The Explication of Process-Product Relationships

in DoD-STD-2167 and DoD-STD-2168 via an

Augmented Data Flow Diagram Model

by

Robert Gregory Lavender

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Science and Applications

APPROVED:

Richard E. Nance, Chairman

James D. Arthur

Dennis G. Kafura

March, 1988

Blacksburg, Virginia

The Explication of Process-Product Relationships
in DoD-STD-2167 and DoD-STD-2168 via an
Augmented Data Flow Diagram Model

by

Robert Gregory Lavender
Richard E. Nance, Chairman
Computer Science and Applications

(ABSTRACT)

The research reported in this thesis is an extension and application of the results first introduced by the Procedural Approach to the Evaluation of Software Development Methodologies. The evaluation procedure offers a unique perspective based on the philosophy that a software development methodology should espouse a set of *objectives* that are achieved by employing certain *principles* throughout the software development process, such that the products generated possess certain *attributes* deemed desirable. Further, definite linkages exist between objectives and principles, and principles and attributes.

The work described herein adopts the perspective offered by the evaluation procedure and applies a critical analysis to the process-product relationships in DoD-STD-2167 and DoD-STD-2168. In support of the analysis, Augmented Data Flow Diagrams are introduced as an effective tool for concisely capturing the information in both standards. The results of the analysis offer a deeper insight into the requirements for defense system software development, such that one is able to better understand the development process, and more intelligently assess the quality of the software and documentation produced.

Keywords: Data Flow Diagrams, DoD-STD-2167, DoD-STD-2168, Methodology Evaluation, Software Development Methodology, Software Development Process, Software Quality Evaluation.

Acknowledgements

I wish to thank Dr. Richard Nance, my thesis advisor, for his perfect tutelage, and for requiring of me a professionalism in my character and my work.

I am also grateful to Dr. James (Sean) Arthur for his weekly guidance and the constant encouragement during the times when the ultimate goal was obscured by the details.

To Dr. Dennis Kafura, I extend my thanks for agreeing to serve on my committee.

Finally, with deepest gratitude, I thank the IBM Federal Systems Division for supporting me, through the Systems Research Center of Virginia Tech, during the year of work required to complete this thesis. The interest and patience of was especially appreciated.

Table of Contents

List of Illustrations

1.0 Introduction

The objective of this thesis is to demonstrate the applicability of a data flow analysis, using an *Augmented Data Flow Diagram* model, to a *procedural evaluation* of Department of Defense standards DoD-STD-2167 and DoD-STD-2168 [DODS85a, DODS85b],[1] and selected Data Item Descriptions.

This thesis, and its supporting research, is embodied in part by the statement that "a sound methodology can empower and liberate the creative mind" [BROF87]. This statement, by Dr. Frederick P. Brooks, was made in the context of a discussion about the essence of software engineering and the qualities that constitute a great software designer. Dr. Brooks makes the point that good methodologies are necessary to achieve good software designs, but such methodologies are only as good as the designers that use them. Experienced practitioners of software design can appreciate this fact.

The motivation for this thesis was a result of the experiences and concerns of both researchers and practitioners of software engineering; specifically, the software design experiences of the author, previous research on the procedural approach to the evaluation of software development

[1] The versions of these standards referred to are 4 June 1985 and 26 April 1985, respectively.

methodologies [DANA87], and the concerns and interests of the Federal Systems Division of the International Business Machines Corporation.[2]

With these motivations, the work reported herein attempts to explicate the process-product relationships in DoD-STD-2167 and DoD-STD-2168, and bring to light new information of immediate use to software developers that are attempting to comply with these standards. The explication of the process-product relationships in both standards is achieved by adopting a critical and analytic perspective based on the foundation of a procedural approach to evaluating software development methodologies. The use of a data flow model, in the context of this perspective, yields practical insight into the nature of the software development process defined by DoD-STD-2167 and the collateral nature of the software quality process defined by DoD-STD-2168. The hope is that such information will stimulate creative minds, and result in improvements to software development in the defense software industry.

The following paragraphs outline the major chapters in this work.

Chapter two introduces the reader to background information that is a prerequisite for fully comprehending the remainder of the thesis.

Chapter three discusses the motivation for defining a conceptual tool called the Augmented Data Flow Diagram model, and introduces the syntax and semantics of this model.

Chapter four discusses the application of the Augmented Data Flow Diagram model to an analysis of DoD-STD-2167 and reports the results derived from such an analysis.

Chapter five is a similar discussion of an analysis applied to DoD-STD-2168 in view of its relationship to DoD-STD-2167.

---

[2] The ideas promulgated in this thesis are those of the author and not necessarily those of IBM.

Chapter six outlines the major conclusions drawn as a result of applying the ADFD model to the analysis of these standards, and suggests future research directions based on the conclusions.

Appendix A contains a glossary of useful terms and acronyms that the reader may refer to as needed. The remaining appendices are referenced accordingly in subsequent chapters.

2.0 Background

The intent of this chapter is to introduce the reader to the prerequisite information relevant to understanding the remaining chapters of this work. Concise characterizations of key concepts are provided as an aid to gaining a perspective similar to that acquired during the research that is reported in later chapters. The hope is that, by assimilating this information, the reader can appreciate the basis for this thesis, collaterally share in its discoveries, and subsequently be able to extend these ideas with ideas of his or her own.

To begin, the reader is introduced to a procedural approach to the evaluation of software development methodologies, which will often be referred to as either the "evaluation procedure" or the "procedural evaluation". The concepts presented in this section are a recurring theme. Subsequently, a brief discussion of the role of software development standards is provided. Such a discussion is merited when one wishes to consider not only the *letter* of a standard but also the *intent*. The final two sections of this chapter focus on the natures of DoD-STD-2167 and DoD-STD-2168, Department of Defense standards for software development and software quality evaluation, respectively. The purpose for introducing these standards, prior to their analysis in later chapters, is to establish their importance, and to delineate those aspects that are relevant to the subsequent analysis.

2.1 The Procedural Approach to the Evaluation of Software Development Methodologies

The procedural approach to the evaluation of software development methodologies is based on a philosophical argument which states:

> a set of *objectives* can be defined that should be postulated within any software engineering methodology, from which certain *principles* are derived that characterize the *process* by which software is created. Adherence to a process governed by those principles should result in a *product* (programs and documentation) that possesses *attributes* considered desirable and beneficial [HENS85].

This philosophical argument forms much of the basis for the perspective maintained during the research activity reported here, and is the basis for the primary results of the thesis by Dandekar [DANA87].

At first introduction, the concepts of objectives, principles, and attributes may not be all that clear. In order to obtain a better intuitive notion of this triad, it is necessary to first define precisely what is meant by a methodology in this context, and to then relate the usefulness of the concepts of objectives, principles, and attributes to a procedural evaluation of a software development methodology. Once one has an intuitive grasp of the applicability of these ideas, the basis for the analyses described in later chapters should appear tenable.

2.1.1 Methods and Methodologies

The proliferation of software engineering techniques has led to varying definitions of the term "methodology". Certainly, in a room full of experienced software designers, one could not obtain a consensus of opinion as to a precise definition. Among academicians, agreement is probably

more remote. However, in order to proceed in a constructive manner, a definition is necessary at this point.

First consider that a *method*, in the software engineering sense, is a procedure for accomplishing some measurable quantity of work. A method may consist of following some well defined set of instructions or an algorithm. Such a method is deterministic; that is, the outcome from following the algorithm, the solution, is predictable. This is not to suggest that all methods are deterministic. Most current software engineering problems can not be reduced to the point where an algorithmic solution becomes feasible. Rather, the staggering complexity of these problems requires the use of heuristic methods for obtaining a solution.

One should be able to reasonably accept that, in general, a method is characterized by:

1. the decisions that are to be made,

2. how to make them, and

3. the order in which they are made.

More specifically, in the context of this thesis, a method is defined to be a *process* for achieving an objective.[3]

A *methodology* can then be defined as a collection of complementary methods, and a set of rules for applying them. By extension, a methodology defines a collection of processes, and prescribes an order in which decisions are made so that performing these processes and making the appropriate decisions leads to the achievement of the desired objective(s). Furthermore, a methodology is guided by generally accepted software engineering principles that govern the defined collection of

[3] The term "process" connotes different things to different people. It may be more beneficial here to think of a process as an *activity* performed by some entity in the generation of some tangible result (e.g., the act of programming is a process that results in code).

processes. Adherence to these principles should lead to products which possess attributes that indicate that the desired objectives have been achieved.

Within the discipline of software engineering, different methodologies can be applied during the various phases of software development. For example, a methodology used during the requirements analysis phase is likely to be different from the methodology used during the implementation phase. The distinction of a methodology from the tools and environments used to *support* the application of methods is important. Such tools and environments also typically differ across phase boundaries. For example, program design language compilers may be used in the design phases; whereas symbolic debuggers are typically used only during the implementation and testing phases.

2.1.2 The Evaluation Procedure

Having presented the philosophy of the procedural approach to the evaluation of software development methodologies, the notions of objectives, principles, and attributes, which define the evaluation procedure, are developed in this section.

Figure 1 is a representation of the evaluation procedure. At the top of this figure is a set of objectives that embodies the needs and requirements one might have for the development of some particular software and related documentation. For example, the requirement for maintainable software has received recent emphasis, due to the alarming costs associated with making changes to software. Maintainable software is an example of an objective one might have in software development.

Suppose that prior to the initiation of software development, a set of objectives is proposed for the software and documentation to be developed. These proposed software development objectives

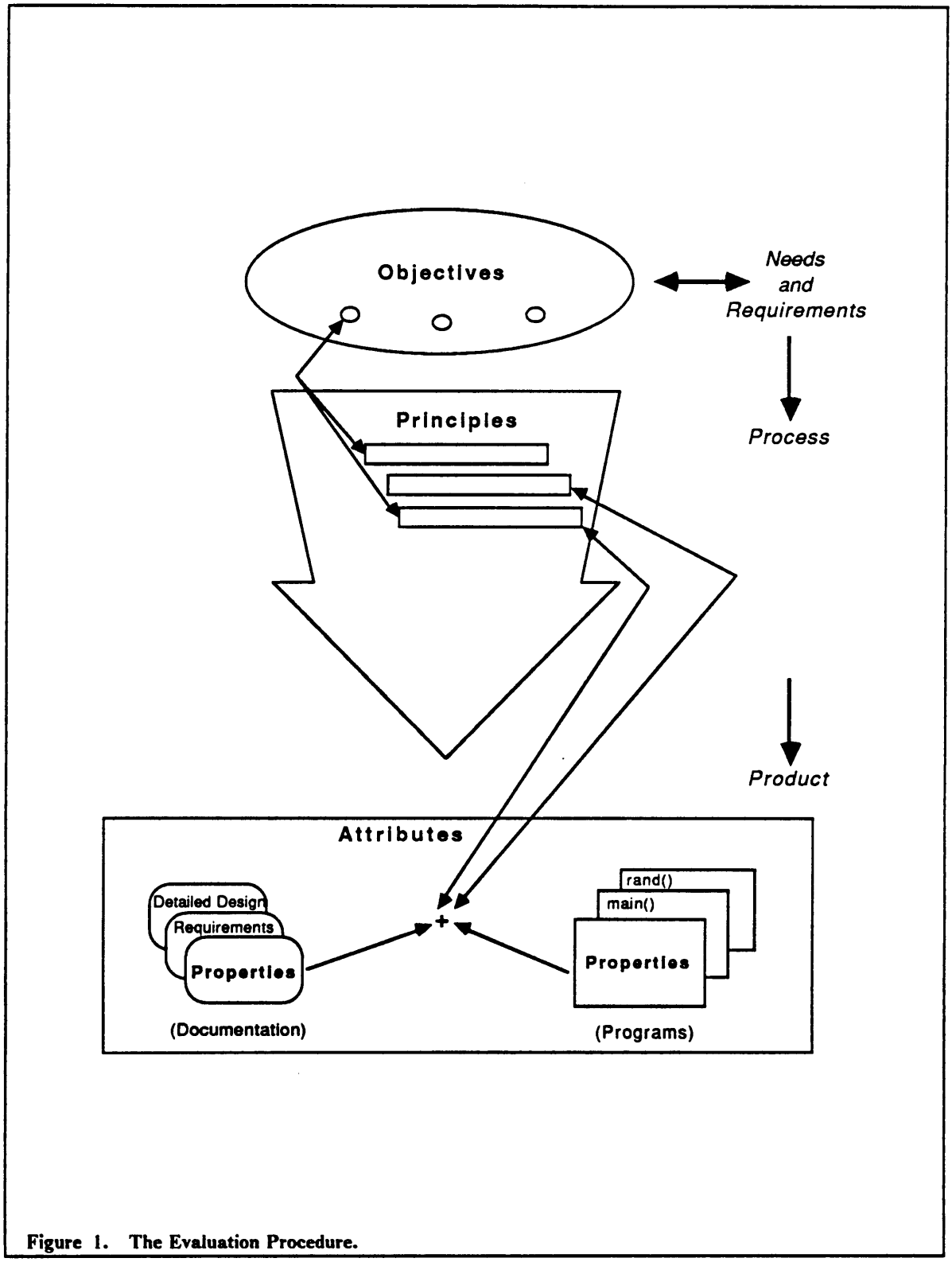

Figure 1. The Evaluation Procedure.

are likely to be some subset of the following list of objectives which have appeared in the software engineering literature:[4]

- Adaptability - the degree to which software can adapt to change.
- Correctness - the degree to which software adheres to specified requirements.
- Maintainability - the degree to which software corrections or modifications can be made.
- Portability - the degree to which software can be transferred to another environment.
- Reliability - the degree of error-free performance of software over time.
- Reusability - the degree to which software can be used in other applications.
- Testability - the ability to evaluate conformance with requirements.

Ideally, one would like to have each of these objectives be complementary, and cost effective. In practice, this is usually not the case. Rather, different objectives are often competitive and cost prohibitive, depending on the particular environment. For example, the objective of portability may be contrary to the objective of correctness if one considers that strict performance requirements exist. Such performance requirements may imply that some components are tightly bound to particular hardware constraints. One attempt at a compromise might be to use different levels of abstraction such that the functional interfaces of the software are hardware independent. The cost then of incorporating sufficient levels of abstraction to make the software have some acceptable degree of portability and correctness (i.e., adherence to the performance requirement), may be prohibitive, since additional design, coding, and testing must take place.

---

[4] See [DANA87] for an extensive bibliography.

The notion of defining a technique such as abstraction, leads to the idea of principles. Referring again to Figure 1, objectives are linked to principles. Principles define the process by which software is developed. Continuing with the previous example, suppose that one is attempting to meet the portability objective, and at the same time provide the required level of performance. Further assume that the approach taken is to provide multiple levels of abstraction, using abstract data types or packages to implement machine-specific functions. Years of research have revealed that the way to accomplish modularity is by employing principles not unlike some of those listed below:

- Documentation - management of system/software/life-cycle specifications.
- Functional Decomposition - partitioning of components along functional boundaries.
- Hierarchical Decomposition - top-down structuring of high-level components into lower-levels.
- Information Hiding - defining abstractions for component behavior.
- Life Cycle Verification - confirmation of requirements throughout the software development phases.
- Stepwise Refinement - incorporation of progressively finer component details in successive steps.
- Structured Programming - application of a restricted set of programming control constructs.

The result of adhering to such principles during the development of software should induce certain attributes in the final product. These attributes provide an indication of the extent to which the objectives have been met.

Continuing with the previous example, the effect of providing additional levels of abstraction by employing the principle of functional decomposition induces the *reduced complexity* and *ease of change* attributes. The complete set of attributes induced by the above set of principles is listed as follows:

- Cohesion - the binding of statements within a component.
- Complexity - an abstract measure of work associated with a component.
- Coupling - interdependence among components.
- Early Error Detection - indication of discrepancies in requirements and design specifications.
- Ease of Change - accommodation of enhancements or extensions to a component.
- Readability - difficulty in understanding a component (related to complexity).
- Traceability - ease of establishing the requirements and development history of an implemented component.
- Visibility of Behavior - provision of a well-defined review process for detecting errors and inadequacies in a component and its specifications.
- Well Defined Interface - completeness and clarity of an interface boundary between components.

In accordance with the ongoing example, one would expect the produced software to have the property that machine dependent packages could be replaced without causing any adverse effects on machine independent modules that utilize these packages. To reconfigure the software system for another hardware environment, only the machine dependent packages need be replaced. The relationship of product properties and attributes is shown at the bottom of Figure 1.

2.1.2.1 *Linkages Between Objectives and Principles*

In the previous discussion of the evaluation procedure, the chain of reasoning proceeds from objectives to principles, and then principles to attributes. Definite linkages exist between the set of objectives and the set of principles listed above. In the simplistic example presented, the portability objective required the employment of the principle of functional decomposition. There are several other principles that are related to this objective. Figure 2 is a table of the linkages between the set of objectives and the set of principles [DANA87].

These linkages are represented in Figure 1 by the double ended arrow joining objectives and principles. Although the previous discussion only followed this linkage in one direction (top-down), one can reason in reverse (bottom-up), from principles to objectives. This double linkage is quite useful when one begins first by examining the properties of a product in order to deduce the principles under which that product was generated, and hence the objectives. Such bottom-up reasoning may provide insight into the development process, given that one may be trying to assess the quality of that process by examining whether or not certain attributes exist in the products generated by the process. However, in order to complete this path from attributes to principles to objectives, the linkages between principles and attributes must first be established.

2.1.2.2 *Linkages Between Principles and Attributes*

Figure 3 shows the relationship of principles to attributes. Again referring back to Figure 1, this linkage is represented by a double ended arrow. Thus, the linkages from objectives to principles to attributes, and attributes to principles to objectives is complete. In the remainder of this work, the concept of linkages is exploited, particularly the linkages that allow a bottom-up chain of reasoning.

Objective	Principles
Adaptability	Documentation Functional Decomposition Hierarchical Decomposition Information Hiding Stepwise Refinement Structured Programming
Correctness	Hierarchical Decomposition Life Cycle Verification Stepwise Refinement Structured Programming
Maintainability	Documentation Functional Decomposition Hierarchical Decomposition Information Hiding Stepwise Refinement Structured Programming
Portability	Documentation Functional Decomposition
Reliability	Hierarchical Decomposition Information Hiding Stepwise Refinement Structured Programming
Reusability	Documentation Functional Decomposition Hierarchical Decomposition Information Hiding
Testability	Functional Decomposition Hierarchical Decomposition Information Hiding Life Cycle Verification Stepwise Refinement Structured Programming

Figure 2. Linkages Between Objectives and Principles.

Principle	Effect on Attributes
Documentation	Reduced Complexity Ease of Change Enhanced Readability Traceability
Functional Decomposition	Cohesion/Coupling Enhanced Reduced Complexity Ease of Change
Hierarchical Decompostion	Cohesion/Coupling Enhanced Reduced Complexity Ease of Change
Information Hiding	Cohesion/Coupling Enhanced Reduced Complexity Ease of Change Well Defined Interface
Life Cycle Verification	Early Error Detection Visibility of Behavior
Stepwise Refinement	Cohesion/Coupling Enhanced Reduced Complexity
Structured Progamming	Reduced Complexity Enhanced Readability

Figure 3. Linkages Between Principles and Attributes.

As alluded to earlier, the bottom-up chain of reasoning yields insight into the process of software development, given that one only has information about the products. The relevance of this statement should become clear when one examines software development within the context of DoD-STD-2167.

2.2 *The Role of Software Development Standards*

The purposes and reasons for software development standards are well-founded [HECH84]. However, the actual development of software standards, and subsequent compliance, is often characterized by confusion and frustration [FIRD87]. One difficulty for the developers of software standards is that of determining the extent of the guidelines in the standard. That is to say, it is difficult to determine at what point guidelines cease to be guidelines and begin to encroach on methodology.

It is often the case that the developers of software standards either fall short of the intended objectives by being too vague and ambiguous in their guidelines, or they overshoot their objectives and the standard is too restrictive. Invariably, the result lies somewhere in between, such that the standard expounds some objectives explicitly and others implicitly. In either case, the users of the standard often find that it is difficult to determine whether or not they are in compliance.

Most major software developers have already invested considerable resources in developing their own procedures for software development. It is of great interest, economically and often contractually, to know how well existing procedures comply with an industry standard. The short-term costs of having to incorporate new procedures and/or change old procedures can be substantial. Of course, existing software development practices may not only be in compliance, but exceed the requirements of the standard.

Whichever the case, one must possess detailed information about the standard in order to intelligently assess one's compliance. The problem is often how to obtain this information in light of the fact that the standard is composed of a combination of explicit and implicit requirements.

2.3 *The Nature of DoD-STD-2167*

DoD-STD-2167[5] claims to establish a uniform software development process based on a life-cycle model for software development. The specific life-cycle model prescribed by DoD-STD-2167 is discussed in this section.[6] The software development process of 2167 defines development activities which result in the following:

1. the generation of different types and levels of software and documentation,

2. the application of development tools, approaches, and methods, and

3. project planning and control [DODS85a].

This thesis is mainly concerned with the first two activities. Before examining these claims, however, the composition of 2167 is briefly reviewed in order to establish a context for subsequent discussions, and to delineate those aspects of 2167 on which this thesis focuses. In addition, a subset of the Data Item Descriptions that relate to 2167 are introduced, and the motivation for conducting a procedural evaluation is explained.

---

[5] For the remainder of this thesis, DoD-STD-2167 shall often be referred to simply as 2167.

[6] Life-cycle models exist in many forms. It is unwise to generalize the applicability of a particular life-cycle model.

2.3.1 Composition of the Standard

DoD-STD-2167 establishes general and detailed requirements for the development of defense system software. In general, the focus is on requirements for the production and acquisition of software *products*.[7] These products are typically associated with a logical entity called a Computer Software Configuration Item (CSCI), which is capable of being independently managed and is contractually deliverable. A CSCI is further subdivided into units of work called Computer Software Components (CSCs). A software system developed in accordance with 2167 is divided into multiple CSCI's. As a result, many of the requirements expressed in 2167 are defined in terms of a CSCI, and its related CSCs.

These requirements are distributed throughout several phases of a development process that is based on a life-cycle model. The specific software development life cycle prescribed by 2167, depicted in Figure 4, consists of the following six phases:

1. Software Requirements Analysis,

2. Preliminary Design,

3. Detailed Design,

4. Coding and Unit Testing,

5. CSC Integration and Testing,

6. CSCI Testing.

---

[7] In the context of this and subsequent discussions, the term "products" refers to both code and documentation.

