Report Number: WUCS-79-10

1979-09-01

Total System Development Framework

Gruia-Catalin Roman

Building on the fundamental assumption that effective methdologies are problem and environment dependent, a suggestion is made to distinguish between methodologies and the methodological frameworks they instantiate. TSD (Total System Development) is put forth as a candidate framework able to assist in the generation and evaluation of specific system development methodologies, where systems are defined as distributed hardware/software aggregates.

TOTAL SYSTEM DEVELOPMENT FRAMEWORK

Gruia-Catalin Roman

WUCS-79-10

September 1979

This paper was produced under contract with Rome Air Development Center, Griffiss AFB, New York 13441, and in collaboration with Defense and Space Systems Group of TRW, Redondo Beach, California 90278, and Defense Mapping Agency, Washington, D.C., 20305.

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

As appeared in Proceedings of the 1981 ACM North Central Regional Conference, September 1981, pp. 135-142

A METHODOLOGICAL FRAMEWORK FOR THE DESIGN OF DISTRIBUTED SYSTEMS*

Gruia-Catalin Roman

Department of Computer Science
Box 1045
Washington University
St. Louis, Missouri 63130
(314) 889-6190

Abstract.

Building on the fundamental assumption that effective methodologies are problem and environment dependent, a suggestion is made to distinguish between methodologies and the methodological frameworks they instantiate. TSD (Total System Development) is put forth as a candidate framework able to assist in the generation and evaluation of specific system development methodologies, where systems are defined as distributed hardware/software aggregates.

Keywords: methodological framework, methodology, system, hardware, software.

* This work was supported by Rome Air Development Center, Griffiss AFB, New York 13441 under contract F30602-78-C-0148.

INTRODUCTION

During recent years the computer field has experienced an extreme proliferation of methodologies, particularly in the system development areas. The result has been an increased awareness of the important role played by methodology, an awareness muddled to some extent by a certain degree of confusion with respect to usage of terms and claims of generality made by various authors. Therfore, it seems reasonable that one ought to stop for a moment, look back and try to put together a global picture of what has been accomplished. However, the rationale behind this paper goes beyond surveying in an attempt to merge concepts, strategies, and issues under a cohesive framework whose role is to provide a philosophical and technical foundation supporting research, development, and evaluation of system development methodologies.

The idea of conceiving the methodological framework as a new investigative tool stems from the firm belief that effective methodologies are environment and problem dependent and, as such, the ability to extract the essence of various approaches for purposes of comparison presupposes some abstraction mechanism which is more or less environment and problem independent. However, the significance of the methodological framework need not be found only in its analytical value, but also in its pragmatics. While methodology exportation may be limited to problems and environments similar in nature and several methodologies may have to coexist within the same organization, the framework may be unique and easily transportable, thus providing a common reference point. Based on the foundation laid by the framework, different organizations and projects may adopt distinct methodologies by recognizing the specificity of their own environment or problem.

The high level of abstraction of the methodological framework manifests itself above all in a less procedural character emphasizing sequences of intermediate results and not the techniques used to generate them. Furthermore, the framework assumes an idealized perspective stressing fundamental relationships between intermediate results and not optimal ways in which they may be produced. For instance, the fact that some parts of the same project may have to be at different steps in the development cycle is of no relevance from the point of view of the framework. Managerial issues and techniques are expressly disregarded due to their strong dependence on the organizations's environment. Once the technical issues are clarified, well-matched management strategies may be selected in order to maximize the effectiveness of the methodology instead of proceeding in the opposite order as is commonly done today.

A global view of the methodological framework reveals its topology to be a directed acyclic graph where each vertex denotes a distinct methodological phase and each arc indicates a dependency relation between phases. In this context, a <u>phase</u> is defined as a set of activities whose overall effect is the generation of one or more well-defined specifications required by its immediate successors in the graph. The reason for the acyclic nature of the graph lies in the fact that interphase feedback has little significance at the high level of abstraction employed by the framework. This fact does not preclude specific methodologies from exploiting freely the use of interphase feedback loops or from considering look-ahead strategies.

To define a particular phase one has to identify its position in the framework, its interactions with the neighboring phases, and its functionality, i.e., the relationship between the specifications received and produced by the particular phase. Furthermore, it is the contention of this paper that judicious separation into phases ought to reflect fundamental relationships between activities and the body of knowledge required to carry them out; sequences of activities employing the same know-how should be part of the same phase.

