Report Number: WUCS-84-10

1984-05-01

# A Total System Design Framework

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

# A TOTAL SYSTEM DESIGN FRAMEWORK

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

WUCS-84-10

May 1984

Department of Computer Science
Washington University
St. Louis, Missouri 63130

As appeared in Computer 17, No. 5, May 1984, pp. 15-26.

# COMPUTER

May 1984 Volume 17 Number 5 (ISSN 0018-9162)


## FEATURE ARTICLES


Cover design: Jay Simpson



## SPECIAL FEATURES



## DEPARTMENTS

*System inadequacies may be the fault of designers who didn't look at the whole picture. To be successful, a design must merge hardware and software concerns into a single, unified perspective.*

# A Total System Design Framework

**Gruia-Catalin Roman, Mishell J. Stucki, William E. Ball, and Will D. Gillett**
**Washington University in Saint Louis**

Successful computer systems are usually the result of many months of careful planning and development. Such an intense planning effort requires integrated design methodologies that cover the entire system development life cycle—from problem definition to final testing—and span the traditional hardware/software boundaries. Furthermore, to be effective, these methodologies must be specialized to the particular application, technology, and organization.

When developing such methodologies, we need to consider a number of issues. The first is that the application drives the methodology. For example, designers must deal with response-time requirements in real-time systems, the validation of security kernels in information systems, and special test procedures for systems that do not allow "fixes" during deployment (such as spacecraft, satellites, and certain weapons). Second, methodologies must exploit available technology to enhance both design productivity and quality. Program generators and microprocessor-based architectures are examples of such exploitation for business data processing and industrial control. Finally, we must remember that if corporate resources such as available manpower, the training and experience of personnel, available support tools, and project management practices are not taken into account during design, the resulting methodology will be severely limited.

In this article we discuss a total system design (TSD) framework that supports the development of integrated system design methodologies. Issues that must be addressed during system development are identified and presented in a helpful structure (1) to aid in our understanding of the design process, (2) to serve as a foundation for the development of new methodologies, and (3) to provide a means for methodology analysis and selection. The TSD framework's broad scope makes it useful in establishing company-wide system-development standards, even for organizations whose concerns span several application areas. The framework is compatible with Department of Defense standards[1] and with research results described in various sources of professional literature.

Our presentation of the TSD framework begins with an overview of its stages, phases and steps, followed by a discussion of hardware-software trade-offs. We then show the development of a system design methodology for a particular type of company and a particular application area. This example serves to identify the key issues facing the methodology developer and to outline a strategy useful for developing custom methodologies and for establishing standard system design practices at the company level.

## Stages and phases

In the TSD framework, system development is partitioned into stages and phases as shown in Figure 1. The stages constitute a natural structuring based on major differences in applied technology. The phases, which make up a stage, impose an ordered, layered approach to design, reducing the risk of error and producing systems that are easier to understand and maintain.

**Problem definition stage.** During this stage the functional and the nonfunctional requirements of the computer system are determined. We believe that successful system design proceeds from a clear understanding of the problem being addressed and, therefore, consider this stage to be of supreme importance. Two phases of development occur during this stage to ensure the accurate


0018-9162/84/0500-0015$01.00 © 1984 IEEE

**System design stage.** During this stage the hardware and software requirements are established for each component of the system. Requirements for the functional and nonfunctional capabilities of the components are specified, including the interfaces between these components (such as communication protocol, programming language support, and operating system primitives). These requirements are closely followed (1) during the procurement and/or design of the individual system components (such as computers, operating systems, and peripherals) and (2) during the subsequent integration of these components into the final system.

The system design stage includes two phases: system architecture design and system binding. The main concern of the developer during the *system architecture design* phase is to investigate system design alternatives and their potential impact on the various system configurations being considered. In the case of a distributed database system, for instance, the developer may use this phase to identify data and processing distribution and the number of nodes present in the network. Particular software or hardware components are not selected during this phase, but the system-level components are evaluated to assure that a set of reasonable binding options exists.

During the *system binding* phase, the actual mix of hardware and software is selected. The hardware and software requirements generated during this phase may combine off-the-shelf and custom-built hardware and software components. How these components are selected is determined by the design 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. The system designer may postulate, for instance, the use of minicomputers having certain capabilities, price range, and software support. The actual selection, however, would occur in this phase, which is also charged with specifying the requirements for the software that would still need to be produced.