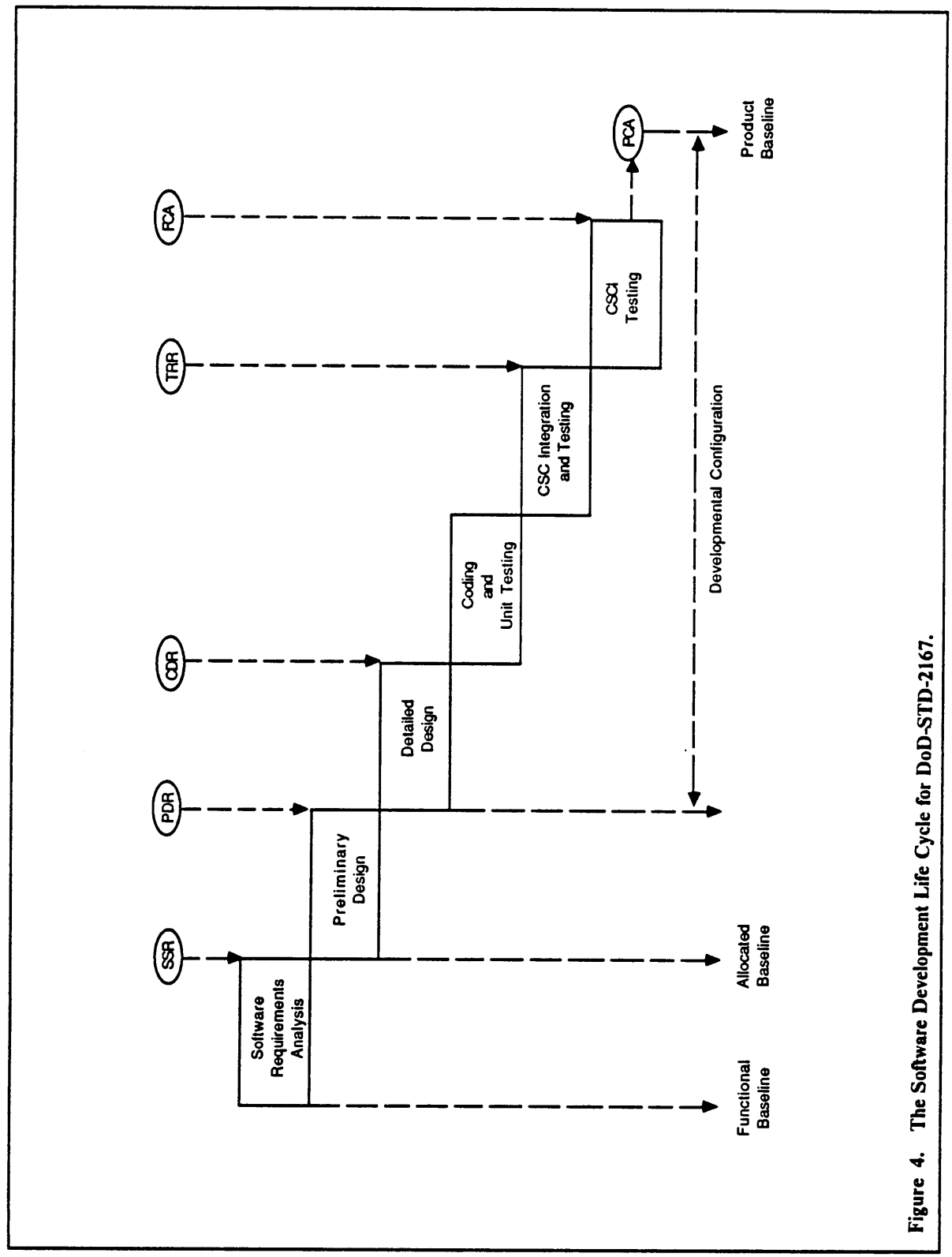

Figure 4. The Software Development Life Cycle for DoD-STD-2167.

Section five of 2167 is composed of subsections that define mostly product requirements for each of these six phases, plus subsections concerned with configuration management, software quality evaluation, and project planning and control.

At different times in the software life cycle, formal reviews or audits are required. During these formal reviews or audits, the software products relevant to a particular phase are evaluated. These reviews are represented in Figure 4 as occurring at the end of selected phases, and are identified as follows:

- SSR - Software Specification Review,
- PDR - Preliminary Design Review,
- CDR - Critical Design Review,
- TRR - Test Readiness Review,
- FCA - Functional Configuration Audit,
- PCA - Physical Configuration Audit.

Depending on the stage of development, the products evaluated represent the current state of the software system. In Figure 4 for example, the products that represent the current state of requirements for a CSCI, following the software requirements analysis phase, define the "allocated baseline". Similarly, the "developmental configuration" represents the current state of the design and development of a CSCI, and eventually establishes the "product baseline".

Given that section five of 2167 defines the detailed requirements for software development, later chapters of this thesis focus on section five; particularly those subsections that correspond to the six phases of the software development life cycle. Furthermore, since section five specifies product

requirements, it is necessary to collaterally focus on the Data Item Descriptions that are related to the life cycle phases. The following section introduces the major Data Item Descriptions that are of interest in following chapters.

2.3.2 Relationship to Data Item Descriptions

As just mentioned, 2167 contains the requirements for the products related to each of the phases in the software development life cycle, as well as the requirements for formal review of these products. However, to find the detailed product requirements, one must refer to the Data Item Description (DID) for a specific product. In general, a DID describes the format and content requirements for a product.

Many of the products required by 2167 occur in more than one phase of software development. Either the product is initially generated during a phase, or the product is updated. Several of these products are so closely related that they can be thought of as a set of products, and referred to collectively. The following sections define these related collections of products.

2.3.2.1 Software Development Specifications

At the outset of software development a set of specifications is produced to guide the subsequent development phases. Collectively, these specifications, called *software development specifications*, are listed as follows:

- Software Development Plan (SDP),
- Software Standards and Procedures Manual (SSPM),

- Software Configuration Management Plan (SCMP),

- Software Quality Evaluation Plan (SQEP).

By examining the DID for the Software Development Plan, one finds that each of the other specifications can be included as part of the SDP. In many instances these development specifications may be separate documents; however, a collective reference is still useful.

2.3.2.2 Allocated Baseline

The *allocated baseline* establishes the requirements for the development of a CSCI. In particular, the specifications that make up the allocated baseline consists of the following:

- Software Requirements Specification (SRS),

- Interface Requirements Specification (IRS).

The SRS specifies both the allocation of requirements to a CSCI, as well as quality requirements. One or more IRSs define the interface requirements allocated to a single CSCI. The allocated baseline is eventually incorporated into the final *product baseline* prior to delivery of a CSCI.

2.3.2.3 Developmental Configuration

The *developmental configuration* is a collection of specifications that represents the current state of development of a CSCI. As such, the developmental configuration, in its final form, contains all of the following:

- Software Top Level Design Document (STLDD),

- Software Detailed Design Document (SDDD),
- Interface Design Document (IDD),
- Data Base Design Document (DBDD),
- developed software.

Both the design specifications and developed software eventually become the final software product in the form of a Software Product Specification (SPS). However, throughout most of this work, these related products are referred to as the developmental configuration. The point when one should cease to think of these products as the developmental configuration and consider them collectively as the SPS is explicitly noted.

2.3.2.4 Testing Specifications

Closely related to the developmental configuration is a set of *testing specifications*. These specifications are developed collaterally with the developmental configuration, but are distinct in their own right. The testing specifications include the following:

- Software Test Plan (STP),
- Software Test Description (STD),
- Software Test Procedure (STPR),
- Software Test Report (STR).

The testing specifications establish the description and format of the formal testing that occurs in the later stages of the software development life cycle.

2.3.2.5 *Life-Cycle Support Specifications*

Aside from the developmental configuration and testing specifications, other specifications exist that are referred to collectively as *life-cycle support specifications.* Some of these documents are required as early as preliminary design and continue to evolve during the remaining phases. Others are not required until the later stages of the life cycle. The life-cycle support specifications are listed as follows:

- Computer Resources and Integrated Support Document (CRISD),
- Computer System Operator's Manual (CSOM),
- Software User's Manual (SUM),
- Computer System Diagnostic Manual (CSDM),
- Software Programmer's Manual (SPM),
- Firmware Support Manual (FSM),
- Operational Concept Document (OCD).

These documents define the information that is related to the support and operation of the system once it is delivered.

2.3.3 Motivation for a Procedural Evaluation

As just demonstrated, DoD-STD-2167 places substantial emphasis on requirements for products, particularly documentation, and the process by which these products are acquired. The process by which these products are generated receives less emphasis.

Suppose that one is interested in the generation process. That is, given the requirements for the products, what can one determine about the processes required to generate these products? Formulated in terms of the procedural evaluation, what properties do products generated in accordance with 2167 possess? By extension, what attributes are indicated? Further, based on linkages of principles to attributes and bottom-up reasoning (as discussed previously), what can be inferred about the process? Even further, based on the linkages of principles to objectives, what can be said about the methodological characteristics of 2167? Is it a software development methodology? That is, does it define a set of objectives for software development? If 2167 is not a methodology, what is it?

These questions can be approached by adopting a perspective based on the objectives, principles, and attributes offered by the procedural evaluation, and then critically analyzing the content of 2167.

2.4 The Nature of DoD-STD-2168

DoD-STD-2168[8] establishes the requirements for evaluating the quality of software and documentation developed in accordance with 2167. Similar to 2167, the software development life

[8] For the remainder of this thesis, DoD-STD-2168 will often be referred to as simply 2168.

cycle is the basis for the software quality process prescribed by 2168. Therefore, 2168 contains requirements that pertain to each of the six phases of software development as prescribed by 2167.

2.4.1 Composition of the Standard

In general, 2168 requires a software quality evaluation process that includes the following:

1. the evaluation of software requirements,

2. the evaluation of software development methodologies,

3. the evaluation of software and documentation, and

4. the mechanism(s) necessary to ensure that changes are made.

In specifying the requirements for software development, a set of software *quality factors* may be included. These quality factors are defined in 2168, and correspond almost exactly to the set of objectives espoused by the procedural evaluation.

Similar to 2167, the detailed requirements for the evaluation of the development process are defined for each of the six phases of the software development life cycle. Thus, 2168 is in one-to-one correspondence with 2167.

The analysis of 2168, in chapter five of this work, is based in part on the explicit relationship of software quality evaluation to software development. The next section introduces those aspects of 2167 that are particularly important to establishing this relationship.

2.4.2 Relationship to DoD-STD-2167

In addition to the six phases of the development life cycle, 2167 has requirements for configuration management and software quality evaluation. Given the above list of 2168 requirements, a direct relationship exists between configuration management and software quality evaluation prescribed by 2167, and the requirements of 2168. Exactly how this relationship can be established is the subject of chapter five. It suffices at this point to introduce the configuration management process and software quality evaluation process as background for the more detailed analysis to follow.

2.4.2.1 Configuration Management

Configuration management (CM) is the mechanism which controls the changes to software and documentation. The CM requirement of 2167 corresponds to the fourth requirement of 2168 listed above.

A *configuration item* is the primary focal point of the configuration management process. For example, the developmental configuration for a CSCI, introduced in the previous discussion on the nature of 2167, is a primary configuration item. In general the following are all configuration items:

- functional and allocated baselines,
- developmental configuration,
- product baseline.

Changes to a configuration item follow a specific procedure whereby a problem is reported via a formal software problem or change report. Changes may then be incorporated via two formal mechanisms:

- an Engineering Change Proposal (ECP),

- a Specification Change Notice (SCN).

Within the context of 2167, the CM process is formally defined in the Software Configuration Management Plan, described earlier as a development specification related to the Software Development Plan. During each of the development phases, the CM process is embodied in the formal reviews that occur at the culmination of a particular phase. These formal reviews are depicted in Figure 4, and discussed previously in the context of the nature of 2167.

2.4.2.2 Software Quality Evaluation

The requirements for software quality evaluation prescribed in 2167 are precisely those prescribed by 2168, only more briefly stated. A clear indication of the relationship of 2167 and 2168 is in terms of the following software quality factors defined in 2168:

- Correctness,

- Efficiency,

- Flexibility,

- Integrity,

- Interoperability,

- Maintainability,

- Portability,

- Reliability,
- Reusability,
- Testability,
- Usability.

If one examines the Data Item Description for the SRS, one finds these same software quality factors listed as possible quality requirements for a CSCI. As stated above, the quality factors correspond closely to the objectives defined during the discussion of the procedural approach to evaluating software development methodologies.

2.4.3 Motivation for a Procedural Evaluation

Having previously established a motivation for a procedural evaluation of 2167, it follows that one would be interested in the relationship of 2168 to 2167. That is, how is software quality evaluation, as required by 2168, integrated with the software development processes and products required by 2167? Given that 2168 defines software quality factors that closely resemble the objectives espoused by the procedural evaluation, what impact do these quality factors have on the development process?

The analysis of 2168 in chapter five focuses on these questions by viewing the software quality evaluation process in light of its relation to software development and the procedural evaluation.

3.0 The ADFD Model

Most scientific investigations into the nature of complex problems require models as conceptual tools for representing an understanding of the fundamental nature of the objects under investigation. Such a conceptual tool allows the investigator to characterize the objects being studied by explicating the structure of the objects and/or their relationships to other objects. Often an added benefit of applying a model to a complex problem is that one is able to discover some unknown or hidden properties of the problem being modelled.

With this in mind, the motivation, background, and definition of the Augmented Data Flow Diagram (ADFD) model as a conceptual tool forms an integral part of this thesis.

3.1 Motivation and Background

To the uninitiated, a first encounter with a technical specification, such as a software standard, is often less than a pleasurable experience. Usually, after several readings, one begins to obtain some idea as to the substance of the information being read. The problem is often a result of

unfamiliarity with basic concepts, and the complexity and detail with which the concepts are presented. The problem is further aggravated by the fact that the task of expressing complex concepts in written language often results in some loss of meaning.

The language of mathematics is an example of a tool developed to aid in coping with the representation of complex ideas. However, no precise mathematical formulas currently exist for characterizing the objects and relationships found in many software specifications. For this reason, models such as the ADFD model are formulated. Such models do not display the precision of mathematical formulas, since the problems to which the models are applied are not precise.

Thus the definition of the ADFD model is brought about by a need to characterize the objects (processes and products) and relationships (data flow) found in both 2167 and 2168. Fortunately, the existence of a data flow model facilitated the definition of the ADFD model. The *Yourdon Data Flow Diagram* (DFD) model [DEMT79] possesses features that, with extensions, are sufficient to characterize the objects found in 2167 and 2168. The DFD model is augmented to yield the ADFD model which is used to characterize the process-product relationships of 2167 and 2168.[9]

3.2 The Definition of the ADFD Model

3.2.1 What are Augmented Data Flow Diagrams?

ADFDs, as just discussed, are adapted from DFDs. Therefore, ADFDs exhibit some of the same characteristics as DFDs. However, the syntax and semantics of ADFDs differ, to a degree, from the syntax and semantics of DFDs.

---

[9] Augmentations to the DFD model are not without precedent [WARP86].

Like a DFD, an ADFD is a graph model of the relationships among processes and the data that processes generate and/or consume. In such a graph, processes are represented by nodes, data flow is represented by directed edges or arcs, and a special type of node is used to represent a repository which contains data.

An important distinction stems from the notion that an ADFD is a model of reiterative data flow through processes that refine data. The refinement of the data changes, in a qualitative sense, the concepts presented by the model. Thus, data in an ADFD is regarded as a dynamic component; whereas data in a DFD is typically regarded as a static component.

3.2.2 The Syntax of Augmented Data Flow Diagrams

ADFDs are constructed from three basic entities:

- process bubbles,
- data stores, and
- data flow arcs.

A data flow is represented graphically by a directed arc. Referring to Figure 5, arcs between process bubbles and data stores are not labeled (Figure 5a). Further, an arc from a data store must terminate in a process bubble.

An arc representing the same data flow may originate as a single arc from a process bubble and split, terminating at multiple process bubbles or data stores (Figure 5b). Note also in Figure 5b that a data flow arc from a process bubble to another process bubble must be labeled.

An arc from a data store directly to another data store is undefined and syntactically incorrect, and an arc looping on a process bubble or data store without first having entered another process bubble is also undefined (Figure 5c).

A process bubble must be labeled. A process bubble with no identifying label is undefined and syntactically incorrect. Data may flow bidirectionally between two or more process bubbles and is represented by a labeled arc. Additionally, data flows bidirectionally between process bubbles and one or more data stores.

A data store must be labeled and be incident with one or more arcs. Data flows from a process bubble to another process bubble, from a process bubble to a data store, or from a data store to a process bubble.

Figure 6 illustrates a syntactically correct ADFD. At this point, a complete understanding of Figure 6 is unnecessary; rather, one should focus on the syntactic components being illustrated: process bubbles, data stores and data flow arcs. Further, note that each process bubble and data store is labeled. Note also that a process bubble may itself be an ADFD.

All process bubbles in Figure 7 are subordinate processes of the process bubble labeled 5.2:1 in Figure 6, and each of the data stores in Figure 7 collectively compose the data store labeled STLDD in Figure 6. The nature of ADFDs thus allows one to represent data flow in a hierarchical manner by refining top level data flow diagrams into successively lower levels.

The following section describes the meaning associated with each graphical entity found in an ADFD.

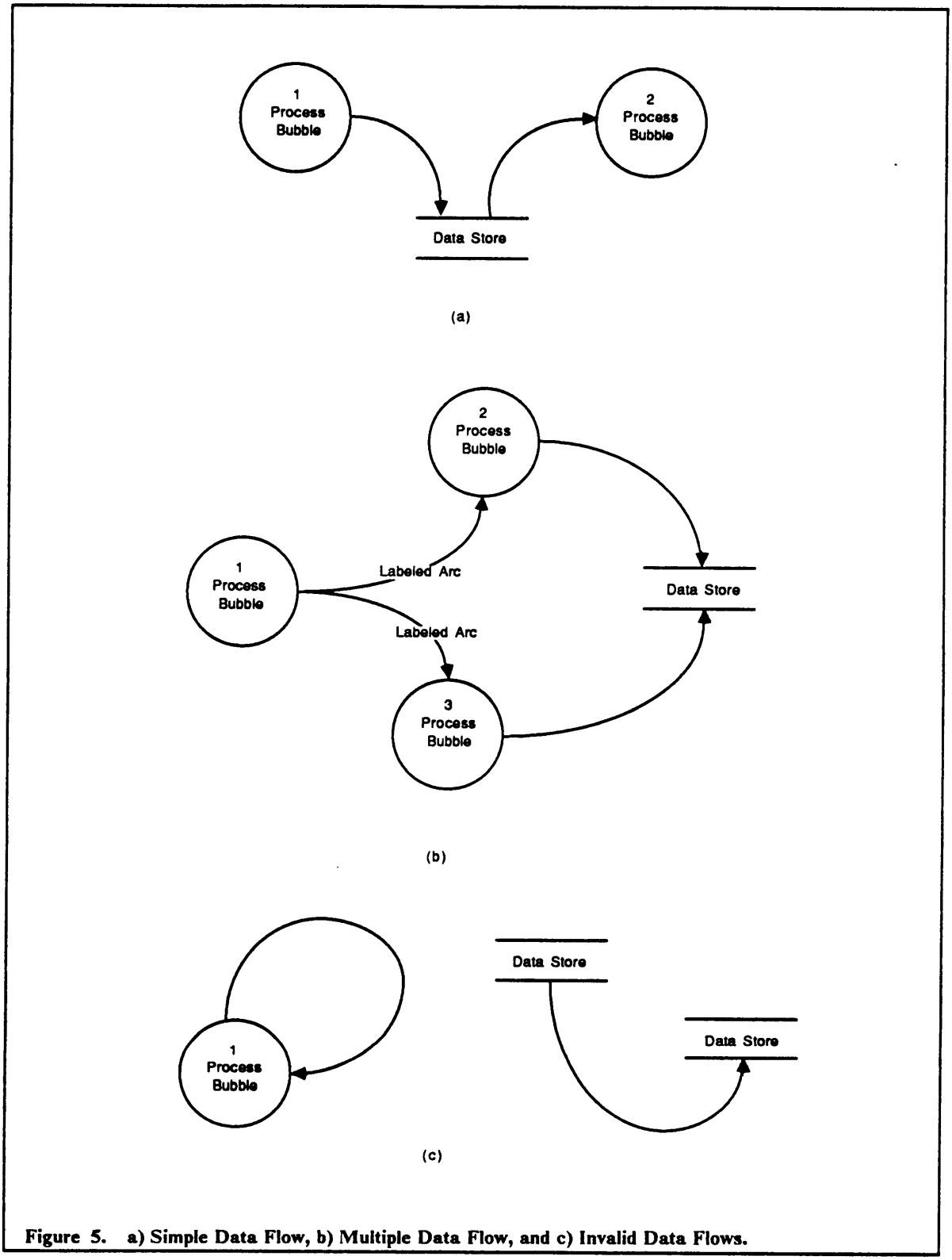

Figure 5. a) Simple Data Flow, b) Multiple Data Flow, and c) Invalid Data Flows.

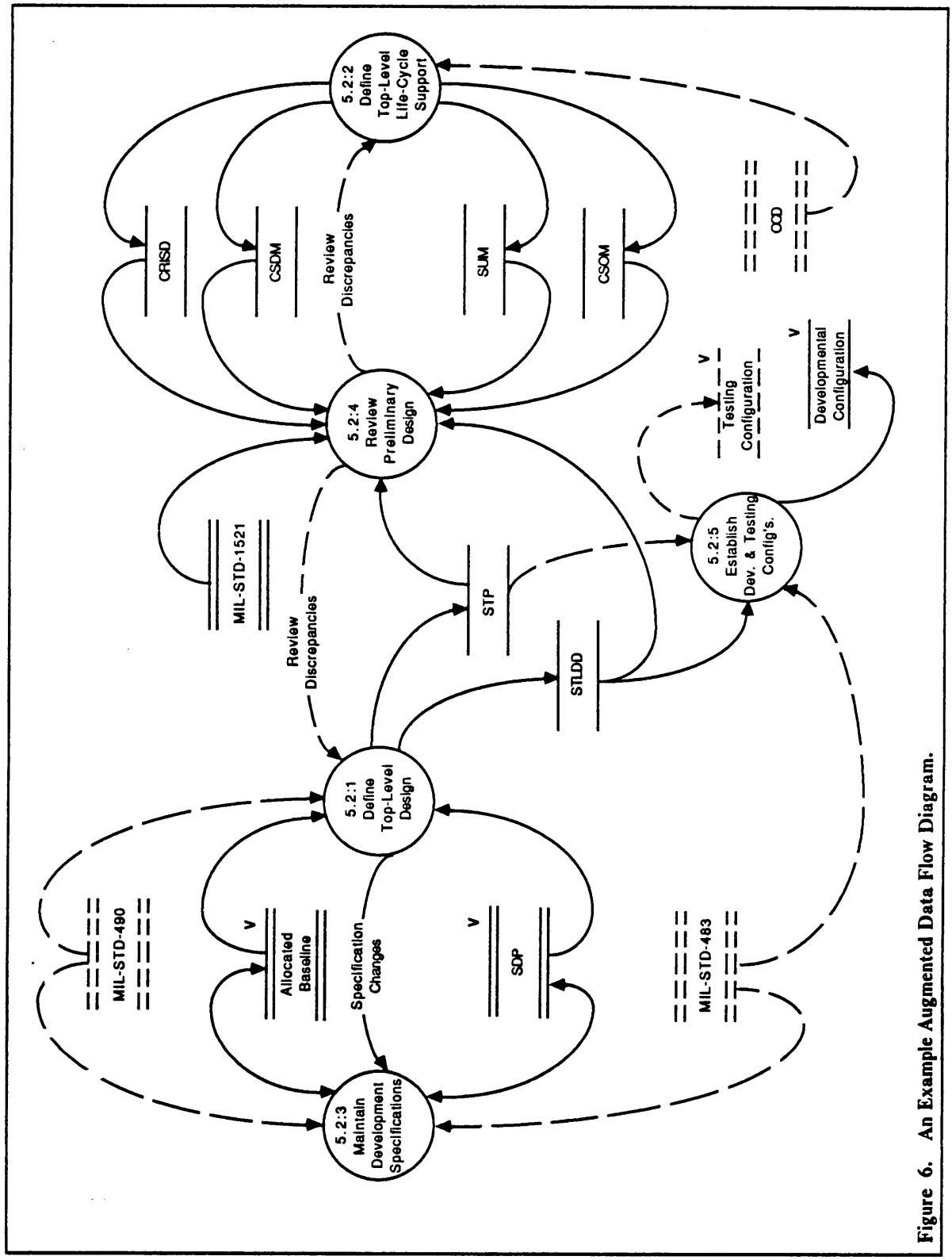

Figure 6. An Example Augmented Data Flow Diagram.

