Report Number: WUCS-82-4

1982-02-01

The Total System Design (TSD) Methodology from Problem Definition to Hardware/Software Requirements

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

This paper presents an informal description of a methodology called the Total System Design (TSD) Methodology. It consists of two phases that deal with the transformation of a system requirements specification to a processing model (in the architecture design phase) and the subsequent generation of hardware and software requirements (in the binding phase). The proposed design strategy is based primarily on an extension of the concept of abstract machine hierarchies to distributed systems, while the binding strategy could be viewed as a "most constrained first" policy.

THE TOTAL SYSTEM DESIGN (TSD) METHODOLOGY FROM PROBLEM DEFINITION TO HARDWARE/SOFTWARE REQUIREMENTS

GRUIA-CATALIN ROMAN
MISHELL J. STUCKI
WILL D. GILLETT

WUCS-82-4

DEPARTMENT OF COMPUTER SCIENCE

WASHINGTON UNIVERSITY

SAINT LOUIS, MISSOURI 63130

FEBRUARY, 1982

This work was supported by Rome Air Development Center and by Defense Mapping Agency under contract F30602-80-C-0284.

ABSTRACT

This paper presents an informal description of a methodology called the Total System Design (TSD) Methodology. It consists of two phases that deal with the transformation of a system requirements specification to a processing model (in the architecture design phase) and the subsequent generation of hardware and software requirements (in the binding phase). The proposed design strategy is based primarily on an extension of the concept of abstract machine hierarchies to distributed systems, while the binding strategy could be viewed as a "most constrained first" policy.

Acknowledgements: This work was supported by Rome Air Development Center and by Defense Mapping Agency under contract F30602-80-C-0284.

Keywords: system design methodology, specification languages.

INTRODUCTION

Increased reliance on distributed systems, growing interdependence between hardware and software, stronger emphasis on the use of existing software (due to soaring software development costs), and availability of an ever increasing variety of hardware choices are several factors that have contributed to making the development of the hardware/software requirements the most critical issue faced by the system developers today, aside from the problem definition which must precede all design activities. The system architecture design and system binding are two development phases that have been proposed in [ROMA82a] to deal with the design decisions relating to this issue. [ROMA82a] investigated the nature of the hardware/software partitioning problem, characterized the dynamics of hardware/software trade-offs, and put forth a proposal for a class of design methodologies covering this aspect of system development.

The system architecture design phase deals with the selection of an overall system architecture which accomplishes the intended system functionality and which, under a reasonable set of technological assumptions, meets the performance and other constraints originating with the system requirements. The proposed architecture and all the design decisions taken during this phase form a processing model used as input to the binding phase.

The binding phase, based on the limited degrees of freedom still left open by the system architecture design phase and based on market availability, identifies a particular mix of software and hardware and produces specifications for all needed components. The nature of the specifications, however, may vary from component to component depending on its intended realization (software or hardware) and on the manner in which it is to be obtained (off-the-shelf, through customization, or custom-made). The system design stage is also concerned with the integration of the system components from the point when both the software and the hardware components are available and up to the point when the system is offered for customer acceptance testing.

This paper presents an informal description of a methodology covering these two key phases of system development, the Total System Design (TSD) Methodology. The following three sections provide the reader with informal definition of the type of specifications employed by the TSD Methodology: the system requirements, the processing model, and the hardware/software requirements (formal definitions may be found in [ROMA82c]). The design strategy is first introduced in this paper in tutorial form, but is formalized in the Annex to this paper using the methodology definition approach of [ROMA82b]. (A formal characterization of the relationship between the design strategy and the nature of the specification languages involved is presented in [ROMA82c].) The proposed design strategy is based primarily on an extension of the concept of abstract machine hierarchies to distributed systems, while the binding strategy could be viewed as a "most constrained first" policy. Conclusions, references, and the Annex appear at the end of the paper.

INFORMAL DEFINITION OF SYSTEM REQUIREMENTS

The system requirements are generated in the problem definition stage. They consist of a conceptual model and a set of constraints which together define the acceptability criterion for any proposed system realization: a system is said to meet its requirements if and only if it carries out the functionality described by the conceptual model and satisfies all the constraints present in the system requirements.

The role of the conceptual model is to capture in finite and precise terms the nature of the interaction between the needed system and its environment. In general, the conceptual model must have the ability to describe the relevant environmental states, an abstraction of the states of the system, and the way in which both the environmental and system states change. The approach to describing the states and the state transition rules varies from one specification language to another. The language discussed in [ROMA82d], for instance, employs a set-theoretical notation to describe both the environmental and the system states and uses predicate calculus to define the state transition rules. By contrast, other languages promote operational approaches based on data flow graphs [BELL77], applicative methods [ZAVE81], etc.

