Report Number: WUCS-82-3

1982-02-01

The Total System Design (TSD) Framework: An Approach to the Development of Distributed Systems Design Methodologies

Gruia-Catalin Roman, Mishell J. Stucki, William E. Ball, and Will G. Gillett

A methodological framework is an abstraction over a class of design methodologies. The framework characteristics the problem solving approach shared by the methodologies belonging to that class: it identifies the nature of their common design concerns and the fundamental logical interdependencies between these concerns. The paper proposes a particular framework called the Total System Design (TSD) Framework. It represents a specification for a class of design methodologies which view computer-based systems as potentially distributed hardware/software aggregates. As such, the TSD Framework consolidates under a unified perspective two traditionally separate concerns: software design and hardware design. Furthermore, it establishes the... **Read complete abstract on page 2.**

The Total System Design (TSD) Framework: An Approach to the Development of Distributed Systems Design Methodologies

Gruia-Catalin Roman, Mishell J. Stucki, William E. Ball, and Will G. Gillett

Complete Abstract:

A methodological framework is an abstraction over a class of design methodologies. The framework characteristics the problem solving approach shared by the methodologies belonging to that class: it identifies the nature of their common design concerns and the fundamental logical interdependencies between these concerns. The paper proposes a particular framework called the Total System Design (TSD) Framework. It represents a specification for a class of design methodologies which view computer-based systems as potentially distributed hardware/software aggregates. As such, the TSD Framework consolidates under a unified perspective two traditionally separate concerns: software design and hardware design. Furthermore, it establishes the role played by hardware/software trade-offs in system design. A strategy for deriving methodologies from the TSD Framework is outlined and illustrated.

THE TOTAL SYSTEM DESIGN (TSD) FRAMEWORK: AN APPROACH TO THE DEVELOPMENT OF DISTRIBUTED SYSTEMS DESIGN METHODOLOGIES

Gruia-Catalin Roman
Mishell J. Stucki
William E. Ball
Will D. Gillett

WUCS-82-3

Department of Computer Science

Washington University

St. Louis, Missouri 63130

February 1982

THE TOTAL SYSTEM DESIGN (TSD) FRAMEWORK: AN APPROACH TO THE DEVELOPMENT OF DISTRIBUTED SYSTEMS DESIGN METHODOLOGIES

Gruia-Catalin Roman
Mishell J. Stucki
William E. Ball
Will D. Gillett

WUCS-82-3

Department of Computer Science

Washington University

St. Louis, Missouri 63130

February 1982

ABSTRACT

A methodological framework is an abstraction over a class of design methodologies. The framework characterizes the problem solving approach shared by the methodologies belonging to that class: it identifies the nature of their common design concerns and the fundamental logical interdependencies between these concerns. The paper proposes a particular framework called the Total System Design (TSD) Framework. It represents a specification for a class of design methodologies which view computer-based systems as potentially distributed hardware/software aggregates. As such, the TSD Framework consolidates under a unified perspective two traditionally separate concerns: software design and hardware design. Furthermore, it establishes the role played by hardware/software trade-offs in system design. A strategy for deriving methodologies from the TSD Framework is outlined and illustrated.

Acknowledgements: This work was supported in part by Rome Air Development Center and by Defense Mapping Agency under contract F30602-80-C-0284. The contributions of R. K. Israel and J. T. Love to the investigation of certain aspects of the TSD Framework are also acknowledged.

Keywords: methodological framework, methodology, system design, hardware/software trade-offs.

INTRODUCTION

The software crisis of the 70's played a significant role in increasing the general awareness of, and interest in, design methodologies. In particular, it brought about a wide-spread belief that large system development without strong methodological support involves unacceptable risks. As a result and in a relatively short time span, major advances have been achieved in the areas of methodology development, project control and review, specification techniques, and automated documentation and analysis tools [CHAN78, WASS78, WEGN79, YEH77]. The complexity of the design issues facing system developers, however, also has grown. The ever increasing interdependency between hardware and software has led to the view that a system is a hardware/software (H/S) aggregate in which the hardware and software aspects must be treated together and not separately as has been traditional.

Today's system design methodologies must consider the relation between hardware and software and its impact on the performance characteristics of the total system. The design of high performance systems (e.g., for graphics, avionics, etc.) demands careful tuning of the hardware structure and primitives to the needs of the application software. The design of distributed systems forces traditional software designers to consider the nature of the support hardware to a larger degree than in the past. Microprogrammable machines blur the distinction between hardware and software. VLSI technology allows traditional software functions to be realized in hardware. Changes in the relative cost of hardware and software affect the system's useful life span. These and many other similar considerations strongly suggest the need to consolidate methodological advances in the software and hardware design areas.