The complexity of individual phases may require further elaboration in terms of steps, partitions of the set of activities involved in some given phase. While the relation between phases is one of dependency, the relation between steps is of a sequencing nature. It describes the set of acceptable step sequences and assumes the representation of a directed graph. The presence of cycles is due to the recognition of feedback loops within a phase. As a matter of fact, the steps are partitioned into two classes: synthetic and analytic steps. Synthetic steps represent groups of activities whose task is that of generating intermediate versions of some specification and, as such, involve no feedback. Analytic steps are associated with the process of evaluating some specification in order to decide if backtracking is necessary. The latter group plays a key role in quality control and also in assuring compliance with any predefined constraints.

As stated in the beginning of this section, the goal of this paper is to propose a specific methodological framework, not only as an illustration of how such a framework might be conceived but also as an attempt to make available a new investigative tool for the study of system development methodologies. The framework under consideration is called the Total System Development (TSD) framework and is intended to emphasize particularly issues related to distributed systems and the hardware/software partitioning problem. By providing a unified treatment of a large class of system development methodologies, TSD should prove useful both to researchers and policy makers by enabling them to carry out a more judicious evaluation of existing and new methodologies as well as fruitful comparisons.

The remaining sections are dedicated exclusively to describing the TSD framework. The next section discusses the separation of the development cycle into phases and includes a short description of each phase. Some of the front-end phases are later detailed by focusing on the key issues faced by each one of them, its outputs, and the availability or lack of adequate techniques to support its activities.

TOTAL SYSTEM DEVELOPMENT (TSD) FRAMEWORK OVERVIEW

Since 1968, the year when the term "software engineering" was born, significant progress has been made in understanding and controlling the software development cycle. A wealth of methodologies has evolved along with design principles and techniques, specification languages, and so on. However, the late seventies marked the emergence of two powerful trends: distribution and VLSI. As a result, the software/hardware relationship has grown more complex while, at the same time, systems development has begun to demand a greater understanding of hardware architectures and the impact they might have on a system's design and characteristics. Consequently, a critical point has been reached where software development and hardware selection and design must be brought together under the umbrella of a unifying conceptual framework. Total System Development (TSD) is put forth as a candidate framework particularly well suited for this task.

TSD is meant to establish the basis for an integrated approach to computer systems development and to contribute to the development of new methodologies which treat systems as distributed software/hardware aggregates, thus breaking the barrier between software and hardware design. Furthermore, it is the author's hope that TSD might stimulate greater collaboration between software and hardware engineers by indicating points of commonality between the two types of activities and the nature of the hardware/software trade-offs involved in system development.

As suggested earlier, TSD is an abstraction based on the analysis of current software system methodologies, hardware selection methods, and hardware design approaches. TSD phases have been slected by grouping together system development activities related to each other and sharing the use of a common knowledge base (theory, techniques, skills, etc.), a variation on the principle of separation of concerns. As a result, some phases deal with issues of the application domain, others involve computational structures, etc. Figure 1 contains a diagram depicting the TSD phases and their fundamental interrelationships. Short descriptions of each phase are included in the remainder of this section. A phase by phase detailed presentation follows starting with the next section.

The problem IDENTIFICATION phase is informal in nature and has an exploratory flavor. It denotes the gathering of information about those activities of the application domain which need to be supported by some proposed system. A good understanding of the application domain is essential for the successful completion of this phase.

During the CONCEPTUALIZATION phase, the information previously gathered is organized as a formal model called a conceptual model. The entire phase is application domain dependent and rests on the ability to formalize the problem domain. The conceptual model becomes the basis for the rest of the development process and, as such, there are good reasons to propose it also as a foundation for all contractual agreements.

The system REALIZATION phase moves the development process from the application domain modeling into the computational domain. For the first time, a realization of the system is proposed. Performance evaluation plays an important role during this phase by helping in the selection between alternate realizations. It ought to be stressed at this point that the realization phase results in a technology independent solution to the problem at hand. Considering the rapid technological changes which have occurred during the last decade and continue to be manifest today, it becomes necessary to separate the technology independent aspects of the

