Ada®COMMUNITY CONCERNS REGARDING DOD-STD-2167

Donald G. Firesmith
Magnavox Electronic Systems Company, Fort Wayne, Indiana, 46808

1 INTRODUCTION

DEFENSE SYSTEM SOFTWARE DEVELOPMENT (DOD-STD-2167) contains requirements for the development of Mission-Critical Computer Resource (MCCR) software and establishes a uniform software development process which is applicable throughout the system life-cycle. It was approved for use by all departments and agencies of the Department of Defense on 4 June 1985 and is therefore now being applied to a great many Ada projects. Although largely ignored by the Ada Community until the formation of the SIGAda Software Development Standards and Ada Working Group (SDSAWG) in late 1985, DOD-STD-2167 came under increasing scrutiny as part of the review of the draft Revision A during the Fall of 1986.

This paper outlines those concerns raised to the SDSAWG by members of the Ada Community regarding DOD-STD-2167. Because it represents the input of many people, probably no one will agree with all of its contents. Due to space limitations, this paper only contains the concerns raised and does not include the arguments that were brought against some of the concerns during the ensuing debate. It therefore does not represent the official position of the ACM, SIGAda, or even the SDSAWG. For more information, please contact the author at:

Magnavox Electronic Systems Co.
Dept. 566 10-C-3
1313 Production Road
Fort Wayne, IN 46808
(219) 429-4327

for a copy of the official SDSAWG report.

A copy of DOD-STD-2167 will prove a useful reference during the remainder of this paper.

®"Ada" is a registered trademark of the U.S.Government (Ada Joint Program Office)

2 Ada AS DEFAULT IMPLEMENTATION LANGUAGE

DOD-STD-2167 is currently language independent, and many members of the Ada Community feel that this should be changed. The following arguments have been raised in favor of having DOD-STD-2167A take Ada as the default programming language:

1. It would bring DOD-STD-2167 into compliance with DoD policy. The 10 June 1983 letter from Under Secretary of Defense DeLauer states "Pending formal coordination and publication of this directive [DODD 5000.31], I request that it be implemented as DoD policy..." Specifically, the directive states that "The Ada programming language shall become the single, common, computer language for defense mission critical applications..." and "... the use of the Ada programming language is ACTIVELY encouraged."

2. Making Ada the default language of the standard will significantly promote Ada usage.

3. Both Ada and DOD-STD-2167 have been developed and mandated to apply to the same class of software applications (i.e., all software that is part of Mission-Critical Computer Resources). Thus, the vast majority of software to which DOD-STD-2167 will be applied will be Ada projects. The Ada-dependent parts of the standard can be tailored out for non-Ada projects.

4. A strong precedent already exists for the extensive use of the Ada language in military standards. Both "Internet Protocol" (MIL-STD-1777) and "Transmission Control Protocol" (MIL-STD-1778) use a subset of Ada constructs common to most high-level languages for the declaration of data structures, etc.

One person suggested that making Ada the default language should not be regarded as a denial of language independence, but rather a way of ensuring a

more effective way of communicating the concepts of software development and design.

2.1 DEFAULT CODING STANDARD

The default coding standard in DOD-STD-2167 is neither consistent with nor based upon Ada. According to DOD-STD-2167, "If the contractor has not ... received ... approval for [his] internal coding standards, then the coding standards of Appendix C shall apply." According to Appendix C, "Code shall be written using only the five control constructs illustrated in Figures 5 through 9" and if the "higher order language does not contain the[se] control constructs ..., the contractor shall use [a] precompiler..."

These control constructs in the present default coding standard do not exactly match those of Ada. Specifically, Ada's sequence construct has exceptions which introduce secondary exits. Ada's version of the **if-then-else, do-while, do-until, case, select**, and **return** all permit secondary exits via exceptions. Additionally, Ada also has the **for-loop** and **task** constructs.

This seems to either prohibit the direct use of Ada or requires the contractor to use a pre-processor to transform the present control constructs into valid Ada. It is not reasonable to require a precompiler to transform classic constructs into Ada for many reasons, the least of which being that the classic constructs do not convey sufficient structural semantics.

The current draft DOD-STD-2167A contains a new default Ada coding standard (Appendix D). Because both Ada and DOD-STD-2167 have been developed and mandated to apply to the same class of software applications, the primary default coding standard (Appendix C) should be based on the Ada programming language, and any other coding standards (whether language independent or specific) should be treated as special cases.

2.2 DEFINITION OF *UNIT*

The definition of the term *unit* in DOD-STD-2167 resulted in several concerns being raised by members of the Ada Community.

2.2.1 DOD-STD-2167 AND Ada INCONSISTENCY

Although both DOD-STD-2167 and ANSI/MIL-STD-1815A (Ada Programming Language) have been mandated for the same class of applications, their definitions of the important term *unit* differ in several ways. This inconsistency produces significant confusion which is worsened because no natural, standardized mapping exists from one type of unit to the other.

According to paragraph 3.23 of DOD-STD-2167, a *unit* is the "smallest logical entity specified in the detailed design which completely describes a single function in sufficient detail to allow implementing code to be produced and tested independently of other Units. Units are the actual physical entities implemented in the code." and according to paragraph 4.2 of DOD-STD-2167, units are the "smallest logical entities, and the actual physical entities implemented in the code." According to ANSI/MIL-STD-1815A, an Ada programming unit (the primary type of unit defined in the standard and thus the type of unit most likely to be confused with the DOD-STD-2167 *unit*) is either a subprogram, package, generic unit, or task unit.

The following are examples of various problems encountered when trying to map DOD-STD-2167 units into Ada programming units:

1. Although packages are the main structural entities in Ada programs, a package can not be a DOD-STD-2167 unit because it is not the "smallest logical entity specified in the detailed design which completely describes a single function" (see 3.23). It is not the smallest logical entity because it may well contain subprograms, etc. It should not be the smallest entity specified in the detailed design because the detailed design should usually document the Ada programming units contained within the package. It also is probably not associated with a single function. For example, a package containing five subprograms would probably perform five different functions while a package containing only type definitions performs no functions as such.

2. However, the "smallest logical entity" which one would want to individually document in the de-

tailed design might be an entire Ada package because:

(a) Of coupling/cohesion constraints and the sharing of data types, etc.

(b) The package may be extracted as a "black box" from a reuse library, where no information about its internal structure is available.

3. On the other hand, any subprograms contained in a package would not be DOD-STD-2167 units since they are not "produced and tested independently of other units."

4. Although some have suggested that Ada specifications and bodies be interpreted as DOD-STD-2167 units because they are the "smallest logical entities", other problems arise. If a subprogram specification is a compilation unit, the corresponding body cannot be compiled without it. Thus, the subprogram body can not be "produced and tested independently" of the specification.