**Software design stage.** There are three phases involved in this stage. During the first phase, *software configuration design,* off-the-shelf software is procured and the overall high-level software system design for each programmable system component is established. This involves (1) the structuring of the software into such divisions as subsystems, virtual layers, and tasks, (2) the definition ot interfaces between components; and (3) the generation of requirements for each component. The *program design* phase takes these requirements and produces the program design (data and processing structures), which, together with all pertinent assumptions and constraints, makes up the implementation requirements. These are used by the *coding phase* to build the actual programs.

**Machine design stage.** This stage is similar to the first two phases of the software design stage. During this first phase of machine design, the *hardware configuration design* phase, off-the-shelf machines are procured and the high-level architecture of custom hardware is designed. Component requirements are developed for all entities that are part of the custom hardware and passed on to the component design phase. A register-transfer level machine description, which is used to determine the circuit design requirements and the firmware requirements, is generated during this stage.

**Circuit design stage.** Three phases of system development occur during this stage: *switching circuit design, electrical circuit design,* and *solid state design.* During each phase, design requirements are generated for the phase immediately following. In this article, we are concerned mainly with system design. Since this area deals with issues traditionally faced by the electrical engineer, we will not go into further detail.

**Firmware design stage.** This stage consists of three phases that are an analog to program design, coding, and compilation. These phases are *microcode design, microprogramming,* and *microcode generation.* Like the circuit design stage, this area is of limited concern to us here.

* * *

Table 1 summarizes the results of comparing the TSD framework with the DoD system development review standard,[1] a distributed system design methodology,[2] and two software development methodologies.[3,4] The following are distinguishing features of the TSD framework:

**Table 1. The correspondence of system development phases proposed by various authors.**

| TSD (Assuming off-the-shelf hardware) | DoD-USAF (based on MIL-STD-1521A[1]) | BOEHM[2] | METZGER[3] | FREEMAN[4] |
|---|---|---|---|---|
| IDENTIFICATION | SYSTEM REQUIREMENTS | | DEFINITION | NEEDS ANALYSIS |
| | | SYSTEM REQUIREMENTS | | |
| CONCEPTUALIZATION | | SOFTWARE REQUIREMENTS | | SPECIFICATION |
| SYSTEM ARCHITECTURE DESIGN | SYSTEM DESIGN | PRELIMINARY DESIGN | | |
| BINDING | | | | |
| SOFTWARE CONFIGURATION DESIGN | PRELIMINARY DESIGN | | DESIGN | ARCHITECTURAL DESIGN |
| PROGRAM DESIGN | DETAILED DESIGN | DETAILED DESIGN | | DETAILED DESIGN |
| CODING | CODE PRODUCTION | CODE AND DEBUG | PROGRAMMING | IMPLEMENTATION |
| —* | INTEGRATION AND VALIDATION | TEST AND PREOPERATIONS | SYSTEM TEST | |
| | | | ACCEPTANCE | |

*The TSD equivalent of these phases falls in the integration step, which is outside the framework proper (see p. 21).

**Formalism selection.** This step encompasses the activities involved in selecting a formalism for a particular problem domain. Each phase may involve one or more different formalisms: programming languages for coding; pseudocode for program design; logic diagrams for switching circuit design; and stimulus-response graphs and logic models for conceptualization. Candidate formalisms are chosen on the basis of their expressive power in that domain and their ease of use, lack of ambiguity, ease of analysis, and potential for automation. This selection step often occurs long before the other steps, sometimes because the methodology used is based on a particular formalism, but more often because of policy or the availability of tools tailored to that formalism.

**Formalism validation.** In this step, we determine whether a formalism has the expressive power needed for a particular task. We also evaluate how easy formalisms are to use. These tasks may involve both theoretical and experimental evaluations. Theoretical results may indicate the power and the fundamental limitations of the formalism; experience gained from similar projects may provide insight into the formalism's appropriateness and ease of use. One project may reject Fortran because it does not support recursion, while another may consider it a source of potential maintenance problems because it has limited support of structured programming. Finite-state machines may be employed in defining some communication protocols but are inappropriate when unbounded queues are present and prove cumbersome when the number of states is large. The validation step also includes evaluations of the formalism's potential for design automation (as a way to increase productivity) and its ability to support hierarchical specifications (as an aid to controlling complexity).

**Exploration.** This 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 significantly restricted.

**Elaboration.** In this step, ideas produced in the exploration step are given form through the use of various formalisms. Coding, specification writing, and circuit layout drawing are typical activities associated with this step, but its scope extends to the building of a concrete object such as a piece of hardware. The effectiveness of this step may be greatly increased through the use of a variety of design and manufacturing aids.

