner produces the "proof obligations" which must be discharged to show that the code meets its specification. Many of these proof obligations can be proved automatically by the SPADE Automatic Simplifier, and the rest can be proved interactively using the SPADE Proof Checker. (Alternatively, they can be justified manually, by "rigorous argument"). In the C-130J project, the generation and automatic simplification of proof obligations for SPARK text has so far been straight-forward - probably because of the simplicity of the required input-output functions. Thus, by using extensions to Ada involving no more than "formal comments", and tools to check the relationship between these and the Ada code, we have been able to bind together the requirement specification, high-level and detailed design and the executable code, with strong rules of well-formation of construction, and verification procedures, all applicable prior to compilation and conventional unit test.

As an illustration of the code verification process, Figure 4 presents a SPARK subprogram which implements the CoRE table of Figure 1. Using the post-condition given in Figure 2, the SPARK Examiner generates 11 proof obligations ("verification conditions") for this code, one of which is shown in Figure 5. This is discharged automatically by the SPADE Automatic Simplifier.

```
procedure Example(i1,i2,i3 : in     integer;
                  Output   :    out integer)
is
begin
   if    IsInSubrange_3_1(i3) then Output := f1(i1, i2, i3);
   elsif IsInSubrange_2_1(i2) then Output := f2(i1, i2);
```

```
      elsif IsInSubrange_1_1(i1) then Output := f3(i3);
      else                            Output := f4(i1, i3);
      end if;
   end Example;
```

Figure 4. Subprogram Implementing Example CoRE Requirement

```
procedure_example_11.
H1:     isinsubrange_1_1(i1) or isinsubrange_1_2(i1) .
H2:     not (isinsubrange_1_1(i1) and isinsubrange_1_2(i1)) .
H3:     isinsubrange_2_1(i2) or isinsubrange_2_2(i2) .
H4:     not (isinsubrange_2_1(i2) and isinsubrange_2_2(i2)) .
H5:     isinsubrange_3_1(i3) or isinsubrange_3_2(i3) .
H6:     not (isinsubrange_3_1(i3) and isinsubrange_3_2(i3)) .
H7:     not (isinsubrange_3_1(i3)) .
H8:     not (isinsubrange_2_1(i2)) .
H9:     not (isinsubrange_1_1(i1)) .
         ->
C1:     isinsubrange_3_1(i3) -> (f4(i1, i3) = f1(i1, i2, i3)) .
C2:     (isinsubrange_2_1(i2) and
            isinsubrange_3_2(i3)) -> (f4(i1, i3) = f2(i1, i2)) .
C3:     (isinsubrange_1_1(i1) and (isinsubrange_2_2(i2) and
            isinsubrange_3_2(i3))) -> (f4(i1, i3) = f3(i3)) .
C4:     (isinsubrange_1_2(i1) and (isinsubrange_2_2(i2) and
            isinsubrange_3_2(i3))) -> (f4(i1, i3) = f4(i1, i3)) .
```

Figure 5. Example Proof Obligation

In this illustrative example the correspondence between the specification and the code is almost obvious. In general the functions to be implemented are more complex, for example the abstract variables may be realised not simply by integers but by structured variables, which may themselves be subject to transformations, for instance to perform scaling, range-limiting and data-packing.

7 Compilation and Testing

Only after formal verification should the code be compiled and tested. Although the burden of proof is increased if verification is attempted for code that does not yet meet its specification, the cost savings obtained by catching errors early, and especially before system integration and costly MC/DC testing, are very great in comparison.

Compilation is performed using an Alsys C-SMART compiler. SMART is a small Ada run-time system, for safety-critical applications, and C-SMART is a certifiable version of this, with documentation designed to satisfy RTCA DO-178B Level A requirements. Since all SPARK features are supported by C-SMART, no problems of compatibility arise.

To test the end product, a set of tests has been developed rigorously from the CoRE formal specification. These tests are implemented as "scripts", to be run in a tool that simulates the environment of the computer executing the product software. The simulation tool directly employs a database created during system requirements

engineering. The scripts and the database constitute a "product validation suite" that can easily be re-executed at any stage in the development and subsequent life of a product, to confirm that the product requirements are being properly met.