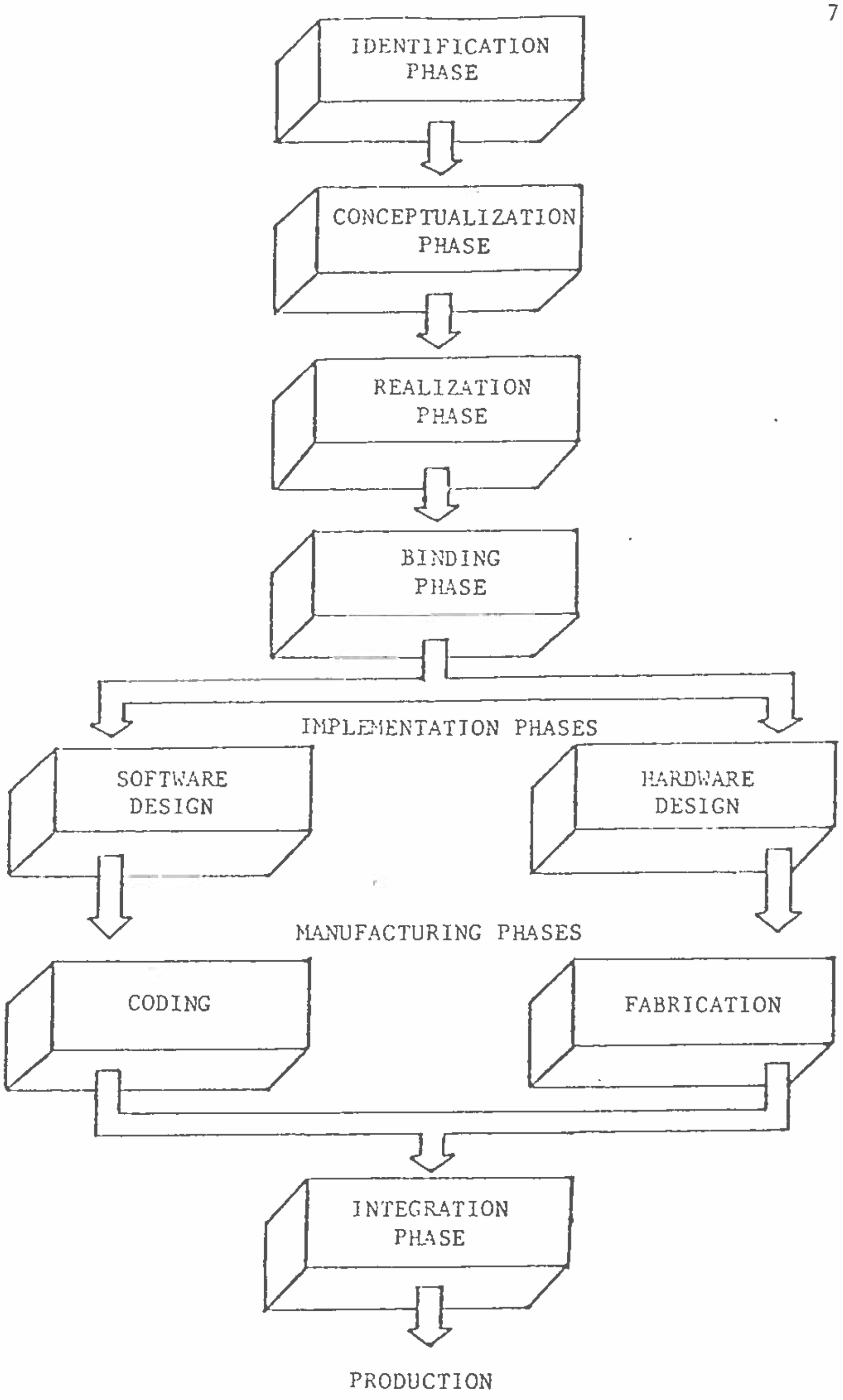

Figure 1: Total System Development (TSD) Framework

design from those decisions which are technolgy dependent. The criterion used in this document naturally separates these phases which in many methodologies today are not yet differentiated.

In contrast with the realization phase, BINDING is a heavily technology dependent phase. It recognizes the unique characteristics of hardware and software, but also the hardware/software duality. The role of the binding phase is that of assigning processors to specific hardware and, during this process, to optimize the system based on hardware/software trade-offs analysis. This process may take a variety of forms from vendor selection to choosing custom hardware and software. During the same phase, the characteristics of the communication medium are also bound.

IMPLEMENTATION denotes two phases: SOFTWARE DESIGN and HARDWARE DESIGN. They both deal with the selection of a particular implementation (composition of functions or building blocks) for custom built components. At the present time there exist significant differences between hardware and software design procedures (despite the duality principle), but it is conceivable that in the future the two phases may resemble each other more and more as the two knowledge bases converge.

Because of the software/hardware differences MANUFACTURING denotes two phases corresponding to two technological domains: CODING, which refers to the process of materializing a particular software implementation through the use of some programming language, and FABRICATION, which is the process by which hardware components are built based on solid state and electronics technologies.