The notion of a methodological framework is employed in this paper as the means by which this consolidation is accomplished. A methodological framework is an abstraction of a class of system design methodologies. The framework is hierarchical in structure, being composed of stages which are, in turn, composed of phases, which are composed of steps. The stages represent broad design areas such as system design, software design, and hardware design, while the phases represent finer divisions of these design areas. For example, a stage dealing with software design could contain separate phases for software architecture, program design, and coding. The steps represent design activities that go on within the design areas. They include activities such as performance evaluation, functional verification, documentation, and acceptance.

The framework characterizes the problem solving approach shared by a group of related methodologies by identifying the nature of their common design concerns and by establishing the fundamental logical interdependencies between these concerns. This fact makes the framework useful as a specification for a class of methodologies having predefined characteristics. The Total System Design (TSD) Framework, for instance, is a methodological framework that has been developed in order to specify a class of distributed systems design methodologies which emphasize a rational and systematic resolution of the hardware/software partitioning issue. The definition of the TSD Framework and its significance for system design is the topic of this paper. The next two sections contain a brief description of the TSD Framework stages, phases and steps. The exposition is introductory in nature, with a detailed description of the TSD Framework being given in [ROMA82]. A separate section summarizes the TSD Framework perspective on H/S trade-offs.

A subsequent section considers the issue of developing specialized methodologies via successive refinements of the TSD Framework. These refinements take into consideration the nature of the application area, the available technology, and the characteristics of the organizations involved in the development and maintenance of the systems. The approach draws heavily on past experience with the development of design methodologies in general (e.g., [ROMA82]) and with the development of several organization specific methodologies (e.g., [ROMA79]). A sample application area and a hypothetical organization are used for illustrative purposes. The concluding section reviews the experience to date with the use of this approach.

STAGES AND PHASES

Figure 1 shows the logical structure of the TSD Framework. The stage boundaries are drawn along traditional lines and the concern of each is obvious from the stage name. Each stage is composed of two or more phases which represent well known design areas. The downward arrows represent requirements specifications that define the problem to be solved by a subsequent stage. Each specification has two parts, a functional requirement and a set of implementation constraints. The upward arrows indicate the flow of finished products during the integration portion of system development. The idea here is that each stage is responsible for the integration of its portion of the design. The integration process thus begins at the lowest level of detail and works upward until all components of the system have been assembled and tested.

PROBLEM DEFINITION STAGE.

This stage is composed of two phases: identification and conceptualization. Both phases are application domain dependent and their successful completion rests on a good understanding of the application. The IDENTIFICATION phase is informal in nature and has an exploratory flavor. Its objective is to produce an identification report which contains all the information available with regard to the system support required by the application at hand, as well as any relevant constraints. Despite the fact that the level of formalization and abstraction of the identification report is relatively low, the report serves two important functions: it establishes the communication link between the designer and the user and provides the necessary base for the development of a formal definition of the problem. This formal development is done in the conceptualization phase.