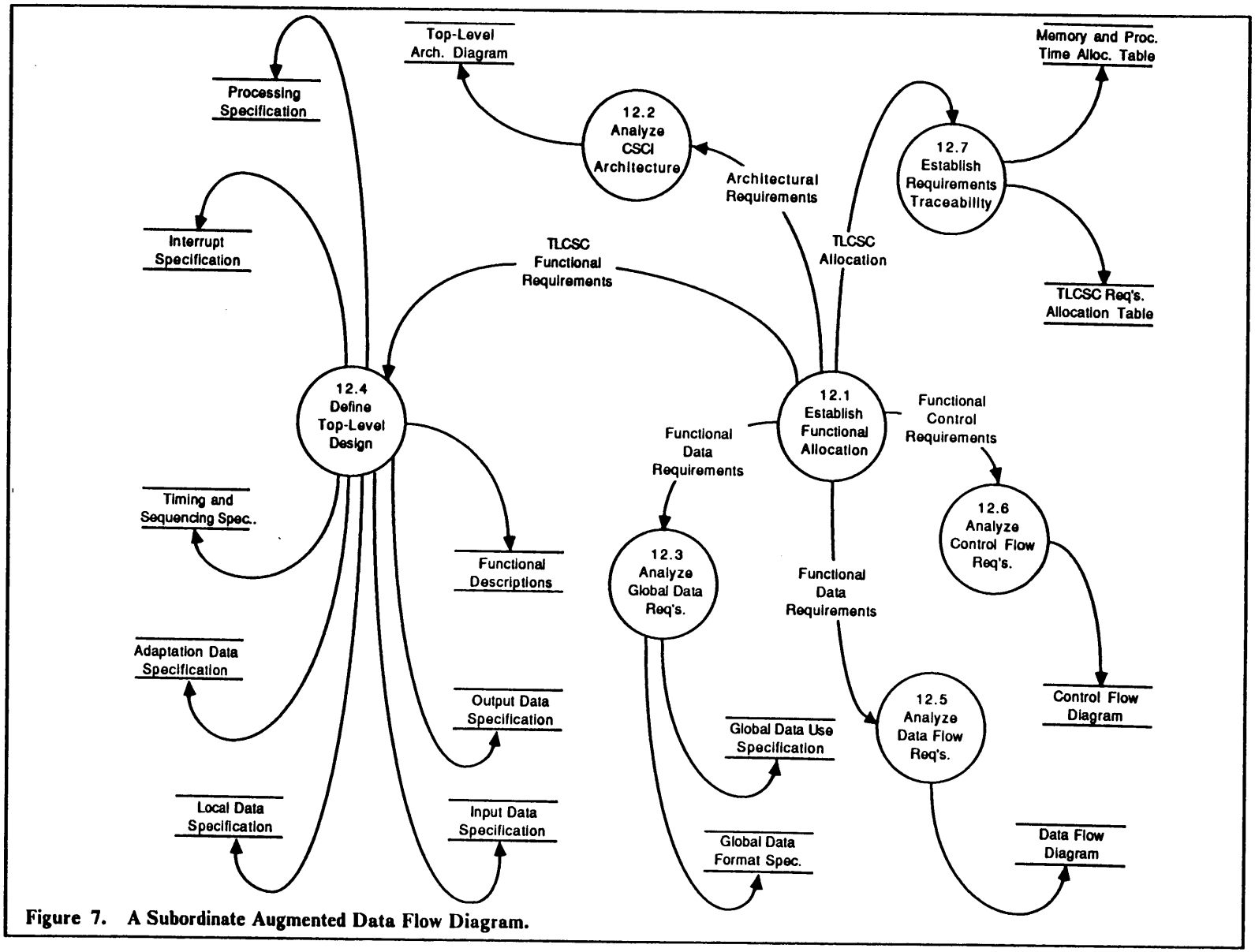

Figure 7. A Subordinate Augmented Data Flow Diagram.

3.2.3 The Semantics of Augmented Data Flow Diagrams

The semantics of ADFDs are described in term of the basic entities: process bubbles, data stores, and data flow arcs. Semantically, each entity denotes a different meaning depending on its graphical representation. The following sections describe the semantics associated with each entity.

3.2.3.1 Process Bubbles

A process bubble denotes a process that generates and/or consumes data. A process may be either *explicit* or *implicit*. An explicit process is denoted by a process bubble with a solid line perimeter, as depicted in Figure 8. Implicit processes are represented by process bubbles with a broken line perimeter.

Every process bubble in an ADFD must be labeled. The label of a process bubble is usually stated in the imperative and identifies the nature of the action(s) performed by the process. For convenience, a process bubble is also given a number by which the process is referenced.

As mentioned previously, an important feature of the ADFD model is the ability to hierarchically decompose an ADFD into subordinate ADFDs. The process bubbles are the entities that are hierarchically decomposed. Hence, a process bubble may represent a subordinate ADFD.

3.2.3.2 Data Stores

Data stores may also be explicit or implicit, as shown in Figure 8. Similar to process bubbles, a data store with solid bars represents an explicit data store while broken bars signify an implicit data store.

A data store representing the same information may occur in many ADFDs. It is sometimes beneficial to make a distinction between data stores that appear for the first time in an ADFD and those that have appeared in other diagrams. A data store that occurs in multiple ADFDs is called a *recurrent* data store and one is depicted at the bottom of Figure 8.

Data stores are defined to be collections of related data. Alternatively, a data store may also be a collection of data stores. In this instance, the data store is referred to as a *virtual* data store. A virtual data store is indicated by the presence of the symbol V just above the data store, as illustrated in Figure 8.

3.2.3.3 Data Flow Arcs

As one might expect, data flow arcs in an ADFD can represent either explicit or implicit data flow. Explicit data flow is graphically represented by a solid arc; a dashed or broken arc represents implicit data flow.

A data flow is also distinguished based on its source. A *generative* data flow originates from a process bubble and remains in the generative state until the data flow terminates in a data store or another process bubble. A *consumable* data flow originates from a data store and is consumed by one or more process bubbles.

Semantically, a data flow from a process bubble to a data store represents the generation of data that produces an initial data store, or the refinement of data which is added to an existing data store. Data flow in which data is consumed, refined, and then added to the same data store may be represented as a single data flow. Such data flow may be represented by a data flow arc from a data store to a process bubble and then a separate data flow arc back to the data store. An abbreviated representation is to represent the data flow with a bidirectional arc (arrows at each end of the arc).

A process bubble may be incident with only generative data flow arcs. In this case, the process denotes a generating process, or *source*, which does not depend on outside information, unless the process is in a subordinate ADFD. A subordinate process is interpreted as consuming any data the higher-level process consumes. A process bubble incident with only consumable data flow arcs represents a *sink*. A sink process is not particularly informative unless the process is decomposed into subordinate processes, in another ADFD, that generate data flow.

A process may also generate data flow to another process. Data flow in this manner is represented by a data flow arc from one process bubble to another. The generative process provides input data to the consuming process denoting a data dependency between the two processes.

3.2.4 Augmented Data Flow Diagrams versus Data Flow Diagrams

An ADFD is built by combining basic graphical entities in a manner consistent with the syntactical restrictions outlined in a previous section. Figure 8 illustrates the explicit, implicit, recurrent and virtual ADFD entities. Referring back to Figure 6, one can identify implicit data flow as the dashed arcs, implicit data stores by the dashed bars, and virtual data stores by bars with a superscripted V.

Note the ability to combine syntactic characteristics of a data store to obtain semantically different entities. For example, in Figure 6, the data store labeled "allocated baseline" is a virtual recurrent data store, and the data store labeled "testing configuration" is an implicit virtual data store. Such combinations of basic entities prove to be powerful tools in the application of the ADFD model.

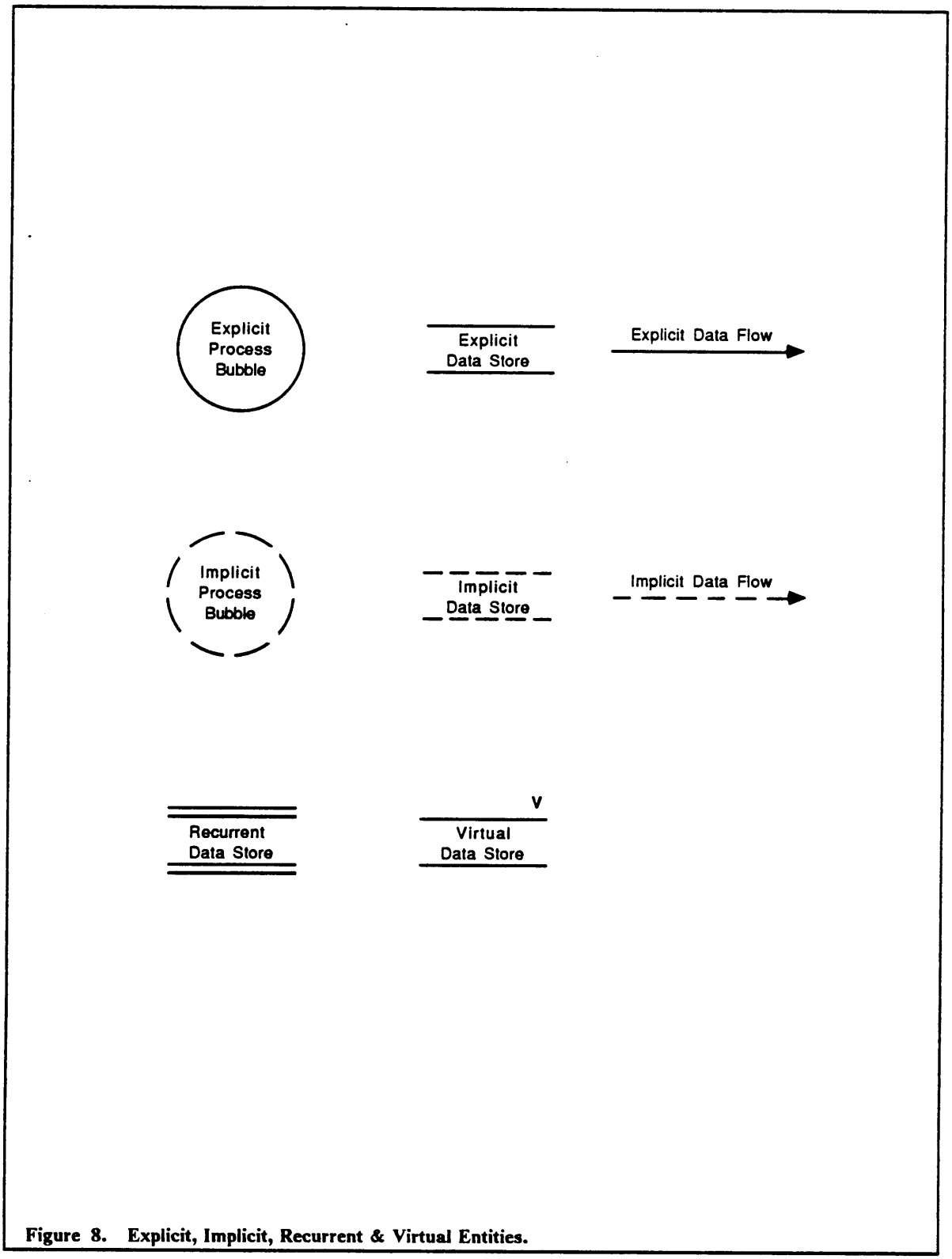

Figure 8. Explicit, Implicit, Recurrent & Virtual Entities.

3.2.4.1 Similarities

ADFDs, like DFDs, do not represent control flow among process bubbles (although the data flow may imply control in a different context). A common difficulty in formulating or interpreting data flow diagrams is the presumption of control flow.[10]

Pictorially, both ADFDs and DFDs are graph models and are therefore similar. Similar terminology for ADFD graphical entities is adopted from DFDs. Likewise, the graphical entities in ADFDs are similar to those in DFDs.

Relationships between processes and data can be clearly represented using a data flow model because of the graphic nature of flow diagrams. The ability to clearly represent relationships also makes it clear when a relationship is missing or in error. ADFDs possess this feature which contributes to their power as a modeling tool.

3.2.4.2 Differences

DFDs typically represent the flow of data from a source, through one or more process bubbles and/or data stores, to a sink; cycles do not normally occur. An ADFD typically contains many cycles. In an ADFD, source data is often unrefined and by a reiterative flow of the data through process bubbles the data becomes more refined. The concept of data refinement is a distinguishing factor of an ADFD.

Additional differences of the ADFD model that pertain to its use in this work are explained in the next section.

---

[10] For example, as just described, data flow between two processes denotes a data dependency. One may view data dependency as implying an ordering in time; which is a form of control.

3.3 *The Application of the ADFD Model*

In this thesis, the ADFD model is used as an analytic tool. Chapter four and chapter five present ADFDs constructed by analyzing particular sections of DoD-STD-2167, DoD-STD-2168, and related Data Item Descriptions. The application of the ADFD model to these standards and DIDs results in a concise representation of an otherwise lengthy and imprecise description.

In the typical application of the DFD model, one constructs a data dictionary (which defines the processes, data flows, and data store contents) in conjunction with the design of the DFD model. The ADFD model can also be used in this manner. However, in the analyses that follow, the ADFD model is used in a slightly different fashion.

One could take the point of view that both 2167 and 2168 are data dictionaries. One might then be interested in constructing an ADFD model of each data dictionary in order to visualize the relationships that exist; in effect, a type of "reverse engineering".

The application of the ADFD model to 2167 and 2168 in this manner results in a series of ADFDs that capture the relationships among development processes and products. The graphic representation of these relationships removes much of the ambiguity and imprecision that is inherent in the written text of 2167, 2168, and related DIDs (approximately five hundred combined pages). Having explicated the process-product relationships using the ADFD model, one can begin to reason, in the context of the evaluation procedure, about these relationships.

In the analyses that follow, the ADFDs presented include instances of explicit and implicit entities, as well as virtual and recurrent data stores. The reader should be familiar with the syntax of each ADFD entity and their combinations (e.g., a virtual recurrent data store). However, before proceeding, a more intuitive explanation of the need for implicit entities is warranted.

In chapter two, the issue of the *letter* of a standard versus the *intent* of a standard is introduced. In the application of the ADFD model to 2167, 2168 and related DIDs, the ability to capture processes and data relationships that are perceived to be implied is considered useful. Hence, implicit process bubbles are used to represent processes that appear to be implied by 2167 and 2168. Likewise, implicit data stores represent implied data repositories and implicit data flow arcs represent implied relationships between processes and data stores.

The selection of implicit entities in the ADFDs presented in the next two chapters is subjective. Based on experience and intuition, implicit entities are defined and incorporated into ADFDs when an implied process or relationship is perceived to be important for establishing the continuity of the explicit processes and relationships being modeled. More generally stated, the implicit ADFD entities are useful for "filling in the gaps" due to the nature of 2167 and 2168.

4.0 Analysis of DoD-STD-2167

Having previously discussed the nature of DoD-STD-2167, the analysis in this chapter focuses on the software development processes and products outlined in section five of 2167. The first six subsections correspond to the six phases of the software development life cycle illustrated in Figure 4, plus subsections on configuration management, software quality evaluation and project planning and control.

In the analysis that follows, ADFDs are presented for each of the six phases of software development, along with ADFDs for selected Data Item Descriptions. In examining the ADFDs for each of the phases of software development, one can infer that the processes of configuration management, software quality evaluation, and project planning and control somehow underlie each of these phases. That is, intuition suggests that these processes must be going on concurrently with the development processes depicted in an ADFD. However, how this "underlying concurrency" should be represented with the ADFD model, if at all, is unclear. In the following chapter, the relationship of the configuration management and software quality evaluation processes, described in 2167, to the processes associated with software development is critically examined. Thus in this chapter, configuration management and software quality evaluation are only generally discussed, and then reintroduced within the context of DoD-STD-2168. The explicit processes associated with project planning and control, as defined in section five of 2167, are not discussed in this work.

In the following sections, several ADFDs are presented, and the discussions in these sections are centered on the ADFDs. In those ADFDs that represent a particular section of 2167, each process bubble is labeled with the section number followed by a colon and number. The colon-number pair is used as a mechanism for distinguishing process bubbles within a single diagram to aid in the discussion. Thus a reference to "process one", in the context of a single ADFD, refers to the process bubble with ":1" following the 2167 section number. The colon-number pair does not refer in anyway to a subsection of 2167.

Each process bubble in an ADFD for a Data Item Description is labeled with the DID number and a second number that allows one to distinguish the activity represented by a particular process bubble. The DID number corresponds to a reference for that DID in the "References" section of this work, and may also be found in "Appendix B. A Synopsis of Data Item Descriptions". The reader may find it necessary to occasionally refer to this appendix. The reader may also find it useful to refer to "Appendix A. Glossary". The Glossary is particularly useful in this chapter (and the next) as the use of acronyms for specifications is predominant.

4.1 Software Requirements Analysis

Software requirements analysis is generally recognized as the most important phase of the six phases of the software development life cycle. This statement is not meant to preclude the importance of the system planning and concept formation phases that usually precede software development. However, the initiation of those activities precede the activities described by 2167, and are excluded from consideration in this work.

Figure 9 is the ADFD of the software requirements analysis process as prescribed by section 5.1 of 2167. The major activities in the analysis of software requirements are depicted by four process bubbles; each is discussed individually below.

4.1.1 ADFD Analysis

The activity of defining development plans and software requirements is represented by process one, labeled 5.1:1, in accordance with the aforementioned labeling scheme. Prior to software requirements analysis, the System/Segment Specification, Prime Item Specification, and Critical Item Specification, which represent potential requirements sources, must be developed. In addition, requirements may be drawn from other unspecified sources. In concert with the semantics of the ADFD model, these specifications are graphically represented by recurrent data stores since they exist prior to the overall process described here.[11] Similarly, MIL-STD-490, the military standard for specification practices, is represented in this manner. As indicated, each of these data stores provides input to the process of defining development plans and software requirements.

More importantly, the focus is on the products generated by this process. In particular, the

- SDP - Software Development Plan [DID30],
- SSPM - Software Standards and Procedures Manual [DID11],
- SCMP - Software Configuration Management Plan [DID09],
- SQEP - Software Quality Evaluation Plan [DID10],

[11] Presumably, if the ADFD model were applied to the *system life cycle*, each of these specifications would appear as data stores generated by some process in a higher level ADFD. Such an application is outside the scope of this research.

- OCD - Operational Concept Document [DID23], and

- informal records and reports.

Each of these products is represented by a single data store indicating that each is first generated during requirements analysis.[12] In actuality, if one were to examine the DID for the SDP, one would find that the SDP may include the SSPM, SCMP, and SQEP. Each specification is represented here as a separate data store to stress its individual importance at the outset of software development.[13]

In particular, the SSPM plays an important role in later analyses. For this reason, the ADFD for the SSPM is presented in Figure 10 in order to introduce the reader to the processes required in the generation of this specification. Each of the processes in the ADFD of the SSPM are subordinate processes of the "define plans and requirements" process in Figure 9. One may also suppose that the SSPM is a virtual data store composed of each of the data stores shown in Figure 10. Technically this is true. However, the convention adopted here is that the granularity of data store composition is at the specification level; otherwise, the use of virtual data stores would be prolific and nothing would be gained.

The importance of the SSPM in this discussion derives from the shaded process bubble and the column of data stores, at the extreme left of Figure 10. The process and data stores represent the definition of methodologies and tools used during each phase of software development. Thus, part of the process of software requirements analysis is to define the methodologies and tools that are to be used throughout software development. In addition, testing criteria, development criteria, design and coding criteria, exception criteria, and configuration management criteria are defined and documented.

---

[12] Technically, the OCD may already exist prior to software requirements analysis.

[13] The reader may notice that this is a perfect example for use of a *virtual* data store, which is done in later ADFDs.

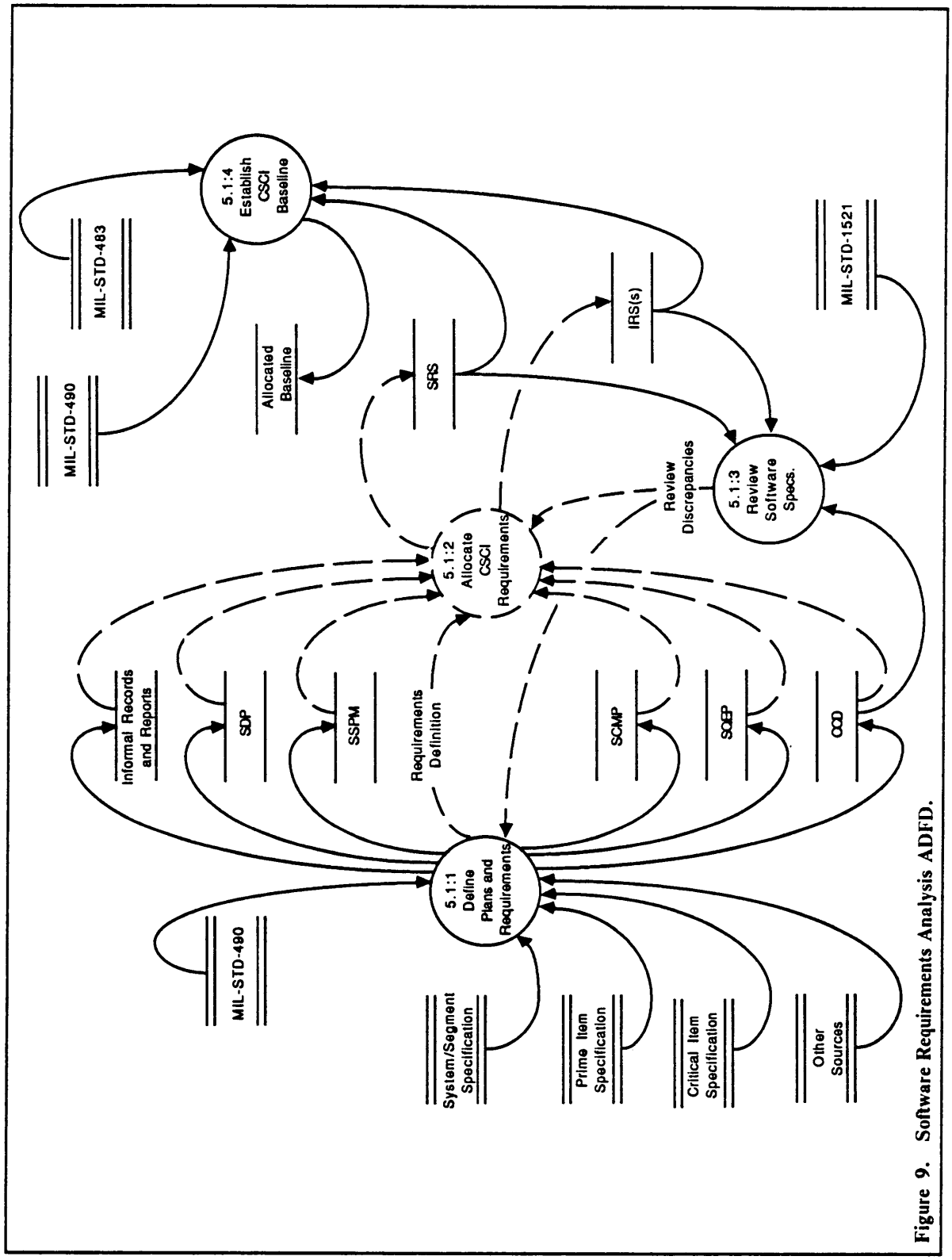

Figure 9. Software Requirements Analysis ADFD.