**Consistency checking.** This step encompasses activities such as checking for incorrect uses of formalisms; for contradictions, conflicts, and incompleteness in specifications; and for semantic errors. Checking includes verifying consistency between different levels of abstraction in a hierarchical specification and reconciling multiple viewpoints. Two examples of the types of problems addressed in this step are a subroutine that is never called and a mismatch between the number of input lines for two instances of the same device on a logic diagram.

**Verification.** In this step, we demonstrate 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 verifying program correctness. The well-known difficulty of this task is representative of the difficulty of the verification step in general.

**Evaluation.** In this step, we determine if a design meets a given set of constraints. Constraints include both those that are part of the requirements specification for the phase and those that 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.

**Inference.** In this step, the potential impact of design decisions is assessed. Questions are addressed such as (1) How will the system impact the application environment? Can we afford the implementation? Is personnel retraining too expensive?; (2) Can subsequent phases accommodate the decisions made in this phase? Is the bandwidth choice reasonable?; (3) How does the design affect our ability to maintain and upgrade the system? Will parts be available five years from now?; and (4) How does the design affect implementation options? Is there a good reason for ruling out mainframes? These issues must be considered in every phase, but they are particularly critical in stages that define architectures.

Moreover, the inference step forces a review of the implications of current design decisions on the hardware technology and the supporting computational techniques that subsequent phases will use. This approach allows top-down design with minimum risk of having to backtrack, and at the same time, requires that the capabilities of current technology be reviewed from both the computational and cost standpoints.

**Invocation.** This step encompasses the activities associated with releasing the results of the phase. It includes quality control activities involving tangible products and review activities that lead to the formal release of output specifications. The release of output gives the step its name, since this release in effect "invokes" subsequent phases.

**Integration.** In this step, the portion of the total system designed in the phase is configured and tested. Traditionally integration is considered a design area, and would therefore qualify as a stage in the framework. However, we have chosen to distribute integration activities among the phases because (1) the expertise needed to test a portion of the system is similar to the expertise needed to create its requirements, (2) the assumptions made in a phase about the nature of the products that could be delivered by subsequent phases must be checked once the subsequent phases complete their tasks, (3) all models used to make these assumptions must be validated, and (4) errors found during integration must be resolved in the phase that created the requirements.

and an adequate database management system is usually preferred over a faster and cheaper machine that lacks software support—even if the choice means greater investment.

In the system architecture design phase of the TSD framework, we systematically reduce binding options so that when we enter the binding phase we can concentrate on selecting components from a few feasible alternatives. Every decision about system architecture design impacts the type of technology needed to realize the system. Furthermore, hardware and software need to be partitioned as part of this phase because all performance models used in the evaluation and inference steps demand, as a minimum, information about how the system's functions are to be distributed among various processors and how much delay is incurred due to interprocessor communication. All such design decisions are actually subject to explicit review and analysis in the inference step. Of particular concern is the rejection of any design solutions that unncecessarily limit the range of feasible binding options.

Although this perspective on hardware-software trade-offs is new, we are not implying that hardware-software trade-offs have been ignored. More often, authors have simply failed to make this issue explicit in their methodology definitions. Let us consider again SREM[2] and SARA[6].

Central to using SREM is the process of breaking down and allocating system functions. Although a function may have many workable decompositions, the designer selects one over the other to exploit the performance opportunities offered by one or another system support structure, that is, the hardware/software mix that might ultimately be part of the system. SREM even provides the means for evaluating the choice relative to the assumptions made about the system support structure. (Bear in mind that unless we consider the nature of the underlying support system, we can say very little about the potential performance of the system—for example, the time required to execute two potentially parallel computations may be either $(a + b)$, if a single machine is available, or $\max(a, b)$ if two machines are used.)

SARA goes one step further. When building blocks with known characteristics are available, SARA explicitly addresses the issue of biasing the design toward a particular hardware/software mix. Furthermore, when building blocks are not available, one system design task is to generate a set of appropriate building blocks—hardware and software components. The composition/partition evaluations have many elements of the TSD inference step built into them.