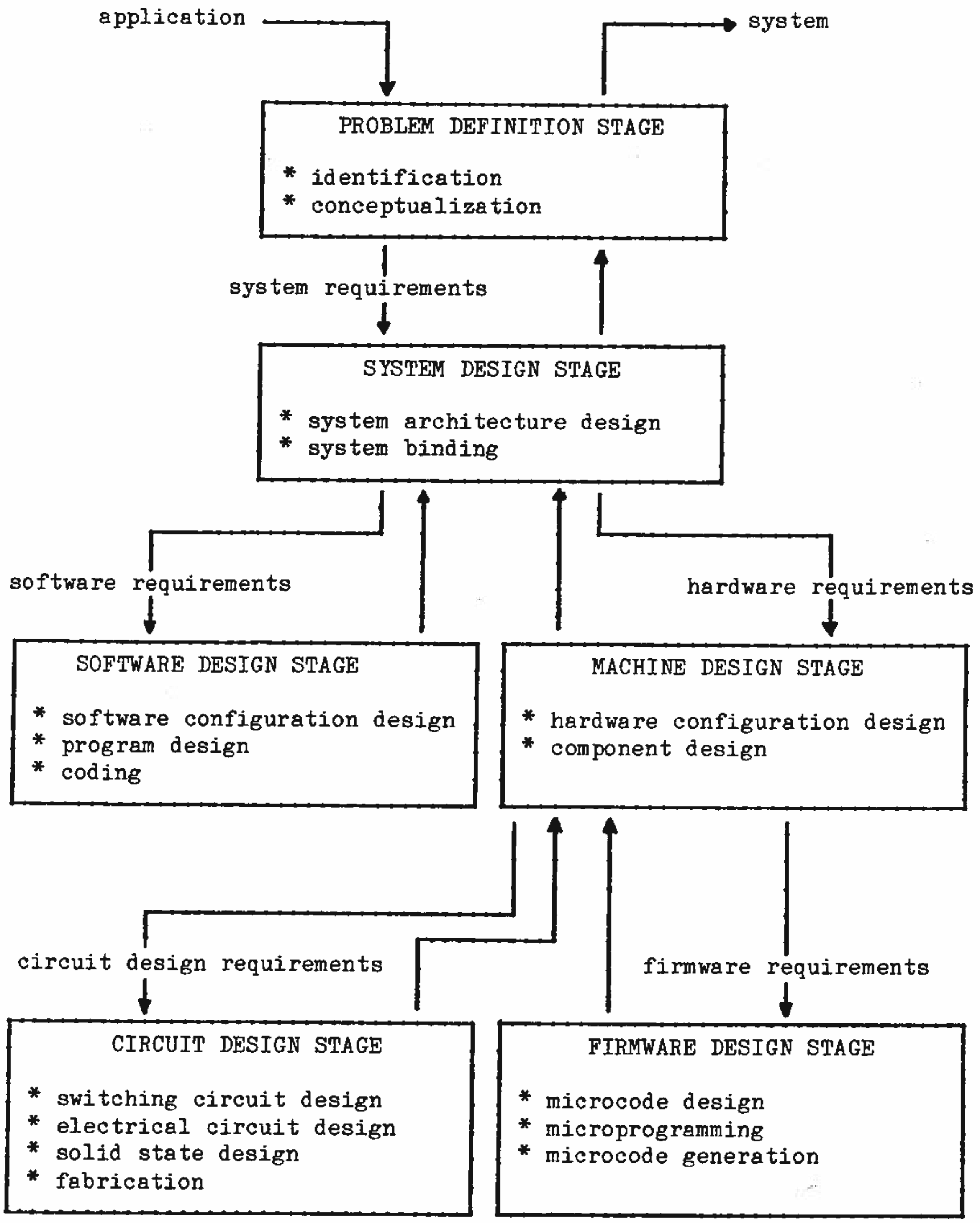

FIGURE 1: TSD FRAMEWORK STRUCTURE

The CONCEPTUALIZATION phase uses the identification report in order to generate the system requirements. These requirements contain a conceptual model which formalizes the system's role from a user perspective and the application constraints identified earlier. Because of its formal nature, the conceptual model provides a solid basis for the entire design process and represents the ultimate correctness criterion against which the final system is judged. The ability to meet all the stated constraints is a second fundamental evaluation criterion.

SYSTEM DESIGN STAGE.

This stage includes two phases: system architecture design and system binding. The main concern of the SYSTEM ARCHITECTURE DESIGN phase is to investigate system design alternatives and their potential impact on the choices for a feasible system configuration (i.e., H/S mix). Without making any explicit choices with respect to the selection of particular software or hardware components, this phase is involved in the performance of H/S trade-offs to the extent that design decisions taken here affect the class of feasible configurations in a manner too significant to be left to chance.

The functional/performance specifications generated by the system architecture design, as part of the system configuration requirements, form the basis on which a particular H/S mix is selected during the SYSTEM BINDING phase. The hardware and software requirements being generated by this phase may assume a variety of H/S combinations from off-the-shelf complete systems to custom built components. The election of one option over another is determined by the nature of the system design, the constraints to be met, and the available technology. Binding options are identified in the system architecture design phase, but the selection of specific components is done in the binding phase.

SOFTWARE DESIGN STAGE.

This stage includes all activities relating to software design and procurement. There are three phases involved in this stage. The first one, SOFTWARE CONFIGURATION DESIGN, is responsible for the procurement of off-the-shelf software as well as the overall high level design of the software system. The software requirements are the basis for these activities which result in the development of program requirements specifications, including the complete design of its data and environment interfaces. The PROGRAM DESIGN phase, in turn, takes these requirements and produces the program design (data and processing structures) which, together with all pertinent assumptions and constraints, make up the implementation requirements. They are used by the CODING phase to build the actual programs.

MACHINE DESIGN STAGE.

This stage plays a role similar to that of the first two phases of the software design stage. The HARDWARE CONFIGURATION DESIGN phase is concerned with the procurement of off-the-shelf machines and the design of the high level architecture of custom hardware. Component requirements are developed for all entities that are part of the custom hardware and passed on to the COMPONENT DESIGN phase. This phase generates a register transfer level machine description that will be included in the circuit design requirements and in the firmware requirements.

CIRCUIT DESIGN STAGE.

This stage follows a generally accepted scenario involving four phases: SWITCHING CIRCUIT DESIGN, ELECTRICAL CIRCUIT DESIGN, SOLID STATE DESIGN, and FABRICATION. Each phase generates design requirements for the phase listed after it.

