A Synthetic Approach to the Design of Information-Systems Software

Roberto R. Kampfner

School of Management and Computer and Information Science, University of Michigan—Dearborn

This paper presents an approach to the design of information-systems software in which alternative designs can be created, as necessary, until specified requirements are met and specific objectives achieved. This approach takes advantage of, and in fact complements, the abstraction process that characterizes the abstraction-synthesis methodology of information-systems development. A broad concept of function support, as provided by the information system, and a design-independent specification of information-systems requirements, are basic features of this methodology. The view of design presented here takes advantage of these features by providing the necessary flexibility. Design itself is viewed as a search on the space of possible software-system structures until one which satisfies the requirements of the information system and achieves the project's objectives is found. The design space is defined on four dimensions that correspond to important layers of information-system software implementation.

1. INTRODUCTION

The abstraction-synthesis methodology of information-systems development [1-3] aims at the development of information systems for effective function support. The first two steps in this four-step methodology: (1) information needs analysis, and (2) analysis of information-system requirements, correspond to the abstraction part. The synthesis part includes steps (3) software design, and (4) system testing and implementation. Underlying this methodology is the view of information systems as collections of activities and resources aimed at the support of control and coordination of function in organizational systems. According to this view, the information-systems design problem becomes one of interfacing automated information-processing functions with organizational activities in a manner as consistent as possible with the achievement of organizational goals and objectives. In this paper, we discuss a synthetic approach to the design of information-systems software. According to this approach, a number of design alternatives are synthesized on the basis of a design-independent specification of information-system requirements. A design satisfying these requirements is then selected for implementation on the basis of its contribution to the attainment of specific objectives.

Emphasis on function support is a distinctive feature of the abstraction-synthesis methodology. Current approaches to information-systems analysis and design base the software-design process on requirements determined with no explicit reference to the support of specific organizational functions. SADT [4], for example, provides tools and techniques for a top-down analysis of data flows, activities, and their interrelationship in complex systems. However, it does not, explicitly, refer to the organizational functions the information system is intended to support. Structured-analysis techniques [5, 6] base the specification of requirements (i.e., the structured specification) on a data-flow model of information processing that identifies the basic components of the information-processing system. Again, these techniques make no explicit reference to specific functions supported by the information system.

The concern with adequate support of organizational function is not apparent in current approaches to software design, either. Structured design [7], for example, is based on the data-flow model developed in the structured-analysis phase, and provides evaluation criteria such as the strength, or cohesion, of modules, and the looseness of their coupling. However, the lack of a link between the processes and data flows defined in the structured specification, and the organizational functions affected, makes it extremely difficult to incorporate effectively function-support considerations at the design stage. Clearly, function-support aspects of

Address correspondence to Roberto R. Kampfner, School of Management and Computer and Information Science, University of Michigan—Dearborn, Dearborn, MI 48128.

0164-1212/89/$3.50

software system must be captured, first, in the requirements specification. Since requirements specifications do not normally allow for an explicit reference to function support, this aspect is lost in traditional approaches to the software cycle, such as the phased-cycle or "waterfall" model [8].

Another important feature of the abstraction-synthesis methodology is the design-independent character of the requirements specification it produces. Current approaches to information-systems development do not give enough emphasis to design independence. Structured analysis, for example, includes in the structured specification a number of design assumptions concerning the user interface, file structures, and features of the physical implementation of information-processing functions.

Our design approach capitalizes on the mentioned features of the abstraction-synthesis methodology by providing software designers with the necessary flexibility. Conceptually, we view the design process as a search on a space of possible software-system structures. The goal of this search process is to find a software-system structure that satisfies the requirements of the information system and, additionally, contributes adequately to the attainment of specific design objectives. Clearly, the design-independent character of the requirements specification is essential to our design approach. It provides the designer with the necessary flexibility. This, in turn, makes it possible to incorporate high standards of software quality so that at the same time function-oriented information-system requirements are met.

This paper is organized as follows. In Section 2, we give a brief review of the abstraction part of the methodology. Section 3 describes the three basic types of requirements considered. This is followed, in Section 4, by a description of the proposed four-dimensional software-design space. The view of design as a synthetic process is characterized, in Section 5, as a search on this space whose aim is to find points (i.e., software-design alternatives) associated with high-performance values. These values must, of course, reflect the degree to which it is assessed that the proposed designs fulfill specific design objectives. A sales and inventory system is considered in the following sections in order to illustrates various aspects of the abstraction-synthesis methodology. The synthetic approach to design is illustrated in Sections 7 and 8.