5. It is important that the DOD-STD-2167 definition of unit be such that the placing of both specification and body in the same compilation unit is neither mandated nor disallowed.

6. It is unclear how the `IS SEPARATE` clause is impacted by the present DOD-STD-2167 definition of unit.

The problems with the DOD-STD-2167 definition that make it difficult to map it into Ada programming units thus seem to fall into the following areas:

1. The mere use of the same word to describe both concepts adds unnecessary confusion because of the temptation to map DOD-STD-2167 units directly into Ada programming (or compilation) units. As it now stands, this problem will only be the cause of endless non-productive arguments, across a broad range of projects, as to what a DOD-STD-2167 unit is in terms of software. That the government will be forced to pay for this needless confusion is intolerable.

2. Any design enitity small enough to be "the smallest logical entity specified in the detailed design" is likely to be too small to be "produced and tested independently".

3. Requiring a unit to be restricted to "a single function" ignores units containing only type information or units (e.g., Ada packages) for collecting related operations (i.e., functions and procedures) and hiding the operations on private types.

4. **Note that the above mentioned problems also exist for languages other than Ada.**

Perhaps the best way to deal with the problem of confusion between DOD-STD-2167 units and Ada units would be to change DOD-STD-2167 terminology. If Lower-Level Computer Software Component (LLCSC) were renamed Mid-Level Computer Software Component (MLCSC) and DOD-STD-2167 Unit were renamed Lower-Level Computer Software Component, then the DOD-STD-2167 static hierarchy would be CSCI, TLCSC, MLCSC, and LLCSC. This would make the static hierarchy terms more mutually consistent and avoid confusion with the Ada unit.

2.2.2 DATA UNITS

According to paragraph 3.23 of DOD-STD-2167, a unit is the "smallest logical entity ... which completely describes a single function...." This definition does not adequately address data units, because they do not perform "a single function". DOD-STD-2167 thus emphasizes procedural units (as well as CSCIs, TLCSCs, and LLCSCs) although data units are often as important or more so.

There is a tendency in many modern systems to embed the requirements and/or control in the data instead of in the executable code. The software is often so flexible and general that it solves a very large class of problems. To be able to apply such programs to a particular instance, the data are used to dynamically (i.e., at run time) conform/configure the actions of the program. For example, in many expert systems, the inference engine per se is almost trivial, and it is the data in the knowledge base and inference rules that embodies the system requirements and control. Another example is a compiler, which is so general that the grammar, semantics, and code generators are selected dynamically by reading the appropriate tables off the disk files. A third example would be the case where the files of a DBMS are key CSCs

or CSCIs. In order to produce and verify such systems for specific uses, it is necessary to design, document, integrate, configuration manage, and verify these data units (e.g., files) in order to ensure that the system-level requirements have been adequately allocated and satisfied. Yet the development, testing, and documentation of the content of data units is not adequately described in DOD-STD-2167, but could be added to sections 5.4 and 5.5.

In Ada, one standard use of packages (an Ada programming *unit*) is to store type and object (i.e., data) definitions. The definition of DOD-STD-2167 should be expanded to cover such units.

The term "Computer PROGRAM Component" has been replaced by "Computer SOFTWARE Component" due to the recognition that the term "software" is more inclusive than "program" (note: this change has also not yet been incorporated in MIL-STD-483 and MIL-STD-490). The definition of "computer data definition" (3.5) includes the value, or content, of the data elements, perhaps defined apart from the representation/format of those items (as embodied in the code). Yet the data aspects or generality of the definitions of "computer software" (3.6), "computer software component" (3.7), and "computer software configuration item" (3.8) are not carried over into the definition of *unit* (3.23).

The Software Requirements Specification DID does not adequately address data units, either as CSCI-internal design entities or as CSCI-external entities with which the CSCI must interface.

DOD-STD-2167 requires that all units be tested (5.4.1.2 and 5.4.1.6). Data units, however, are not tested in the ordinary sense of the word, but rather verified.

In the case of data units, different methods and tool sets may be required (i.e., not simple compilers and linkers). In particular, the specific references to "source code and object code" may not apply to units of data.

Note that this concern about data units applies to all languages, not just Ada.

The definition of unit should be generalized to the point that units may contain no data, contain only data, or any mixture of both.

2.2.3 LOGICAL VS. PHYSICAL UNITS

The distinction between logical and physical units is ignored in DOD-STD-2167. There is no guarantee that the logical design will (or should) map one-to-one into the physical design. In fact, the mapping from the smallest logical design entities into the physical program units is strongly methodology- and language-dependent.

The static hierarchy is a logical structure. Because the entire static hierarchy is based on the "composed of" relationship and because the higher levels of the hierarchy (e.g., TLCSC and LLCSC) are logical, the concept of unit in DOD-STD-2167 should also be logical only.

When code is manually optimized, one method of reducing overhead is to map the logical design into a physical (i.e., code) design. For example, logically concurrent units may be serialized and called as procedures. Logical units may also be inserted "inline" to eliminate subroutine calls. Another instance in which the logical and physical designs are not identical is when a unit is repeated in separate overlays (i.e., one logical copy maps into multiple physical copies). Thus, the final, physical units may not be the same as the originally designed logical units.

It appears that DOD-STD-2167 requires one to document the logical units in the Software Detailed Design Document and Software Development Files (5.3.1.2, 5.3.1.8, 5.3.2.3) and to update the logical design to ensure compatibility with the physical design (5.4.1.9) only in the case where the incompatibility is due to unit testing. There appears to be no requirement to document the physical design when it differs from the logical design for other reasons, nor does there appear to be any requirement to document the mapping between logical and physical units.

By forcing the logical and physical designs to be identical as documented (i.e., by documenting only the as-built design), one loses the original design and the rationale behind it. Just as having the individual pieces of a jig-saw puzzle does not immediately show you the entire picture, the listings of the as-built software (a part of the Software Product Specification) do not clearly show the physical software architecture. The as-built design therefore may not be adequately documented.

Note that this concern is language independent, and in fact, may be even more important when using lower-level languages.

2.2.4 UNIT TESTING

DOD-STD-2167 currently has an inappropriate emphasis on unit-level testing. In light of the high modularity and low unit-level complexity of well-designed Ada software, a requirement to individually test each Ada programming unit seems inappropriate and wasteful. For example, a relatively monolithic Fortran unit may correspond to (i.e., have the same function, size, and complexity as) several Ada programming units. It may not be cost effective to individually unit-test all (or even most) Ada programming units if they consist of little more than a short sequence of calls and associated exception handling. Thus for example, one may wish to *unit* test an entire package rather than individually test each procedure and function nested within it.

The appropriate level at which to commence unit-level testing is language, method, and design dependent.