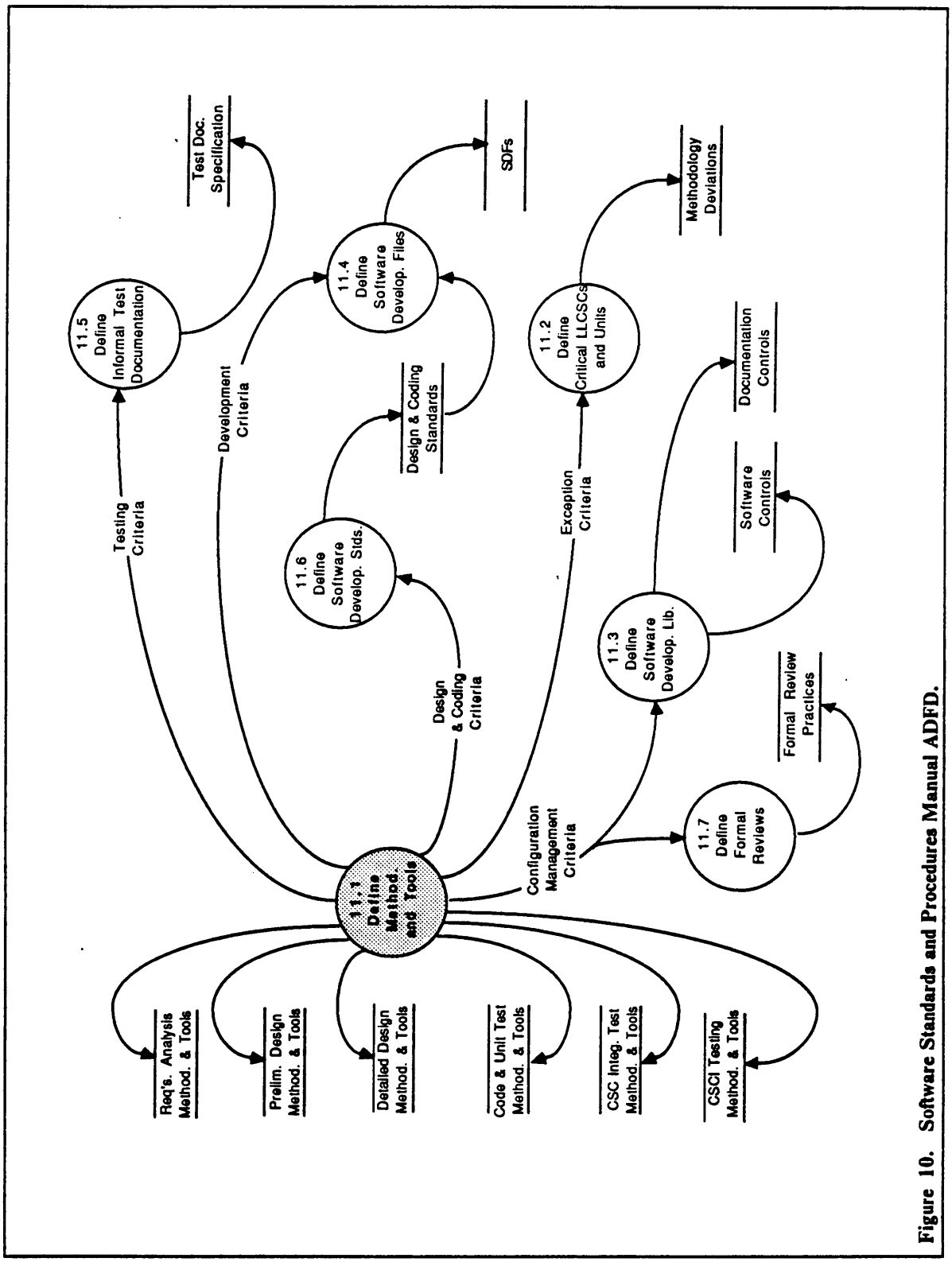

Figure 10. Software Standards and Procedures Manual ADFD.

Here is evidence for an initial justification of the previous comment that configuration management underlies all of the phases of software development. That is, configuration management criteria form the basis for the formal reviews which in turn are an integral part of each phase of the software development life cycle, as well as the basis for establishing software and documentation controls as part of a software development library.[14]

Referring again to Figure 9, process 5.1:2 is an implicit process that relies on information from the data stores generated by process 5.1:1. This process is represented implicitly, as it is considered to be a necessary peer process distinct from the analysis phase. Consequently, the allocation of CSCI requirements relies implicitly on the information obtained from the data stores generated during the analysis process. Particularly, the OCD is often a valuable source of information during allocation of CSCI requirements, when one considers that a CSCI is defined more along managerial guidelines rather than functional guidelines.

The main purpose of the requirements allocation process is to define for a CSCI both the

- SRS - Software Requirements Specification [DID25], and the

- IRS(s) - Interface Requirements Specification(s) [DID26].

The allocation of CSCI requirements results in an SRS for each of the potentially numerous CSCIs.[15] In addition, several IRSs may be defined for each SRS. Figure 11 is the ADFD for the DID describing the activities required to generate an SRS. This ADFD shows a set of process bubbles that are subordinate processes to process 5.1:2 in Figure 9. Each process in Figure 11, with the exception of process 25.8, defines a specific category of requirements. Process 25.8

---

[14] DoD-STD-2167 does not mandate a formal review for code and unit test; however, a developer may find it beneficial to incorporate an internal review process of similar formality.

[15] For simplicity, the ADFDs in this chapter model the development of a single CSCI. Extending the semantics of these ADFDs to include multiple CSCIs is trivial.

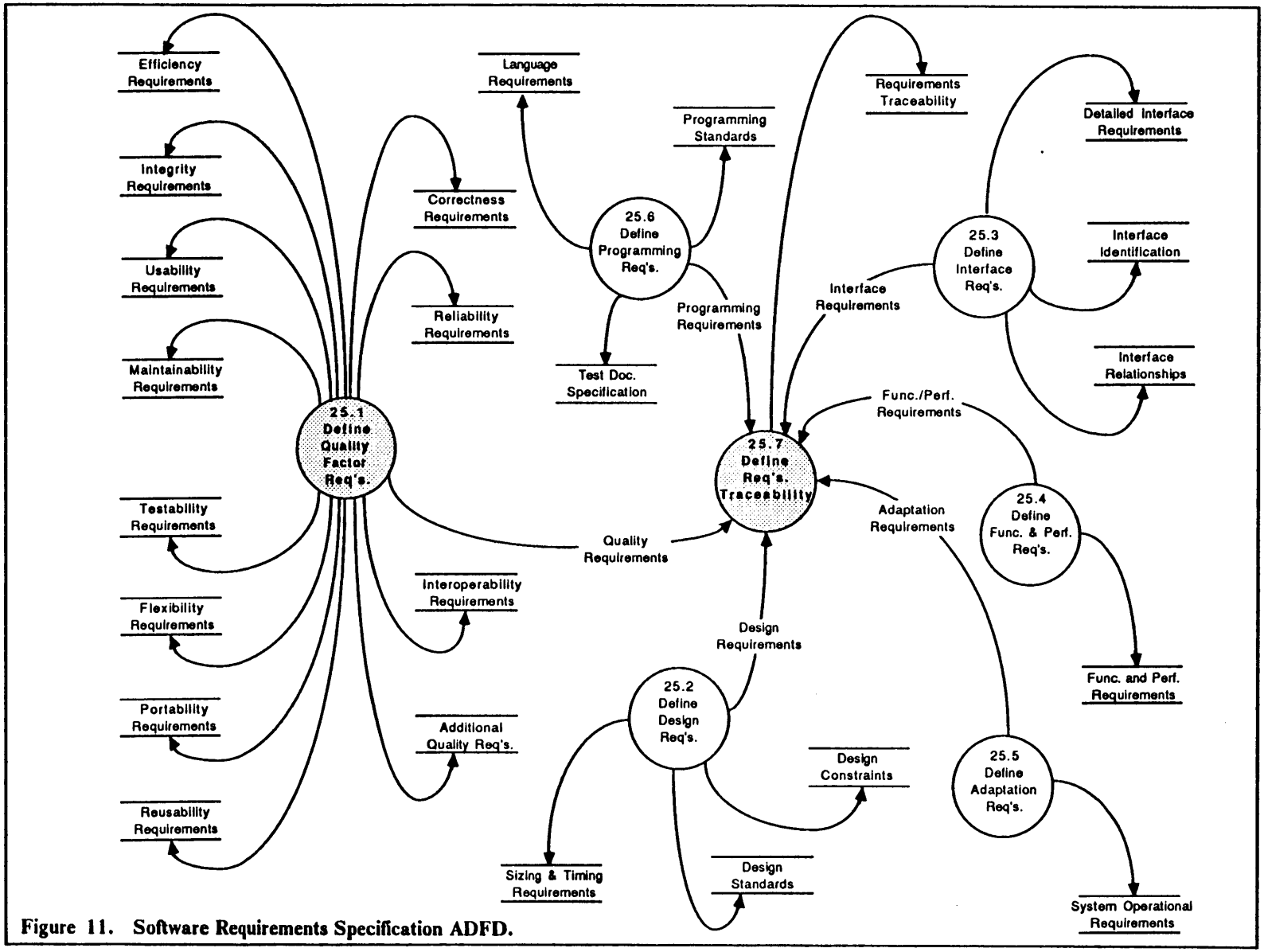

Figure 11. Software Requirements Specification ADFD.

establishes the requirements traceability for each of these categories. Again, this is an indication of the underlying software quality evaluation process. Notice also that process 25.1 defines a set of *quality factor requirements* which are a superset of the objectives defined as part of the evaluation procedure. During the allocation of requirements, unmatched requirements or discrepancies are represented by a data flow returning to the analysis process resulting in a reiteration of that process.

Upon completion of the SRS and IRS(s) for a particular CSCI, a formal review process (represented by the process bubble labeled 5.1:3), occurs for each SRS, IRS, and the OCD. Discrepancies in these specifications are represented by an implicit data flow returning to the analysis processes (an implicit indication of the underlying nature of software quality evaluation and configuration management). The result is a reiterative data flow through all of the previous processes.

After satisfactory completion of the formal review process, the SRS and the IRS(s) are the first components of the allocated baseline for a CSCI. Thus process 5.1:4 establishes the allocated baseline; represented in Figure 9 as a virtual data store.

4.1.2 Conclusions

The analysis of section 5.1 has demonstrated the applicability of the ADFD model, identified several important aspects of DoD-STD-2167, and set the basis for a procedural evaluation.

Specifically, the ADFD model has been used to establish a clear and concise graphic representation of the requirements for one of the most important phases of software development prescribed by 2167. Further, by applying the model to the DIDs for the products generated during requirements analysis, the model has explicated the need for methodology definition as an integral part of the software requirements analysis phase. In addition, evidence of a relationship to configuration

management and software quality evaluation processes, at this relatively early stage, provides an indication of the importance of both to the whole of software development.

By examining section 5.1 of 2167, the relevant importance of the both the SSPM and the SRS is not readily apparent. That is, the requirements for establishing methodologies for each of the phases of software development (as shown in the ADFD for the SSPM), and the establishment of software quality factors (or objectives), and requirements traceability (as shown in the ADFD for the SRS), do not become obvious until one examines the DIDs for these products in detail. All of this information is concisely represented in three ADFDs.

In terms of the procedural evaluation, the ADFDs for the SSPM and the SRS have identified both methodological characteristics and objectives. The importance of these discoveries becomes clearer in the next chapter.

4.2 Preliminary Design

Having established the SDP (including the SSPM, SCMP, and SQEP), and the allocated baseline for a CSCI (SRS and IRSs) during requirements analysis, the preliminary design phase represents the first step towards the actual implementation of a CSCI. Here a top level design is produced for a CSCI, and CSCI test plans are first defined. In practice, preliminary design is rarely, if ever, an activity independent of the phases that precede or follow it. Recognition of this fact is important.

Typically, activities that fall under the umbrella of preliminary design are often initiated during software requirements analysis. Likewise, some of the activities of detailed design are initiated during preliminary design. A similar statement holds for subsequent phases. Hence, some subset of the set of software development phases are usually going on concurrently.

In general, 2167 recognizes this fact; however, due to its emphasis on product generation and acquisition, this perspective is obscured by the time one reaches the discussions on the specific requirements for each development phase. As a consequence, the ADFD for preliminary design, and subsequent ADFDs, do not explicitly represent this concurrency among phases. However, one can visualize this concurrency by imagining process bubbles in both the same and different ADFDs as representing concurrent activities. By extension, data stores can be perceived as continually evolving entities, since all process bubbles are represented as either producing or consuming information from one or more data stores.

4.2.1 ADFD Analysis

Figure 12 presents the ADFD for preliminary design, section 5.2 of 2167. Here the SDP is represented as a virtual recurrent data store. Furthermore, following the Software Specification Review that occurs at the end of software requirements analysis, both the SRS and IRS(s) for a CSCI establish the allocated baseline. Hence, the virtual recurrent data store labeled as the allocated baseline represents the SRS and IRS(s) for a single CSCI.

With regard to the previous discussion, one might keep in mind that software requirements activities may be going on concurrently with preliminary design such that these data stores are continually evolving. In part, this evolution is represented by process 5.2:3, which maintains the allocated baseline and SDP based on specification changes from process 5.2:1, bidirectional data flow from both virtual data stores, and guidance from standards. As in the production of these specifications, MIL-STD-490 and MIL-STD-483 [MILS85a, MILS85b] provide the necessary guidance. Each is represented as a recurrent data store since they exist prior to preliminary design. Both military standards are represented implicitly since they are not explicitly mentioned in this section of 2167.[16]

---

[16] In later ADFDs, MIL-STD-490 and MIL-STD-483 are deliberately omitted. This omission is not meant

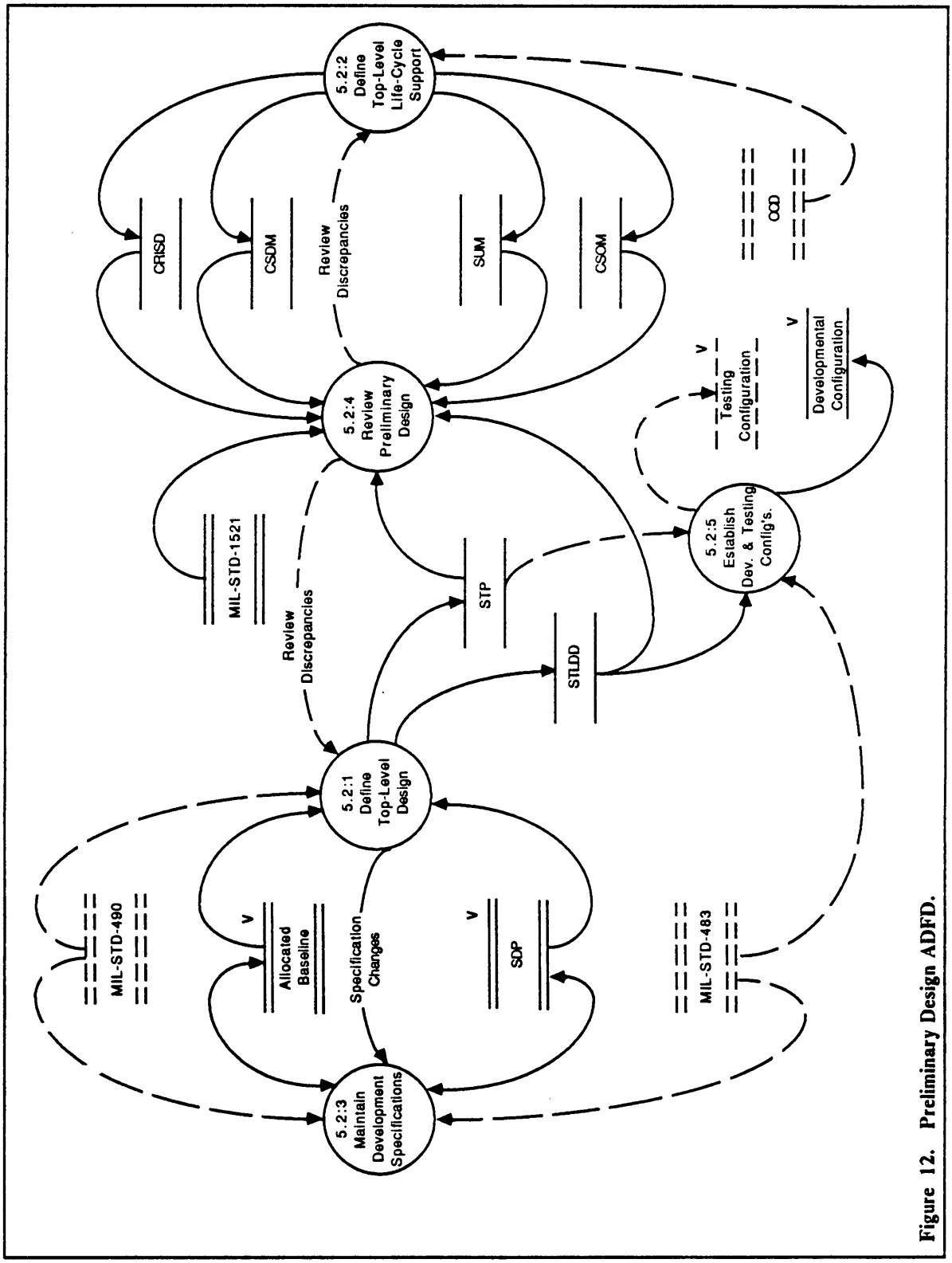

Figure 12. Preliminary Design ADFD.

Process 5.2:1 is the primary process of interest in Figure 12. As seen previously (Figure 10), the SSPM defines a distinct methodology for preliminary design. Therefore, process 5.2:1 depends, in particular, on the information found in the SSPM and more generally on the information in the SDP.

The products generated by this process are the

- STLDD - Software Top Level Design Document [DID12], and the

- STP - Software Test Plan [DID14].

An STLDD and STP are produced for each CSCI. The generation of these products follows from the requirements contained in the allocated baseline.

Figure 13 is the ADFD that captures those processes which result in the generation of the STLDD. Process 12.1 allocates functional requirements to processes that generate specific properties of the top level design. In the STLDD evidence is first found for the location of product attributes. For example, process 12.4 defines functional descriptions of the top level design. Typically, such functional descriptions are in the form of a program design language (or PDL). One of the principles identified in Figure 3 is functional decomposition, which has linkages to several attributes (e.g., reduced complexity). This is an indication that the products of preliminary design, as prescribed by 2167, are created by processes that are governed by principles not unlike those described by the evaluation procedure. Furthermore, such principles (if employed properly), should result in specific attributes being induced in the product; in this example, reduced complexity in the top level design.

---

to diminish their respective importance to the development process; rather it is more convenient to omit them from the figures so that the emphasis may be on those data stores that are deemed more significant. Similarly for MIL-STD-1521. The reader is advised to assume that the data stores for these standards are present.

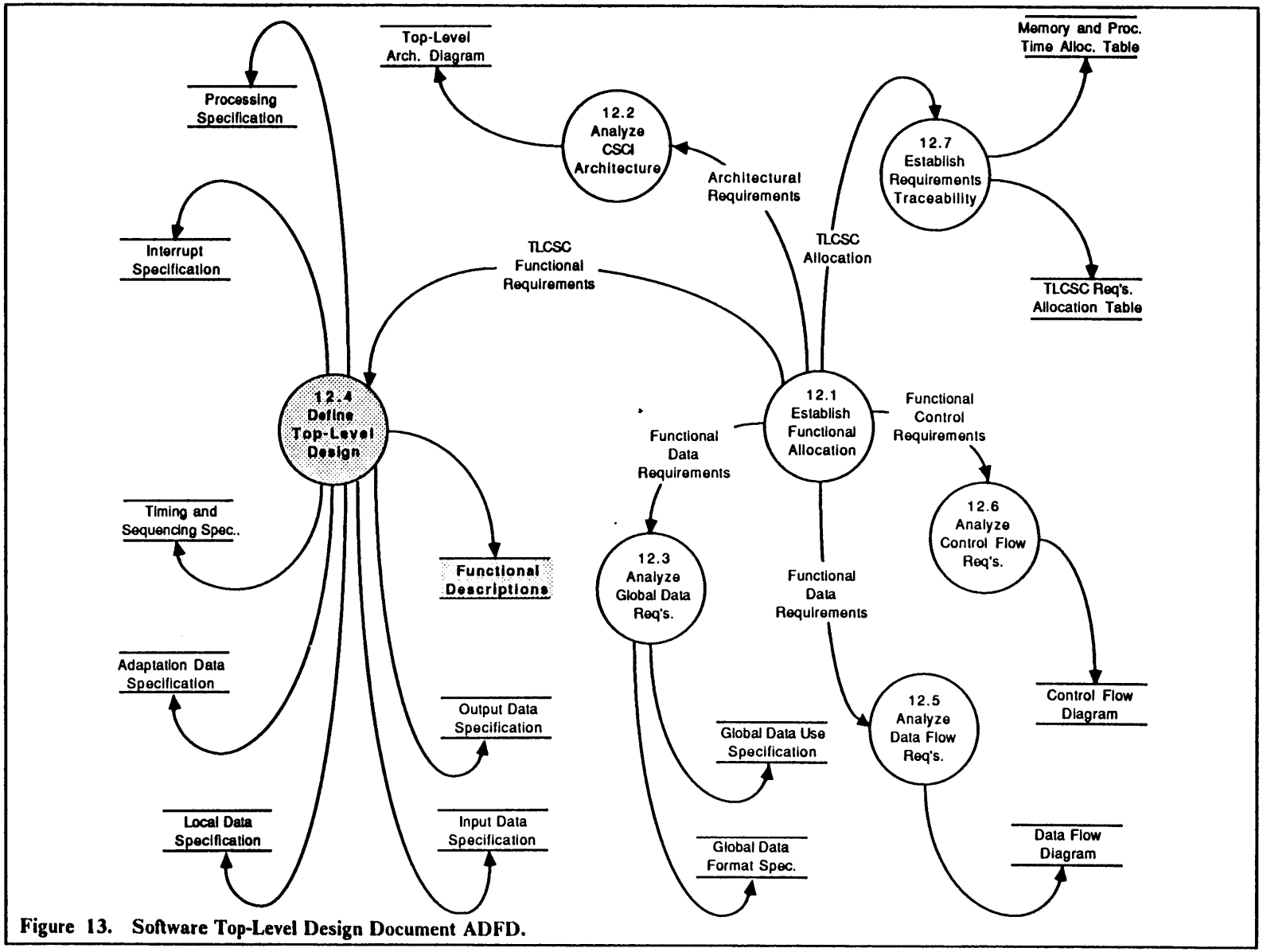

Figure 13. Software Top-Level Design Document ADFD.