8 Lessons Learned

The constructive approach adopted for the C-130J MC and BIU OFP software has had the effect of keeping the development effort in pace with systems-engineering additions and modifications, and with the program schedule. This is in contrast to the "software crisis" experience of many if not most aerospace (and other application-domain) projects, where software becomes the main cause of overall project delays. The approach works well in an environment where there are difficulties in stabilising system requirements.

The constructive approach involves the entire software development team more deeply with systems engineering: it requires software developers to verify code as it is developed, possibly chasing errors and inconsistencies in the requirements, rather than relying on unit and integration testing to find problems. Of course, this is a consequence of the formal methods and the constructive approach: they simply won't allow systems-level inadequacies to propagate into the software product. While this costs some extra time in the short term, it always more than regains it later.

While most of the errors uncovered in requirements specifications and code by formal analysis would have been discovered during integration testing of the software, it has been found that tracing such bugs from traditional test results is more expensive than formal analysis of the same code. Some errors immediately uncovered by formal analysis, such as conditional initialization errors, may only emerge after very extensive testing.

The next lesson relates to software changes. When the need for domain-level requirements changes is identified, the amount of affected software can be large. Yet, because of the functional decomposition in the design, and the DSDL/template-driven implementation, C-130J experience has shown that the places where code must be changed are clear from examination of the design template(s) relevant to the affected structures. Comprehensive changes can rapidly be effected by changing the appropriate section of code in each of the implementation instances of the affected template(s). Unlike non-domain-oriented architectures, where functionality is distributed in software in an irregular way, there is objective confidence that the software changes will correctly implement the requirements changes.

Finally, we draw attention to the significance of this software project, as one of the first in which the use of formal methods has proved to be advantageous from an economic standpoint, rather than simply being desirable or even essential to achieve a particularly high level of integrity. In what way has the application, or the technology deployed, really been exceptional, to allow us to achieve this result?

The essential factors are firstly the architectural design features, which factorise this sizeable system into a large number of functionally distinct components, so that the scaling-up difficulties often associated with formal methods do not apply; secondly, the use of formal specification of the requirements of these components; thirdly, the rigidly-enforced design rules which preserved the functional separation of the components, and "kept the requirements specifications alive". Translation of the CoRE into formal specifications of program units also supported the functional separation; and finally, the technology for generating and discharging the proof obligations, based on the formal definition of the SPARK component of Ada, was crucial, in binding the code to the initial requirements.

References

1. Ada 95 Reference Manual, ISO/IEC 8652:1995(E)-RM95; version 6.0, December 1994. (See especially Annex H "Safety and Security".)
2. Alspaugh, S. Faulk, K. Heninger Britton, R. Parker, D. Parnas, J. Shore: Software Requirements for the A7-E Aircraft. Report NRL/FR/5530-92-9194. Naval Research Laboratory, Washington, D.C., 1992.
3. B.A. Carré, J.R. Garnsworthy: SPARK - An annotated Ada subset for safety-critical programming. In: Proceedings of Tri-Ada Conference, Baltimore, December 1990.
4. S. Faulk, L. Finneran, J. Kirby, Jr., S. Shah, J. Sutton: Experience applying the CoRE method to the Lockheed C-130J software requirements. In: Proceedings of Ninth Annual Conference on Computer Assurance, Gaithersburg, MD, 1994, pp.3-8.
5. J.R. Garnsworthy, I.M. O'Neill, B.A. Carré: Automatic proof of absence of run-time errors. In: Proceedings of Ada UK Conference, London Docklands, October 1993.
6. Program Validation Ltd.: The Formal Semantics of SPARK (Volume 1: Static Semantics; Volume 2: Dynamic Semantics). Praxis PVL, 20 Manvers Street, Bath BA1 1PX, U.K., 1994.
7. RTCA: Software Considerations in Airborne Systems and Equipment Certification. RTCA/DO-178B, 1994
8. Software Productivity Consortium: Consortium Requirements Engineering Guidebook, SPC-92060-CMC version 01.00.09. Software Productivity Consortium, Herndon, VA, U.S., 1993
9. Software Productivity Consortium: ADARTS Guidebook, SPC-94107-N, version 02.01.00 Software Productivity Consortium, Herndon, VA, U.S., 1991.