The INTEGRATION phase deals with the assembly of the system from component parts and the associated testing procedures. Subsequent to integration the system is installed and put into production.

The remainder of the paper considers the first four phases of the framework and outlines the fundamental steps involved in carrying out the activities related to these front-end phases.

IDENTIFICATION PHASE

Identification is the first phase of the TSD framework. It encompasses activities whose goal is that of establishing the groundwork required to start system development. This phase attempts to identify why a system should be built, what it is supposed to do, how it relates to the production environment, and what the constraints to be met by the system are. The identification phase has a highly informal character (formalization requires first an understanding of the application), is facilitated by a prior knowledge of the application domain, and requires careful considerations of the human factors involved in the process of creating a good communication link between the developer and the customer.

The output of the identification phase is a report detailing, in some organized but informal way, the results of the developer's explorations of the customer's world. It corresponds, in some sense, to the experimental or field data used in disciplines like physics or anthropology. As such, the viewpoint tends to be local, the level of abstraction is low, and the correlations are few. The completeness and accuracy of the data are the key issues facing the writer of the report.

The fundamental steps involved in the identification phase are listed below:

(a) Exploration

(b) Report generation

(c) Report evaluation

The exploration step is meant to accomplish two highly interrelated tasks: to create a communication link between developer and customer and to establish the role played by the proposed system. The report generation step must produce a complete and unambiguous description of the issues revealed during the exploration step. The report evaluation is an analytical step which attempts to determine the successful completion of the identification phase. Rejection of the report indicates the need to revisit previous steps.

Typically, the identification phase of most system development methodologies also includes steps which, while technically non-essential, have great practical significance. They tend to be oriented toward the generation of preliminary studies regarding anticipated benefits or market value, cost estimation, development time and needed resources, potential environmental, social and legal implications, etc. Such investigative work is important in assisting decision making processes both in the developer and customer organizations. Based on these studies unwise system development may be stopped before committing too much money and resources. However, one must emphasize that the informal nature of the identification phase can hardly support a formal study of complicated issues such as those mentioned above. More definitive studies become possible only in the conceptualization phase.

An adequate treatment of the identification phase is hard to find in the current literature. The identification phase is most often neglected in favor of other pahses which exhibit more formal qualities and have been studied in some depth. A successful case study by Heminger(1) is recommended to the reader interested in techniques supporting the identification phase. The approach, as expected, is application domain dependent. The case study involves an existing working system (flight software for the NAVY's A-7 aircraft) which was subjected to the identification phase. Nevertheless, the techniques could also be used in the more common situation when a new system needs to be built. In the area of business systems, Taggart and Tharp(2) survey a number of techniques for "information requirements analysis" which deal with the identification phase. However, most of them go beyond the pure identification activities into conceptualization and even further.

CONCEPTUALIZATION PHASE

The information obtained in the identification phase is formalized during the conceptualization phase. The result is a conceptual model which formally defines all aspects of the application domain to be supported by the system under development. The conceptual model is intended to play a multitude of roles: it defines the system's functionality, boundaries, and interfaces to the application environment; it forms the basis for analytical studies of a technical (e.g., feasibility) or managerial (e.g., cost prediction) type; it establishes a firm foundation for all contractual agreements; and it helps in understanding the customer's problem representing the most important communication link between developer and customer. Informal models can rarely satisfy all these needs. The conceptual model ought to be very precisely formulated, to provide concise means of expression and powerful notation, to be analyzable by humans and/or machines, and to support a variety of levels of abstraction. Occasionally more than one conceptual model may be required for a complete description of the problem.

Jumping directly from identification to realization is an important factor in the failure of many systems being developed today. Informal descriptions of the problems tend to be incomplete, self-contradictory, hard to verify, and ambiguous. Furthermore, many fundamental problems are obscured by the verbosity of the reports and the simplicity of the examples. Such issues can be resolved only by systematic formal modeling and analyses.

The steps comprising the conceptualization phase are as follows:

(a) Formalism selection

(b) Formalism validation

(c) Conceptual model construction

(d) Conceptual model verification

(e) System boundaries selection

The conceptualization phase starts with the selection of a formalism able to represent the problem. The formalism may be newly developed or already available from some source. In either case the selection needs to be validated theoretically or experimentally as being suitable for the task before too much effort is invested in using it. Subsequent to validation, a conceptual model of the problem is built and later made subject to systematic verification by both developer and customer. The verification ought to consider issues such as: proper use of the formalism, lack of contradictions, completeness, and consistency with the identification report. Finally, the conceptual model may be used to select, in agreement with the customer, the boundaries of the system to be developed.