Of equal significance, is that process 5.2:5 in Figure 12 establishes the developmental configuration for a CSCI, which at this stage consists of only the STLDD.[17] Additional software products are eventually added to the developmental configuration, hence a virtual data store is used to represent this collection of software products.

Process 5.2:5 is also shown as providing an implicit data flow to the *testing configuration.* The testing configuration is not explicitly defined in 2167; therefore, the data store representing the testing configuration is shown implicitly. The belief is that a testing configuration is as important to the entire development process as is the developmental configuration.

At this point, the testing configuration consists of only the STP, which is shown as providing an implicit data flow into process 5.2:5. As development continues through subsequent phases, the requirements of 2167 mandate additional testing documentation. Hence, the testing configuration is a virtual data store.

Aside from the STLDD and STP, other products are generated during preliminary design. These documents include the

- CRISD - Computer Resources and Integrated Support Document [DID24],
- CSDM - Computer System Diagnostic Manual [DID20],
- SUM - Software User's Manual [DID19], and the
- CSOM - Computer System Operator's Manual [DID18].

These products are generated in preliminary versions by process 5.2:2, represented as depending implicitly on the OCD. The CRISD, CSDM, SUM, and CSOM, along with the STLDD and STP,

---

[17] The importance of the developmental configuration is explained further as the analysis continues through the remainder of this chapter. The developmental configuration ultimately contributes to the final results of this thesis.

are part of the formal review conducted during preliminary design (process 5.2:4). As before, the review process may uncover discrepancies that result in a data flow returning to a development process indicating the need for a reiterative flow of data to correct the discrepancies. For conciseness, the CRISD, CSDM, SUM, and CSOM are collectively referred to as life-cycle support specifications and represented by a single virtual data store.

The life-cycle support specifications, like the developmental and testing configurations, are potential candidates for the location of attributes induced by principles employed during the process of product development. The recurrence of the developmental configuration in subsequent ADFDs is paralleled by the recurrence of the life-cycle support specifications.

4.2.2 Conclusions

Having established the requirements and development methodologies during the requirements analysis phase, the preliminary design phase produces the top level design for a CSCI, establishes the developmental configuration, the testing configuration, and defines preliminary life-cycle support specifications.

The developmental configuration should be produced in accordance with the methodology defined in the SSPM for preliminary design. The STLDD, which forms the initial developmental configuration, has been shown to possess properties which lead one to infer that certain principles must be employed during its production. In the context of a procedural evaluation, these principles are known to have linkages to specific attributes and objectives.

As in the previous section, the application of the ADFD model to a section of 2167, and a corresponding DID, has concisely captured the explicit requirements of preliminary design. More importantly, by adopting the critical perspective offered by the evaluation procedure, the ADFD model has allowed one to establish that a methodology that embodies at least the principles of

functional and hierarchical decomposition is required to ensure that the products produced during preliminary design possess the required properties. In effect, these are implied requirements of 2167. The ADFD model has explicated the intent of 2167 with respect to the relationship of the product to the process by which the product is generated.

The relevance of the above statement may not be clear until one recalls, from Chapter 2, that 2167 is *product* oriented and not *process* oriented. The explicit requirements are product requirements. The ADFD model reveals also process requirements, at least implicitly.

With this in mind, consider the question of whether or not 2167 is a methodology. Does this also imply the methods one should use? One might argue in the affirmative, but consider that 2167 requires that methodologies are to be established for each phase of software development. Under the definition offered earlier in this thesis, a methodology is a collection of methods.

A credible proposition is that 2167 establishes a *meta-methodology*. That is, 2167 is product oriented as originally claimed, and that the process requirements inferred from the products are methodology requirements. This suggests that the selection of the methodologies defined in the SSPM during software requirements analysis should not be a casual affair. Rather, it is necessary to examine critically and carefully the product requirements of a particular phase prescribed by 2167 in order to determine the desired product properties. One can then determine the necessary principles by following the linkages defined in Figure 3. Based on these principles, one can establish a set of objectives by examining the linkages in Figure 2. Having established these linkages using bottom-up reasoning, one should be able to define or select a methodology which, when applied top-down in conjunction with the activities prescribed by 2167, leads to products with the desired attributes.

The issue of a testing configuration is a subjective one. The argument is made here that testing is of equal importance as an item of configuration as is the developmental configuration. The requirements of 2167 call for an amount of testing documentation proportional to the development

documentation, yet it does not appear explicitly as though the same level of control is applied to the set of testing specifications.

The testing configuration demonstrates the usefulness of the implicit entities provided by the ADFD model. A conjecture is made that the intent of 2167 is to require a testing configuration. Such a supposition seems plausible, especially if one considers that 2167 requires that CSCI testing be performed by parties sufficiently removed from the development process; which leads one to speculate on the potential for mismanagement if such a configuration is not provided. In this thesis, the testing configuration is held to be as important as the developmental configuration. Thus, in subsequent ADFDs, the testing configuration appears implicitly along with the explicit developmental configuration.

4.3 Detailed Design

The previous two sections have set the stage for much of what follows in describing the remaining phases of the 2167 software development life cycle. The arguments for the importance of both the developmental configuration and testing configuration have been made. In this section, both of these configurations continue to evolve, along with the life-cycle support specifications.

The importance of the SDP, particularly the SSPM, continues to be affirmed given that a separate methodology may be defined that governs the detailed design process. As always, the subverted nature of configuration management and software quality evaluation exists, but as mentioned earlier, a detailed discussion of each is deferred until the next chapter.

4.3.1 ADFD Analysis

The detailed design phase is captured in the ADFD shown in Figure 14. Considering process 5.3:1, note that the developmental configuration, testing configuration, and allocated baseline are all represented implicitly (the reason that each of these is recurrent and virtual should by now be obvious). Each is required since the products that are generated at this phase are in some way related to components of each.

Again the SDP is represented indicating that the methodology defined in the SSPM for detailed design is employed during this phase. The products generated by process 5.3:1 are as follows:

- SDDD - Software Detailed Design Document [DID31],
- DBDD - Data Base Design Document [DID28],
- IDD - Interface Design Document [DID27],
- STD - Software Test Description [DID15],
- SDF(s) - Software Development File(s).

The generation of the SDDD is predicated on information found in the allocated baseline and the developmental configuration. The SRS and IRS(s), along with the previously generated STLDD, possess the information from which the SDDD is derived. Figure 15 illustrates the processes, which are subordinate to process 5.3:1, that result in the SDDD. Note that these processes are very similar to those that result in the generation of the STLDD (see Figure 13).

The DBDD and IDD, of which there may be many, are specifications that are closely related to the STLDD and are also generated in accordance with requirements from the allocated baseline,

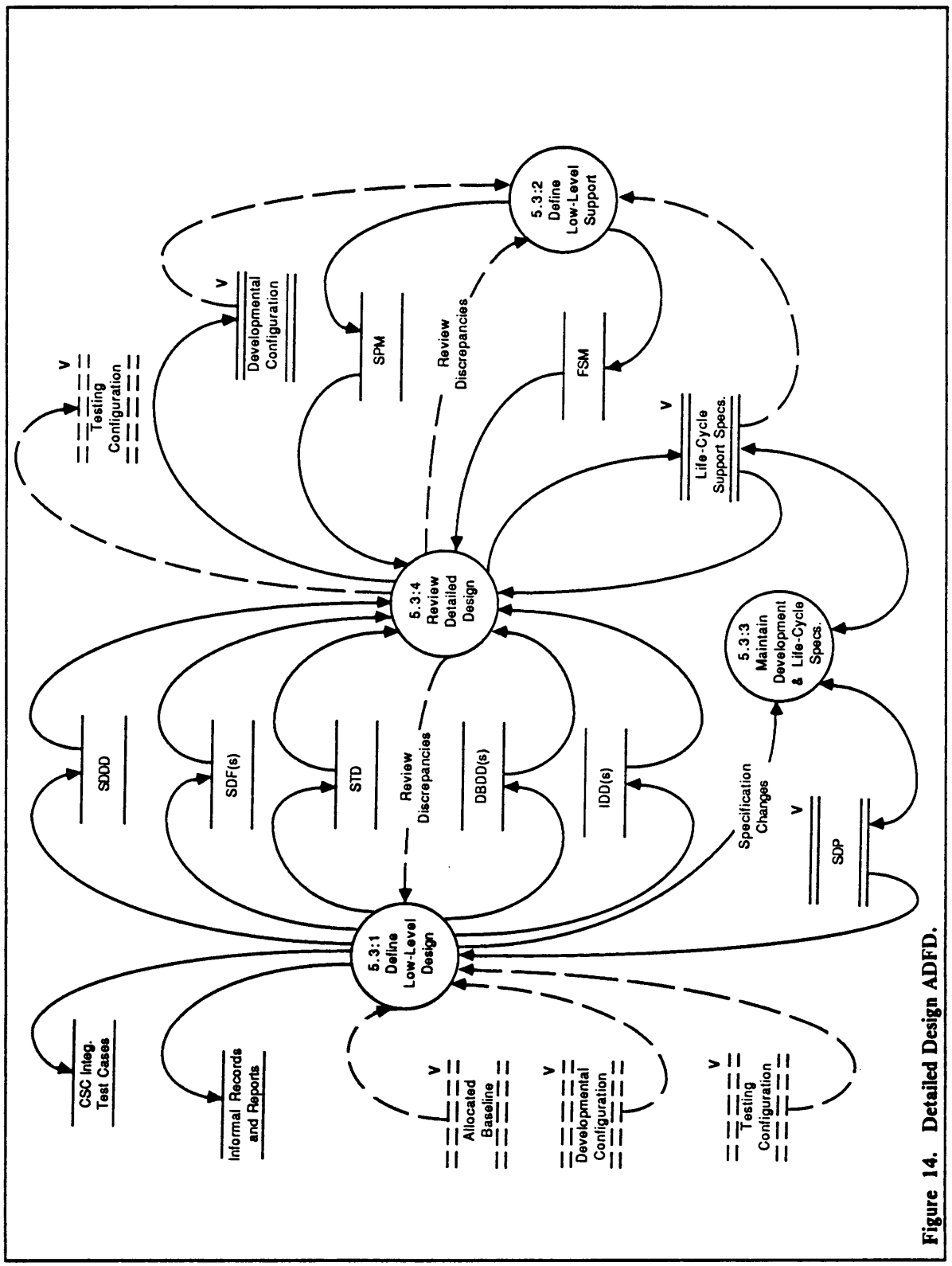

Figure 14. Detailed Design ADFD.

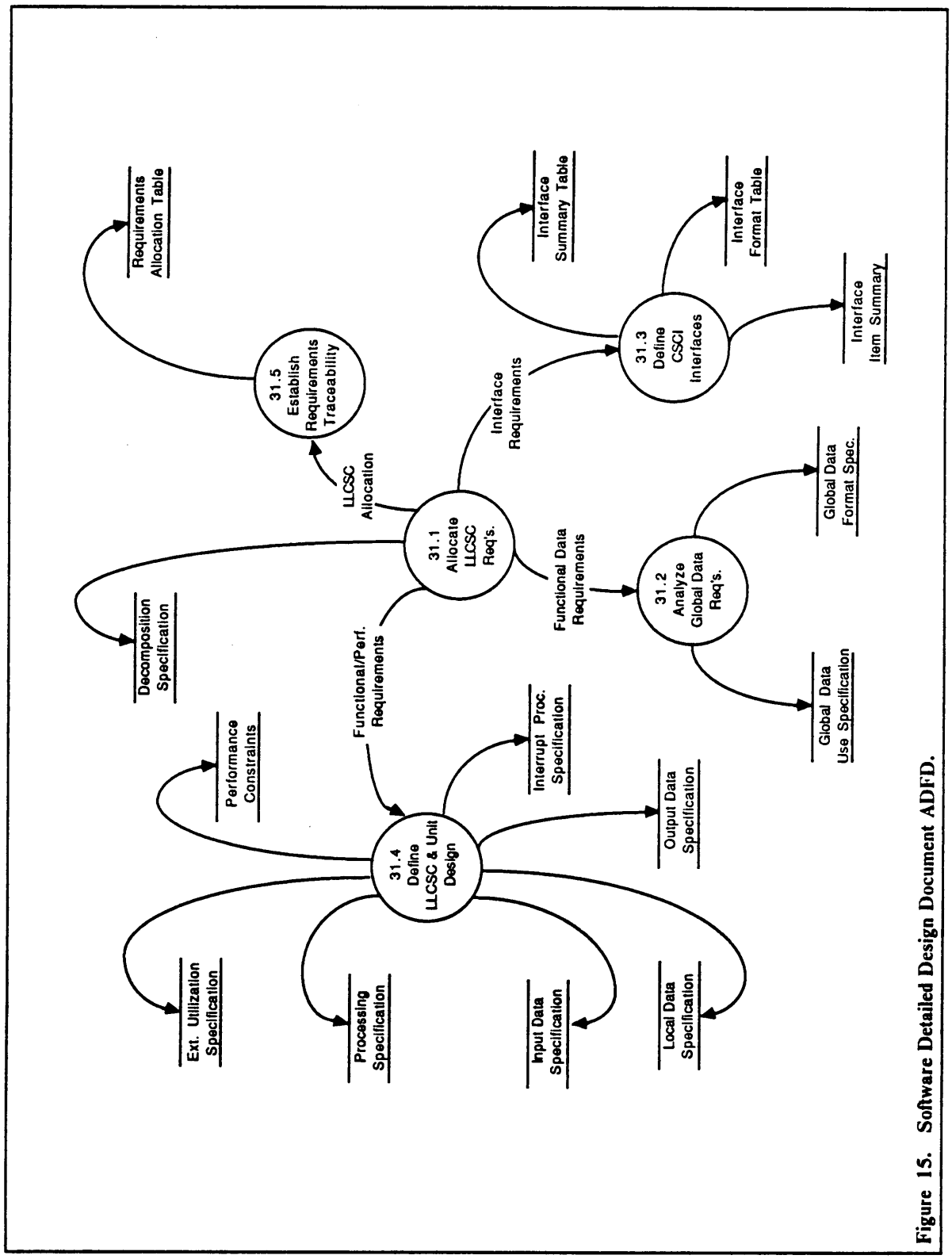

Figure 15. Software Detailed Design Document ADFD.

particularly from the IRS(s). The ADFD for a DBDD is shown in Figure 16, and the ADFD for an IDD is shown in Figure 17. Figure 16 captures the requirements from the corresponding DID. The requirements are allocated to four processes that define the various data base structures and interfaces. Consider that process 28.3 establishes a requirements traceability table. Thus a traceability is established from the design document to the allocated baseline.

Notice that the process requirements for generating the IDD, as shown in Figure 17, are not well developed. One might speculate on additional interface design activities to remedy this apparent shortcoming.[18]

Focusing again on Figure 14, the STD consists of the test descriptions for CSCI testing. The information found in this specification is derived from the STP generated during preliminary design and entered into the testing configuration. This substantiates to some degree the claims made earlier about the testing configuration, since its evolution parallels that of the developmental configuration. The significance of the testing configuration becomes more apparent during the CSCI testing phase (section six of this chapter).

The final items generated by process 5.3:1 are Software Development Files. The requirements for SDFs are not found in a separate DID. Rather, the format and content of SDFs are defined in the SDP. The perceived tendency is to downplay the importance of the SDFs since they are not usually deliverable items. The SDFs are typically a repository for documented design decision, incorporating the latest requirements and design changes until the formal documentation in the developmental configuration can be updated.[19]

---

[18] In practice, many integration problems arise from ill-defined CSCI interfaces. In keeping with the intent of 2167, one might find it beneficial to define an *interface design methodology*, and incorporate such a methodology into the SSPM as a peer to the detailed design methodology.

[19] Experience has shown that the SDFs often become the final product since in many cases the time and resources are not available to maintain a large number of development documents over a long period. Hence, the SDFs are often crucial to the development effort.

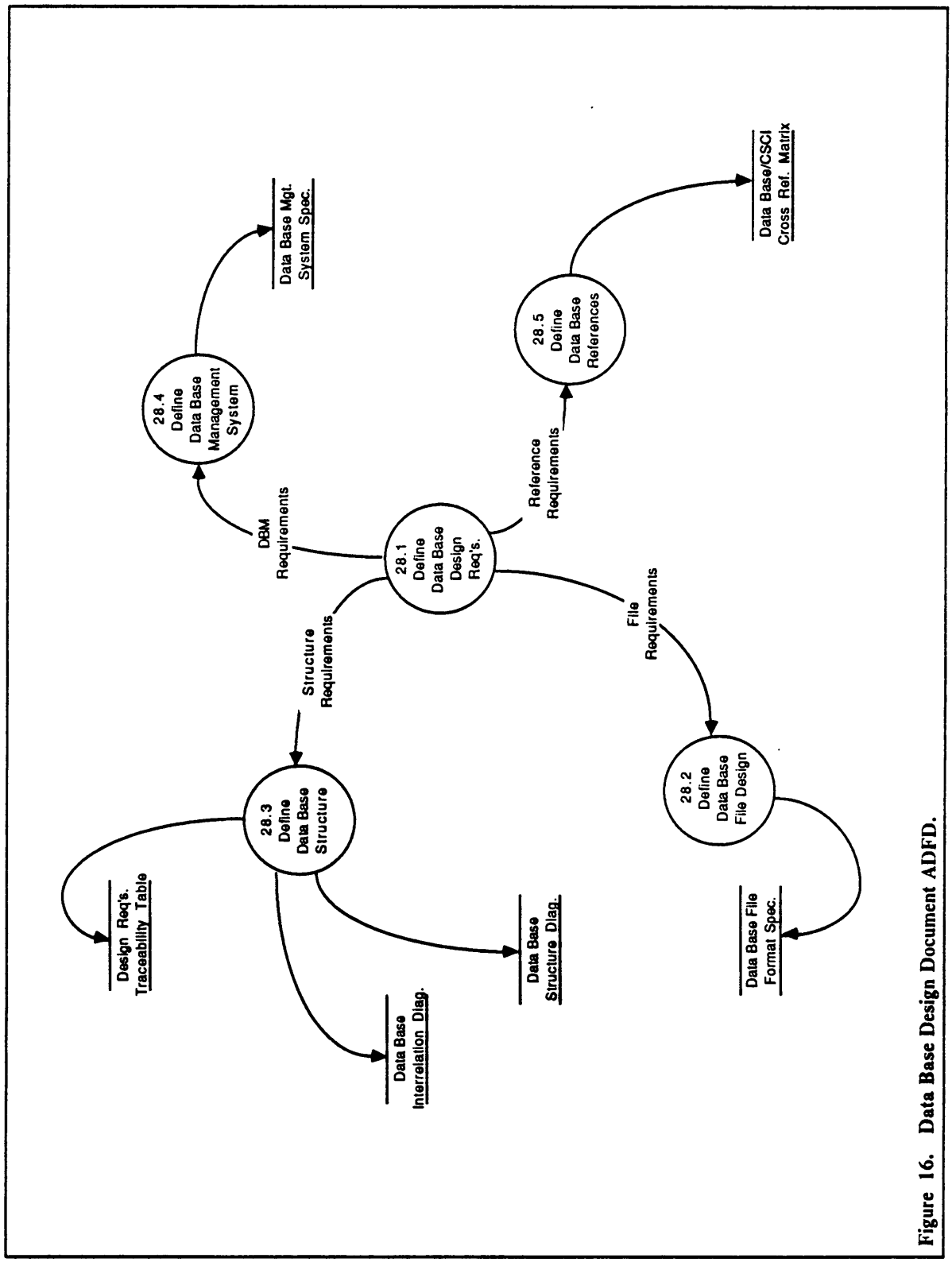

Figure 16. Data Base Design Document ADFD.

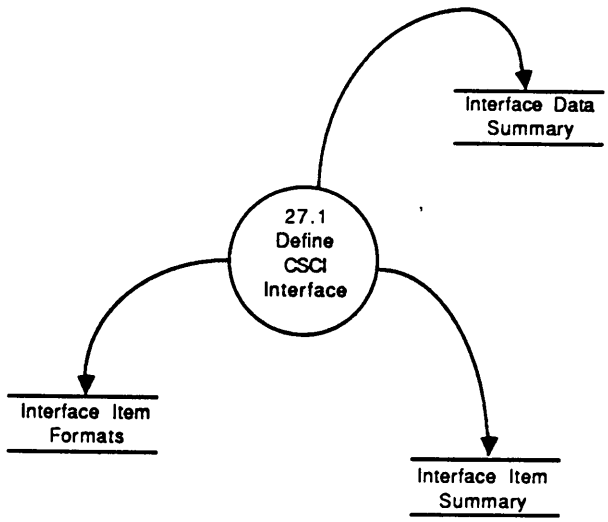

Figure 17. Interface Design Document ADFD.

Process 5.3:2 in Figure 14 is a peer process to process 5.3:3. The implication is that the developmental configuration and life-cycle support specifications play an important role in defining the low-level support specifications. The following products define the operational support for the software components under development:

- SPM - Software Programmer's Manual [DID21],

- FSM - Firmware Support Manual [DID22].

The SPM, FSM, and other life-cycle support specifications, along with the products generated by process 5.3:1, are part of a formal review process indicated by the bubble labeled 5.3:4. The formal review process is a critical design review of all these specifications. Upon successful completion of this process, the SDDD, DBDD(s), and IDD(s) are entered into the developmental configuration, the SPM and FSM become part of the virtual data store representing the life-cycle support specifications, and the STD is implicitly entered into the testing configuration.

The completion of the critical design review signals the end of the formal design phases. However, the software development life cycle is only a model. Typically design processes continue after the critical design review and overlap into the coding and unit testing phase which follows.

4.3.2 Conclusions

At this point the detailed design phase is essentially completed. However, as discussed previously, there are rarely such clear cut boundaries between phases. Processes from all phases are typically proceeding concurrently and interacting. What are the common points for concurrent interaction among phases? Exactly those virtual data stores that have been represented as being recurrent:

- the allocated baseline,

- the developmental configuration,
- the testing configuration, and
- the life-cycle support specifications.

That is, 2167 presents each phase in a sequential manner, methodically laying down requirements for the generation of deliverable products. The interaction points across phase boundaries and the relationship of processes at these interaction points is not readily apparent from an examination of 2167.

The ADFD analysis offers a tool for obtaining a clearer picture of these relationships through its ability to concisely represent the processes required in each phase and the products generated. Functionally related products can then be represented as single entities that are seen to evolve throughout the software development life cycle. These single entities (recurrent virtual data stores) enable one to define and focus attention on relevant aspects of the development process.

In the context of a procedural evaluation, the ability to isolate products in this manner can provide insight into the principles employed in their production. That is, one may be able to identify specific attributes in a particular collection of products. For example, one might be interested in examining the developmental configuration apart from the testing configuration. Such an examination might focus on the principles one would employ so that attributes desired in the developmental configuration are present. The attributes desired for the products composing the testing configuration may be different from those for the products in the developmental configuration. Hence, using the previously established linkages from principles to attributes, the principles common to both the testing configuration and the developmental configuration could be identified.