FIRMWARE DESIGN STAGE.

This stage consists of three phases that are an analog to program design, coding, and compilation. These phases are called MICROCODE DESIGN, MICROPROGRAMMING and MICROCODE GENERATION.

STEPS

The previous section gave a general introduction to the design areas covered by the TSD stages and phases. This section gives a general introduction to the activities that occur during the design process. The major design activities within a phase are called STEPS. There are ten steps which collectively represent the activities within any phase, regardless of the nature of the phase. Some of the steps represent activities that are common practice among good designers and appear to be fundamental to the design process. The other steps represent activities that are needed to meet the objectives of the TSD Framework. The names of these steps are listed below. The dashed lines are used to indicate groups of related steps.

```
     -----------
 formalism selection
formalism validation
     -----------
     exploration
     elaboration
consistency checking
    verification
     evaluation
      inference
     -----------
     invocation
     -----------
     integration
     -----------
```

The FORMALISM SELECTION step encompasses the activities involved in selecting a formalism for a particular problem domain. Candidate formalisms are evaluated for their expressive power in that domain and also for qualities such as simplicity of use, lack of ambiguity, analyzability, and potential for automation. While this step must take place before other steps in the phase, it often occurs long before them. This is sometimes due to the use of a methodology that is based on a particular formalism, but is more often simply a matter of policy or is due to the availability of tools tailored to that formalism.

The FORMALISM VALIDATION step encompasses activities involved in determining whether a formalism has the expressive power needed for a particular task. It also includes the evaluation of formalisms from the standpoint of ease of use. These tasks are generally non-trivial and may involve both theoretical and experimental evaluations. Theoretical results may indicate the power and the fundamental limitations of the formalism while past experience with it on similar projects may provide insight into its appropriateness and ease of use. The step also includes evaluations of the formalism's potential for design automation (as a way to bring about productivity increases) and its ability to support hierarchical specifications (as an aid to controlling complexity).

The EXPLORATION step encompasses the mental activities involved in synthesizing a design. These activities are creative in nature and depend on experience and natural talent. They cannot be formalized or automated unless the problem domain is restricted to a significant degree.

The ELABORATION step encompasses the activities involved in giving form to the ideas produced in the exploration step. In general, this step involves the use of formalisms and its activities are facilitated by design aids such as text editors and formatters. This step includes the building of a concrete object such as a piece of hardware.

The CONSISTENCY CHECKING step encompasses activities such as checking for incorrect uses of formalisms, checking for contradictions, conflicts, and incompleteness in specifications, and checking for errors of a semantic nature. It includes checking for consistency between different levels of abstraction in a hierarchical specification and the reconciliation of multiple viewpoints.

The VERIFICATION step encompasses activities involved in demonstrating that a design has the functional properties called for in its requirements specification. Since each phase has a requirements specification and produces a design, this step is equally important for all phases. A common example of this type of activity is the proving of program correctness. The difficulty of this task is well known and is also representative of the difficulty of the verification task in general.

The EVALUATION step encompasses activities involved in determining if a design meets a given set of constraints. This includes constraints which are part of the requirements specification for the phase and constraints which result from design decisions. The nature of the evaluation activities depends on the type of constraints being analyzed. They include classical system performance evaluation of response time and workload by means of analytical or simulation methods; deductive reasoning for investigating certain qualitative aspects like fault tolerance or survivability; and construction of predictive models for properties such as cost and reliability.

The INFERENCE step encompasses activities involved in assessing the potential impact of design decisions made in the phase. The domain of these activities includes: impact on the application environment, ability of subsequent phases to live with decisions made in this phase, effect on system maintainability and enhanceability, and effect on implementation options. While these issues must be considered in every phase, proper treatment is particularly critical in those stages defining architectures.

The INVOCATION step encompasses the activities associated with releasing the results of the phase. It includes quality control activities where tangible products are involved and review activities leading to the formal release of output specifications. It is this latter aspect that gives the step its name, since the release of specifications in effect invokes subsequent phases.

The INTEGRATION step encompasses the activities associated with the configuring and testing of that portion of the total system that was designed in the phase. Although it is traditional to consider integration to be a design area that would qualify as a stage in the framework, the integration activities have been distributed among the phases in recognition of the following facts: the expertise needed to test a portion of the system is similar to the expertise needed to design it; the assumptions made in a phase about the nature of the products that could be delivered by the subsequent phases must be checked once the subsequent phases complete their respective tasks; all models used to make these assumptions must be validated; and, finally, design errors found during integration must naturally be referred back to the phase in which they were introduced. It is therefore fitting that integration be considered a phase activity.