Furthermore, some languages make implicit assumptions about either or both the nature of the states and of the state transition rules; the loss in generality is motivated by increased specificity in the handling of a particular application area. As an example, a system that responds to stimuli from the environment in a manner which is independent of the history of previous stimuli and responses may be easily described in a language which equates the state of the environment with the current stimulus, has no ability to describe system states, and is capable of defining a mapping from the set of stimuli to the set of responses. Yet another example could be used to illustrate the fact that there is also great variability in the way state transitions may be described: in a biomedical simulation system a new state is generated as a result of the integration of a set of differential equations.

Increases in the ability to formally define the desired functionality are not accompanied by commensurable advances in the definition of system constraints. There are four important reasons contributing to this. First, there is a great diversity of types of constraints (e.g., response time, space, reliability, cost, schedule, weight, power, etc.). Second, some of them are related to possible design solutions which are not yet formally stated at the time the system requirements are being conceived. Furthermore, their relevance differs at different points in the design. Third, many constraints (e.g., maintainability) are not formalizable given current state-of-the-art. Finally, not all constraints are explicit. For instance, the designer is expected to follow generally accepted rules of the trade in designing a system without having them explicitly stated.

INFORMAL DEFINITION OF PROCESSING MODEL

The methodology put forth here treats systems as being describable by a hierarchy of related design specifications where the specification at one level reveals a design solution for some problem which is formally defined within the level above. The processing model reflects this view by assuming a similar structure: a total order over a finite set of design specifications. The total ordering is not really necessary but has been adopted in order to simplify the presentation of both the processing model and the system design strategy. Furthermore, the extrapolation to an upside-down tree (a tree in which the level number of each node is defined as the longest distance from a leaf rather than root) is trivial.

By definition, each design specification is viewed as corresponding to a subsystem in the overall system. The support relation between subsystems is explained later and is formally defined in [ROMA82c]. The remainder of this section focuses on the informal definition of the design specifications.

Regardless of its position in the hierarchy, each design specification consists of same six components:

- PROCESS STRUCTURE
  - -- network topology in terms of processes and links
  - -- definition of system and external processes
  - -- definition of links
  - -- definition of link communication protocols

- PROCESSOR STRUCTURE
  - -- network topology in terms of processors and interconnections
  - -- definition of processors
  - -- definition of processor interconnections
  - -- definition of interconnection communication protocols

- PROCESS/PROCESSOR ALLOCATION
  - -- allocation and reallocation rule

- PERFORMANCE SPECIFICATIONS
  - -- performance requirements of processes and links
  - -- performance requirements of processors and interconnections
  - -- performance characteristics of processes and links
  - -- performance characteristics of processors and interconnections
  - -- performance models

- BINDING OPTIONS
  - -- set of feasible realizations of the process and processor structures
  - -- set of binding constraints

- CONSTRAINTS.

The PROCESS STRUCTURE describes the subsystem functionality by means of a network of communicating processes interconnected via links. Each link provides a logical connection between two or more processes. The

message traffic on each link, however, behaves in accordance with a communication protocol specified by the designer. In the top level subsystem the processes may correspond to successive transformations of the input data in a data processing system or to query processing in a database system. At other levels the process structure may be describing operating system capabilities. In all cases, however, the description is independent of the way in which the processes are distributed within a realization of the system and of the manner in which they may be implemented.

The PROCESSOR STRUCTURE, in conjunction with the process/processor allocation explained below, is an abstraction of all the subsequent levels in the hierarchy. In its simplest form, the distinction between the process and the processor structures is like the distinction between an application program and the operating-system/hardware combination that enables it to execute. Furthermore, processors are assumed to correspond to separate distributed collections of system components. In other words, given the final system realization and any one of the processor structures present in the hierarchy, one should be able to uniquely partition all system components into equivalence classes and to establish a meaningful one-to-one correspondence between these equivalence classes and the entities (processors and interconnections) of the chosen processor structure.

The PROCESS/PROCESSOR ALLOCATION captures the distribution of the processes among the available processors. In its simplest form, the allocation may be static, i.e., does not change during the execution of the system. In such cases, all processes are partitioned among the available processors with the links being partitioned accordingly between processors and their interconnections. Reliability, workload balancing, and other design considerations, however, often require dynamic changes in the allocation of processes and links among the available processors and interconnections. (Note: An additional degree of complexity may be noticed in systems which permit a process to be mapped simultaneously on several processors. This occurs, for instance, when the code associated with a particular realization of some process and the execution of the corresponding instructions are the responsibilities of two separate processors. The definitions from [ROMA82c] do not rule out such cases.) The separation of the allocation/reallocation issue from the functional details of the process structure has the potential to significantly reduce the complexity of analyzing both the individual subsystems and their relationships.