2. INFORMATION-NEEDS ANALYSIS

The abstraction-synthesis methodology is discussed in detail elsewhere [2, 3]. Here, I shall only review its essential concepts. The first step, information-needs analysis, deals essentially with the identification of objects in the organizational system, which are relevant to the information-function relationship. The view of the information system as a collection of activities and resources aimed at providing information for function support in adaptive, dynamic systems, is central to the abstraction-synthesis methodology. The use of models that facilitate the analysis and specification of organizational features relevant to the information-function relationship is therefore correspondingly important. The organizational control-systems model (OCSM), for example, is a model of hierarchical control that can be used to represent adaptive-control structures in an important class of goal-oriented, organizational systems [3]. This model can be used in information-needs analysis, especially in connection with the support of planning and control functions in the organization. Other conceptual models are also possible, depending on the type of function to be supported by the information system. In fact, hierarchies of conceptual models of specialized subsystems of the organization are often required in order to describe appropriately the network of control and communication processes underlying function support in complex organizational systems. According to the OCSM, the objects identified include goals to be achieved by some part of the organization, functions required for the achievement of such goals, organizational units that realize these functions, and information flows that intercommunicate these units. In the sales and inventory system, for example, the information system supports operational, planning, and control functions in a typical manufacturing organization. These include organizational functions such as those related to finance, production, personnel, and marketing. The main outcome of the information-needs analysis step is the information-needs specification. Representative results obtained as part of this specification are shown in Figure 1, which depicts informational interactions between the order processing and inventory units. These units perform specific organizational functions within the sales and inventory system. The control units that monitor the performance of these operational units are also shown in Figure 1. The information flows described in the information-needs specification are required in order for the organizational units concerned to perform their functions adequately. It is in this sense that they represent information needs for the support of organizational function. The information-function relationship is further determined by the place occupied by each organizational unit in the functional and control hierarchies defined by the OCSM.

Semantic models have been increasingly recognized as an effective means of expressing complex objects and their relationships for the purposes of conceptual schema

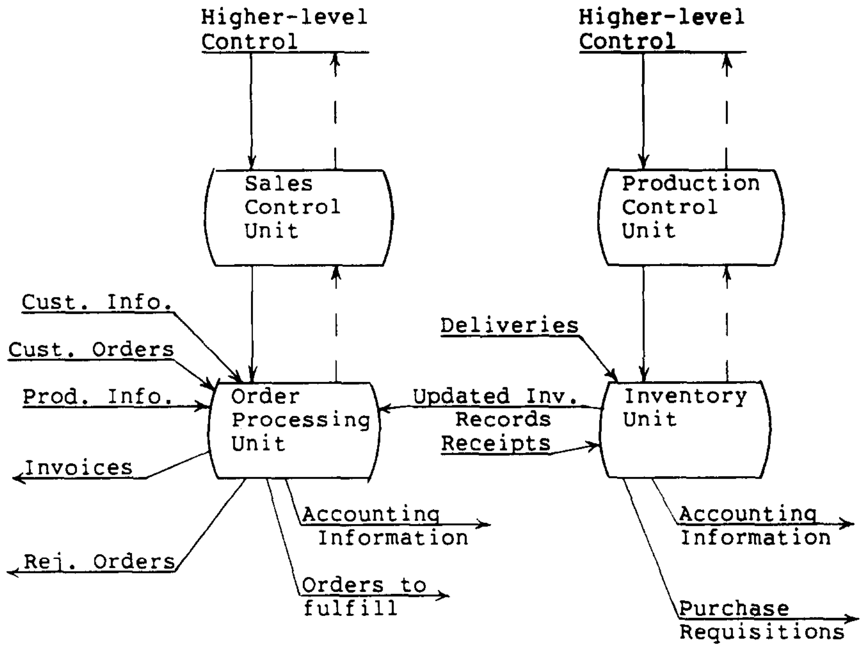


Figure 1. Informational-interactions diagram for the sales and inventory system. Horizontal arrows represent information flows due to operational interactions. Vertical arrows represent control information between operational and control units.

design [9]. An extension of semantic models, aimed at capturing broader aspects of human-oriented models of a world, is referred to as conceptual modeling. The TAXIS project [10], for example, proposes a methodology for building conceptual models based on generalization/specialization techniques.