Since the range of binding options has been significantly reduced in the previous phase, system binding is primarily selecting specific components among those still eligible. (Timmreck[9] gives a survey of available techniques.) The selection of individual components must be in the context of the entire system; it should not be simply the optimization of decisions about a certain part of the system. The focus thus remains 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. Appropriate performance models that can be used for both performance evaluation and in making technological inferences can be of significant assistance. (They could become part of libraries such as the one containing the available building blocks in SARA.) Granted, the number of conceivable binding options may be overwhelming, but we can still develop reduction strategies and performance models for a few of the more common options, although the process is complex. Similar challenges face us in component selection during the binding phase. We need to develop adequate selection strategies for both software and hardware components, and we must establish meaningful mappings between the performance attributes of the models and those recognized in actual component candidates.

## From framework to methodology

The TSD framework can be used in generating company-wide system development standards and in developing custom methodologies. The strategy described here is rooted in our experience with the development of custom methodologies for several organizations, but we can use it to identify the types of issues faced by all methodology developers and evaluators.

The strategy consists of the following steps:

- *Context identification.* Establishment of the context in which the methodology will be used.
- *Framework pruning.* Removal of stages and phases that are not required by the situation at hand, and the redefinition of the remaining ones.
- *Selection and validation of the specification language.* Selection of specification and programming languages that suit the class of systems to be developed and the abilities of the designers.
- *Selection of design/analysis techniques.* Selection of techniques that are cost-effective and meet the overall constraints of the class of applications.
- *Sequencing of design/analysis activities.* Determination of how activities will be interleaved to optimize design productivity.
- *Addition of project management components.* Addition of management activities that support system design.

Of course, methodology development has to be followed by the even greater effort of methodology integration. Introducing the methodology into the organization involves tasks such as retraining, retooling, using experimental pilot projects, and gradually introducing the methodology in production. These issues, however, are outside the scope of this article.

The following example is used to illustrate the concepts discussed in this section: A vendor employs 10 software designers and programmers and produces turn-key financial data processing systems. The systems are hardware/software packages and are intended for use by small organizations. They are maintained by the vendor and should not be modified or expanded by the customer.

various phases must generally adhere to certain standards of presentation, which vary in formality. At one extreme, there are no standards, or the standards that do exist are concerned with the general form and content of the document, which otherwise is written in natural language. At the other extreme is the use of formally defined specification languages, and all specifications are maintained on line and checked mechanically for adherence to language syntax and semantics—through compilers and interpreters, for example. Most methodologies fall somewhere in the middle of these two extremes. Furthermore, the specification language used by a methodology is considered its cornerstone—so much so, that people often talk about a specification language as if it were a methodology, when in actuality the specification language is independent of the methodology.

There are several important reasons that specification languages occupy such a central role in methodology development. First, they establish the basis for precise communication among designers. Second, they greatly influence the way a designer approaches a problem because they define a certain point of view and the concepts used to present a model of the system. Third, specification languages determine the nature of available design and analysis tools and thus affect the productivity of those involved in the design proper and in the review process. If various design/analysis tasks are highly mechanized, and the interaction with the language processor is engineered with careful attention to human concerns, many solutions can be explored and the level of confidence in design quality increases.

Given the major influence of specification languages and the investment required in making them available, we usually select them a priori rather than according to the individual project. (A project-by-project selection is more common when there is a lack of commitment to or experience with specification languages, or when individual projects have a high degree of independence, such as in large and diversified organizations.) The selection is affected by (1) the nature of the application, (2) the background of the available personnel, and (3) the availability of supporting tools. 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 example is the selection of a single specification language over the use of several languages, even though each language is better suited for a particular subclass of projects than the single one chosen. The rationale for this decision may be a 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, or any combination of these.

In our example, the use of identical formalisms on all system design projects has obvious advantages. Therefore, we could adopt several specification languages as company standards. 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,[10] which could provide computer-aided assistance in the development of data-flow specifications for system requirements (conceptualization phase);
- 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
- one or more standard programming languages for coding.

**Selection of design/analysis techniques.** The language selection depends on 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 that emphasizes top-down design, for instance. The language is usually selected with a view toward particular design/analysis techniques. Nevertheless, the language must be chosen before we can attempt to select and tune such techniques. Design techniques generally help the designer avoid dead-end paths in the design process. The techniques are more guidelines for the designer rather than algorithms, providing tools that assist in the rapid development of design specifications. The analytic techniques used to evaluate design properties also provide feedback with regard to potential problems, weaknesses, and strengths of particular design alternatives.

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 selecting stages and phases directly reflect the principle of separation of concerns. For example, the separateness of hardware and software design is preserved by identifying the distinct stages associated with each. However, when we partition system functions between hardware and software, we need to consider the two together. The system design stage provides for this view by separating the selection and specification of the hardware and software from the design of the hardware and software.