The PERFORMANCE SPECIFICATIONS deal with the performance attributes of the system and with the models used to relate the performance attributes to the selected system architecture and to each other. A performance attribute may be associated with either the process or the processor structure and represents either a performance requirement originating with some performance constraint or a performance characteristic that has been established to be true, i.e., it was validated. Performance requirements (i.e., constraints) are assumed to propagate top-down from the process structure to the processor structure, and from one subsystem to the next. The performance characteristics, however, propagate bottom-up; only when the exact characteristics of the

processor structure are known may one deduce with certainty the characteristics of the process structure. Moreover, an acceptable design demands that all performance characteristics imply the satisfiability of the corresponding performance requirements. In this context, performance models assume a dual role. First, they assist one in determining the performance requirements of the processor structure from those of the process structure. Second, they propagate the performance characteristics of the processor over the process structure.

The BINDING OPTIONS represent a non-empty (possibly infinite) set of system realizations that are still feasible at a given point in the design process. This set is very large at the start of the system architecture design phase and, through successive design refinements, is systematically reduced to a manageable size upon entering the binding phase. Because at no point in time it is possible to enumerate the members of this set, the designer specifies it indirectily via a distinguished category of constraints called binding constraints. They are formulated during the design process as a result of explicit design choices (which rule other out alternatives) and due to conclusions drawn from various design studies and analysis of the stated system requirements, available technology, anticipated operating environment, etc.

Finally, the CONSTRAINTS that appear in each design specification are inherited from the original system requirements and carried along throughout the entire design. Different constraints, however, affect the design at different points in time. Some represent the origin of the performance requirements while others may affect certain aspects of binding. It is the designer who brings into consideration the appropriate constraints at the right place in the design.

INFORMAL DEFINITION OF HARDWARE/SOFTWARE REQUIREMENTS

The hierarchy of design specifications present in the processing model is mapped during the binding phase into off-the-shelf, customized, and custom-made software and hardware. Separate software requirements specifications are generated for each subsystem. In addition, hardware requirements specifications are produced for the lowest level subsystem in the processing model hierarchy. While there is great variability in the way in which both software and hardware requirements need to be specified, they generally include the following:

- a specification of the functional and performance requirements of the hardware or the software (present, for the most part, in the respective design specification);

- a specification of all relevant interfaces (between subsystems, between components residing on different machines, between components developed separately, etc.);

- a mapping from parts of the proposed design onto existing hardware or software;

- a list of existing hardware or software to be used.

A simple inventory system may be used to illustrate the nature of the hardware/software requirements:

SOFTWARE REQUIREMENTS.

LEVEL 1 (Application Program).
- functionality given by an inventory control language whose syntax and semantics have been fully specified; no performance constraints;
- user interface via the inventory command language; access to the database defined by the INGRES user manual; the implementation language C;
- all database manipulations are relegated to INGRES;
- off-the-shelf software to be used: INGRES -- a relational database package.

LEVEL 2 (Operating System).
- functionality given by the UNIX user manual;
- user interface via UNIX standard commands; UNIX version supported by the PDP 11/40 machine and compatible with INGRES;
- no changes or enhancements to the UNIX operating system permitted;
- off-the-shelf software to be used: UNIX operating system.

HARDWARE REQUIREMENTS.

LEVEL 2 (Hardware Configuration).
- the hardware configuration consists of a PDP 11/40 with 64k bytes of main memory, a VT52 compatible CRT terminal, a 1200 baud printer, and two disk drives for 2.5 megabytes disk cartridges;
- one serial port for interfacing the crt and the processor; one parallel port for the printer; two direct memory access ports for the disk drives;
- the mapping of functions to components is trivial in this case;
- specific printer, crt, and disk drives could be listed here.

The relation between the processing model and the hardware/software requirements is further analyzed in the next section.

METHODOLOGY OUTLINE

GENERAL REMARKS

This section discusses a proposal for a class of TSD Methodologies focused on the design activities leading to the identification of the hardware and software components in distributed systems. The presentation starts with a statement of objectives. It is followed by the design strategy to carry out the system architecture design phase. The strategy covers both the design of the individual subsystems and the sequencing of design activities between subsystems. Finally, the discussion turns to a systematic way of accomplishing the task associated with the binding phase.

The principal goal of the proposed methodology is to increase the quality and productivity of the design of large distributed systems. Reaching this goal, however, places the following demands on the nature of the TSD Methodologies:

- an ability to explore in a systematic manner a large design space by separating system level issues from those involved in the design of hardware and software and by placing the selection of hardware and software (i.e., hardware/software trade-offs) on a more rational base than it has been done in the past;

- a structuring of the design process in a way which assures a great degree of control over design complexity and promotes incremental verification of both the functional and performance aspects of successive system design refinements;

- a strategy which is based on general design principles rather than the peculiarities of a specific class of applications but which, at the same time, is adaptable, i.e., may be tuned to a given application.

It is our contention that the design strategy we have selected indeed has all these attributes. The remainder of this section describes the proposed strategy. The ultimate validation, however, has to come from empirical studies in which the methodologies are applied to specific problems. Furthermore, specification languages and an appropriate assortment of techniques need to be developed in order to provide the designer with a computer aided environment that would assure the high productivity to which the TSD Methodologies aspire.