The abstraction-synthesis methodology also involves the use of conceptual models of organizational activity and function, such as the OCSM, as the basis for the determination of information needs. In the abstraction-synthesis approach, however, the information provided by the conceptual model is not immediately translated into a conceptual schema representation of some kind. Rather, it is used to develop a design-independent specification of information-system requirements. The idea is that it is at the design stage, and based on this kind of specification, where the data base structure of the whole system should be determined, and the required conceptual schemata or file structures defined.

3. THREE TYPES OF INFORMATION-SYSTEM REQUIREMENTS

As mentioned above, the *information-needs analysis* step deals with the determination of the information flows necessary to control and coordinate the organizational units realizing the various organizational functions. In the second step, *analysis of information-system requirements,* the information-processing structure required to sustain appropriately these flows is determined, together with its performance and interface requirements. Recall that these requirements address broad aspects of organizational function support. Thus, not only the algorithmic description of the information-processing functions to be incorporated in the software system is important, but also the role they play in the support of the organization's functional and control structures, their throughput and performance requirements, and the modes of interaction required for the user interface.

In order to capture these vital aspects of function support, the abstraction-synthesis methodology incorporates the specification of *logical, quantitative,* and *user-interface* requirements. Each of these types of requirements conveys an important aspect of information needs, as explained below.

Logical requirements defined the algorithmic structure of the computational models to be incorporated in the software system. They represent invariant, design-independent properties of these models and lend themselves to formal specification. The LIPS/LIPN model [2], for example, is a formalism that describes structural and dynamical features of the information-processing structure required to satisfy an organization's information needs. The LIPS model allows for the representation of logical requirements as a structure.

$$P=(X, Z, \sigma, \tau) \qquad (1)$$

of sets and relations. Here, $X = \{x_i\}$ is the set of information items representing objects of the organizational system that are relevant to the information system.

$Z = \{p_j\}$ is the set of information-processing functions (or IPFs) that transform these information items. $\tau = \{t_m\}$ is a set of triggers, or necessary conditions for the activation of specific information-processing functions $p_j \in Z$. $\sigma = (\sigma_I, \sigma_O, \sigma_P, \sigma_H, \sigma_T)$ is a set of relations that define the logical structure of the information system. The input relation $\sigma_I \subseteq X \times Z$ associates information items $x_i \in X$ with information-processing functions $p_j \in Z$ to which they serve as inputs. The output relation $\sigma_O \subseteq X \times Z$ is defined similarly for information items output from an IPF p_j. The precedence relation $\sigma_P \subseteq Z \times Z$ defines an IPF as preceding another if at least one of the outputs of the former is an input to the latter. The hierarchy relation $\sigma_H \subseteq Z \times Z$ relates an IPF to a larger one including it as a subprocedure. Finally, $\sigma_T \subseteq \tau \times Z$, the trigger relation associates IPFs with triggers, which represent necessary conditions for their activation. The LIPS specification provides also a formal basis for the verification of the validity of proposed designs.

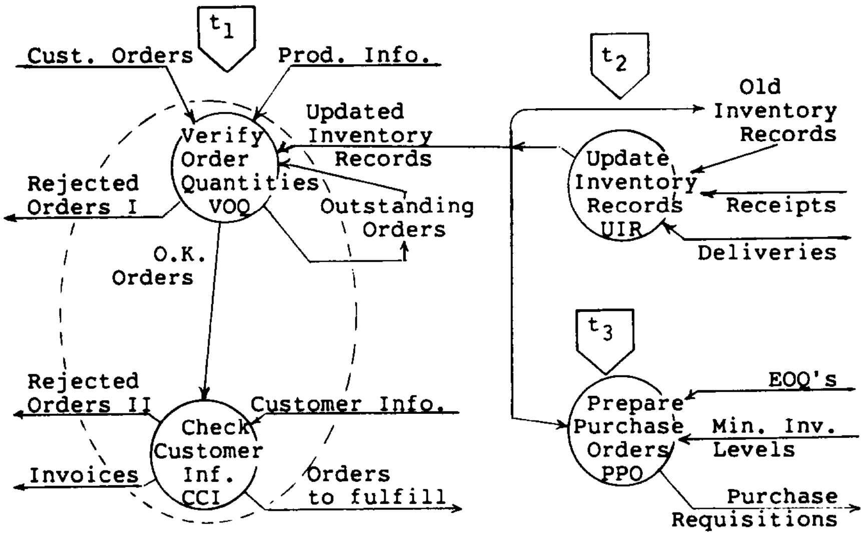
Quantitative requirements represent performance targets imposed on the information-processing structure defined by the logical requirements, such as throughput and response times required at various points of the user interface. Requirements such as volume/time patterns of information usage and spatial distribution of information-system users fall into this class.