Extending this bottom-up reasoning approach, one could determine the linkages between principles and objectives. Hence an assessment could be made as to whether or not certain objectives have

been achieved. Thus, the assessment of product quality (i.e., the extent to which products possess attributes indicating that objectives have been met) can focus on not just the products, but also on the processes by which the products are generated. The processes one must examine can be determined by examining the ADFDs.

The analysis up to this point has partially established that software quality is related to both the products (i.e., the testing and development configurations), and the process of software development. Furthermore, if one takes the point of view of a strong configuration management process, then software quality is also related to configuration management, since the product configurations are maintained by the configuration management process.[20]

4.4 Coding and Unit Testing

Following the detailed design phase, the developmental configuration contains the necessary information required to produce code for a particular CSCI. As shown in the ADFD for the STLDD (Figure 13), a CSCI is decomposed into Top-Level Computer Software Components (TLCSCs) during preliminary design. Further refinement occurs during detailed design such that the TLCSCs are first decomposed into Lower-Level CSCs and then units. The main objects of interest at this point of the development process are the units that represent the physical code.

The coding and unit test phase is closely tied to the subsequent CSC integration and testing phase. Many of the integration and testing activities that are formally defined during CSC integration testing occur during coding and unit testing.

---

[20] This notion of the dual nature of software quality evaluation is more fully developed within the context of 2168 in the next chapter.

Again the pattern of overlapping processes is recognized by the occurrence of the developmental configuration and the testing configuration, further adding evidence to mutual importance of these configuration items to the overall development process.

4.4.1 ADFD Analysis

The coding and unit testing phase is captured by the ADFD in Figure 18. Process 5.4:1, like all of the previous development processes, relies on the SDP for guidance during the development of code and unit testing. As shown previously, a separate methodology and tool set is defined in the SSPM.

The SDFs appear again, this time providing information to the development process as well as evolving from new information generated during coding. Similarly, the developmental configuration contains all of the design documentation which represents the functional requirements for a CSCI. Particularly, the SDDD at this point is required to be of a sufficient level of detail to allow code development directly from the detailed design specification. The SDFs may be of significance here if one considers that the translation of a logical unit into code could require more than the PDL or structured English description in the SDDD.[21]

The testing configuration is depicted implicitly, as explained before. However, the data flow from the testing configuration to process 5.4:1 is explicit since 2167 states that STDs are required. Since the STDs are not explicitly part of the testing configuration, this juxtaposition of an implicit data store and explicit data flow captures exactly what is intended.

The products generated by process 5.4:1 consists of the following:

---

[21] For example, the SDDD may specify in PDL a "wait for interrupt". The actual mechanisms for interrupt processing may be more appropriately defined in the SDF.

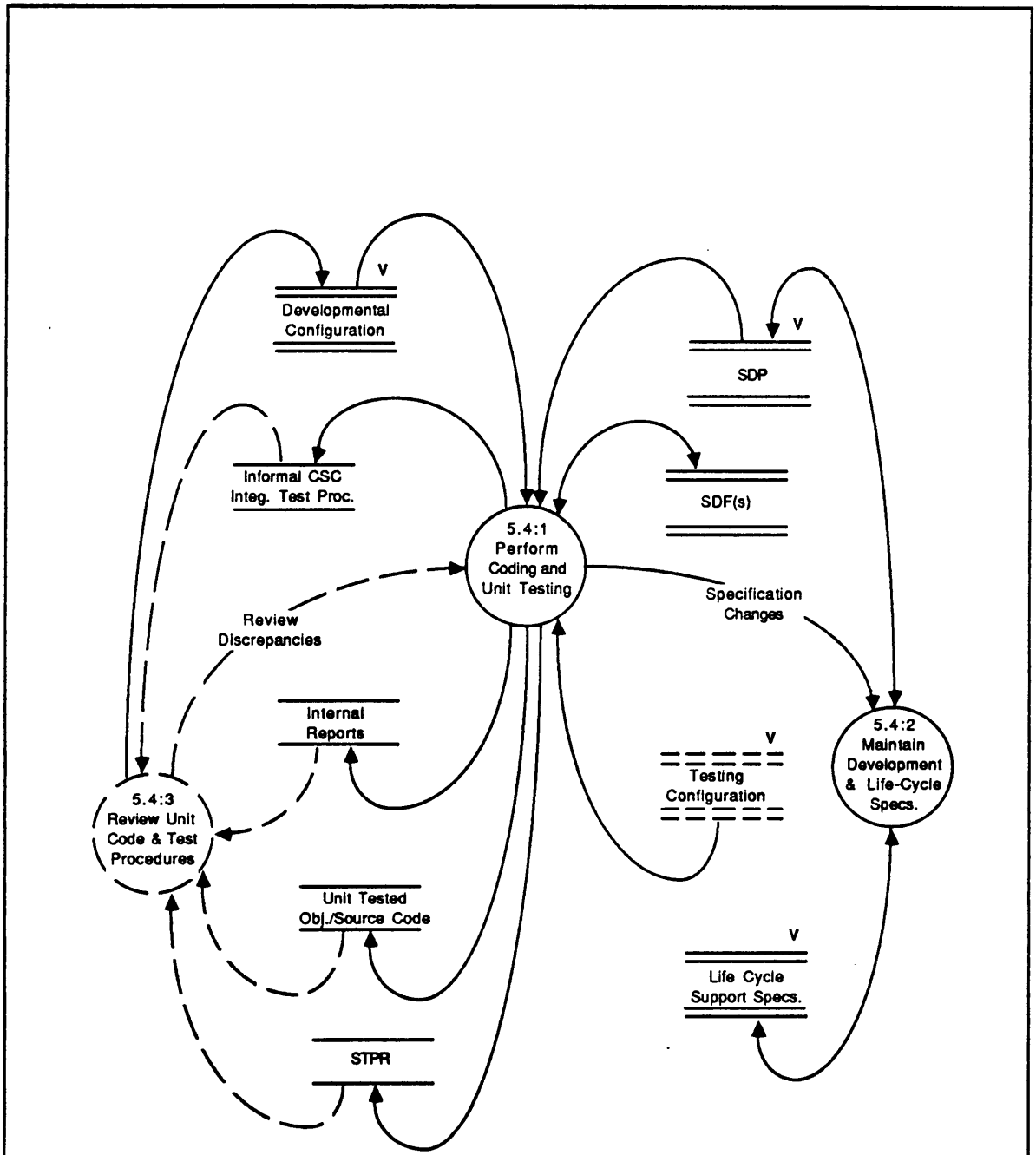

Figure 18. Coding and Unit Testing ADFD.

- informal CSC integration test cases,
- internal reports,
- unit tested source and object code, and the
- STPR - Software Test Procedures [DID16].

Each of these is shown as providing an implicit data flow into process 5.4:3, since 2167 does not require an explicit formal review process during the coding and unit testing phase. However, a review process of similar formality is considered implicit during coding and unit test, and should thus be incorporated. In support of this argument, consider that 2167 requires that the unit tested source and object code be entered into a CSCI's developmental configuration. Recalling that the developmental configuration is a recurrent data store, and is an interface point of the concurrency across phase boundaries, a formal review process is essential to maintaining a level of control over the developmental configuration commensurate with its importance. The relationship of the coding and unit testing phase to the developmental configuration is represented by the explicit data flow from process 5.4:3 to the virtual recurrent data store representing the developmental configuration.

The STPR is not entered into the testing configuration at this point. Rather 2167 requires that the STPR be formally reviewed during CSC integration testing. For this reason, the entry of the STPR into the testing configuration is deferred until the next section.

4.4.2 Conclusions

The number of recurrent data stores in the previous figure, and the lack of newly generated data stores indicates that most of the specifications required by 2167 have been developed. The ADFD

model affirms what one would expect at this point in the life cycle: that the design may still be evolving, but the emphasis is no longer on documentation.

A conjecture is made that at this stage the underlying processes of configuration management and software quality evaluation are critical. Up to this moment, the developer is required to generate and formally exhibit the ongoing development and testing configurations, and life-cycle specifications. All efforts of the previous development are now focused on coding and unit testing. Here is where many design flaws and discrepancies are first encountered. The incorporation of an implicit review process into the code and unit testing phase is meant to emphasis the need for configuration management and software quality evaluation during the development.

Consider again that the SSPM requires a methodology for code and unit testing. Such a methodology embodies the principles one should apply to the coding process. Thus, one could examine the coded units and test results from this phase and assess how well those principles are being applied. If one can establish that principles are indeed being employed correctly, one can infer which objectives are being met (by examining the linkages). Hence, one can assess the quality of the product by examining the process in terms of the principles employed and taking the necessary steps to correct any discrepancies.

4.5 CSC Integration and Testing

The CSC integration and testing phase consists of the integration of CSC units and CSC testing, followed by a formal review of the readiness of the CSCI for formal testing. The recurrence of the SDP, developmental configuration, testing configuration, and life-cycle support specifications continues in this phase, further demonstrating their respective criticality.

4.5.1 ADFD Analysis

Similar to the code and unit testing phase, the recurring data stores outnumber the newly generated data stores. Two processes are of primary interest in Figure 19. The first is process 5.5:1 which depends on the developmental configurations for information from which to conduct CSC integration, and further depends on the testing configuration for CSC testing. Again the SSPM provides the methodological guidance for this phase, as evidence by the SDP.

The newly generated items at this point consist of the following:

- records and reports,
- integrated CSCI source and object code, and the
- integration test results.

Note also that the SDFs are updated by process 5.5:1 to contain the most recent information pertaining to the results of the integration of each unit into a CSC. As mentioned in the discussion of unit testing, the STPR is carried over to this phase and updated with further information from the CSC testing process.

Process 5.5:3 follows from the completion of process 5.5:1. The STPR and the results from integration and testing are formally reviewed in order to determine the readiness of the CSCI for testing. Furthermore, any life-cycle support specifications in their final form, such as the SPM and FSM, are reviewed. The culmination of the test readiness review results in an implicit data flow to the testing configuration, which represents the entry of the STPR (generated in the previous phase) into the configuration item. In addition, the developmental configuration evolves to reflect any additional information that updates the current state of the CSCI.

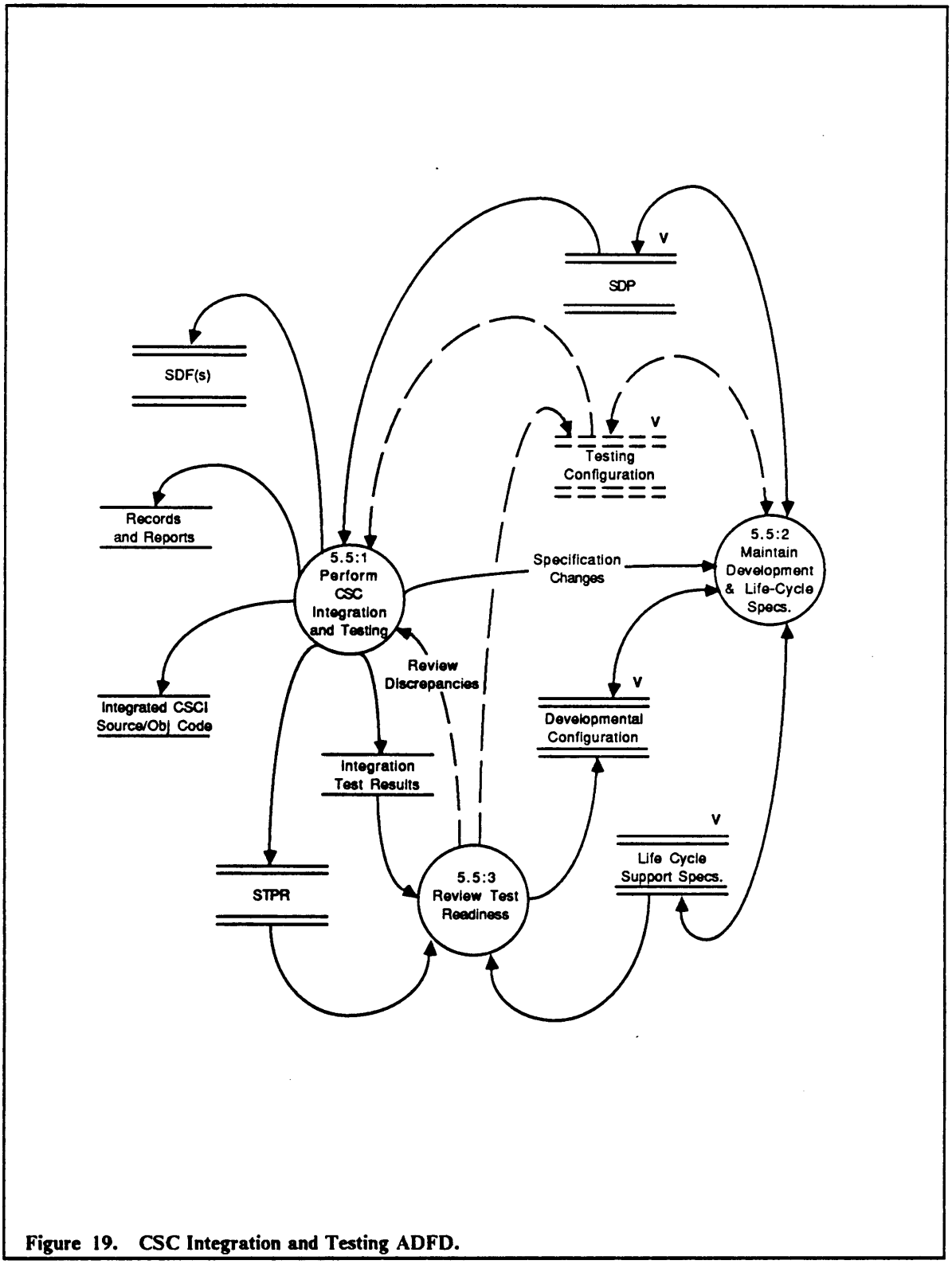

Figure 19. CSC Integration and Testing ADFD.

4.5.2 Conclusions

At this stage the testing configuration is of great significance. Prior to CSC testing, the testing configuration has been evolving, but it is now completed pending CSCI testing. Similarly the developmental configuration has continued to evolve, and at this point in the development reflects the complete CSCI. The life-cycle support specifications are also in final form.

The arguments made previously, concerning the location of product attributes and the use of principles embodied in the methodology, still hold during the CSC integration and testing phase. Thus, the potential exists for a software quality evaluation process to concurrently assess the development process, using the virtual recurrent data stores as focal points.

4.6 CSCI Testing

The CSCI testing phase results in completed CSCIs being formally reviewed and subsequently delivered. The virtual recurrent data stores, existing in various evolutionary states throughout the development process, are coalesced into a final product specification. Subsequent configuration audits result in a final product baseline which represents the delivered product.

4.6.1 ADFD Analysis

Figure 20 depicts the processes and data stores relevant to the CSCI testing phase. Process 5.6:1 represents the accumulation of the results of all previous development phases. The allocated baseline which consists of the CSCI requirements, along with the developmental configuration and

implicit testing configuration, provide the information from which the final CSCI products are produced. Once again, the SDP governs this phase of development in the form of the CSCI testing methodology defined in the SSPM.

The products generated by process 5.6:1 include the following:

- VDD - Version Description Document [DID13],
- SPS - Software Product Specification [DID29],
- STR(s) - Software Test Report(s) [DID17],
- updated source and object code, and
- records and reports.

The SPS consists of all design documentation and code. In effect, the developmental configuration becomes the SPS. Hence, the SPS is represented as a virtual data store.

The SPS, VDD, updated source and object code, and the life-cycle support specifications represent the items that are reviewed as part of the physical configuration audit, denoted by process 5.6:4. In addition to the physical configuration audit, a functional configuration audit is also conducted (process 5.6:3) at which the STR(s) and life-cycle support specifications are reviewed. These audits represent the final formal review processes conducted for a CSCI. The culmination of these review processes is the establishment of the final product baseline, depicted in Figure 20 as being generated by processes 5.6:3 and 5.6:4.

The final product baseline may optionally be installed and checked out, in which case process 5.6:6 is incorporated into this last phase of the software development process. Process 5.6:6, if included, is shown as producing the delivered software.

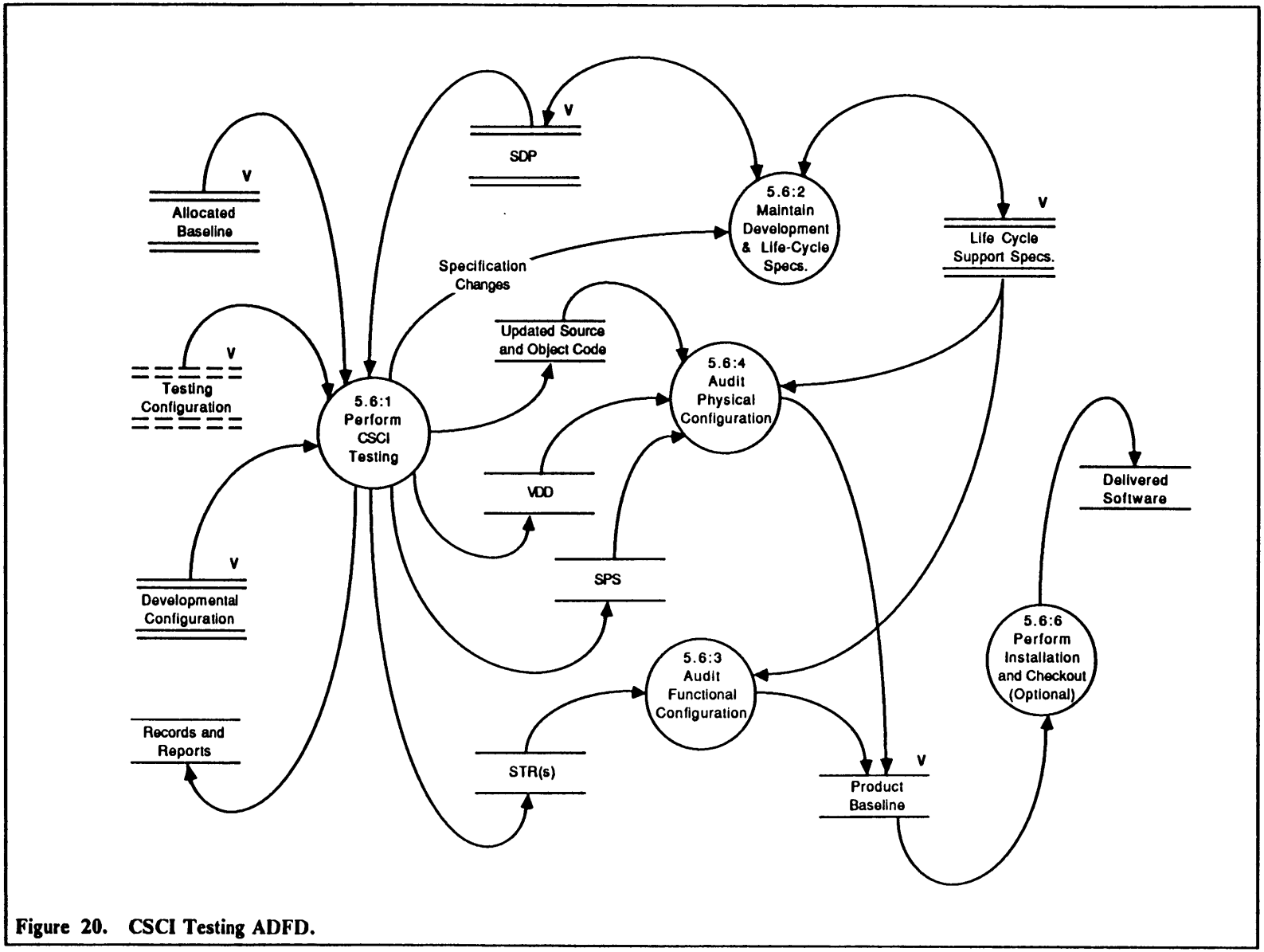

Figure 20. CSCI Testing ADFD.

4.6.2 Conclusions

The amount of information captured by the ADFD in Figure 20 is deceptive. The use of virtual recurrent data stores throughout the previous ADFDs has abstracted the results of the entire development process into just a few virtual products. Such a technique is useful for capturing a lot of information and presenting it concisely. The ADFD model has proved to be a convenient tool for representing features of the development process prescribed by 2167. However, one must be cognizant of the entire development process to appreciate the volume and complexity of the information represented in just a few virtual data stores.

In this final phase of the development process, the recurrent virtual data stores that are critical to each of the previous phases, and which are the focal points for the underlying software quality and configuration management processes, result in a final software product.

With the generation of the final software product, the typical approach in the past has been to evaluate the development process by evaluating the quality of the final product. The research presented here has attempted to show, through the ADFD based analysis, that the software quality evaluation process can occur concurrently with the software development process.

The virtual recurrent data stores represent the focal points for this evaluation, since they represent the final product in its evolutionary form. Furthermore, the evaluation procedure can lead one to infer the principles that must be employed, provided that attributes can be located in the evolving products. These principles are embodied in the methodologies and tools that are defined in the SSPM. In order to assess the quality of a product at any particular stage, one can examine the product for indications of particular attributes. The existence of established linkages then allows one to assess how well particular objectives are achieved. In addition, one can also assess the process by which the products are being generated by determining whether or not the principles required to achieve a particular objective are properly employed during the development process.

Furthermore, the meta-methodological characteristics of 2167 allow one to establish this evaluation framework within the development process

4.7 Conclusions

The viability of the ADFD model as a useful tool for concisely representing process-product relationships is demonstrated. Applying this model to the six sections of 2167, which correspond to the six phases of the 2167 software development life-cycle, explicated information about the processes and products required by 2167.

The nature of 2167 is such that having a concise representation of the requirements, and the implicit and explicit relationships of processes and products yields insight into the intent of the standard, such that one may better assess the degree to which one's own development process is in compliance with 2167. This information may allow one to better meet the requirements for quality defense system software development. For example, 2167 does not require a testing configuration as a separate configuration item. Such a procedure, if included into the development process, may provide a clear indication of the desire of the developer to not only comply with the requirements of the standard, but also augment those requirements so that a better product is produced and delivered.

The claim is made that 2167 is perhaps better viewed as a meta-methodology. From the perspective of a meta-methodology, one can define the actual development methodologies that are to be employed. The opportunity is thus presented for the developer to augment the requirements in a manner just described. For example, the CSCI interface design appears to be lacking in requirements for procedures to aid in alleviating the problem of ill-defined interfaces. Under the meta-methodology provided by 2167, one can perhaps increase the effectiveness of the SSPM by

providing additional methodologies and tools to assist in the interface design process. Such an effort would be a clear indication of the interest on the part of the developer in complying with the intent of the standard.

On several occasions, reference has been made to both software quality evaluation as well as configuration management, and their relationship to DoD-STD-2168. Having developed the idea of the underlying nature of these two processes with respect to software development, chapter five relates the software quality process to the software development process, including configuration management.