Before presenting the methodology it is necessary to point out that, for the sake of clarity, certain simplifying assumptions are being made throughout the paper:

- design backtracking due to errors receives limited coverage;

- parallel development of portions of the design by different teams on the project is ignored despite the great opportunities for concurrency within a project;

- most project management activities are omitted;

- system integration is not discussed.

While they do not alter the overall flavor of the strategy, they may make the methodology appear somewhat inflexible. We hope that by pointing them out early in the presentation, the reader will have no trouble in discerning the difference between the overall design strategy and the artifacts of the simplifying assumptions.

SINGLE SUBSYSTEM DESIGN STRATEGY

As indicated earlier, systems are described in terms of a hierarchy of design specifications. They force a structuring of the system in terms of a number of subsystems, each supporting the subsystem above. The methodology requires the design of individual subsystems to proceed top-down. Within the general context of top-down design, however, several related activities are interleaved. These design activities are outlined below.

Successive and concurrent refinement of both the process and the processor structures. The fact that a given system function may be decomposed in more than one way is well-known. This design freedom is not a menace, as seen by some (e.g., [BERG81]), but rather a degree of flexibility essential to good design. The selection between alternate decompositions is not intrinsic to the decomposition itself, but depends upon the designer's objectives (maintainability, clean abstraction, simple interfaces, reliability, etc.). Among them, the availability of certain (existing or postulated) means of support may also affect the functional decomposition. The case when the process structure is affected by earlier choices of the processor structure is illustrated by the way in which the solution for a certain computational problem may take different forms if one assumes the use of a high speed minicomputer with or without an attached array processor. The converse situation (which occurs frequently in the data processing field) is where the needed hardware is selected based on the result of a functional decomposition of the application problem, where the decomposition is guided by some modularization principle.

Due to the interdependence between the selection of the process structure and of the processor structure, TSD Methodologies emphasize the concurrent refinement of both structures. While accommodating the special cases where the peculiarities of the application force one structure or the other to be dominant, this approach offers the system designer the added flexibility required by an unbiased treatment of the hardware/software trade-offs problem. Furthermore, the balance is allowed to shift in one direction or another, not due to personal prejudices, but due to constraints that affect the range of acceptable system realizations.

Top-down propagation of performance requirements. Fundamental to the conception of the TSD Methodologies is the assumption that performance constraints direct to a large extent the designer's activities. Performance requirements recognized at the top level of a design specification propagate from one level to the next through the assumptions the designer makes at one level about the characteristics of the next. The assumptions later become requirements and the cycle continues. In order for the designer to make reasonable assumptions, two things are needed: past experience and adequate performance models that relate the presumed performance characteristics of entities of some level and the performance requirements placed over the particular level of the design specification. The nature of the performance models has to change according to the level of functional detail. When the level of abstraction is high, the models are less detailed, less accurate, and also less costly than when lower levels of the specification are reached. The scheme has two advantages. On one hand, it allows performance considerations to influence design decisions early on. On the other hand, it holds the promise that this may be achieved in a cost effective manner. (This idea has received some endorsement in recent publications [KUMA80, SANG79].)

Bottom-up propagation of performance characteristics. While the performance requirements flow top-down, the validated performance characteristics (once available) propagate in the opposite direction [BOOT80]. The use of the performance data is important in making immediate readjustments of the subsystem design and establishes the accuracy of the assumptions that were made and the feasibility of the proposed design. (The way in which the performance characteristics become available is discussed later.)

Binding constraints accumulation. The hardware/software trade-offs dynamics manifest themselves during the system design stage as a gradual narrowing down of the range of feasible realizations, i.e., binding options. This takes effect through a growth in the set of recognized binding constraints. The set, originally inherited from the subsystem above, is augmented from several sources. First, each design decision taken (e.g., successive refinements, allocation, etc.) rules out all realizations which have adopted different approaches. Second, design studies that look ahead to low level but potentially difficult components of the system also affect the directions the designer is willing to consider. If, for instance, there were no totally distributed concurrency coordination algorithms, a database design based on their potential availability would have to be discarded. Third, inference studies may suggest that the use of some technological alternatives may be unfeasible (due to their impact on other aspects of the system or on its operation environment, etc.) or, although feasible, not recommended (due to anticipated technological trends, for instance). Finally, the availability of certain software or hardware may dictate a design solution which takes advantage of such off-the-shelf components in order to reduce development costs.

Systematic error detection. Error detection is supported via a number of checks placed at various critical points in the sequence of design activities within the tasks/subtasks and in the tasks/subtasks review sections of the methodology specification. They involve consistency checks between adjacent levels of a design specification and between related components of the specification (e.g., process/processor allocation versus the process and the processor structures). The checks also include logical verification between the design specification of one subsystem and its requirements which, in general, are established by the specification of the subsystem above, if any. For the top subsystem in the hierarchy, however, the subsystem requirements are the same as the system requirements. This issue is considered again in the discussion of the subsystems' design dependencies which follows.