User-interface requirements describe yet another essential aspect of information needs. The required or acceptable modes of interaction between automated information-processing functions and the information system's users. Clearly, these three types of requirements must be specified in a closely interrelated manner. This is important for the adequate verification and evaluation of each of the design alternatives synthesized.

Let us consider some information-system requirements for the sales and inventory system. Typical *logical* requirements of a portion of this system are shown schematically in Figure 2. Corresponding *quantitative* and *user-interface* requirements can be stated at a very broad, general level, as follows:

1. Concerning customer orders: It is important to minimize the time required to process customer orders. This process includes the editing, verification, and actual fulfillment of customer orders.
2. Inventory transactions are of two kinds: (a) Queries concerning the availability of products ordered must be responded to promptly, so that a good estimate of the data of delivery can be given to the customer, at the time an order is placed. (b) Updates to inventory records concerning receipts and deliveries of specific items must be performed at the end of the day they occur, the latest.

Figure 2. Some logical requirements of an information system. The information-processing functions shown support sales and inventory control operations. Both graphical and formal representations are shown.



Some corresponding entries in the LIPS specification are:

<Cust. Orders,VOQ> $\epsilon\ \sigma_I$, <O.K.Orders,VOQ> $\epsilon\ \sigma_O$

<VOQ, CCI> $\epsilon\ \sigma_P$, <t_2, UIR> $\epsilon\ \sigma_T$

<VOQ, PO> $\epsilon\ \sigma_H$

Table 1. Some Quantitative Requirements in the Sales and Inventory System

	Transactions during peak period				
Branch	Customer orders	Inventory transactions	Total	Total orders per day	Total number of customers
Cleveland, OH	472	712	1184	1416	11,900
Pittsburgh, PA	448	707	1155	1344	11,560
Akron, OH	171	295	466	523	4700
Buffalo, NY	300	702	1002	900	10,010
Cincinnati, OH	329	650	979	987	9820
St. Louis, MO	329	652	981	987	9820
Louisville, KY	356	401	757	1068	7580
Total	2405	4119	6524	7215	65,390

Some additional *quantitative* requirements are described in Table 1. Considerably more detail is, of course, normally included in a complete specification of information-system requirements. For brevity, however, we are only including in our discussion what we think of as most relevant to the design operations considered.

4. THE DESIGN SPACE OF INFORMATION-SYSTEMS SOFTWARE

By the structure of a software system we mean a specific pattern of arrangement of its parts, such as higher-level functional components, programs, and modules. As mentioned above, we consider four dimensions for the representation of software-system structures: (1) *applications,* (2) *applications support,* (3) *systems software,* and (4) *hardware/firmware.*

The applications dimension is that part of the software system that directly incorporates the information-processing structure defined by the logical requirements. Options on the applications dimension can be conveniently described at three levels as follows:

1. *System architecture.* This level describes highly aggregated functional components of the software system and their interrelationships.
2. *Program structure.* Computer programs are the basic units considered at this level, which describes the modular structure of each program, the function performed by each module, and the communication interfaces between modules.
3. *Module structure.* This level describes the structure of each individual module considered in a given design alternative.

Each of these levels describes the same system, although with a different emphasis. Consequently, consistency between levels becomes an important aspect of this description.