Although DOD-STD-2167 does not define "unit testing", it seems to mean the testing of individual "2167" units (5.4.1.2, 5.4.1.6) rather than either the testing of individual Ada programming units or unit-testing in the classic sense. This is consistent with the definition of unit (3.23) which implies that each unit may be "tested independently of other Units." It is thus unclear exactly what DOD-STD-2167 requires as far as unit testing is concerned. Even if the intent is to allow the contractor to determine the appropriate level at which to commence unit-level testing, there is a concern that the program office may not take such an enlightened view.

According to paragraph 5.2.1.6 (a) of DOD-STD-2167 Revision A:

> "A Unit test shall test an individual Unit or, subject to contracting agency disapproval, a logically related group of Units."

Although the above modification allows the contractor some latitude in determining the level at which to commence unit testing, it is still too restrictive. The contracting agency should disapprove at the level of the SDP or SSPM that documents the contractor's unit test methods - NOT on an individual test-by-test basis. Unit testing is supposed to be informal, and allowing contracting agency disapproval of unit testing on a case-by-case basis is an improper "how-to" constraint for the following reasons:

1. Unit testing is a contractor-internal activity and the contractor should be allowed to manage this without contracting agency interference. This violates the DoD Acquisition Streamlining directive.
2. DOD-STD-2167A allows the contractor this control when Software Development Files (SDFs) are concerned. No contracting agency approval is required to use SDFs to document multiple units.
3. On large programs, units tend to be designed, coded, reviewed, and tested in logical groups. Previously negotiated contract costs may well be based on the contractor's past experience with group testing of units.
4. DOD-STD-2167 is unclear as to whether each individual unit is required to have its own unit test plan, procedure, and report or whether a generic plan and procedure is allowed.
5. The requirement to prepare unit test procedures is not cost-effective for most unit-testing. Test cases are usually sufficient. For example, on one large Ada project (AFATDS), it was determined that unit testing was not cost-effective if the unit had a McCabe's Complexity of 3 or less. Inspection proved quite sufficient.
6. Paragraph 5.3.1.9 implies isolation-testing of all units.
7. Timing and sizing assessments are usually meaningless at the unit level.

Part of this concern may result from the following facts:

1. Unit testing is never defined in DOD-STD-2167.
2. The definition of *unit* in DOD-STD-2167 may well not be the same as that which is meant by *unit* in classic unit testing.

The contractor can NOT avoid unit testing by specifying in the Software Requirements Specification (10.2.6.1.d) that the unit test method may be inspection. Inspection is one of the "qualification methods used to show that the requirements of a CSCI have been satisfied". This is not the purpose of unit testing and therefore irrelevant to the issue of unit testing.

The argument that the Software Test Plan specifies the type of unit testing to be performed does not answer this concern for the following two reasons:

1. Informal testing (e.g., unit testing) is beyond the scope of the Software Test Plan (STP).
2. The contractor can not arbitrarily group design entities for unit testing without identifying that collection as a DOD-STD-2167 unit or, according to 5.2.1.6 (a) of DOD-STD-2167 Revision A, being "subject to contracting agency disapproval".

The question is not whether the extensive independent testing of each unit mandated or implied by DOD-STD-2167 is the best current method of low-level testing , but rather whether any single method should be mandated or made the default (i.e., preferred). Even the best current method is subject to obsolescence in a rapidly evolving industry. By singling out any specific method in the standard, one is ensuring that the standard will become obsolete sooner than is necessary. Because the standard can not be easily and rapidly updated, this will have a major negative impact on the development of DoD software.

According to the forward of DOD-STD-2167 Revision A, "The intent of this standard is to permit any systematic, well-documented, proven software development methodology."

Informal testing must be controlled by the contractor in order to minimize cost and schedule risk. The contracting agency should only be concerned with formal testing against specifications. The requirement for independent testing creates improper "how-to-manage" constraints on the contractor in violation of the DoD Acquisition Streamlining directive.

While the contracting agency should stipulate the required reliability of the delivered software, it is the contractor's, and not the government's, responsibility to define how software should be tested.

2.2.5 NESTED UNITS

There is currently no requirement to document subunits nested within a DOD-STD-2167 unit. In spite of the problems listed above in paragraph 2.2.1 of this paper, some contractors choose to implement and document an Ada package containing many procedures, functions, tasks, and other packages as a single DOD-STD-2167 unit. Thus one could argue that, by the definition in DOD-STD-2167, the "push" and "pop" routines in a stack package are in reality nested subunits. In addition, it is frequently useful during coding to introduce nested subunits that logically encapsulate some functionality unique to the unit (e.g., to code a PDL statement as a nested procedure). In this case, the fact that the unit contains nested (Ada) subunits may not be documented.

Although some have stated that nesting is a poor practice that should be prohibited, others feel that it is a legitimate way of implementing information hiding. Because it is a legal part of the Ada language and not prohibited by DOD-STD-2167, the standard should be able to handle it. Use of the Ada `IS SEPARATE` clause also invalidates many of the arguments raised against nesting.

Note that this concern also applies to other languages (e.g., Pascal, PL1, JOVIAL) that allow the nesting of subunits.

2.2.6 VISIBILITY BETWEEN UNITS

There is currently no requirement to document the visibility relationship between DOD-STD-2167 units.

If a contractor chooses to consider the nested contents (e.g., subfunctions and tasks) of a package to be separate DOD-STD-2167 units, the visibility and scope relationships between these units are not required to be documented and would therefore not be obvious to maintenance personnel.

It would be useful to have a requirement that could, for example in Ada, be satisfied by including in the Software Detailed Design Document the WITH and USE clauses as part of the PDL (something that

does not appear to be required now). This critical design information should be included in the SDDD as well as in the source code.

Note that this concern also **applies to other languages** that allow the nesting of subunits (e.g., PL/I) although the problem is more critical with Ada. The need to document and limit scope and visibility should be addressed for all languages.

Part of the confusion is due to the differences between DOD-STD-2167 *unit*s and programming *unit*s.

2.2.7 LLCSC AND UNIT DISTINCTION

The distinction between LLCSC and Unit needs clarification. There is currently no requirement for a contractor to define or document his criteria for determining that a design entity is a unit rather than a Lower-Level Computer Software Component (LLCSC). There is also no clear guide in determining how low to take the design since DOD-STD-2167 units may not map clearly into language-specific programming units. Although paragraph 4.2.1 of DOD-STD-2167 states that Appendix XVII of MIL-STD-483 gives guidelines for selecting CSCIs, TLCSCs, LLCSCs, and Units, the guidelines it contains are primarily intended for CSCIs and do not contain much that is useful for determining CSCs and Units. As a *reductio ad absurdum* example, there appears to be nothing in DOD-STD-2167 or the DIDs to prevent a contractor from circumventing much of the standard by arbitrarily having each TLCSC consist of a single LLCSC consisting of a single unit. This problem of "deciding when to stop" is inherent throughout the development process.