OVERALL SYSTEM DESIGN STRATEGY

The structuring of the system design in terms of the proposed hierarchy of design specifications, is motivated by the desire to control complexity through a systematic and strict separation of concerns. The idea originates in part with the already common concepts of virtual machine and stratified design (layers of virtual machines [ROBI77]). When using a programming language, the designer is not in the least concerned with the implementation details of the language (if the language is properly designed). Similarly, when working at one level of a stratified design the designer deals only with the semantics of the operations available at that point and not with their possible realizations. The TSD Methodologies attempt to exploit this approach in the context of distributed systems by adapting it accordingly.

The designer starts from the system requirements and, through successive refinements of the process and processor structures, defines both the way in which the functionality specified in the conceptual model is implemented and the support needs for such an implementation (e.g., message exchange capability, process reallocation due to failures, storage management, etc.). The top design specification is said to describe the application subsystem, due to the nature of its functionality which is directly relevant to the application at hand. All subsequent specifications are said to describe support subsystems.

As already stated, the construction of each design specification takes place in a top-down manner. However, it is often the case that, prior to completing the specification, the support needs required by the process structure may become clear. In such cases, the generation of the current design specification may be temporarily suspended and the design of the supporting subsystem may proceed. Despite the fact that the strict top-down design strategy could be followed, the designer may chose to move to the next subsystem in the hierarchy in order to minimize the risk that some of the assumptions made about the support subsystem may prove to be wrong. However, once the designer decides to move to the subsystem below, the design discipline prescribed by the TSD Methodologies requires one to complete the design of the support subsystem prior to resuming the design of the subsystem above. This reduces thrashing between subsystems and

enables the designer to make use of the performance characteristics of the support subsystem in the adjustment and completion of the specification for the subsystem being supported.

The diagram that follows depicts the result of applying this strategy to the design of a computer graphics system. Design activities are represented by groups of slashes. The left most column of slashes corresponds to the design of the top subsystem, i.e., the application subsystem.

```
/ design of
/ graphics
/ language
/
     // design of
     // graphics
     // language
     // interpreter
     //
          /// design of
          /// graphics
          /// and
          /// communication
          /// primitives
          ///
               //// design of
               //// graphics
               //// hardware
               ////
          ///
     //
/
```

Aside from the global sequencing of design activities, an understanding of the role that the processing model plays in the system design stage requires the definition of three important concepts. The first one is the notion that the top design specification (the application subsystem) implements the system requirements. The second is the support relation between the design specifications within the processing model hierarchy. Finally, the concept of superficial binding makes the transition to the binding phase.

A design specification is said to implement the system requirements when its process structure is logically equivalent to the functionality captured in the conceptual model. The definition may be actually extended to the individual levels developed during the top-down design of the specification. A given level in the specification implements the system requirements if its process structure is logically equivalent to an abstraction of the conceptual model. These definitions establish the correctness criteria to be employed during the design of the application subsystem and form the foundation for future automated checking of the top design specification.