The ability to develop and communicate the conceptual model presupposes the availability of an adequate formalism. It may be selected from among existing formalisms or it may be the result of abstracting from the application domain the key fundamental concepts required to understand and describe the information gathered during the identification phase. Formalisms used for problem conceptualization tend to fall into distinct groups.

Some formalisms are application domain specific. Many of them are powerful enough to reach the status of problem oriented languages such as BIOSSIM(3) which is used in biological research or BDL(4) which is intended for business applications. Other formalisms are only partially processable by machines but have been proven to be valuable tools in the development of conceptual models. A second group is represented by languages such as HOS(5) which, while unrelated to any particular application domain, allow for user defined entities which can later be invoked during the development of the conceptual model. Finally, there are formalisms which are the result of abstracting over large classes of applications systems common features which help one in better understanding the type of processes involved and fundamental limitations or trade-offs. Examples of such formalisms are formal language and automata based models of communication,(6,7) relational algebra,(8) and database abstractions.(9) These formalisms play a crucial role in the development of appropriate conceptual models for multi-purpose computer systems as well as for systems supporting specific applications whose conceptual models depend heavily on communication, database, etc. For instance, the development of a distributed medical information system may require a conceptual model which includes concepts from medical record keeping but also from communication and database areas.

Despite its importance, many system development methodologies do not even recognize the conceptualization phase and, as such, are guilty of a very serious omission. Furthermore, conceptualization is most often confused with one or another design activity and dealt with as a computational problem instead of being treated as a business, military, or medical problem. Consequently, many formalisms have little to do with the application domain. They are design specification languages misused for a purpose outside their range. More recognition needs to be given to the fact that problem specification languages used as conceptualization tools ought to reflect and incorporate application domain knowledge.

REALIZATION PHASE

System design activities start with the realization phase whose main function is that of generating a technology independent system specification called the processing model. Technology independence is achieved by maintaining the design activities at a very high level of abstraction. The processing model is conceived as the result of a highly interactive sequence of synthetic and analytic steps. The synthetic steps represent design activities aimed either at altering an unsatisfactory design solution or at adding new details to the model. The analytic steps establish, on one hand, the logical correctness of the design and, on the other hand, conformity with given qualitative and quantitative constraints which may have originated with the customer or developer, or may represent generally accepted rules of the trade.

The motivation behind developing the processing model is two-fold. First, the model describes top level system design from the point of view of functionality and behavior at an abstract enough level so as to be independent of detailed technical considerations. Secondly, the processing model is envisioned to become the basis for both hardware/software partitioning and hardware selection, which take place in the binding phase. As such, in addition to functionality, the processing model ought also to include performance related information such as data volume, message rates, data access patterns, etc. This information could originate, in part, in the conceptual model and among customer provided constraints while the rest would have to be generated during the performance checks to which the model is subjected in the realization phase.

The fundamental steps involved in developing the processing model are listed below.

(a) Elaboration

(b) Logical verification

(c) Partitioning

(d) Performance check

(e) Qualitative check

When considering the general case of a potentially distributed system, the scenario defined by the steps above is as follows. The elaboration step starts by establishing the set of events relevant to the particular level of abstraction and groups them into parallel processes. The resulting specification is then subject to logical verification, which attempts to determine its consistency with respect to the level above and with respect to the conceptual model. During partitioning, processors are approximated in terms of the processes assigned to them. This partitioning of the various processes among hypothesized processors is based on estimating relative communication costs within the same processor and across processors. Subsequently, performance and qualitative checks are carried out in order to evaluate the design's conformity to the set of realization pertinent constraints.

Because of the size of the systems and the complexity of the decisions involved in the realization phase, the same steps would have to be repeated for each successive level of abstraction employed in the processing model. Consequently, as the need for further refinement and distribution is determined, one or more processors on level n may be further refined by employing the same procedure, thus generating level (n + 1). As an example, processors on level one may represent network nodes while on level two they may abstract single machines. The level of detail at which the refinement and distribution activities (i.e., the realization phase) are curtailed is a function of the ability to assess compliance with all relevant constraints; however, failures in being able to carry out the binding may require later resumption of this phase.