The size and nature of LLCSCs and units are both application, software development method, and language dependent.

Note that unlike the language-independent coding standard, the new proposed Ada coding standard (Appendix D) of DOD-STD-2167 does not have a size restriction on units.

2.3 FORMAT EXAMPLES IN THE DIDS

None of the formats specified in the DIDs show how to use Ada to present information and there are no examples of how to optimally present such information. This is inconvenient at best because Ada will be the primary language for future software development. Some contractors are currently experiencing great difficulty applying the DIDs when Ada is used as either the PDL or implementation language.

Ada is a design, as well as an implementation, language. It is therefore reasonable to use Ada to present the design information required by the Software Detailed Design Document and possibly the Software Top Level Design Document. The Data Item Descriptions (DIDs) should encourage the presentation of design information in the form of either Ada PDL or full Ada.

Ada examples should be used either in addition to, or in place of, existing examples (e.g., tables). Benefits of this would be to:

1. Bring DOD-STD-2167 into compliance with DoD policy. The letter of June 10, 1983 from Under Secretary of Defense DeLauer states in part "... use of the Ada programming language is actively encouraged."
2. Decrease errors by:
   (a) Increasing the rigor of the information presented so that it is less subject to interpretation.
   (b) Enabling automatic verification of syntax and some semantic information using an Ada compiler.
   (c) Promoting consistency and uniformity between code and the documentation.
   (d) Reduce the effort to produce code and documentation.
3. Improve the understandability, readability, and maintainability of the code and documentation.

The DIDs inhibit the automation of documentation production by suggesting formats that do not easily map into the way relevant information is stored at the code level.

As mentioned previously, a strong precedent already exists for the extensive use of the Ada language in military standards. Both "Internet Protocol" (MIL-STD-1777) and "Transmission Control Protocol" (MIL-STD-1778) use a subset of Ada constructs

common to most high-level languages to declare data structures, etc.

Because Ada will be used on all future DoD computer systems, it will be advantageous to define all fields and messages in the interface control and design specifications as Ada data type specifications. This would allow the Ada programs in the interfacing systems to directly `with` and reference the interface data definitions into the code and thus be assured that the expected interface data exactly matches the ICD/ISD specification. When the interface data definitions are changed, the Ada compiler and linker will automatically declare "obsolete" all compilation units that reference the changed data items until the programs are modified (if necessary) and recompiled. This greatly enhances the configuration control of the interface specifications. Compiling these portions of the interface specifications would also be a significant validation of their correctness. When interfacing to hardware devices or between different target processors whose compilers allocate the underlying bit representations differently, Ada representation specifications should be included as part of the data definitions.

This concern should also be addressed in "Defense System Software Development Handbook" (DOD-HDBK-287).

2.4 USE OF ADA IDENTIFIERS

DOD-STD-2167 does not adequately support identifier consistency. There appears to be no requirement that the same entity always have the same identifier, both in the documentation and code. The use of consistent Ada identifiers would enhance the traceability of entities from requirements through design into the resulting code.

3 PROCESS VS. PRODUCT STANDARD

According to the Foreword of DOD-STD-2167, it is a process standard that "establishes a uniform software development process". By mandating a standard life-cycle and activities based upon the phases of this life-cycle, DOD-STD-2167 contains a significant number of "how to" requirements.

Process standards must, by definition, contain "how to" restrictions and inhibit innovation. It is not proper for the government to mandate how the contractor is to develop software. Software development methods that significantly deviate from the defined process may not even be proposed, regardless of their technical merit, because of the perceived political and economic risks to the contractor that proposes anything different than expected. In a rapidly evolving industry in which advances in software engineering come almost daily and in which major improvements are necessary in order to solve the software crisis, innovation is necessary and should be promoted. In fact, many Requests for Proposals (RFPs) state that an innovative methodology is favorably considered in the contractor selection criteria.

By its very nature, most of the DoD (certain advanced R&D efforts excluded) will always be several years behind industry, and even further behind the research community. Thus it is vital that the contractor be encouraged to apply the methods most appropriate for producing the product and associated documentation that the DoD needs at the least possible cost, while still providing the government adequate oversight into the contractor's development effort.

Just as contractor personnel must always keep up with a rapidly evolving technology if they are to remain competitive, government personnel must do so also. One hardly expects government personnel familiar only with vacuum tube technology to manage and maintain modern computer systems, and what applies to hardware applies equally to software.

If the contractor does not have the expertise to propose and implement a software development method (either the current default of DOD-STD-2167 or a more modern alternative), then the contractor should not be developing software. If the government is not qualified to evaluate contractor proposed processes, life-cycles, and methods, they should hire an independent expert. After all, this is one of the standard duties of IV&V contractors.

The classic software development process mandated by DOD-STD-2167 has not been proven to work well on large projects. It has the following well-known disadvantages:

1. You never have a demonstrably useful product (i.e., one that validates the requirements and

design) until the end of the development lifecycle (i.e., near the end of a release or project). This is one of the incentives to rush coding without a methodology or adequate preparation.

2. On large projects, the government monitoring personnel are likely to change, leading to different "hot buttons" and large changes in requirements.

3. On large projects, turnover of contractor personnel leads to a loss of why certain key decisions were made.

Any advantages that the government would gain from maintaining a common, single development process, lifecycle, or set of methods throughout a variety of software development projects (e.g., ease of training) would be outweighed by the inhibition of innovation that would result.

The ability to propose an alternate software development process, life-cycle model, or methods in the Software Development Plan (SDP) and thus ignore all or part of DOD-STD-2167 is an insufficient loophole since contractor's may well feel pressured to comply with the standard in order to win the contract.

A default software development process and lifecycle model is NOT necessary because DOD-STD-2167 (5.1.1.3.c.1) already requires the contractor's proposed software development methods and techniques to be documented in the Software Standards and Procedures Manual (10.2.5.1). Although not mentioned in DOD-STD-2167, the exact same requirement is also REDUNDANTLY stated in the Software Development Plan (10.2.7.1.1).

It is clearly the contractor's, and not the government's, responsibility to define how software should be developed and what life-cycle and software development methods should be used. When a contractor does not propose any process, life-cycle, or methods, the SSMP and SDP should be rejected rather than mandating a single, standard process and life-cycle that may well not be appropriate.