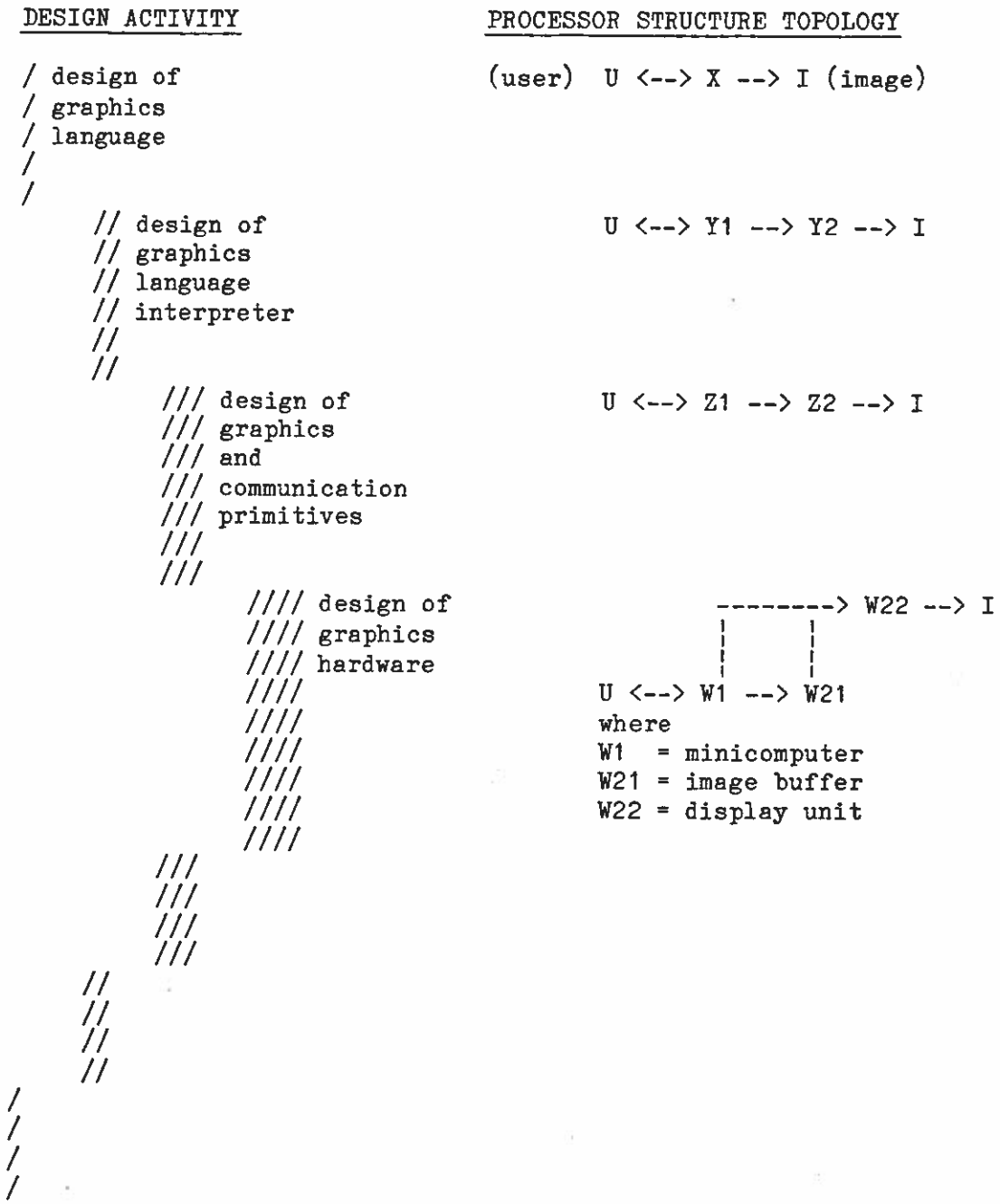
In the most basic terms, "subsystem B supports subsystem A" implies two things about B: (1) it contains the design of functions (unrelated to the application area) which were assumed to be available during the design of subsystem A and (2) it may represent a further refinement of the degree of distribution within the system. With regard to the first role of a support subsystem, it must be pointed out that the ultimate realization of the support relationship may assume a great variety of forms. Consider, for instance, the special case when both subsystems are eventually implemented in software:

- the programs of A may actually invoke the programs of B either as procedure calls or as macros -- such is the case when B realizes the communication protocol assumed by the message sending and receiving commands used by A;

- the programs of A may be interpreted by programs in B -- the availability of a LISP interpreter may be one of the support functions assumed by A;

- the programs of A may by objects (i.e., data) manipulated by programs in B -- programs in B may have the responsibility to monitor and relocate the programs of A in case of equipment failure or for load balancing purposes.

About the potential increase in the degree of distribution from one subsystem to the next, the idea may be rendered easily through the use of the earlier graphics system example.



The third important concept, superficial binding, must be considered in conjunction with the binding strategy.

BINDING STRATEGY

A processing model is considered bound when all its entities are mapped into software and hardware to be obtained (some by purchasing off-the-shelf components, others by customizing available components, and yet others by custom building them). The binding process (also called hardware/software trade-offs) starts in the system architecture design phase and reaches its conclusion in the binding phase. In the first of these two phases the growth in the set of binding constraints (explained earlier in this section) reduces the set of feasible alternatives, thus biasing the design toward certain technological alternatives the designer considers to be most promising. This biasing becomes very strong when the designer choses to structure the system around the potential use of available components; the system entities tentatively associated with such components are said to be superficially bound to them. Note that one entity may be superficially bound to one or more alternatives.

At the point when the binding phase is entered, large parts of the processing model may be superficially bound. The strategy used to accomplish the binding could be called a "most constrained first" approach. The designer starts by identifying binding alternatives for the most constrained areas of the specification. This results in the immediate generation of new binding constraints over the remaining parts of the design which, in turn, eliminates from consideration many fruitless alternatives. Even if one is careful to always limit the investigation to a tractable number of alternatives, the total number of system configurations being evaluated at one time could grow rapidly. If, for instance, one needs to merely select three machines and there are four alternate candidates for each, the total number of system configurations reaches sixty-four. While some configurations may be ruled out by incompatibilities between some candidates associated with areas of the design which are interfaced to each other, the designer needs to weed out many more by employing guidelines such as cost minimization, maintainability, uniformity, etc. Once the entire specification is superficially bound to several alternate configurations, their number needs to be reduced to one by evaluating the weak and the strong points of each of them. Now the system specification is bound.

A last task still to be carried out is the generation of the software and the hardware requirements. They have to include such things as the functionality of various components, performance and other constraints, interface definitions, etc. The exact contents and form of these requirements is hard to formalize due to significant variability between systems. This concludes the informal presentation of the design strategy proposed for the system design stage.