5.0 Analysis of DoD-STD-2168

The analysis of DoD-STD-2168 is motivated by the relationships indicative of the configuration management and the software quality evaluation requirements expressed in 2167. As explained in chapter two, the software quality evaluation processes prescribed by 2167 is defined in the context of the six phases of the software development process. In the examination of 2168, a one-to-one correspondence emerges between the description of the software quality process in 2167, and the software quality evaluation requirements of 2168. As defined earlier in this work, 2168 requires:

1. the evaluation of software requirements,

2. the evaluation of software development methodologies,

3. the evaluation of software and documentation, and

4. the mechanism(s) necessary to ensure that changes are made.

The analysis presented in this chapter focuses on capturing, with the ADFD model, the relationship of the development process and the above requirements for software quality evaluation. This analysis demonstrates the underlying nature of the configuration management process and software quality evaluation process in relation to the software development process.

As in the analysis of 2167, the theme of objectives, principles, and attributes forms the perspective on which the analysis of 2168 is based. In fact, this perspective is necessary for fully comprehending the requirements of 2168 for the evaluation of software development methodologies.

5.1 Relationship to DoD-STD-2167

The analysis of 2167 reveals that at each phase of the development process, a set of data stores is continually evolving, concluding with the CSCI testing phase. These data stores are represented as virtual recurrent data stores. The argument is made that, in reality, any one phase of development may be going on concurrently with a previous or subsequent phase. The virtual recurrent data stores identified by the ADFD analysis as focal points for interaction across phase boundaries include:

- the SDP,
- the allocated baseline,
- the life-cycle support specifications,
- the developmental configuration, and
- the testing configuration.

In the ADFD presented in Figure 21, each of the above virtual recurrent data stores should be identifiable. Focussing attention only on those process bubbles that are shaded, one observes data flow from each of these data stores to the shaded processes (parenthetically labeled 2167). The process labeled "perform software development process," represents an abstraction of one of the

six phases of software development. This process embodies each and all processes defined by 2167 for the development of a CSCI.[22]

The SDP data store represents the recurring nature of the software development specifications. The definition of methodologies for each phase, as part of the SSPM, is particularly stressed. The allocated baseline represents the requirements for the development of a CSCI and is particularly relevant when one considers the quality factor requirements identified previously in chapter two.

The emphasis of 2167 on product generation is manifested by the recurrence of the life-cycle support specifications, the developmental configuration, and the testing configuration, throughout each development phase. The importance of each of these virtual recurrent data stores is again stressed as a common focal point for the concurrent activities of each development phase.

In the context of the procedural evaluation, the argument is advanced that attributes may be located in each of the life-cycle, development and testing specifications. The importance of the potential for locating product attributes in these specification is emphasized in Figure 21 by the shaded "attributes" label above the respective virtual recurrent data stores. For notational convenience, both the developmental configuration and testing configuration are denoted as a single explicit virtual recurrent data store.[23]

Having abstracted the development process into a single process bubble, and representing all development products as virtual recurrent data stores, one can now see the relationship of the configuration management process to the development process. In the ADFD analysis of 2167, it is difficult to represent this relationship. The configuration management process appears to encompasses all of the processes that occur in any of the ADFDs presented in the previous section. A conjecture is made that in order to clearly define the configuration management responsibilities,

---

[22] In this chapter, the abstraction features of the ADFD model are fully exploited.

[23] Having developed the importance of the testing configuration, one should accept the testing configuration as being explicit at this point.

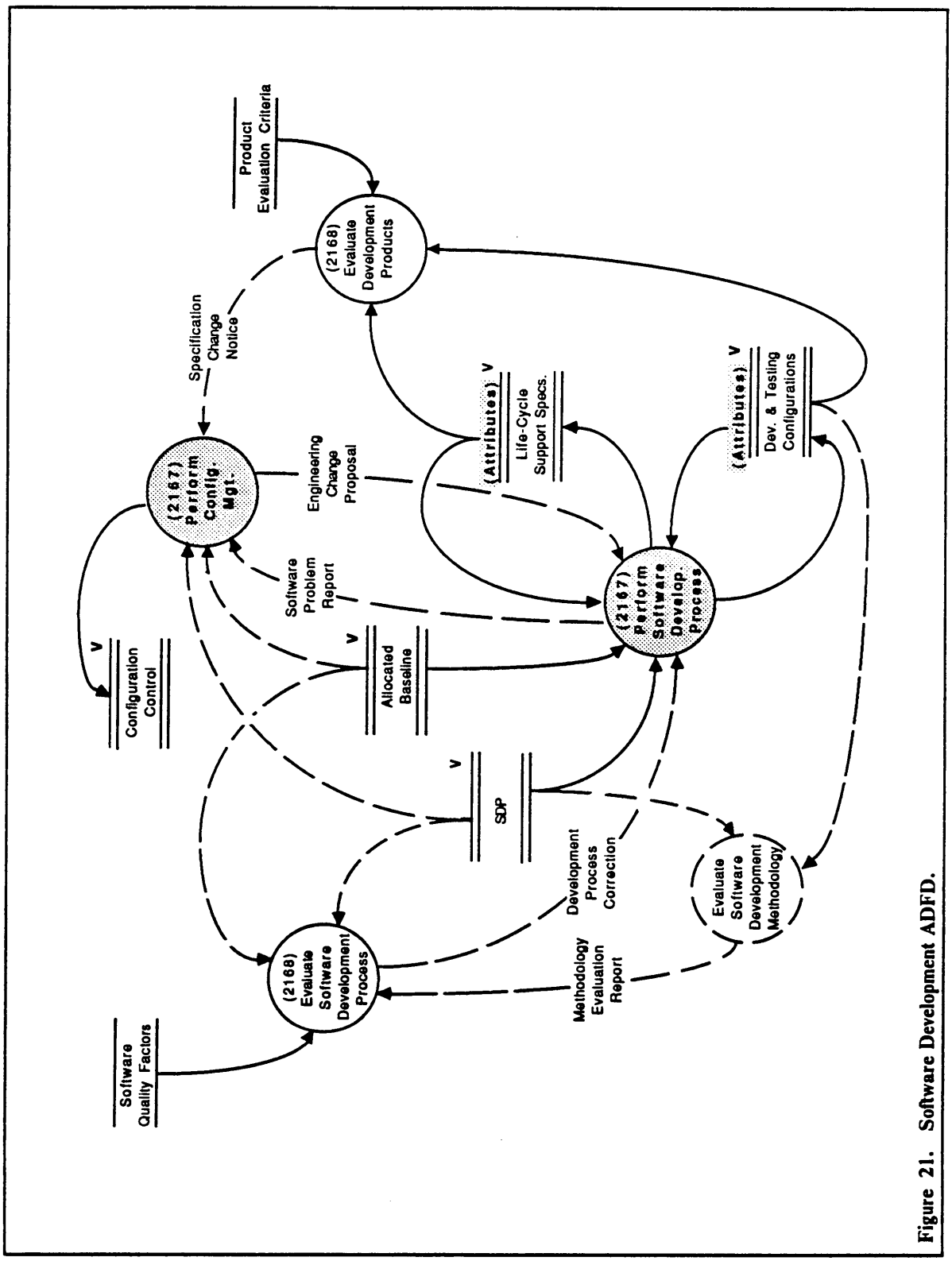

Figure 21. Software Development ADFD.

one must abstract from the details associated with any particular development process. This is precisely the relationship present in Figure 21, in which the configuration management process is depicted as a shaded process bubble and denoted parenthetically as a 2167 process. Having developed such an abstraction, one can then see the implicit data flow associated with presenting a software problem report and receiving an engineering change proposal that results in changes to any development process.

The establishment of the relationship between the processes of configuration management and software development leads to the second issue of how the software quality evaluation process relates to both configuration management and software development. In the next section, this relationship is established by considering an integrated view of software development, configuration management, and software quality evaluation.

5.2 An Integrated View

Figure 22 is the same ADFD of Figure 21 except that the emphasis is now on software quality evaluation. Two software quality processes are depicted by the shaded bubbles and denoted parenthetically as 2168 processes. These two software quality processes represent the duality of software quality evaluation. A distinct process evaluates the development products (at the far right of Figure 22), and a distinct process evaluates the development process (at the far left of Figure 22).

First consider the evaluation of the products of software development. The perspective adopted by the procedural evaluation, requires the identification of specific attributes. The life-cycle support specifications, as well as the developmental and testing configurations represent the potential locations for these attributes. The evaluation of the products is then an evaluation of the types of

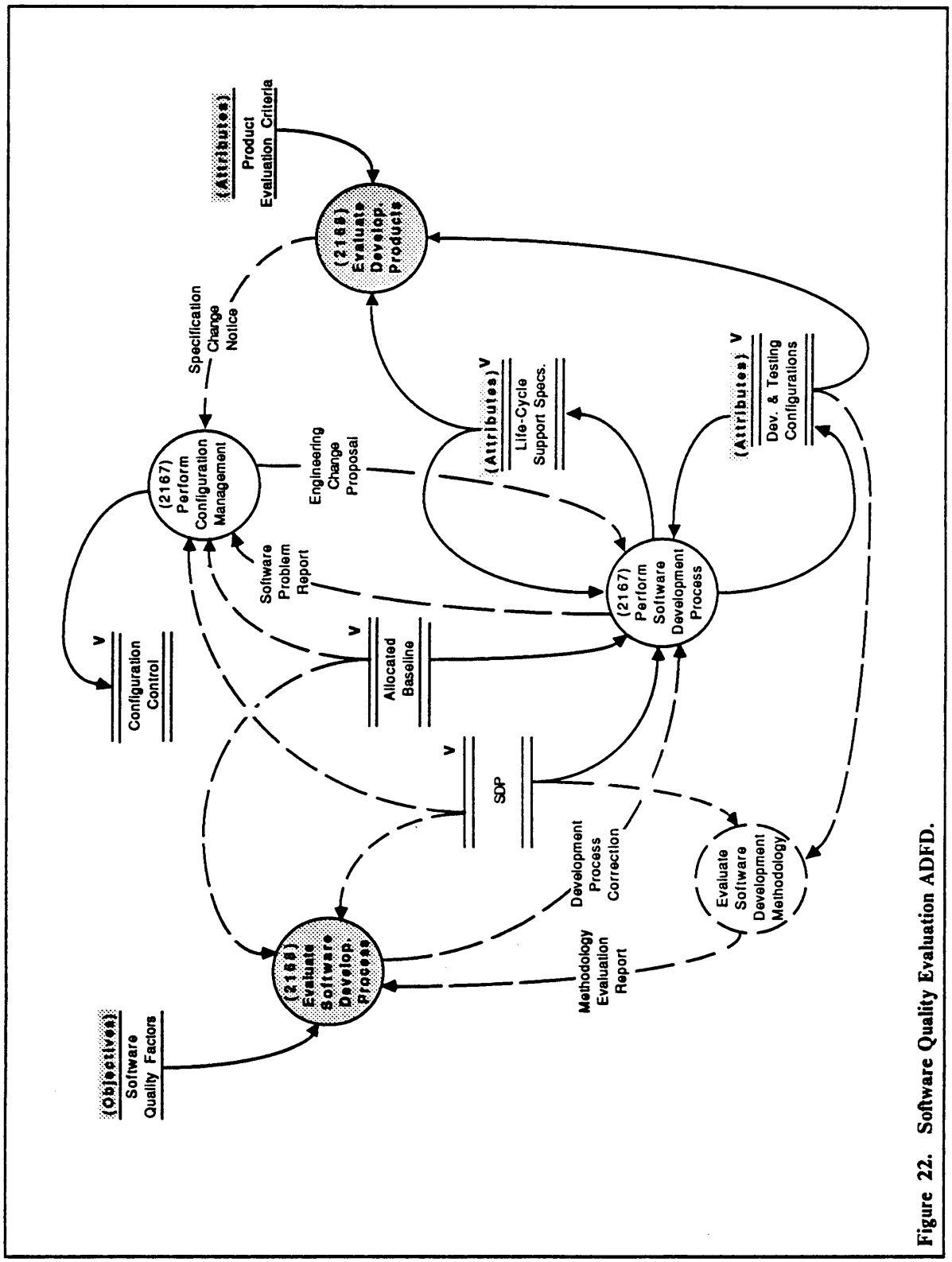

Figure 22. Software Quality Evaluation ADFD.

attributes that the products possess. As part of this evaluation, 2168 describes explicit *product evaluation criteria* which are also a potential source of attributes. The evaluation of products relates to the development process via an implied data flow to the configuration management process consisting of a Specification Change Notice (SCN). An SCN typically results in an Engineering Change Proposal (ECP). Hence, the product evaluation process indirectly interacts with a particular development process through the configuration management process.

Now consider the shaded process bubble in Figure 22 labeled "evaluate software development process". The evaluation of the development process is predicated on two virtual recurrent data stores. These are the allocated baseline and the SDP. Specifically, the allocated baseline contains the CSCI requirements in the form of the SRS and IRS(s). As previously established, quality factor requirements are defined in the SRS and these quality factor requirements are the same as the quality factors defined in 2168. Furthermore, these quality factor requirements correspond closely to the set of objectives defined by the procedural evaluation.

The relationship of the SRS, specifically the quality factor requirements, to the evaluation of the development process is represented by the implicit data flow from the allocated baseline. The relationship of the quality factor requirements of 2168 is represented as an explicit data flow. The correspondence of the quality factor requirements to objectives is indicated by the shaded "objectives" label above the data store representing these requirements.

The implicit relationship of the SDP is derived from the perspective of the procedural evaluation which maintains that a methodology espouses a set of objectives. Hence, one would expect that the methodology definitions in the SSPM would correspond to the quality factor requirements (objectives) defined by both the SRS and 2168.

Thus, the ability to evaluate the development process critically depends on the methodology definitions found in the SSPM. However, the methodologies do not represent the entire development process. One must also consider, in addition to the objectives, the principles

employed and the attributes that can be identified at any moment in the configuration items under development.

The next section draws together the final results of the analysis of the software quality evaluation process by examining the evaluation of the software development methodologies in the context of the procedural evaluation.

5.3 *Methodology Evaluation*

Capturing the relationship of the software quality evaluation process to the software development process requires a level of abstraction similar to that required for recognition of the configuration management relationship. The conclusion drawn from this result is that software quality evaluation encompasses all of the processes of software development as well as the products. In this section, the final result of the ADFD analysis of the relationship of 2168 to 2167 is presented.

The final requirement of software quality evaluation is represented by the shaded implicit process bubble labeled "evaluate software development methodology," in Figure 23. This process embodies the concepts of objectives, principles, attributes, and the linkages among them. The process of evaluating a software development methodology follows from the examination of product attributes (indicated in Figure 23 by the implicit data flow from the development and testing configurations). By examining product attributes, one can use the linkages of principles to attributes to deduce, in a bottom-up fashion, the principles employed in the generation of those products. The linkages from principles to objectives can then be established.

Assuming that the objectives are available from the methodology, one can assess how well the process is performed by ascertaining the degree to which the attributes in the products reflect the

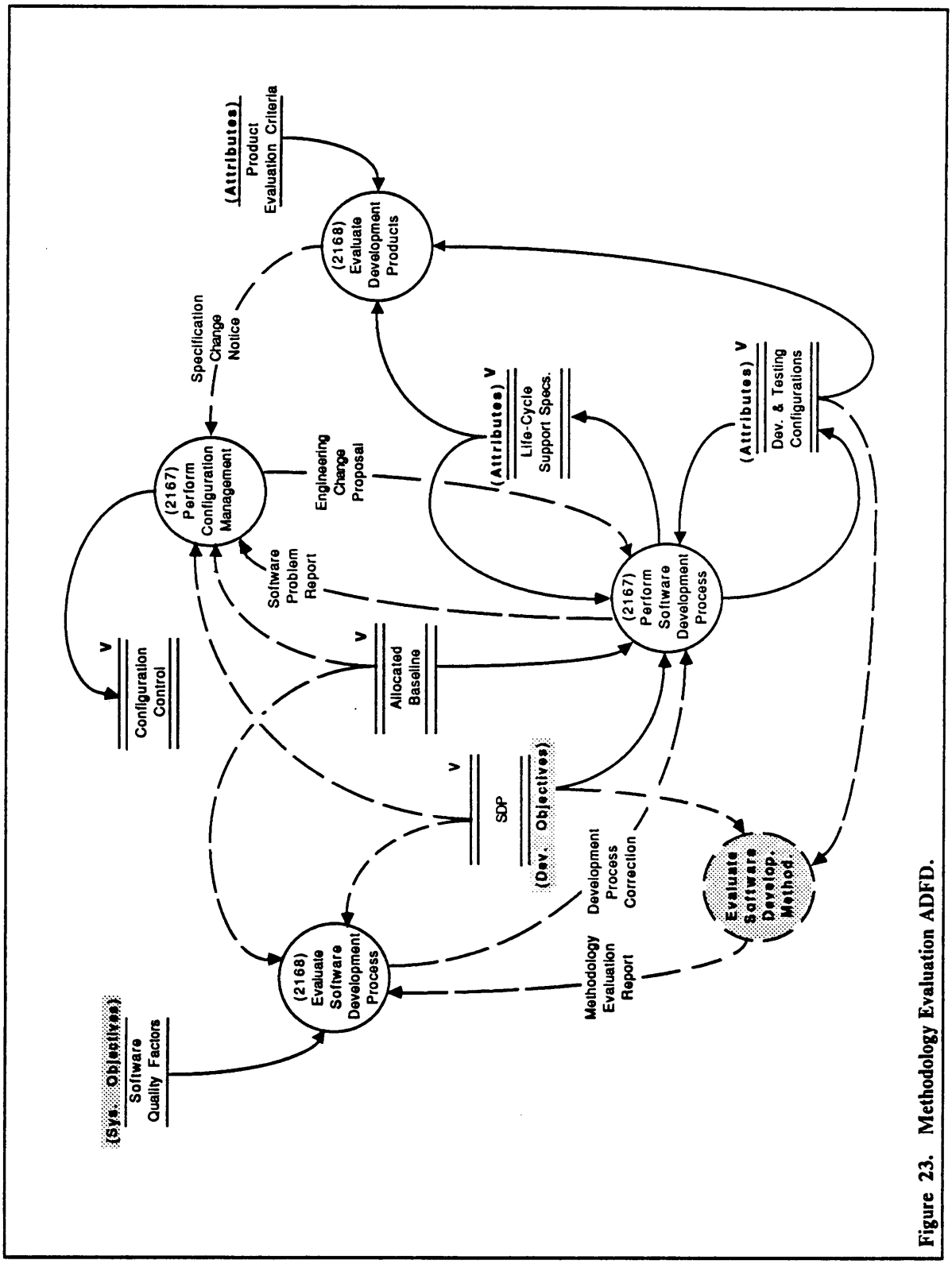

Figure 23. Methodology Evaluation ADFD.

required objectives. The objectives should be obtainable from the methodology definitions found in the SSPM (represented in Figure 23 as an implicit data flow from the SDP). The assessment results in a "methodology evaluation report" as an implicit data flow to the 2168 process for evaluation of the software development process. The result is an implicit data flow to the development process corresponding to the methodology that was evaluated.

However, an interesting question then arises. Since it has been established that 2167 establishes a meta-methodology, are the software quality factors the same as the objectives one would expect of a methodology? The indication is that they are not identical. In Figure 23, this distinction is represented by recasting the software quality factors (found in the SRS and 2168) as *system objectives* and defining *development objectives* that correspond to the methodologies defined as part of the SSPM.

Consider that in order to recognize the underlying relationship of software quality evaluation to software development, it is necessary to develop a higher level of abstraction for the development processes and associated products. The claim is made that the system objectives correspond to this higher level of abstraction, and the development objectives are more appropriately viewed at a lower level commensurate with specific development processes.

Having made this important distinction between system objectives and development objectives, one has to assess the impact on the procedural approach to the evaluation of software development methodologies. The indications are that additional work is required to understand exactly how development objectives relate to system objectives. It is outside the scope of this work to offer more than speculation on this result.

To conclude the analysis of 2168, Figure 24 presents the final ADFD which depicts the complete integration of the software development process, the configuration management process, the software quality evaluation process, and the virtual recurrent data stores that provide the interaction points for all of these processes.

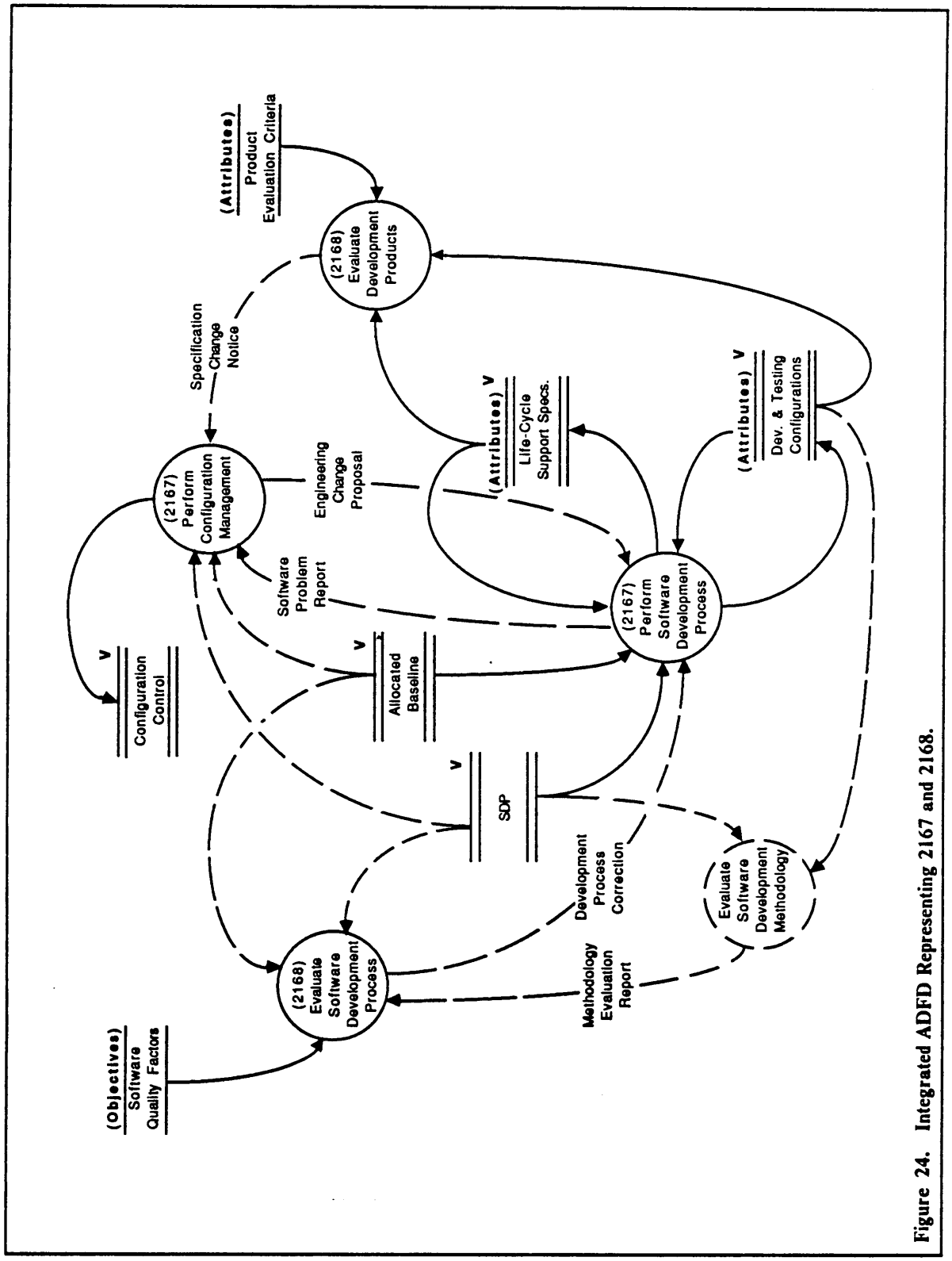

Figure 24. Integrated ADFD Representing 2167 and 2168.