HARDWARE/SOFTWARE TRADE-OFFS

This topic was already introduced earlier in the context of the system design stage. In this section we provide a brief review of the framework's perspective on this issue in its own right undiluted by all the other details involved in the presentation of the framework. The discussion starts with the definition of H/S trade-offs, outlines the approach prescribed by the framework, and identifies the main problem areas.

The problem embodied in H/S trade-offs is that of allocating the system's functionality between hardware and software components (be they off-the-shelf or custom designed) in a manner that satisfies all system design constraints. Because systems are perceived as H/S aggregates, the consideration of H/S trade-offs is perceived to be a central system design issue. Its complexity is so high, however, that few methodologies make any attempt to deal with it, and most existing work focuses solely on computer systems selection, itself a difficult problem.

As far as the TSD Framework is concerned, the activities related to H/S trade-offs are distributed across the two phases of the system design stage. The system architecture design phase is engaged in a systematic process of reducing the binding options to the point where the binding phase is left to deal strictly with component selection from among a few feasible alternatives. Every system architecture design decision, taken in the exploration step, has implications with respect to the type of technology that would be needed to realize the system. Furthermore, partitioning into hardware and software needs to be carried out as part of this phase because all performance models used in the evaluation and inference steps demand, as a minimum, information about the distribution of the system's functions among various processors and about interprocessor communication costs. All such design decisions are actually subject to explicit review and analysis in the inference step. Of particular concern for the inference step is the rejection of any design solutions which limit the range of feasible binding options unnecessarily. Since the system architecture is presumed to be developed top-down, the option elimination process is characterized by an iterative sequence of refinements and inferences.

Having the range of binding options significantly reduced by the previous phase, system binding concentrates on selecting specific components among those still eligible. It is critical to proceed with the selection of individual components in the context of the entire system, and not by optimizing local decisions. This enables the focus to remain on the performance objectives of the system as a whole (cost included), where it belongs.

Neither option reduction nor component selection is a simple task. The former requires significant experience with system design and a good grasp of existing technology and current technological trends, issues that are difficult to formalize. The availability of appropriate performance models applicable both to performance evaluation and to technological inferences could, however, assist the designer in very important ways. While the number of conceivable binding options may be overwhelming, the development of reduction strategies and performance models for a few common ones is believed to be feasible, but nontrivial. Similar challenges are present in dealing with component selection, in the binding phase. On one hand, there is a need to develop adequate selection strategies for both software and hardware components. On the other hand, it is necessary to establish meaningful mappings between performance attributes present in the performance models mentioned above and those recognized in the actual component candidates.

FROM FRAMEWORK TO METHODOLOGY

This section introduces and illustrates a strategy that could assist a methodology developer to instantiate the TSD Framework by developing methodologies customized to particular applications and organizations. The approach draws heavily on previous experience with methodology development (e.g., [ROMA79]) and makes explicit decision processes which occurred informally in the past. Its theoretical foundation, however, is to be found in the following set of basic principles:

- The framework represents a general problem solving approach. Its role is to impose a particular structure over the class of possible problem solving approaches. The framework may limit both the set of acceptable problem solving strategies and the set of acceptable solutions.

- Methodologies are application dependent. The effectiveness of a methodology depends upon the extent to which its specification, design, and analysis techniques are tuned to the nature of a particular application.

- Methodologies are technology dependent. The nature of the technology being postulated, the relative costs of different technological alternatives (e.g., hardware versus software), and the recognized technological trends play a major role in methodology development and selection.

- Methodologies are organization dependent. The level of expertise, the cost of labor, and the structure of the organization affect to a significant degree the effectiveness of the methodology being used, i.e., the product quality and the personnel's productivity.

- Project management objectives impact methodology development.

- The methodology, i.e., the technical approach, must be supported and enforced by the project management techniques adopted.

The way in which these principles have been embodied in the methodology development approach will become evident in its description below.

Context identification.

If one assumes the availability of a framework, the TSD Framework in this case, the first task involved in methodology development is the establishment of the context in which the methodology is to function, i.e., the application area and the type of organization for which it is intended.

The importance of understanding the application area has long been acknowledged by system designers. Defense related systems, for instance, are generally separated into three general categories: embedded (real-time) systems, data processing systems, and command, control, communication, and intelligence systems. Unfortunately, this taxonomy is much too broad to be very useful. Data processing systems, for example, include both logistic command systems, which have features in common with business data processing, and geographic database systems which have a rather unique nature. The methodology developer has to identify both those application features that are unique and those that are common to other applications. Common features may suggest the use of similar techniques to be borrowed from already existing methodologies, while unique features may reveal special problems that need to be addressed. In such cases existing techniques may have to be adapted or new ones may need to be conceived.