The elaboration step as well as all the subsequent steps involved in the realization phase require the support of system design languages capable of describing processes and their interactions in a precise manner and of generating specifications suitable for purposes of analysis. Considerable effort has been and continues to be expended towards the development of such specification languages. Some of the languages mentioned under conceptualization are actually better suited for this purpose, HOS(5) for instance. Others, such as PSL(10), lack the full range of capabilities suggested here but function well for a restricted set of problems. Most languages based on graphic descriptions and informal semantics such as those reviewed by Peters and Tripp(11) are adequate only for the design of systems without concurrency. Highly formalized specification languages such as SRI modules(12), DREAM(13), or SARA(14) have better chances of success and encourage mechanization of some of the analytic steps. Other serious contenders are parallel programming languages such as concurrent PASCAL(15) which have been used for specification purposes. Unfortunately, parallel languages have not been designed with the intent of supporting the specification activity and, as such, tend to be somewhat cumbersome to use and too restrictive.

BINDING PHASE

Determination of the hardware/software boundary and selection of particular hardware components of the system are the goals of the binding phase. All decisions taken during this phase are controlled by market availability, technological state-of-the-art, and manufacturing capabilities. The constraints imposed by these factors are manifest not only in terms of sheer availability of hardware having certain given qualities, but also in the cost of various existing components. Binding and all subsequent phases are heavily technology dependent and are expected to change as long as technological progress takes place. As the cost structure of alternate options varies with time, the binding techniques may change substantially. Such a phenomenon is actually observable today in a trend away from the large mainframes and toward distributed systems.

The complexity of binding has long been recognized(16,17) even for the somewhat simpler situation when a single computer system is to be purchased. The difficulties stem from the growing multitude of feasible alternatives, the virtual impossibility to objectively compare highly dissimilar systems, the lack of sophistication of the performance measurements, the complexity of evaluating cost and human factors, the subjectivity of the selection team, its role, power, and composition, etc. As one considers distributed systems and custom-made components, the complexity grows many fold with the technical expertise required of the selection team increasing accordingly.

A comprehensive and cohesive approach to binding has been proposed by Roman.(18) His strategy takes advantage of the top-down organization of the processing model in order to minimize not only the objective function but also the effort required to carry out the binding. Furthermore, various binding options are considered on a priority basis. For instance, the following sample options are ordered as listed: commercially available computer system, customized computer (changes in microcode or operating system), custom-made machine, and custom-made (VLSI) device. A lower priority option is introduced only when increased levels of distribution cannot satisfy the performance needs given the current option or become prohibitive in terms of cost. Furthermore, given a feasible option, an attempt is made to minimize cost and number of components (when cost differences are small).

The first step of the binding phase (see Figure 2) identifies the minimum distribution requirements, i.e., what processors are prohibited from being implemented on the same (physical) machine. This is done by reviewing pertinent customer imposed constraints (e.g., survivability, fault tolerance, etc.) and by recognizing performance constraints which obviously require distribution. Next, tightly coupled processors of comparable structure, behavior, and performance are grouped into clusters. They represent areas of hardware regularity which ought to result in uniform binding, i.e., each processor in the cluster is bound to the same type of physical machine.

Based upon the characteristics of its members, a binding option is selected for each cluster. Every time an option is considered, selection rules for determining the set of viable candidates are established. They help in reducing the number of candidates to those that have a better than average chance of success and satisfy all the basic binding constraints. The rules have to be simple to apply, making initial candidate selection trivial.

Once the candidate set has been chosen, each candidate is evaluated with respect to the cluster. The evaluation method depends on the current option. Its role is that of determining how good a match the candidate is for the particular cluster. Each processor is mapped into a physical machine. The mapping is verified and later evaluated with respect to performance and other qualities that the system must exhibit. Candidates that are too powerful or not powerful enough are rejected. A good match between some candidate and the cluster results in a successful binding which, along with other successful bindings, is saved for future consideration. Failure to bind at least one of the candidates results in taking one of the following three courses of action: further distribution of the processors in the cluster, selection of a different binding option, or restructuring of one or more clusters.

The strategy above results in a number of possible configurations. Not all of them are viable due to incompatibilities that may occur when communicating clusters are separately bound. As such, all incongruent configurations need to be discarded before continuing with the binding of the communication links.

At this point a final validation of each configuration takes place, and all acceptable configurations receive a cost-value coefficient. The cost estimation ought to include, in each case, hardware costs, software costs (two equally priced machines but having different software result in different software development costs), and communication costs. The final choice is determined by contrasting the cost of the various alternatives (development cost plus any other cost factors) and by considering any other issues perceived to be relevant to the decision in question. The binding phase, however, cannot be completed until precise and complete software and hardware requirements are generated, thus assuring the stability of both types of components during the implementation and manufacturing phases.