DOD-STD-2167 is very much a "how-to-develop" process standard, and is therefore in gross conflict with DoD acquisition streamlining directives. Specifically, the first sentence of the second paragraph of DODD 5000.43, "Acquisition Streamlining", states:

> "As a first priority, this Directive establishes policy for streamlining solicitation and contract requirements by: (a) Specifying contract requirements in terms of the results desired, rather than "how-to-design" or "how-to-manage"...."

All requirements regarding the performance and documentation of contractor-internal activities (e.g., informal testing) are improper "how-to" constraints on the contractor and may well increase acquisition costs and schedules without justifiable benefit. It is time that such "how-to-design" and "how-to-manage" requirements are deleted from DOD-STD-2167.

One should note that often the "process" requirements found in "Activities" sections are really product requirements and are nothing more or less than the REDUNDANT reiteration of the contents of the associated DID. Thus, we have the same product requirement redundantly mandated on the customer three (!) times: once in the activities section, once in the product section, and once in the DID. This creates an unnecessary CM problem for the government and an unnecessary QA problem for the contractor. These redundant parts of DOD-STD-2167 should be replaced with references to the applicable DID.

While it may be reasonable for the government to adopt a process standard to dictate how the government is to PROCURE software, it is something quite different to adopt a process standard to dictate how the contractor is to DEVELOP software.

The question is not whether the software development process, life-cycle, and methods mandated by DOD-STD-2167 are the best currently available, but rather whether any single approach should be mandated or made the default (i.e., preferred). Even the best current approach is subject to obsolescence in a rapidly evolving industry.

By singling out any specific process, life-cycle, or set of methods in the standard, one is ensuring that the standard will become obsolete sooner than is necessary. Because the standard can not be easily and rapidly updated, this will have a major negative impact on the development of DoD software.

According to the forward of DOD-STD-2167 Revision A, "The intent of this standard is to permit any

systematic, well-documented, proven software development methodology."

3.1 "WATERFALL" LIFE-CYCLE

DOD-STD-2167 restricts the contractor to software development methods consistent with the classic "waterfall" life-cycle.

According to DOD-STD-2167 (4.1), "The contractor shall implement a software development cycle that includes the following six phases...". By basing DOD-STD-2167 upon a single life-cycle (i.e., the classical "waterfall" model), innovation and methods based upon alternative life-cycles are prohibited.

Many new life-cycle models have been introduced during the last five years and others will continually be introduced as software engineering evolves. Notable examples of other methods having life-cycles prohibited by DOD-STD-2167 include rapid prototyping methods, recursive object-oriented development methods, and specific life-cycle models such as Boehm's spiral life-cycle.

Many new life-cycle models are fundamentally different from the classic waterfall life-cycle. They can not be mapped into the DOD-STD-2167 life-cycle due to close binding of specific products developed and reviewed during the individual phases of the DOD-STD-2167 life-cycle.

Requiring conformity to a single standard life-cycle model is an improper "how to" restriction placed on the contractor.

Although DOD-STD-2167 allows incremental reviews, the linear nature of the classical life-cycle with its formal reviews that act as bottlenecks between phases (4.1.2) prohibits the use of recursive "design a little, code a little, test a little" methods. Thus for example, although DOD-STD-2167 permits a small number of incremental Preliminary Design Reviews (PDRs) and Critical Design Reviews (CDRs) per CSCI per build or release, it does not permit methods such as Object-Oriented Design in which small amounts of code (e.g., approximately 1KLOC) are recursively designed, coded, and tested during each pass through the method. On large projects (e.g., more than 100KLOC), it is clearly impractical to hold several hundred traditional PDRs and CDRs. The bottleneck nature of the formal reviews also prohibits one from coding and testing as one goes – an important aspect of such methods that permits one to incrementally validate the evolving design.

3.2 STATIC HIERARCHY

The static software hierarchy of DOD-STD-2167 does not map well into the network structure of well-designed Ada software. There is no clear method-independent way of defining the terms of the static hierarchy of DOD-STD-2167 in terms of the structuring concepts of Ada (e.g., packages, nesting, withing).

The static software hierarchy is tied too closely with the software development process, prohibiting one from first developing the proper Ada structure and only then decomposing it into a static hierarchy for purposes of Software Configuration Management.

The static software hierarchy of DOD-STD-2167 implies a hierarchial-decomposition software development method, thus inhibiting the use of more modern non-decomposition methods based on a recursive "design a little, code a little, test a little" approach.

The software hierarchy of DOD-STD-2167 impacts the order and scope of integration and testing. The order and scope of integration and testing, however, should be method-, language-, and architecture-dependent.

The static structure can inhibit software reusability, and it often conflicts with the use of generics in Ada.

It is not at all clear what the static structure represents if not the structure of the implemented code. Since CSCIs are composed of TLCSCs which are composed of LLCSCs which are composed of Units, it only seems reasonable that this structure, which is based upon various levels of software groupings, should represent the software architecture in some natural way. It does not make sense to say that the code "resides" only at the unit level and therefore the static structure has no relationship to the software structure. Otherwise, the static structure is highly misleading and the software design documents should not be based on it.

The current static structure is valuable for Software Configuration Management purposes, and NOT for purposes of design in an engineering sense. Yet the whole design process of DOD-STD-2167 is based on the static hierarchy.

3.3 "TOP-DOWN" DEFAULT

The choice of "top-down" as the single default development approach implies that it is the preferred approach for all software development activities.

A clear distinction should be made between the use of the term "top-down" to describe a contractor's software development method and the use of the term to describe documentation formating (e.g., document structuring and presentation).

DOD-STD-2167 currently states (4.8) that "The contractor shall use a top-down approach to design, code, integrate, and test all CSCI's unless specific alternate methodologies have been proposed ... and received contracting agency approval." This requirement is therefore an improper "how to" restriction on the contractor.

The appropriateness of "top-down", "bottom-up", "outside-in", "inside-out", or "holistic" approaches is method- and activity-dependent.

Examples of situations where other approaches appear preferable include:

1. Extensive reuse often implies a "bottom-up" approach to design and test.
2. Use of commercial software implies "bottom-up" testing.
3. The compilation order restrictions of Ada encourages a "bottom-up" approach to CSC testing.
4. The development of critical software implies "bottom-up" design and testing.
5. The development of test suites requires at least a partial "bottom-up" approach.

The application of the "top-down" approach to activities other than design (e.g., code, integration, and testing) is not a recognized, proven approach.

Software is almost guaranteed not to be reusable if one uses a pure top-down design method because one of the major goals of top-down design methods are to produce units precisely suited to the specific problem being solved. Thus, the requiring of top-down is counterproductive to the goal of producing reusable software.

Even DOD-STD-2167 is not consistent with regard to its own requirement that the "contractor shall use a top-down approach to ... test all CSCI's". It specifically requires an approach to test and integration that is absolutely, vertically bottom-up as is clearly demonstrated by paragraph 4.1 which lists the last three sequential phases of the DOD-STD-2167 software development life-cycle as Coding and Unit Testing, Computer Software Component Integration and Testing, and CSCI Testing.