At the system-architecture level, a plausible option for the sales and inventory system would consist of an order-processing component incorporating IPFs VOQ and CCI, and an inventory-control component incorporating VIR and PPO (see Figure 2). At this level, each component is described in terms of the computer programs it contains and their interfaces. The function of each program must also be indicated at this level, in terms of the IPFs it contains. At the program-structure level, each program is described in terms of the modules it contains, and the interfaces between modules. Each of these modules is then described at the module-structure level, in terms of the procedure it incorporates, and any other information concerning the implementation of the information-processing functions incorporated in the module. Thus, an option on the applications dimension corresponds essentially to an assignment of information-processing functions, at various levels of aggregation, to corresponding levels of software-system description.

To each option on the applications dimension corresponds a number of possible options on the applications-support dimension compatible with it. Points on the applications-support dimension represent resources provided by the system that are not application specific, such as programming language processors, data base management systems, application packages, utility programs, etc. More specifically, a point on this dimension consists of a combination of application-support resources that are compatible with some specific point on the applications dimension. In the sales and inventory system, for example, each one of the order-processing and inventory-control components may be assigned a set of conventional file structures, or a specific kind of data base system. The latter may be centralized, or decentralized, to varying degrees. Specific choices must also be made for programming and query languages and other application-support resources.

For each combination of compatible options on the applications and applications-support dimensions, a number of choices on the systems-software dimension compatible with them is normally available. These choices consist of combination of operating-system services and resources. These include specific types of operating-system functions, including those necessary to support interprocess communication, computer networks control, distributed processing, and data base management functions.

Hardware resources compatible with corresponding options on the other dimensions must also be defined for each software-design alternative. Points on the hardware dimension correspond to possible hardware configurations. Choices on this dimension include computer systems, data communications and network equipment, and data base processors, if any.

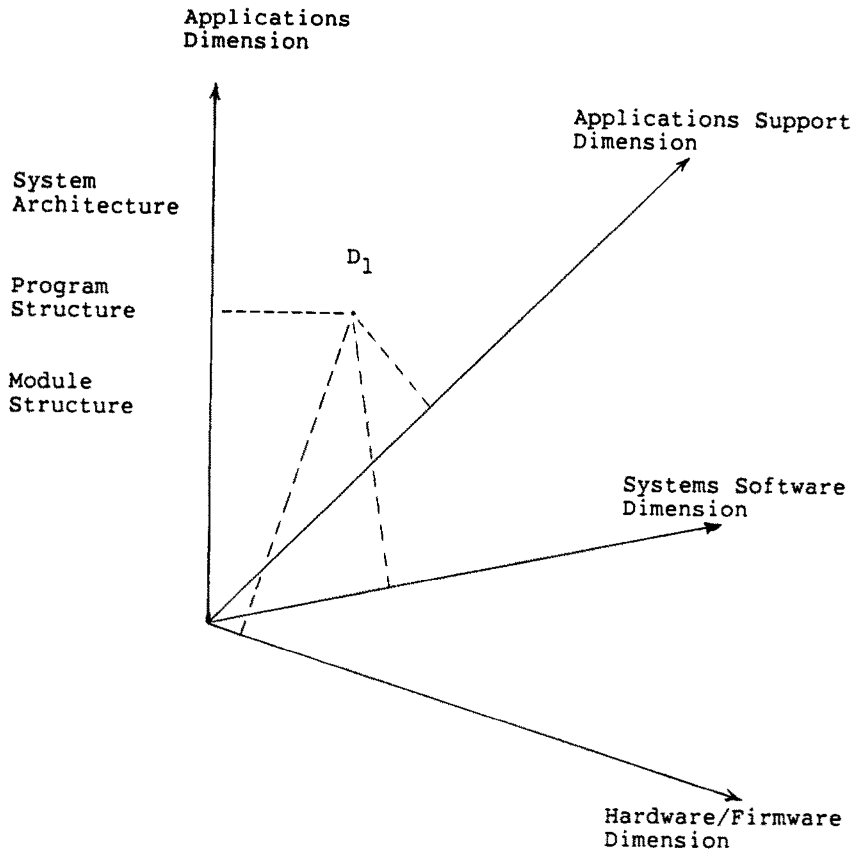

Applications Dimension
Applications Support Dimension
System Architecture
D_1
Program Structure
Module Structure
Systems Software Dimension
Hardware/Firmware Dimension

Figure 3. Schematic representation of the design space of information-systems software. A proposed design, D_1 say, can be represented as a point in this space by specifying its attributes on each of the four dimensions indicated in the figure. On the applications dimension, design attributes are suggested at the levels of system architecture, program structure, and module structure.