The need to look at the type of organization for which the methodology is intended has been considered by few and only indirectly. (It is common practice to compare methodologies which are intended for different organization types and to pass on gratuitous judgements.) Such an omission, however, may lead to methodologies that ignore the obvious fact that the success of the methodology depends to a very large extent upon the people that use it. A low level of sophistication may demand the use of less formal and more mechanical techniques. A small organization may be able to cope with or to afford fewer project controls and reviews. A highly diversified company may not be in a position to impose a common methodology over the entire organization. A defense contractor must have a methodology compatible with the Department of Defense regulations. These are only a few issues that a methodology developer should consider from the start.

In order to illustrate how such considerations affect methodology development, the case of a small data processing organization will be considered. Its characteristics will be brought to light gradually throughout the remainder of this section. At this point, it suffices to introduce the organization by indicating its small size. Furthermore, it will be assumed that the systems to be developed are turnkey systems for relatively small data processing applications and that development and maintenance of these systems is to be the responsibility of the vendor organization.

Framework pruning.

The framework may be used as a methodology skeleton and checklist which is pruned and refined during methodology development based on the nature of the application and organization. The starting point is the discarding of whole phases which may be shown to bare no significance for the type of system being considered. In other words, the domain of acceptable solutions is restricted a priori based upon certain global issues which remove from consideration particular technological alternatives.

In our example three whole stages may be removed from the start: machine design, circuit design, and firmware design. The justification may be found in the following concerns. First, it is the lack of expertise available in the organization--nobody has any hardware background. Second, maintainability and cost considerations strongly suggest the use of off-the-shelf machines if at all possible--the nature of the application makes it possible. Third, even if development and product costs could be reduced and performance could be increased by using customized components, maintenance costs over several years may offset the balance. Finally, having the hardware vendor provide the hardware support through its organization may be not only cost effective but a positive marketing factor.

Phase redefinition.

The pruning of the framework must be followed by a redefinition of the objectives of the phases that are still present. In part, this is because the elimination of some phases and stages (e.g., machine design stage) results in fewer demands on the phases that are supposed to generate their requirements (e.g., system binding phase). However, further reductions in the range of acceptable design solutions also take place. Many reductions are often rooted in economic considerations which are addressed through some sort of standardization. The standardization may cover anything from the use of predefined hardware, software packages, operating systems, databases, file formats and implementation languages to prescribed system architectures. In the first case, a certain amount of "pre-binding" takes place, while the second case goes so far as to carry out a certain level of "pre-design." Neither may be accomplished without in-depth understanding of the application area and the organization involved. Both represent degrees of specialization motivated by the desire to achieve methodology effectiveness, be it measured in terms of quality, productivity, or both.

These issues may be illustrated by continuing the example. The nature of the application, small data processing, makes it possible for the vendor organization to select a single minicomputer as the common hardware support for all systems to be developed. The consequence is immediately felt in the system design stage. The objective of the system architecture design phase is reduced to the determination of how to allocate the system's functions among one or more minicomputers of a given type. The binding phase, in turn, is assigned the task of evaluating the proposed distribution against the characteristics of the actual machines and of generating the hardware and software requirements. The former specify the number of machines and the way in which they are configured. The latter contain descriptions of the software to be placed on each of the machines, the implementation language, and the communication protocols between the software pieces residing on different machines. The last two are always the same due to standardization considerations. At this point the reader may notice that, while none of the specification, design and analysis techniques of the methodology have been identified so far, significant system design decisions have already been taken.

Specification language selection and validation.
The design papers and documentation produced by various phases are, generally, required to adhere to certain standards of presentation. They vary in degree of formality from one methodology to another. At one extreme, there are no standards or the standards that exist are concerned with the general form and content of the document which otherwise is written in natural language. At the other end of the spectrum, formally defined specification languages are employed and all specifications are maintained on line and checked for adherence to the language syntax and semantics by mechanical means (e.g., compilers and interpreters). Most methodologies, however, fall somewhere in between combining formal and informal specifications. Furthermore, the specification language employed by some methodology is seen as its cornerstone--so much so, that often people talk about a specification language as if it were a methodology, when in actuality a specification language exists independent of any methodology that relies on its use.

There are several important reasons why specification languages occupy such a central role in methodology development. First, they establish the basis for precise communication among designers. Second, they influence to a significant degree the way in which a designer approaches a problem by defining a certain point of view and the concepts one uses to present a model of the system being designed. Third, they determine the nature of the design and analysis tools that may be provided to the designer and, thus, they affect the productivity of both the people involved in the design proper and in the review process. A high degree of mechanization of various design/analysis tasks and a properly human-engineered interaction with the language processor result in the exploration of a large solution space and increase the confidence level in the quality of the design.