CONCLUSIONS

This paper outlined a methodology (the TSD Methodology) for generating the hardware and software requirements for a distributed computer-based system. Future refinements of the methodology still need to be carried out based on feedback from empirical evaluations of the approach. Attempts to show the feasibility of the TSD Methodology for two application areas (ballistic missile defense and geographic data processing) were successful. They indicated, however, the need to tune the TSD Methodology to the specifics of the application area under consideration. Plans are under way to do this for one narrowly focused application area soon to be selected.

REFERENCES

[BELL77] Bell, T. E., Bixler, D. C. and Dyer, M. E., "An Extendable Approach to Computer-Aided Software Requirements Engineering," IEEE Trans. on Soft. Eng. SE-3, No. 1, pp. 49-60, January 1977.

[BERG81] Bergland, G. D., "A Guided Tour of Program Design Methodologies," Computer 14, No. 10, pp. 13-37, October 1981.

[BOOT80] Booth, T. L. and Wiecek, C. A., "Performance Abstract Data Types as a Tool in Software Performance Analysis and Design," IEEE Trans. on Soft. Eng. SE-6, No. 2, pp. 138-151, March 1980.

[KUMA80] Kumar, B. and Davidson, E. S., "Computer System Design Using a Hierarchical Approach to Performance Evaluation," CACM 23, No. 9, pp. 511-521, September 1980.

[ROBI77] Robinson, L. and Levitt, K. N., "Proof Techniques for Hierarchically Structured Programs," CACM 20, No. 4, pp. 271-404, April 1977.

[ROMA82a] Roman, G.-C., Stucki, M. J., Ball, W. E., and Gillett, W., "The Total System Design (TSD) Framework: An Approach to the Development of Distributed Systems Design Methodologies," Dept. of Computer Science Report WUCS-82-3, Washington Univ., St. Louis, 1982.

[ROMA82b] Roman, G.-C., "On Reducing Ambiguities in Methodology Definitions," Dept. of Computer Science Report WUCS-82-5, Washington Univ., St. Louis, 1982.

[ROMA82c] Roman, G.-C. and Israel, R. K., "A Formal Treatment of Distributed Systems Design," Dept. of Computer Science Report WUCS-82-6, Washington Univ., St. Louis, 1982.

[ROMA82d] Roman, G.-C., "A Rigorous Approach to Building Formal System Requirements," Dept. of Computer Science Report WUCS-82-7, Washington Univ., St. Louis, 1982.

[SANG79] Sanguinetti, J., "A Technique for Integrating Simulation and System Design," Proc. Conf. on Simulation, Measurement and Modeling of Computer Systems, pp. 163-172, August 1979.

[ZAVE81] Zave, P. and Yeh, R. T., "Executable Requirements for Embedded Systems," Proc. 5'th Int. Conf. on Soft. Eng., pp. 295-304, March 1981.

ANNEX

FORMALIZATION OF THE DESIGN STRATEGY

NOTE: The flow of control constructs employed in this section are explained in [ROMA82b]. Tasks, subtasks and procedures should be treated like recursive procedures present in common programming languages such as PL/1 or Pascal with the special provision that their definition appears at the place of their first invocation. Consequently, at the place of definition and first invocation, parameters are defined and initialized at the same time.

Furthermore, all simplifying assumptions explained earlier are reflected in the way the strategy is formalized here.

TASK System-Architecture-Design.

SUBTASK Subsystem-Design(i=1).

Review subsystem requirements (for i=1 the subsystem requirements correspond to the system requirements and the subsystem is called the application subsystem; otherwise, the requirements are given by the processor structure definition and process/processor allocation defined by the subsystem (i-1)).

Set the set of binding constraints to be the same as the binding constraints of subsystem (i-1), unless i=1, in which case the set of binding constraints starts by being empty.

Identify those technological alternatives that may be ruled out as unacceptable and/or limit the set of technological alternatives only to those that appear to be appropriate; formulate constraints which would reflect these considerations; add these constraints to the binding constraints.

IF the subsystem i is already available THEN DONE.

Develop top-level (i.e., level 1) for the design specification of the subsystem i based on some abstraction of the requirements definition; the process structure includes the modelling of the subsystem's environment; the processor structure topology is inherited from the subsystem (i-1), if it exists.

PROCEDURE Subsystem-Refinement(j=2).