Because no single approach is clearly optimal for all life-cycle activities, choosing "top-down" as the single, default (and therefore preferred) approach appears counterproductive.

The requirement of "top-down" as a default approach is an improper "how-to" restriction on the contractor who may feel pressured to provide the expected, "preferred" approach in order to win the contract.

3.4 PDL REQUIREMENT

The choice of a Program Design Language (PDL) as the single required detailed design tool and default top-level design tool causes various problems.

3.4.1 "HOW TO" CONSTRAINT

The choice of PDL as the single default top-level design technique and the required detailed design technique is an improper "how to" constraint.

DOD-STD-2167 (5.2.1.4) states that "In establishing and defining the top-level and, as applicable, lower-level design of each CSCI, the contractor shall use a program design language or some other top-level design description tool or methodology."

DOD-STD-2167 (5.3.1.5) states that "In the development of the detailed design for each CSCI, the contractor shall employ a program design language."

The choice of any single design technique as default or requirement inhibits contractor innovation and will result in the proposal and use of technically inferior tools due to the natural tendency of contractors to develop designs in the manner expected by the government. The contractor should be free to choose the best detailed design description technique for the language and application. This is an improper "how

to" constraint.

3.4.2 PDL INAPPROPRIATNESS

The use of a PDL as a top-level design description technique is probably inappropriate. Graphics, such as those of Grady Booch and R. J. A. Buhr, are clearly superior in terms of understandability when it comes to presenting the top-level architectural design in terms of software units and their relationships. The use of a higher-level notation should be explicitly encouraged.

The early use of PDL also tends to force the designer to prematurely think in terms of execution sequences when all of the entities may not yet be fully understood.

3.4.3 PDL PURPOSE CHANGING

Due to the high modularity and low complexity of well-designed Ada software, the lack of distinction between Ada PDL and Ada code, and the design aspects of the Ada specification, the nature and purpose of PDL is changing in the Ada community. PDL is not needed to document the logic of the body of many units because of their trivial size and complexity. Although PDL may prove useful in the automatic generation of the Software Detailed Design Document, it is inappropriate to imply that it, or any single detailed design method or tool, is to be preferred under all circumstances. This inhibits innovation and is an improper "how to" constraint on the contractor.

Graphical methods (e.g., the use of decision tables or finite state transition diagrams) are probably better than PDL for specifying the design of logically complex units.

3.5 FUNCTIONAL ORIENTATION

DOD-STD-2167 seems to imply a functional decomposition method of software development.

3.5.1 FUNCTIONAL SRS

The format of the Software Requirements Specification (SRS) seems to imply a functional decomposition method of software requirements analysis.

Many real-time systems are data driven or processing sequence driven. Yet DOD-STD-2167 is oriented almost exclusively towards functionally driven systems. Data structure and the sequencing of operations are inadequately identified and tracked.

The format of the Software Requirements Specification (SRS) is based upon, and implies, a functional decomposition method of software requirements analysis. This causes the following problems:

1. Organizing the requirements along object or hardware, rather than functional, lines is not permitted.
2. If one uses an object-oriented requirements analysis method, one must re-sort the requirements from object into functional order. This results in excess work and a SRS whose structure does not map well into the design.
3. If one uses a functional decomposition requirements analysis method and an object-oriented design method, one encounters method incompatibility problems. One must somehow ignore the functional decomposition design implied by the functional requirement decomposition, something that is psychologically very difficult to do in practice.

The increased readability of a document due to its having a standard format probably does not justify the use of a standard format if the format is inappropriate and adversely impacts the quality of the software. Besides, the table of contents should always allow one to find the relevant portions of the document, and the completeness of the document can always be verified by means of a cross-reference matrix that maps the contractor's format to the content requirements of the DID.

3.5.2 FUNCTIONAL STATIC HIERARCHY

The entities of the static hierarchy are oriented towards functional decomposition software development methods.

According to paragraph 3.7 of DOD-STD-2167, a Computer Software Component is a "*Functional* or

logically distinct part of a computer software configuration item."[1] Although the above wording does allow for the decomposition of CSCIs into TLCSCs and TLCSCs into LLCSCs using other than functional methods, the implication of both the word "functional" and its order (i.e., functional precedes logically) is that functional methods are the preferred default. This represents an improper, if weak, "how to" constraint on the contractor.

According to paragraph 3.19 of DOD-STD-2167, the definition of "Modular" implies that the software "is organized into limited aggregates ... that perform identifiable *functions*." This represents another improper "how to" constraint on the contractor.

According to paragraph 3.27 of DOD-STD-2167, a unit "... describes a single *function* ..." and according to paragraph 5.3.1.2 of DOD-STD-2167, "Each Unit shall perform a single function." The above requirements imply a functional decomposition design method. They ignore the existence of data units and do not allow one to have abstract data type packages (which perform multiple functions) as units. This is a strong improper "how to" constraint on the contractor.

According to paragraph 4.2.1 of DOD-STD-2167, "the partitioning of the CSCI into TLCSCs, LLCSCs, and Units may be based on *functional* requirements, data flow ..." Although the wording does allow for the decomposition of CSCIs into TLCSCs and TLCSCs into LLCSCs using other than functional methods, the implication of both the word "functional" and its order (i.e., functional is mentioned first) is that functional methods are the preferred default. This represents an improper, if weak, "how to" constraint on the contractor.

According to paragraph 5.2.1.2 (b) of DOD-STD-2167, "In defining each TLCSC the contractor shall identify ... [the] *function* allocated to the TLCSC". This implies a functional decomposition design method and ignores the existence of data TLCSCs such as knowledge bases in AI applications. Requirements, rather than functions, should be allocated to TLCSCs. This is a strong improper "how to" constraint on the contractor.

The implication of "functional decomposition" as the default approach is an improper "how-to" restriction on the contractor who may feel pressured to provide the expected, "preferred" approach in order to win the contract.

The question is not whether "functional decomposition" is the best current method, but rather whether any single method should be mandated or implied to be the default (i.e., preferred). Even the best current method is subject to obsolescence in a rapidly evolving industry. By singling out a specific method in the standard, one is ensuring that the standard will become obsolete sooner than is necessary. Because the standard can not be easily and rapidly updated, this will have a major negative impact on the development of DoD software.

The contractor should be free to:

1. Modularize the software along other than functional lines.
2. Determine the appropriate decomposition method for the application and software development method.
3. Determine the appropriate method and criteria for identifying TLCSCs, LLCSCs, and Units.