6.0 Conclusion

In conclusion, the ADFD model is shown to be a useful tool for analyzing the processes and products described by both DoD-STD-2167 and DoD-STD-2168. Furthermore, by adopting the unique perspective offered by the procedural approach for evaluating software development methodologies, and maintaining this perspective throughout the ADFD based analysis, the relationships between both the processes and the products prescribed by 2167 and 2168 is explicated.

In addition, useful insights are gained through such an analysis. Particularly,

1. the notion that 2167 is a meta-methodology,

2. the notion of the duality of software quality evaluation,

3. the concept of methodology evaluation as an integral part of the software quality evaluation process, and

4. the distinction between system objectives and development objectives.

The following sections offer conjectures on the usefulness of these results, and concludes with suggestions for future research.

6.1 The ADFD Model as a Tool

The analyses in chapters four and five are intimately tied to the ADFD model. Progress during the early stages of the research effort reported here suffered due to the verbose, repetitive and confusing natures of both 2167 and 2168. Using a data flow model as a tool for coping with the complexity of these documents proves to be the key for precisely capturing the desired information. The augmentation of the Yourdon DFD model further enhances the ability to clearly present the large amount of information contained in both standards. The incorporation of recurrent virtual data stores, implicit processes, implicit data stores, and implicit data flows proved to be especially helpful during the analysis.

The usefulness of the ADFD model as a representation tool is confirmed through discussions concerning this research. With minimal introduction to the syntax and semantics of the model, one can readily comprehend relationships that are barely visible otherwise. In addition, the ADFDs are surprisingly effective as a vehicle for describing and reasoning about the procedural evaluation.

One key point is the necessity for controlling the complexity of the diagrams constructed with the ADFD model. Naive efforts can result in diagrams that become exceedingly complex as the number of entities one tries to represent in the same diagram increases. However, by careful abstraction, one can represent information distributed in several documents by using virtual data stores and relying on the hierarchical decomposition feature of the ADFD model to represent subprocesses and data stores.

An additional difficulty in working with the ADFD model is the physical layout of the ADFDs. The manual production of diagrams is labor-intensive. Automated tools are essential to produce the high quality figures found in this work. If one is considering applying the ADFD model to a similar analysis, such tools are strongly recommended; otherwise, much time and effort can be spent on purely mechanical exercises.

6.2 *Methodology versus Meta-Methodology*

A point made repeatedly in the analysis of 2167 is that methodologies and tools are defined during the requirements analysis process as part of the SSPM. This fact leads one to deduce that perhaps 2167 is itself not a methodology under the definition presented in chapter two. Rather, due to the product generation and acquisition nature of 2167, one might more correctly define 2167 as a meta-methodology. Furthermore, in order to determine methodological objectives, one must examine the product requirements (which are explicit). Through a process of bottom-up reasoning based on the notion of linkages, one can deduce some of the principles that should be employed in the production of software and documentation. By extension, one can then infer what some of the objectives might be. However, the philosophical argument on which this research is based requires that a methodology be more explicit in stating its objectives. Clearly, the opportunity exists within the guidelines of 2167 to provide such objectives through the definition of methodologies in the SSPM. Hence, the claim is advanced that 2167 establishes a meta-methodology.

Considering 2167 as a meta-methodology presents the potential to the developer for exceeding the requirements of the standard and thereby demonstrating not only compliance, but also a willingness to abide by the intent of the standard. For example, the ADFD analysis indicates an implicit need for a testing configuration during the development process. Such a configuration item is not

required by 2167, but it seems reasonable to conclude that one should be provided. In addition, the requirements in the DID for the interface design document appear to be insufficient. Due to the critical nature of CSCI interfaces, one might be of the opinion that additional methodologies and tools should be incorporated into existing development procedures. The meta-methodological nature of 2167 encourages such augmentations, and it may prove to be contractually advantageous to do so.

A developer may also be interested in determining how well existing methodologies and procedures comply with the requirements of 2167. Accepting the meta-methodology characterization of 2167 might allow one to better substantiate that a candidate methodology is adequate. Similarly, one may be able to establish that existing procedures are inadequate and take the necessary steps to improve such inadequacies. In making such assessments, the ADFD analysis may provide a useful vehicle for comparison, due to its ability to capture concisely and visually many of the requirements of 2167.

6.3 The Duality of Software Quality Evaluation

Throughout the analysis of 2167, reference is made to a software quality evaluation process that underlies the software development process. The analysis of 2168 introduces the notion of a duality of software quality evaluation. The argument is advanced, based on the results of the analysis, that software quality evaluation should be applied to a development process while the process occurs, as well as to the final products.

This claim is made in the context of the procedural evaluation, which bases the evaluation of a process on the principles employed during the process. Thus, the methodologies adopted during software development are crucial to the evaluation of the quality of the process, since the

methodologies embody the principles that are employed to achieve certain objectives. The importance of the methodologies to the software quality evaluation process is a key link between the requirements of 2167 and 2168. Hence the importance of the SSPM in the overall process of defense system software development is demonstrated. In addition, the occurrence of system objectives in both 2168 and the SRS, that correspond to the objectives defined by the procedural evaluation, provide an indication that evaluation of the software development process is the intent of both 2167 and 2168.

A developer may find benefit in incorporating into existing procedures, methods for assessing the quality of the process based on the philosophy of the procedural evaluation. A developer may be able to relate existing methodologies and procedures to specific objectives required under 2167 and 2168, and using the established linkages, define procedures for incorporating an evaluation process based on objectives, principles, and attributes. The ability to demonstrate the existence of such a software quality process might result in a clear indication of the commitment of the developer to improving the quality of defense system software.

6.4 Development Objectives versus System Objectives

At the end of chapter five, a distinction is made between development objectives and system objectives. The recognition of this distinction does not become clear until the final stage of the research reported here. How this distinction may impact the procedural evaluation, if at all, remains unclear. A technical report has been prepared, based on the results of this thesis, which presents preliminary ideas on the role of development objectives in the context of DoD-STD-2167 and DoD-STD-2168 [LAVR88].[24]

[24] This technical report also discusses the relationship of DoD-STD-2167A [DODS87a] and DoD-STD-2168 [DODS87b] to the research reported here.

6.5 *Suggestions for Future Research*

The following are areas in which the work described in this thesis can be extended or applied:

- extensions to the ADFD model,
- application of the results of the ADFD analysis to an actual development activity attempting to conform with 2167 and 2168, and
- investigation into the characteristics for an development environment that supports the development processes and software quality evaluation activities required by 2167 and 2168.

A consequence of the characterization of the process-product relationships, using the ADFD model, is a qualitative sense for the complexity of a relationship based on the number of data flow arcs representing that relationship. An obvious extension would be to assess this complexity using a quantitative measure. The ability to derive a quantitative measure based on data flow is predicated on the ability to first quantify the data that represents the relationship. This would require that the ADFD model be applied not to the standards themselves, but to software development activities that are in conformance with 2167 and 2168. The information in this thesis already provides the basis for beginning such an activity.

Given that the characterization of both 2167 and 2168 by the ADFD model allows one to determine compliance with the standards, the definition and design of a software development environment in which software could be designed and developed in accordance with the prescribed processes and products of both standards is useful. Such an environment would possess imbedded knowledge of the requirements of the standards and thus aid the developers in meeting those requirements.

References

ARTJ85 Arthur, James D., Sallie M. Henry and Richard E. Nance, "Immediate Software Development Issues for Embedded Systems Applications in Surface Combatants," Technical Report TR-85-19, Virginia Polytechnic Institute, 15 March, 1985.

BROF87 Brooks, Frederick P., "No Silver Bullet: Essence and Accidents of Software Engineering," *IEEE Computer,* April, 1987, pp. 10-19.

DANA87 Dandekar, Ashok V., "A Procedural Approach to the Evaluation of Software Development Methodologies," *Masters Thesis,* Virginia Polytechnic Institute and State University, September, 1987.

DEMT79 Demarco, Tom, *Structured Analysis and System Specification,* New York: Yourdon Inc., 1979.

DID08 DI-CMAN-80008, *System/Segment Specification,* United States Department of Defense, 4 June, 1985.

DID09 DI-MCCR-80009, *Software Configuration Management Plan,* United States Department of Defense, 4 June, 1985.

DID10 DI-MCCR-80010, *Software Quality Evaluation Plan,* United States Department of Defense, 4 June, 1985.

DID11 DI-MCCR-80011, *Software Standards and Procedures Manual,* United States Department of Defense, 4 June, 1985.

DID12 DI-MCCR-80012, *Software Top Level Design Document,* United States Department of Defense, 4 June, 1985.

DID13 DI-MCCR-80013, *Version Description Document,* United States Department of Defense, 4 June, 1985.

DID14 DI-MCCR-80014, *Software Test Plan,* United States Department of Defense, 4 June, 1985.

DID15 DI-MCCR-80015, *Software Test Description,* United States Department of Defense, 4 June, 1985.

DID16 DI-MCCR-80016, *Software Test Procedure,* United States Department of Defense, 4 June, 1985.

DID17 DI-MCCR-80017, *Software Test Report,* United States Department of Defense, 4 June, 1985.

DID18 DI-MCCR-80018, *Computer System Operator's Manual,* United States Department of Defense, 4 June, 1985.

DID19 DI-MCCR-80019, *Software User's Manual,* United States Department of Defense, 4 June, 1985.

DID20 DI-MCCR-80020, *Computer System Diagnostic Manual,* United States Department of Defense, 4 June, 1985.

DID21 DI-MCCR-80021, *Software Programmer's Manual,* United States Department of Defense, 4 June, 1985.

DID22 DI-MCCR-80022, *Firmware Support Manual,* United States Department of Defense, 4 June, 1985.

DID23 DI-MCCR-80023, *Operational Concept Document,* United States Department of Defense, 4 June, 1985.

DID24 DI-MCCR-80024, *Computer Resources Integrated Support Document,* United States Department of Defense, 4 June, 1985.

DID25 DI-MCCR-80025, *Software Requirements Specification,* United States Department of Defense, 4 June, 1985.

DID26 DI-MCCR-80026, *Interface Requirements Specification,* United States Department of Defense, 4 June, 1985.

DID27 DI-MCCR-80027, *Interface Design Document,* United States Department of Defense, 4 June, 1985.

DID28 DI-MCCR-80028, *Data Base Design Document,* United States Department of Defense, 4 June, 1985.

DID29 DI-MCCR-80029, *Software Product Specification,* United States Department of Defense, 4 June, 1985.

DID30 DI-MCCR-80030, *Software Development Plan,* United States Department of Defense, 4 June, 1985.

DID31 DI-MCCR-80031, *Software Detailed Design Document,* United States Department of Defense, 4 June, 1985.

DODS87a DoD-STD-2167A, *Defense System Software Development - Draft,* United States Department of Defense, 1 April, 1987.

DODS87b DoD-STD-2168, *Defense System Software Quality Program - Draft,* United States Department of Defense, 1 April, 1987.

DODS85a DoD-STD-2167, *Defense System Software Development,* United States Department of Defense, 4 June, 1985.

DODS85b DoD-STD-2168, *Software Quality Evaluation,* United States Department of Defense, 26 April, 1985.

FIRD87 Firesmith, Donald G., "Software Development Process? Should the DoD Mandate a Standard," *Defense Science and Electronics,* July, 1987, pp. 56-59.

HECH84 Hecht, Herbert, "Computer Standards," *IEEE Computer,* October, 1984, pp. 33-43.

HENS85 Henry, Sallie M., James D. Arthur and Richard E. Nance, "A Procedural Approach to Evaluating Software Development Methodologies," Technical Report TR-85-20, Virginia Polytechnic Institute, 30 March, 1985.

IEEE83 IEEE Standards Board, "IEEE Standard Glossary of Software Engineering Terminology," *ANSI/IEEE Std 729-1983,* February, 1983.

LAVR88 Lavender, Robert G., "Issues Concerning the Explication of Process-Product Relationships in DoD-STD-2167 and DoD-STD-2168," Systems Research Center Technical Report SRC-88-006, Virginia Polytechnic Institute, 31 March, 1988.

MILS85a MIL-STD-483A "Configuration Management Practices for Systems, Equipment, Munitions, and Computer Programs," United States Department of Defense, 4 June, 1985.

MILS85b MIL-STD-490A "Specification Practices," United States Department of Defense, 4 June, 1985.

WARP86 Ward, Paul T., "The Transformation Schema: An Extension of the Data Flow Diagram to Represent Control and Timing," *IEEE Transactions on Software Engineering,* Vol. SE-12 (2), February, 1986, pp. 198-210.

Appendix A. Glossary

The following terms, definitions and acronyms have been compiled from DoD-STD-2167 [DODS85a], DoD-STD-2168 [DODS85b], and the IEEE Standard Glossary of Software Engineering Terminology [IEEE83]. More detailed explanations may be found by consulting these sources.

ADFD: Augmented Data Flow Diagram.

allocated baseline: The initial approved allocated configuration identification. Within the context of DoD-STD-2167, the allocated baseline consists of the SRS and IRS(s) for a CSCI.

Augmented Data Flow Diagram: An extension to Yourdon Data Flow Diagrams that allows one to distinguish between implicit, explicit, recurrent, and virtual entities.

baseline: A set of configuration identification documents fixed at a specific time during a configuration item's life cycle.

CDR: Critical Design Review.

CM: Configuration Management.

configuration identification: The current approved or conditionally approved technical documentation for a CSCI.

Computer Software Component: A functional or logically distinct part of a Computer Software Configuration Item.

Computer Software Configuration Item: An aggregate of software designated as an item for configuration management.

CRISD: Computer Resources Integrated Support Document.

CSC: Computer Software Component.

CSCI: Computer Software Configuration Item.

CSDM: Computer System Diagnostic Manual.

CSOM: Computer System Operator's Manual.

data flow arc: A graphical entity in an ADFD that represents the flow of data among process bubbles and data stores.

data store: A graphical entity in an ADFD that represents a repository of data.

DBDD: Data Base Design Document.

developmental configuration: The software and associated technical documentation that defines the evolving configuration of a CSCI during development. Includes the STLDD, SDDD, IDD(s), DBDD(s), and developed software.

duality of software quality evaluation: The notion that software quality evaluation is applied to not only the products but also the processes of software development; particularly the development methodologies.

DFD: Yourdon Data Flow Diagram.

DID: Data Item Description.

DoD: Department of Defense.

DoD-STD-2167: Military standard for defense system software development of mission-critical computer system software.

DoD-STD-2168: Military standard for software quality evaluation.

ECP: Engineering Change Proposal.

FSM: Firmware Support Manual.

IDD: Interface Design Document.

IRS: Interface Requirements Specification.

life-cycle support specifications: A collection of specifications that includes the CRISD, CSOM, CSDM, SUM, SPM, FSM, and OCD.

LLCSC: Lower-Level Computer Software Component.

OCD: Operational Concept Document.

PDR: Preliminary Design Review.

process bubble: A graphical entity in an ADFD that represents a process.

product: A term used to define code or documentation, or both.

recurrent data store: A graphical entity in an ADFD representing data that occurs repeatedly in successive diagrams.

SCMP: Software Configuration Management Plan.

SCN: Specification Change Notice.

SDDD: Software Detailed Design Document.

SDF: Software Development File.

SDP: Software Development Plan (may include the SSPM, SCMP, and SQEP).

software development specifications: A collection of development specifications which includes the SDP, SSPM, SCMP, and SQEP.

software life cycle: The period of time between the conception of a software product and the termination of its use.

software product: A software entity designated for delivery. May be code or documentation, or both.

software quality: The degree to which the attributes of the software enable the software to perform its specified end use.

SPM: Software Programmer's Manual.

SPS: Software Product Specification.

SQEP: Software Quality Evaluation Plan.

SRS: Software Requirements Specification.

SSPM: Software Standards and Procedural Manual.

SSR: Software Specification Review.

SSS: System/Segment Specification.

STD: Software Test Description.

STLDD: Software Top-Level Design Document.

STP: Software Test Plan.

STPR: Software Test Procedure.

STR: Software Test Report.

SUM: Software User's Manual.

testing configuration: A collection of testing specifications for a CSCI which includes the STD, STP, STPR and STR(s).

TLCSC: Top-Level Computer Software Component.

Unit: The smallest logical entity specified in the detailed design which completely describes a single function in sufficient detail to allow the code to be produced and tested independently. A physical section of code.

VDD: Version Description Document.

virtual data store: A graphical entity in an ADFD representing several combined data stores.

Appendix B. A Synopsis of Data Item Descriptions

This appendix provides a synopsis of the Data Item Descriptions (DIDs) that are referenced in both DoD-STD-2167 and DoD-STD-2168, and appear as data stores in the Augmented Data Flow Diagrams in Chapters four and five. More complete descriptions can be found by consulting the reference, enclosed in brackets, preceding each of the following items.

[DID08] System/Segment Specification (SSS)

- Specifies the functional, performance, and interface requirements for a system, or segment of a system.
- Specifies the requirements for the characteristics, logistics, quality factors, design, qualification, and the delivery of the system or segment.
- Specifies the requirements to potential developers of the system.
- Identifies Computer Software Configuration Items for a system.

[DID09] Software Configuration Management Plan (SCMP)

- Describes the organizations responsible for software configuration management, and the procedures to be followed in identifying the internal developmental configurations, and in identifying and controlling software changes.

- Provides insight into the procedures of the configuration management activity.

- May be included in the Software Development Plan (SDP).

[DID10] Software Quality Evaluation Plan (SQEP)

- Describes the organizations and procedures to be used to determine the quality of software, associated documentation and activities, and to perform related tasks that are an outgrowth of the evaluation activities.

- Used to monitor the procedures and management of the software quality evaluation process.

- May be included in the Software Development Plan (SDP).

[DID11] Software Standards and Procedures Manual (SSPM)

- Contains the standards, methodologies, guidelines, and restrictions used in the development of Computer Software Configuration Items, including design and coding standards.

- Used to ensure uniformity among the CSCIs in the system as they progress through the development cycle.

- May be included in the Software Development Plan (SDP).

[DID12] Software Top-Level Design Document (STLDD)

- Describes the structure and organization of a particular Computer Software Configuration Item.

- Presents the allocation of the CSCI requirements specified in the Software Requirements Specification (SRS) and Interface Requirements Specifications (IRSs) to Top-Level Computer Software Components of the CSCI.

- Defines the interface, data, and processing characteristics for each TLCSC in the CSCI design.

- The STLDD is the first document in the developmental configuration for a CSCI.

[DID13] Version Description Document (VDD)

- Identifies and describes each version of a Computer Software Configuration Item, including initial release, and interim changes to the previously released version.

- Changes are identified by reference to the applicable Engineering Change Proposal (ECP) and Specification Change Notice (SCN).

[DID14] Software Test Plan (STP)

- Defines the total scope of testing for a particular Computer Software Configuration Item, and establishes requirements, describes organizations, management, and planning for CSCI testing.

- Basis for the development of both formal and informal test programs.

[DID15] Software Test Description (STD)

- Identifies the input data, expected output data, and evaluation criteria that comprise the test cases for all of the formal tests of a Computer Software Configuration Item.

- Basis for the development of test procedures for formal CSCI testing.

[DID16] Software Test Procedure (STPR)

- Describes detailed procedures for the performance of formal tests on a Computer Software Configuration Item, and describes detailed procedures for the reduction and analysis of CSCI formal test data.

- Basis for the execution of formal CSCI testing.

[DID17] Software Test Report (STR)

- Contains a record of the formal testing performed for a particular Computer Software Configuration Item.

- Basis for the re-test of a CSCI.

[DID18] Computer System Operator's Manual (CSOM)

- Provides information and detailed procedures for initiating, operating, and monitoring a specific computer system which is part of the overall mission-critical system.

- A CSOM is developed for each computer system in which one or more Computer Software Configuration Items execute.

[DID19] Software User's Manual (SUM)

- Provides users with instructions sufficient to execute the software of one or more related Computer Software Configuration Items.

- Provides the steps for executing the software, the expected output, and the corrective measures required when the expected output is not obtained.

[DID20] Computer System Diagnostic Manual (CSDM)

- Provides procedures and information to identify and isolate a malfunction in a computer system.

- Not required if the information is provided in other documents such as a commercially available manual or the Computer System Operator's Manual (CSOM).

[DID21] Software Programmer's Manual (SPM)

- Provides information to enable a programmer to produce, interpret, troubleshoot, and modify software for specific and host computers.

- Not required if the information is provided in a commercially available document.

[DID22] Firmware Support Manual (FSM)

- Provides the information necessary to program the read-only memories of hardware components.

- Describes the technical aspects of the memory devices, support software, and support equipment required.

[DID23] Operational Concept Document (OCD)

- Describes the mission of the system and its operational and support environment.

- Describes the functions and characteristics of the computer system within the overall system.

[DID24] Computer Resources and Integrated Support Document (CRISD)

- Provides information required to perform life cycle support for deliverable software.

[DID25] Software Requirements Specification (SRS)

- Specifies in detail the complete functional, interface, performance, and qualification requirements of a particular Computer Software Configuration Item, and includes requirements for programming design, adaptation, quality factors, and traceability of the CSCI.

- Allows for the assessment of whether or not the completed CSCI complies with its requirements.

- Basis for the development and formal testing of a CSCI.

- Included in the allocated baseline for a CSCI.

[DID26] Interface Requirements Specification (IRS)

- Specifies the requirements for one or more interfaces between a particular Computer Software Configuration Item and other configuration items, or critical items in a system.

- Allows assessment of whether or not the implementation of an interface complies with the requirements.

- Basis for the development and qualification of the interface(s).

- May be included in the Software Requirements Specification (SRS), or included in the allocated baseline for a CSCI.

[DID27] Interface Design Document (IDD)

- Describes the detailed design of one or more interfaces between a Computer Software Configuration Item and other CSCIs, hardware configuration items or critical items.

- Used along with the Interface Requirements Specification (IRS) to communicate and control interface design decisions.

[DID28] Data Base Design Document (DBDD)

- Describes the architecture and design of one or more data bases for a Computer Software Configuration Item, and describes the relationship among files in the data base.

- Used to communicate and control data base design decisions.

[DID29] Software Product Specification (SPS)

- Consists of the design documents and software listings for a Computer Software Configuration Item.

- Establishes the product baseline for a delivered CSCI.

[DID30] Software Development Plan (SDP)

- Describes the organization and procedures to be used in performing software development.

- May consist of the Software Standard and Procedures Manual (SSPM), the Software Configuration Management Plan (SCMP), and the Software Quality Evaluation Plan (SQEP).

[DID31] Software Detailed Design Document (SDDD)

- Describes in detail the structure and organization of a particular Computer Software Configuration Item, including the decomposition of Top-Level Computer Software Components into Lower-Level Computer Software Components and Units.

- Defines the interface, data, and processing characteristics for each LLCSC and Unit.

- Included in the Software Product Specification (SPS) as part of the final product baseline for a CSCI.

The vita has been removed from the scanned document