We envision the TSD framework as a means by which existing methodologies may be rigorously evaluated against each other before empirical experiments are set up. It is also the basis for a systematic approach to the development and evaluation of design methodologies tuned to the needs of particular application area. *

## Acknowledgments

This work was supported in part by the Rome Air Development Center and by the Defense Mapping Agency under contract F3062-80-C-0284. We acknowledge the contributions of R. K. Israel and J. T. Love in the investigation of certain aspects of the TSD framework. We also thank H. R. Lykins, who edited and assisted in reviewing the final draft of this article.

## References

1. *Technical Reviews and Audits for Systems, Equipments, and Computer Programs,* Department of Defense (USAF), MIL-STD-1521A, Washington, DC, June, 1976.
2. B. W. Boehm, "Software Engineering," *IEEE Trans. Computers,* Vol. C-25, No. 12, Dec. 1976, pp. 1226-1241.
3. P. W. Metzger, *Managing a Programming Project,* Prentice-Hall, Englewood Cliffs, N.J., 1973.
4. P. Freeman, *Tutorial on Software Design Techniques,* IEEE-CS Press, Los Alamitos, Calif., 1976.
5. *Specification Practices,* Department of Defense, MIL-STD-490, Washington, DC, Oct. 1968.
6. G. Estrin, "A Methodology for Design of Digital Systems," *AFIPS Conf. Proc.,* Vol. 47, 1978 NCC, AFIPS Press, Reston, Va., 1978, pp. 313-324.
7. P. M. Mariani and D. F. Palmer, *Distributed System Deisgn,* IEEE-CS Press, Los Alamitos, Calif., 1979.
8. R. W. Jensen and C. C. Tonies, *Software Engineering,* Prentice Hall, Englewood Cliffs, N.J., 1979.
9. E. M. Timmreck, "Computer Selection Methodology," *Computing Surveys,* Vol. 5, No. 4, Dec. 1973, pp. 199-222.
10. D. Teichroew and E. A. Hershey, III, "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems," *IEEE Soft. Engineering,* Vol. SE-3, No. 1, Jan. 1977, pp. 41-48.

**Gruia-Catalin Roman,** an associate professor in the Department of Computer Science, Washington University, has been on the university's faculty since 1976. He is also a software engineering consultant for several large firms. He is currently working on the formal specification of geographic data processing requirements, distributed system design models and methodologies, and the dynamics of software development environments. His prior work includes methodology development and assessment and studies in formal languages, biomedical simulation, computer graphics, and distributed databases.

Roman was a Fulbright Scholar at the University of Pennsylvania, where he received a BS in 1973, an MS in 1974 and a PhD in 1976, all in computer science. He is a member of Tau Beta Phi, ACM, and IEEE.

**Mishell J. Stucki** is a lead engineer in the Computer Systems and Software Engineering Department of the McDonnell Douglas Astronautics Company, St. Louis Division, where he is involved in research on special-purpose computer architectures. From 1964 through 1982, he was on the research staff of Washington University where he worked on synthesis techniques for asynchronous sequential circuits; the definition, design, and development of a set of digital building blocks called Macromodules; coordination techniques for distributed database architectures; and the TSD framework. From 1962 to 1964, he was on the research staff of the Massachusetts Institute of Technology where he participated in the design of an early minicomputer called Linc. Stucki received a BA in mathematics from Harvard in 1963 and an MSEE from Washington University in 1973.

**William E. Ball** is a professor in the Department of Computer Science at Washington University, where his work includes artificial intelligence, knowledge representation, expert systems, and database systems. His experience includes the development of digital signal processing systems and on-line information retrieval systems. Ball received a DSc in chemical engineering from Washington University. He is a member of ACM and the Association for the Advancement of Artificial Intelligence, and an affiliate member of the IEEE-CS.

**Will D. Gillett** is an associate professor in the Department of Computer Science at Washington University, where he works on language design and implementation, algorithm analysis, database development, and software engineering with an emphasis on applying formalisms to the solution of practical problems. Prior to this position, he was a systems analyst for the Office of the Secretary of Defense.

Gillett holds a BA and MA in mathematics from the University of California, Irvine, and a PhD in computer science from the University of Illinois. An active member of ACM, he cochaired the 1977 North Central Regional ACM Conference. He is also a coauthor of *An Introduction to Engineered Software* (Holt, Rinehart, & Wilson, 1982).

Questions about this article can be directed to Gruia-Catalin Roman, Dept. of Computer Science, PO Box 1045, Washington University, St. Louis, MO 63130.