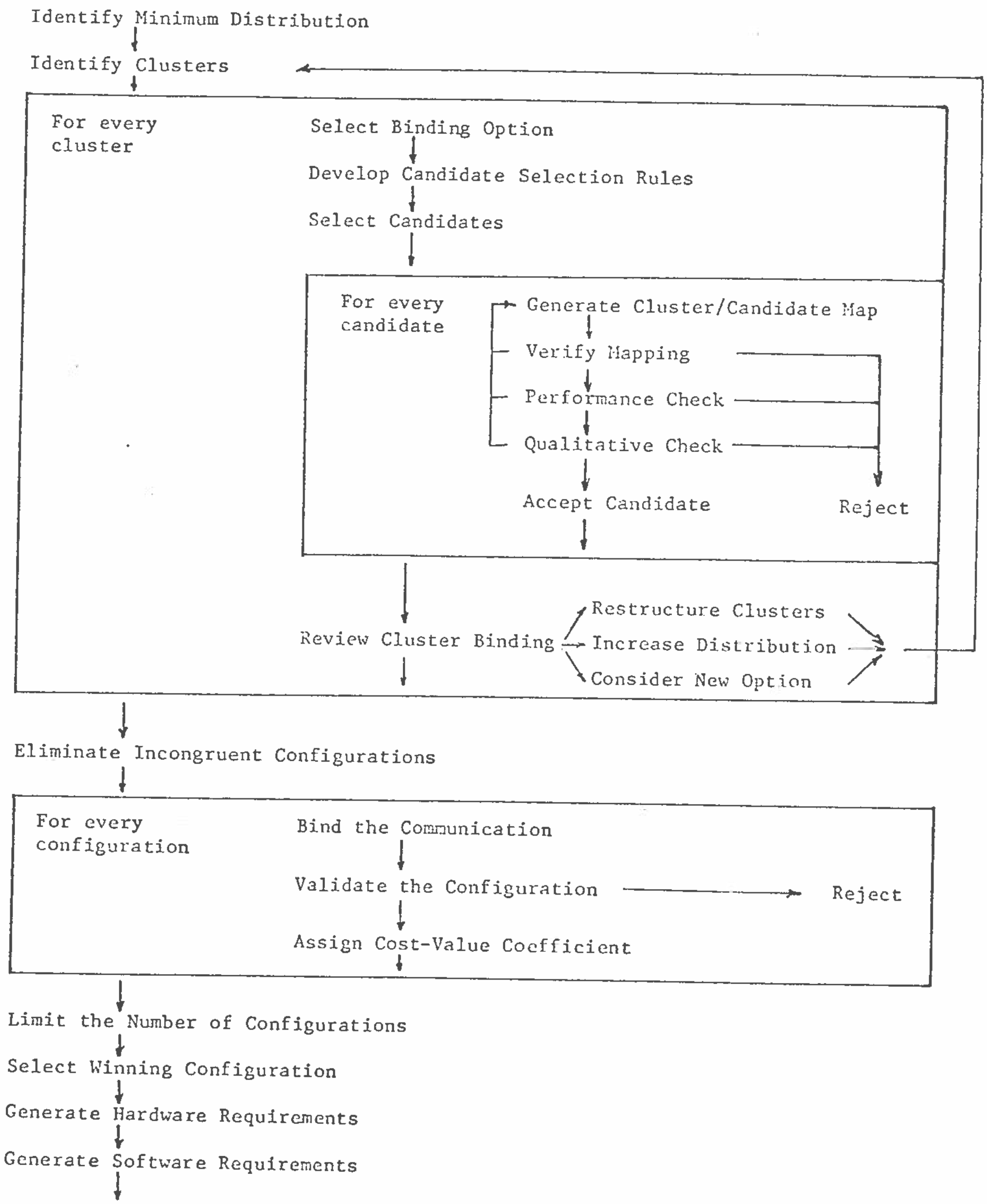
Identify Minimum Distribution
Identify Clusters
For every cluster
Select Binding Option
Develop Candidate Selection Rules
Select Candidates
For every candidate
Generate Cluster/Candidate Map
Verify Mapping
Performance Check
Qualitative Check
Accept Candidate
Reject
Review Cluster Binding
Restructure Clusters
Increase Distribution
Consider New Option
Eliminate Incongruent Configurations
For every configuration
Bind the Communication
Validate the Configuration
Reject
Assign Cost-Value Coefficient
Limit the Number of Configurations
Select Winning Configuration
Generate Hardware Requirements
Generate Software Requirements

Figure 2: Binding Phase

POST-BINDING PHASES

Because the phases following binding have already received adequate coverage elsewhere, the implementation, manufacturing and integration phases are not reviewed here. The interested reader, however, is directed to Wegner(19) for an evaluation of the state-of-the-art in software technology, to Bell et. al., (20) for a treatment of hardware systems design, and to Mead and Conway(21) for issues related to VLSI design.