Because no single type of software development method is clearly optimal for all applications, choosing functional decomposition as the single implied default (and therefore preferred approach) appears counterproductive.

3.6 DOD ACQUISITION PROCESS

DOD-STD-2167 is based upon a DoD acquisition process inappropriate for the development of software using modern methods.

The basic process mandated by DOD-STD-2167 and the remaining standards is based upon the DoD Acquisition process which was historically developed to support hardware and systems acquisition rather than software acquisition. As a consequence, the basic life-cycle and review process is not necessarily consistent with the most modern ways of developing software. This becomes especially important in light of the DoD's recent realization of the important impact of software development upon systems development.

This is a very complex subissue that could require years to properly study and resolve.

[1] Emphasis added here and below by the author.

4 METHOD-SPECIFIC OMISSIONS

Although the previous section documents a strong aversion to "how to" constraints, many members of the Ada Community felt that DOD-STD-2167 lacked sufficient requirements concerning prototyping, reuse, and automation of the development process.

4.1 PROTOTYPING

DOD-STD-2167 does not adequately address software prototyping.

A very important and productive technique in every engineering field is the use of prototypes, mock-ups, models, and breadboards. The usefulness of building and testing models and prototypes, in addition to producing a "paper" design, has long been recognized. Only in this way can one verify the feasibility of the design. Perhaps the use of prototypes is one of the reasons why hardware engineering has advanced beyond software engineering.

Rapid prototyping life-cycles do not map well into the classic waterfall life-cycle of DOD-STD-2167.

Although the value of software prototyping is largely language independent, compilable Ada PDLs and Ada language features (e.g., separate compilation) facilitate the production of software prototypes.

The scope of MIL-STD-1521 includes the review and audit of both hardware and software, yet these two branches of engineering are treated differently. MIL-STD-1521B (40.2.1 r) covers the hardware items to be reviewed during Preliminary Design Review and lists "Mock-ups, models, breadboards, or prototype hardware when appropriate." MIL-STD-1521B (40.2.2) makes no mention of software prototypes. The same applies to Critical Design Reviews (50.2.1 and 50.2.2).

According to page 21 of the Joint Regulation:

> "4.2.3 Development of Prototype Computer Resources. ... Software may be developed to demonstrate critical algorithms, control sequences, timing, operator interfaces, etc. ..."

Some mention of prototyping during Requirements Analysis or Preliminary Design would be useful for assessing key algorithms, Man-Machine Interfacing, scenarios, etc.

4.2 REUSE

DOD-STD-2167 needs to address both the production of reusable software and the use of reusable software. It does not adequately address software reuse, nor does it sufficiently promote reusability in a practical way. Isolated references to the importance of reusability are insufficient.

DOD-STD-2167 does not supply an adequate definition of "reuse". Any requirements for reuse in DOD-STD-2167 should be based on such a definition, a definition general enough to include reusable requirements, designs, architectures, test software and data, etc.

DOD-STD-2167 contains the tacit assumption that all software in a system will be built from scratch for that system. This thwarts one of the major advances of Ada, namely the production and use of libraries of reusable software.

If the DoD's goals for significant code reuse are to be met, an automated means of identifying needed preexisting Ada software is necessary. DOD-STD-2167 can support this by ensuring that all code developed has been marked for easy identification (e.g., through the use of mandatory keyword and abstract fields in the prolog of all units).

Some degree of reuse may be contractually required. Where are the reuse requirements to be specified and how are they to be verified? Where is the quality, applicability, and functionality of reuse candidates to be specified? If reuse candidates prove to be inappropriate, is a waiver procedure needed?

The contracting agency should not simply direct that a system or component is to be reusable or as reusable as is possible or practical. Such a requirement is neither objective nor testable. Components are not absolutely reusable, but rather reusable to a certain degree that is application dependent.

The process described by DOD-STD-2167 seems more appropriate for the development of large applications rather than the development of a reuse library consisting of many small separate unrelated

programs. It is not yet clear how DOD-STD-2167 must be modified to be applicable to the development of reuse libraries.

DOD-STD-2167 does not appear to have incorporated any of the lessons learned from the STARS Reuse Committee.

Because reuse is an area where good tailoring will make a great difference, DOD-HDBK-287 should be updated with significant material on reuse.

A requirement to make reuse and reusability an integral part of all relevant software development activities can always be tailored out if not appropriate.

Paragraph 4.4.d of DOD-STD-2167 inhibits reuse because it implies that contracting agency approval is needed if one wishes to use reuseable software. Sufficient checks and balances are already achieved by 4.4 a, b, and c.

4.3 AUTOMATION

DOD-STD-2167 does not sufficiently promote automation.

In order to increase the efficiency of the software development process and to increase the quality of the resulting software and documentation by reducing human error, significant portions of the process need to be automated. This includes, but is NOT limited to, the production of documentation.

When significant portions of the life-cycle are automated, what effect does this have on the description of required activities and the associated reviews?

The DIDs inhibit the automation of documentation production by requiring, or suggesting via examples, formats that do not easily map into the way relevant information is stored at the code level.

The DIDs should be reviewed for compatibility with the National Bureau of Standards Information Resource Dictionary which defines entities and relationships at four levels of data abstraction and which describe the information which can be used to automatically generate the DIDs.

5 OTHER ISSUES

Various other issues were also raised by members of the Ada Community.

5.1 FORMAL REVIEWS

Many comments were made regarding the formal reviews of DOD-STD-2167 and MIL-STD-1521.

5.1.1 AMOUNT OF MATERIAL

The size and complexity of today's systems overwhelms the current formal review process. It is not humanly possible to properly perform technical reviews of manually-produced "gothic novel" sized specifications. There is often insufficient time for a proper in-depth analysis and the correction of errors found. The reviews tend to concentrate on superficial formatting problems while important technical issues become buried. The forest gets lost for the trees.

By having more reviews and limiting the scope of any single review, a better analysis results for the following reasons:

1. Smaller documents and partial documents are easier to review. There is less reviewer fatigue and the tail end of the documents will be reviewed with the same care as the front end. The tail end of larger documents often "slides by" due to reviewer fatigue, lack of time, etc.

2. Because smaller documents and partial documents can be prepared with less lead time, they will be more current when reviewed.

3. Because smaller documents and partial documents take less time to produce and review and have a more narrow scope, a small percentage of the project's personnel grind to a shorter stop.

4. Having a larger number of smaller reviews makes each single review less important. By becoming part of the (almost weekly) development activities, the developers are less impacted by "non-productive" work and the "dog and pony show" atmosphere is reduced.

5. If any "show stoppers" are discovered, they will likely be limited in scope and result in holding up the development process for a shorter period. For example, corrections can be processed in a recap session. A serious error will also tend to invalidate much less of the work that would have ensued prior to a more major review.