{ ==> { Generate the process structure for level j by decomposing or by copying the process structure of level (j-1).
Generate the processor structure for level j by decomposing the processor structure of level (j-1) w.r.t. the needs of the process structure on level j.} |

==> { Generate the processor structure for level j by decomposing or

```
        by copying the processor structure of level (j-1).
        Generate the process structure for level j by decomposing the
        process structure of level (j-1) w.r.t. the capabilities of
        the processor structure on level j.}}

Define process/processor allocation on level j; the allocation may be
static or dynamic and must be consistent with the allocation rule used
by the subsystem (i-1), if it exists.

iteration:
LOOP{ ==> Adjust the specification of the level j. ¦
      ==> Propagate both process and processor performance
          requirements of level (j-1) over the level j and refine the
          performance models used at level(j-1); analytical,
          simulation and empirical techniques may be required to
          support the requirements propagation activity. ¦
      ==> Investigate inference issues related to decisions taken at
          this level and eliminate binding options that are shown to
          be inappropriate. ¦
      ==> Superficially bind aspects of the subsystem to already
          available software/hardware, if such decisions are strongly
          motivated by constraints or design principles. ¦
      ==> Carry out logical and consistency checks for level j. ¦
      ==> Carry out design studies for this or subsequent
          subsystems. ¦
      ==> BREAK. }

IF level j does not refine correctly level (j-1) THEN BACK.

IF level j is not an implementation of some abstraction of the
subsystem requirements THEN BACK.

{ Process and processor structures are not completely refined
  ==> INVOKE Subsystem-Refinement(j+1). ¦

  Processor structure is completely refined
  ==> { INVOKE Subsystem-Design(i+1).

       LOOP{ ==> Propagate the performance characteristics of the
                 subsystem (i+1) to the processor structure of the
                 subsystem i and, subsequently, to the process
                 structure of subsystem i. ¦
             ==> Adjust the specification of subsystem i. ¦
             ==> BREAK.}

       IF process structure is not completely refined THEN
         { PROCEDURE Finish-Refinement(jf=j+1).

           Generate the process structure for level jf by
           decomposing the process structure of level (jf-1) w.r.t.
           the capabilities of the processor structure on level jf
           (same as on level (jf-1)).
```

```
                    iteration:
                    LOOP{ ==> Adjust the specification of the level jf. ¦
                          ==> Propagate process structure performance
                              requirements of level (jf-1) over the level jf
                              through the use of appropriate performance
                              models. ¦
                          ==> Investigate inference issues related to
                              decisions taken at this level and eliminate
                              binding options that are shown to be
                              inappropriate. ¦
                          ==> Superficially bind aspects of the subsystem if
                              such decisions are strongly motivated by
                              constraints or design principles. ¦
                          ==> Carry out logical and consistency checks for
                              level jf. ¦
                          ==> BREAK.}

                    IF level jf does not refine correctly level (jf-1) THEN
                    BACK.

                    IF level jf is not an implementation of some abstraction
                    of the conceptual model THEN BACK.

                    IF process structure is not completely refined THEN
                    INVOKE Finish-Refinement(jf+1).

                  PEND.}}}

      PEND.

    STREVIEW.

    F(Check the self-consistency of the design specification for the
    subsystem i.) ==> BACK.

    F(Perform logical verification of the design specification with respect
    to its functional requirements.) ==> BACK.

    F(Check that all performance constraints placed on subsystem i are met,
    given the characteristics of subsystem (i+1).) ==> BACK.

    F(Determine that all consequences of the proposed design are
    acceptable.) ==> BACK.

    STEND.

    Develop system testing plan.

TREVIEW.

F(Evaluate the system testing plan.) ==> BACK.

TEND.
```

* * *

TASK Binding.

LOOP{ IF the system is superficially bound THEN BREAK.

Identify those design entities and groups of design entities which are not superficially bound and have fewest degrees of freedom with respect to binding.

FOR all such entities and groups DO
{ // { Identify binding candidate selection rules.

Select a tractable set of candidates.

Establish the mapping between the candidates and the related design entities.}}

Define the compatibility relation between the candidates associated with various parts of the design.

IF Compatibility problems are found THEN BACK.

Keep a reduced list of compatible alternatives based on various guidelines such as cost minimization, uniformity, flexibility, interface complexity, etc. }

Evaluate the possible system configurations and reduce their number to one.

{ Generate software requirements including: functionality; explicit statements with regard to both constraints and degrees of freedom; the specifications of the interfaces between the components of each subsystem, between subsystems, and with the hardware; and the off-the-shelf and customized software to which some of the components are bound. //

Generate hardware requirements including: functionality; explicit statements with regard to both constraints and degrees of freedom; the specifications of the interfaces between the hardware components and with some of the software; and the off-the-shelf and customized hardware to which some of the components are bound. }

Develop integration plan.

TREVIEW.

F(Check the self-consistency of the software requirements.) ==> BACK.

F(Check the self-consistency of the hardware requirements.) ==> BACK.

F(Check consistency between hardware and software requirements.) ==> BACK.

F(Verify the functional aspects of the hardware/software requirements against the processing model.) ==> BACK.

F(Check that all performance constraints placed on the system are met, given the characteristics of the hardware and of the software.) ==> BACK.

F(Determine that all consequences of the proposed hardware/software selection are acceptable.) ==> BACK.

F(Evaluate the integration plan.) ==> BACK.

TEND.

* * *