CONCLUSIONS

The Total System Development (TSD) methodological framework emerges from the need to establish a unified view of the various activities involved in the development of systems that span across hardware/software boundaries. It is an attempt to merge concepts, techniques, and methodologies under a cohesive philosophical and technical framework, thus enabling one to more clearly relate approaches that address disjoint facets of the system development cycle.

TSD is not a specific distributed systems design methodology but a methodolgical framework supporting methodology research, development, and evaluation. TSD provides the researcher with the foundation for a systematic approach to methodology development and with a new conceptual tool. Its high level of abstraction allows for a clearer identification of basic methodological issues transcending individual methods. The policy maker and the manager of research organizations may employ TSD in devising a fixed point of reference against which research planning, trends, and accomplishments may be judged. Finally, the production manager may choose to use TSD as an aid in methodology analysis, selection, and enhancement.

REFERENCES

1. Heninger, K. L. "Specifying Software Requirements for Complex Systems: New Techniques and their Applications." Proceedings of the Conference on Specifications of Reliable Software, April 1979.

2. Taggart, W. M., Jr. and Tharp, M. O. "A Survey of Information Requirements Analysis Techniques." Comp. Surv. 9, No. 4, December 1977.

3. Roman, G.-C. and Garfinkel, D. "BIOSSIM - A Structured Machine-Independent Biological Simulation Language." Comp. and Biomed. Res. 11, pp. 3-15, 1978.

4. Hammer, M., Howe, W. G., Kruskal, V. J. and Wladawsky, I. "A Very High Level Programming Language for Data Processing Applications." CACM 20, No. 11, pp. 832-840, November 1977.

5. Hamilton, M. and Zeldin, S. "Higher Order Software - A Methodology for Defining Software." IEEE Trans. on Soft. Eng. SE-3, No., 1, pp. 9-32, March 1976.

6. Shaw, A. "Software Descriptions with Flow Expressions." IEEE Trans. on Soft. Eng. SE-4, No. 3, pp. 242-254, May 1978.

7. Reif, J. H. "Analysis of Communicating Processes." Technical Report TR 30, Computer Science Department, University of Rochester, Rochester, N. Y. 14627.

8. Codd, E. F. "Relational Completeness of Database Sublanguages." Data Base Systems, R. Rustin editor, Prentice-Hall, 1971.

9. Smith, J. M. and Smith, D. C. P. "Database Abstractions: Aggregation and Generalization." ACM Trans. on Database Sys. 2, No. 2, pp. 105-133, June 1977.

10. 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.

11. Peters, L. J. and Tripp, L. L. "Comparing Software Design Methodologies." DATAMATION, pp. 89-94, November 1977.

12. Robinson, L., Levitt, K. N., Neumann, P. G. and Saxena, A. R. "A Formal Methodology for the Design of Operating System Software." Current Trends in Programming Methodology, edited by R. Yeh, Vol. 1, Ch. 3, pp. 51-110, Prentice-Hall, 1977.

13. Riddle, W. E., Wiledon, J. C., Sayler, J. H., Segal, A. R. and Stavely, A. M. "Behavior Modeling During Software Design." IEEE Trans. on Soft. Eng. SE-4, pp. 283-292, July 1978.

14. Campos, I. M. and Estrin, G. "Concurrent Software System Design Supported by SARA at the Age of One.: Proceedings of the 3rd International Conference on Software Engineering, pp. 230-242, May 1978.

15. Hansen, B. The Architecture of Concurrent Programs, Prentice-Hall 1977.

16. Joslin, E. O. Computer Selection. Addison-Wesley, 1968.

17. Timmreck, E. M. "Computer Selection Methodology." Comp. Surveys 5, No. 4, pp. 199-222, 1973.

18. Roman, G.-C. "Total System Development Framework." Technical Report WUCS-79-10, Department of Computer Science, Washington University, St. Louis, Missouri 63130, 1979.

19. Wegner, P. (editor) Research Directions in Software Technology, MIT Press, 1979.

20. Bell, C. G., Mudge, J. C., and McNamara, J. E. Computer Engineering: A DEC View of Hardware Systems Design. Digital Press, Digital Equipment Corporation, 1978.

21. Mead, C. A. and Conway, L. A. Introduction to VLSI Systems. Addison-Wesley, 1980.