Given the role played by specification languages and the investment required in making them available for use in the organization, their selection is generally done a priori rather than on a project by project basis. (The latter is not uncommon when there is a lack of commitment or experience with specification languages or in large and diversified organizations where individual projects have a high degree of independence.) The selection is affected by three key factors: the nature of the application, the background of the available personnel, and availability of tools supporting a particular specification language. Because no language is equally adequate from all three points of view, the choice is usually a compromise favoring one need over the others. One such example is the selection of a single specification language over the use of several languages each better suited for a particular subclass of the projects of interest. The rationale for this decision may rest with the desire to facilitate personnel transfer between projects, to limit retooling and training costs, to establish a common base for the interpretation of collected statistical data, etc.

In the case of our example, because the use of identical formalisms on all system design projects has obvious advantages, several specification languages could be adopted as company standard after evaluating their appropriateness for the type of projects being envisioned. The formalism selection and validation steps are thus eliminated from the methodology. Our sample organization could decide, for instance, in favor of:

- English text for the identification report;
- PSL/PSA [TEIC77] to assist the conceptualization phase by providing computer-aided assistance in the development of dataflow specifications for the system requirements;
- a modified use of PSL for both the system architecture and software configuration design phases;
- some form of pseudocode (syntactically checked by a locally developed tool) for the program design phase;
- and a standard programming language for coding.

<u>Selection of design/analysis techniques.</u>
The language selection is not independent of the design/analysis techniques being contemplated. A language that does not support the development of hierarchical specifications can hardly be expected to work well with a technique which emphasizes top-down design, for instance. The language is generally targetted toward particular design/analysis techniques. Nevertheless, it has to be chosen before attempting to carry out their selection and tuning. The design techniques are generally intended to help the designer structure the design process in such a way as to reduce the chance that a significant amount of effort is wasted on dead-end paths. The designer is provided most often with guidelines

rather than algorithms and with tools that assist with the rapid development of design specifications. The analytic techniques used to evaluate various properties of the design also provide feedback with regard to potential problems, weaknesses, and strengths of particular design alternatives.

The choice of one set of techniques over another is determined, in principle, by their relative costs. In practice, this is a most difficult task due to the lack of empirical studies and great variability between designers' experience and background. Mechanical and easily taught techniques are needed for less sophisticated designers but they often compromise both the quality and the performance capabilities of the resulting systems. By contrast, complex techniques are hard to teach and demand more experience but are more apt to deal with the design of novel systems and with the design of systems that must meet severe performance constraints.

System life-cycle considerations also affect the nature of the selected techniques. Some loss in efficiency and productivity is often acceptable if it brings about lower maintenance and higher reliability. Company design rules are common practice in the electronics industry and are intended to avoid well-known pitfalls and hard to understand designs. The use of standard parts having known properties also affects the nature of the design/analysis techniques.

Finally, the nature of the design/analysis techniques is also affected by the dominant constraint over the system. The reliability required from an unmanned space craft is quite different from that of an airline reservation system. In the latter case manual intervention is possible; in the former it is not. The result is a significant disparity between the structures used in the design of the two systems and between the scrutiny to which they are subjected during testing.

In our simple illustration, because software design is perceived to be the dominant design activity, a set of design guidelines is adopted based on practices common in the industry. They include top-down stepwise refinement, modularity, etc. Furthermore, structured programming is to be enforced by the nature of the pseudocode and by the implementation language itself which includes all needed structured constructs. The use of goto's is generally prohibited by a set of programming standards.

Driven by the desire to minimize both development and maintenance costs, a systematic quality assurance program is also put in place. Some of its components are listed below. Consistency checking and logical verification are to be carried out as walk-throughs; the performance evaluation step is formally identified only in the architecture design and software configuration design phases--it is limited to questions of time and space and it is done by hand calculations and occasional benchmarks; the inference step is handled in an ad-hoc manner by taking advantage of past experience and the similarity between systems. Without being too sophisticated, these choices may, nevertheless, prove adequate for our hypothetical organization. If not, adjustments will have to be made later. Because the organization and the technology change, the methodology will have to be revised too, even if the original selection were adequate.

Sequencing of design/analysis activities.

Besides the use of particular techniques, another factor that contributes to the effectiveness of some methodology is the manner in which design activities identified by the framework as steps within various phases are to be sequenced on actual projects. To illustrate this point, consider a relatively well-understood activity such as program testing and three testing strategies. The first one requires the complete coding of the program before testing. The second one employs a strict top-down testing approach where the main program is tested first, then all second level modules are added and tested, then the next level is added and tested, etc. The third strategy starts by testing the main program after which a small number of stubs may be replaced by code before resuming the testing process. The last one turns out to be the most cost effective of the three both in terms of the time being expended and the quality of the tests. This is because it combines control and flexibility.