A specific software-system alternative is thus defined by four sets of attributes, each corresponding to a specific design dimension. Figure 3 shows schematically the four-dimensional design space we are proposing for the representation of software-system structures. Each choice on each dimension is unique in the sense that no two points in a given dimension may have the same sets of attributes. For simplicity, it is convenient to consider, as points in the four-dimensional space, only those combinations of options that correspond to plausible design alternatives. An incompatible combination of hardware and systems software, for example, would not contribute to any plausible design.

5. DESIGN AS A SYNTHETIC PROCESS

The four-dimensional design space defined in the previous section allows for the representation of software-system structures in terms of four sets of relevant attributes, one for each dimension. This kind of representation facilitates the analysis of the effect of the interdependencies and interrelationships existing between these dimensions on the topology of the design space. This topology is, in fact, determined by computer and software technology, as well as technical and economic factors. Conceptually, the evaluation of each design alternative assigns to the corresponding point in the design space a performance value that reflects its contribution to the attainment of system requirements and design objectives.

The process of synthesizing and evaluating software design and implementation alternatives (shown schematically in Figure 4) consists of the following steps:

1. *Synthesis*. Software-design alternatives are first defined on the applications dimension. This consists of the allocation of information-processing functions defined in the LIPS to modules, programs, and high-level functional components of the software system. The consistency of each alternative with the logical requirements of the information system must be verified at this step. *Valid* design alternatives, i.e., those passing this test, are then the subject of more detailed considerations concerning their possible design and implementation features.
2. *Evaluation*. This is done with respect to quantitative

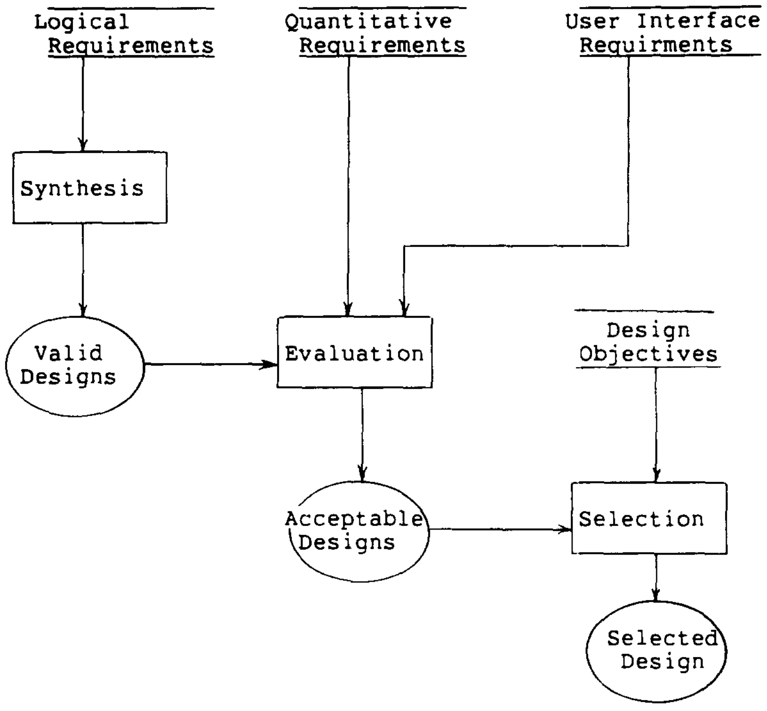


Figure 4. Schematic view of design as a synthetic process. In the synthesis step, proposed designs are synthesized by assigning information-processing functions defined in the logical requirements specification to high-level functional components, programs, and modules. *Valid* designs are those preserving the logical requirements of the information system, that is, the relations specified in the LIPS specification. *Acceptable* designs are *valid* ones that also meet quantitative and user-interface requirements. A design may be further selected for implementation if it gives adequate trade-offs between specified objectives.

and user-interface requirements. Valid design alternatives can be implemented in different ways. To each option considered on the applications dimension may correspond a number of alternatives, each corresponding to a combination of compatible choices on the other three dimensions. Each of these alternatives must be evaluated with respect to the quantitative and user-interface requirements of the information system. Those satisfying these requirements are referred to as *acceptable* alternatives.