6. Major process problems will show up earlier, when they will be easier and less expensive to correct.

7. Replacing a single massive review with a sequence of smaller reviews permits better contractor man-power leveling by overlapping the requirements analysis, design, and coding of separate elements. The same advantages offered in DOD-STD-2167 now for the separate review of different CSCIs and incremental reviews (e.g., for each build or release) would also result if applied to smaller, relatively independent "chunks" of software (e.g., those resulting from each recursion of the Object-Oriented Design process).

5.1.2 EVOLVING CDR

MIL-STD-1521 and DOD-STD-2167 do not account for the evolving nature of the CDR.

Due to the high modularity and low complexity of well-designed Ada software, the lack of distinction between Ada PDL and Ada code, and the design aspects of the Ada specification, the classic purpose of the CDR (i.e., to review and approve unit-internal logic prior to coding) is no longer relevant. Coding the Ada specification is a design activity. PDL is not needed to document the logic of the body of many units because of their trivial size and complexity. One should go ahead and code the body once started since it involves little added work and allows one to use the compiler to partially check the results prior to any (semi)formal review. By performing the unit testing immediately, one can also validate the design as one goes.

With the use of the same language for both design and implementation (e.g., Ada), there exists a very real non-trivial problem of defining what is design and what is code. This has a very real impact on determining the scope of the CDR as currently defined.

By requiring at CDR and prior to coding and unit test, a formal review of the "detailed design", one is prohibiting the contractor from using RECURSIVE software development methods that result in the hierarchical top-down design, code, and test of very small amounts of software (e.g., approx. 1KLOC). This is a very major and improper "how to" constraint on the contractor.

5.1.3 LINEAR NATURE OF REVIEWS

The linear nature of the formal reviews is an improper "how to" constraint on the contractor.

Although DOD-STD-2167 allows incremental reviews, the linear nature of the classical life-cycle with its formal reviews that act as bottlenecks between phases (4.1.2) prohibits the use of recursive "design a little, code a little, test a little" methods. Thus for example, although DOD-STD-2167 permits a small number of incremental PDRs per CSCI per build or release, it does not permit methods such as Object-Oriented Design in which small amounts of code (e.g., approximately 1KLOC) are recursively designed, coded, and tested during each pass through the method. On large projects (e.g., >100KLOC), it is clearly impractical to hold several hundred traditional CDRs and PDRs. The bottleneck nature of the formal reviews also prohibits one from coding and testing as one goes – an important aspect of such methods that permits one to incrementally validate the evolving design.

Because all methods do not produce the same intermediate products in the same order, the scope of the current reviews is sometimes inappropriate.

The timing and the scope of the results of certain design activities are set by the timing and nature of the formal reviews. This is an improper "how to" constraint on the contractor.

5.1.4 INCONSISTENT PURPOSES

The reviews have too many inconsistent purposes (e.g., finding errors, educating customer, obtaining customer input, obtaining customer approval, etc.). Specifically, some people felt that the formal reviews have both a management and technical nature that is often contradictory.

Note that it may be useful to have a single, formal

management-oriented review at the end of a series of less formal, technically-oriented CSCI reviews. Such a summary review would serve as a major scheduling milestone and provide a forum for management to get summary feedback from the technical personnel.

5.1.5 SIMULTANEOUS REVIEWS

Not all CSCI's are designed, coded, or tested at the same time, yet DOD-STD-2167 implies that all CSCI's must be reviewed at the same time.

Note that it may be useful to have a single, formal management-oriented review at the end of a series of less formal, technically-oriented CSCI reviews. Such a summary review would serve as a major scheduling milestone and provide a forum for management to get summary feedback from the technical personnel.

5.2 CONTENT EXAMPLES IN THE DIDS

Certain DID examples imply the requirement to document irrelevant information.

Some content examples are not relevant. Examples of machine representations and actual storage location of data are often no longer applicable for software written using a good High-Level Language (HLL).

In numerous places thoughout DOD-STD-2167, monitoring of memory size and processor time allocation is mandated. In modern software technology, these items are frequently neither a critical or meaningful measure of the software. For example, in a virtual memory system, memory and address space is huge and the 'channel' for transfer in/out of 'real' memory is the critical resource. Many other examples could be given. The correct requirement should be to:

1. Identify the critical resources,
2. Allocate those resources based on the known limitations, and
3. Monitor, as soon as is feasible and realistic, the utilization of these resources.

One should only have to document the relevant resources.

The DIDs seem to require the documentation of useless information or information that can be derived from the use of Ada as a PDL.

The content shown in the DID examples may be irrelevant in some cases, and information not depicted may be extremely useful in other cases. It is the contractor's responsibility to provide the relevant information in a meaningful format.

5.3 DOCUMENTATION OF CONCURRENCY

Nowhere is there a designated place to document the issues of concurrency associated with a particular software component.

Traditionally, an entire CSCI, or at least a TLCSC, would run on a single CPU, but with the existence of Ada and the increased use of distributed architectures, this premise no longer holds. It is now not unusual for a CSCI to consist of many parallel threads of control.

There is no requirement to document the concurrency architecture of a TLCSC, LLCSC, or unit. One needs to identify the roots of each thread of control, communication among concurrent entities (e.g., Ada tasks), and scheduling characteristics (if unusual or beyond the semantics of the implementation language).

A "Concurrency Features" paragraph for TLCSCs should be added to the Software Top-Level Design Document DID, and similar paragraphs for LLCSCs and Units should be added to the Software Detailed Design Document DID.

6 CONCLUSION

It is hoped that the above information will be useful to those who must develop Ada software under DOD-STD-2167. It may be used as either tailoring guidelines or as a warning of possible pitfalls to be managed.

The situation with regard to DOD-STD-2167 Revision A offers considerably more hope to the Ada Community. The initial draft of Revision A released for formal industry and government review during the Fall of 1986 successfully answered several of the

above listed concerns, notably those regarding the default language-independent coding standard. Due to the results of that review, a second draft Revision A is now being produced for a second formal industry and government review cycle scheduled to begin during April of 1987. This draft is intended to address many (but not all) of the remaining concerns of the Ada community. Most notably, the "top-down" and PDL issues should be completely answered and the majority of the unit concerns should also be addressed. The author, for one, is very pleased with the DoD's current plans and is confident that the second draft will be a major improvement.

MCC '87
Military Computing Conference

CONFERENCE PROCEEDINGS

Disneyland Hotel
Anaheim, California
May 5, 6 & 7, 1987

Published by: EW Communications, Inc.
1170 East Meadow Drive
Palo Alto, California 94303
Telephone: (415) 494-2800

The Military Computing Conference is organized and sponsored by
The Military Computing Institute
P.O. Box 428
Los Altos, California 94023