Similar considerations are generally involved in sequencing all activities prescribed by some methodology. Complex sequencing strategies have been observed in some existing design methodologies (e.g., [MCCL75]). The right answer depends again upon the particulars of the application and the organization involved. Here are some examples of issues that need to be addressed: the selection of an overall design strategy (top-down, bottom-up, etc.); the points in the design when performance evaluation should be carried out through the use of benchmarks rather than simulation; the frequency and placement of design reviews; the circumstances under which parallel development of parts of the system may be permitted; the placement of project planning activities; etc. The choices are generally made based on past experience and deductive reasoning. A more judicious selection method needs to be based on models (of the system design process) which would permit the designer to establish the potential impact of altering such parameters as error frequency, error recovery costs, review costs, etc.

Considering the example again, project control objectives may dictate that all relevant phases are to be done in the order in which they appear in the framework except for the case when corrections to earlier work are deemed necessary. Different subsystems, however, are permitted to be in different stages of development as long as their interfaces are clearly identified. On the other hand, within a single phase, all steps are to be repeated, in the same sequence as in the framework, for every level of the hierarchical specification being produced. Formal project reviews are scheduled at the completion of each phase for each subsystem involved.

Adding the project management components.

Methodology development must also include the managerial perspective on system design. It adds (to the methodology) activities which, without being directly related to the design process, are essential to managing any reasonably sized project: project organization, scheduling, resource allocation, reporting procedures, configuration control, etc. Their placement in the methodology has to be coordinated with the design activities but their role is fairly well-understood and, consequently, no further elaboration is necessary at this point.

The development of the methodology has to be followed by the even greater effort represented by its integration into the organization. This includes retraining, retooling, the use of experimental pilot projects, gradual introduction of the methodology in production, etc. These issues, however, are outside of the scope of this paper.

CONCLUSIONS

The TSD Framework builds directly on the current understanding of system design methodologies with respect to both the phases and the steps that make up its structure. Its steps represent a taxonomy of the design activities generally encountered in system design. Its phases, aside from those included in the system design stage, have been recognized already by other authors. There are, however, two important distinctions between the way phases and steps are used here and elsewhere. First, the grouping of activities into a phase is based upon the nature of the technical expertise they require rather than upon considerations related to project management. The latter are relegated to methodologies and are not part of the framework. Second, the steps are abstractions over classes of design activities and not specific actions to be carried out by the designer in some prescribed order. These differences stem from the fundamental distinction between frameworks and methodologies.

The criteria used in the selection of both phases and steps are a direct reflection of the principle of separation of concerns. The traditional separation between hardware and software design, for instance, is captured by the identification of distinct phases associated with each. At the same time, however, because judicious partitioning of the system functions between hardware and software demands the two to be considered together and to perform certain trade-offs, the system design stage has been included. It separates the selection and specification of the hardware and software from hardware and software design.

The TSD Framework has been used primarily as a way of specifying a class of distributed system design methodologies called the TSD Methodologies. The development of one such methodology has been carried out following the approach described in this paper. Unfortunately, the methodology was so general in scope that it exercised the approach only to a very limited extent. However, a systematic evaluation of the approach in an industrial environment is anticipated to take place in the near future. In the meantime, the approach is now being used again in the development of a distributed data-processing methodology that could later be easily adapted to the needs of certain types of organizations. Despite the limited evaluation of the approach, the fact that it is based on the earlier experiences with the selection of design methodologies for two very different organizations recommends it for serious consideration.

REFERENCES

[CHAN78] Chandy, K. M. and Yeh, R. T. (editors), Current Trends in Programming Methodology, vol. 3, "Software Modeling," Prentice Hall, 1978.

[MCCL75] McClean, R. K. and Press, B., "The Flexible Analysis Simulation and Test Facility: Diagnostic Emulation," Technical Report TRW-SS-75-03, TRW, Redondo Beach, CA. 90278, 1975.

[ROMA79] Roman, G.-C., "Verification Procedures Supporting Software Systems Development," 1979 NCC Proc., pp. 947-956, June 1979.

[ROMA82] Roman, G.-C. et al, TSD Methodology Assessment, Final Report, Contract F30602-80-C-0284, Washington University, Saint Louis, Missouri 63130, 1982.

[TEIC77] Teichroew, D. and Hershey, III, E. A., "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems." IEEE Trans. on Soft. Eng. SE-3, No. 1, pp. 41-48, January 1977.

[WASS78] Wasserman, A. I. and Belady, L. A., "Software Engineering: The Turning Point," Computer, pp. 30-41, September 1978.

[WEGN79] Wegner, P. (editor), Research Directions in Software Technology, MIT Press, 1979.

[YEH77] Yeh, R. T. (editor), Current Trends in Programming Methodology, vol. 2, "Program Validation," Prentice Hall, 1977.