3. *Selection.* The design selected for construction and implementation must, clearly, be a valid and acceptable one. There are, however, many other factors that may influence this selection process. Several objectives are normally pursued in any design project, such as cost and time constraints to be met, reliability and efficiency targets, increased maintainability, etc. The design selected for construction and implementation must yield adequate trade-offs between the objectives pursued in a particular project.

The verification of the validity of a proposed design, that is, its consistency with the information system's logical requirements, aims at ensuring that the software system incorporates the information-processing structure required to support the organization's information needs. More precisely, this would require that the information-processing functions needed for the support of organizational functions be incorporated in the appropriate parts of the software system, and that the appropriate information flows be made available through the user interface. The validity of a software-design alternative implies that there is a mapping from the set of IPFs into the set of software-system components realizing them, which preserves the relations defined in the LIPS.

Valid design alternatives, that is, those which preserve the relations specified in the LIPS, must also be evaluated with respect to quantitative and user-interface requirements. This evaluation necessarily involves the consideration of design and implementation options concerning the other dimensions of the design space. The LIPN model is a helpful conceptual tool for this evaluation, since it models the dynamics of the information-processing structure defined by the LIPS. Formally, a LIPN, M is a structure

$$M = (Q_M, P, \tau, \delta_l) \qquad (2)$$

in which Q_M is the set of global states, τ is the set of triggers, P is a LIPS, as defined above, and δ_l is the local transition function.

The global transition function, which is in fact a parallel map, is defined in a cell-space-like fashion. The global state $\hat{q} \in Q_M \subseteq \{q_i\}^n$, $i = 1, 2, \cdots, n$, where q_i is the state of $p_i \in Z$, is determined by the collection of local states $q_i \in \{S, W, A, C, I\}$, which represent distinguishable states of the process associated with IPF p_i. An informal description of these states is as follows.

S, the *initial* state. All of the *processes* associated with IPFs included in a LIPN, M, are initially in state S.

I, the *inactive* state. A process p_i is in state I when no computation associated with it is being performed, and it is not affecting the state of other processes in the network. p_i enters state I, after it leaves its *completed* state C.

W, the *waiting* state. A process p_i is in state W if at least one p_j that immediately precedes it, and is in its *active* state A, has not entered the completed state C.

A, the *active* state. A process p_i is in state A if a computation associated with it is being performed. It define the information-processing structure to be automated. Therefore, they constitute the basis of the synthesis process, that is, as mentioned above, all alternatives considered for evaluation must be consistent with these requirements. On the applications dimension, software-design alternatives differ in the way the information-processing functions specified in the logical requirements are assigned to high-level components of the software system, computer programs, and modules. We consider three main levels of description. The system-architecture level describes the software system in terms of the programs each high-level functional component includes, and the interfaces between programs. The program-structure level describes each program as a hierarchy of modules, each performing a well-defined function, and their communication interfaces. The detailed structure of each module is then described at the module-structure level.

Many techniques have been proposed, and used, to help define the structure of software-system components and evaluate their quality. Structured design [7], for example, uses transform analysis and transaction analysis as the main techniques for the derivation of a program structure consistent with a given program specification. The concept of abstractions has also been used in the process of specifying a user model that would satisfy a set of requirements [11]. Such a user model includes the concepts the user has to know to use the system, commands available to the user, and the interactions with external subsystems. An implementor's model, that can be ultimately translated into code that specifies the program behavior, is then constructed. Yet another approach concerns modular and object-oriented methodologies of software engineering, in which specific programming-language constructs and definitions are derived from the requirements specification [12]. Modular and object-oriented design [12, 13] allows the designer to create abstract data types and functional abstractions into which to map relevant aspects of the real-world domain. The Ada package with its private types, and the Modula-2 module with its opaque types, are programming-language features that help the designer by allowing for the complete separation between the specification and implementation of modules, thus facilitating modular and object-oriented design.

As mentioned above, all attributes pertaining to the applications dimension must be consistent with the logical requirements. Modules, programs, and higher-level components of the software system incorporate IPFs at corresponding levels of aggregation. Consistency of a given design alternative with the corresponding logical requirements implies that the relations included in the LIPS are preserved through the assignment of IPFs to software-system components at the various levels.

Let us assume, for example, that the sales and inventory system a sales office has been established at enters this state from state W if all of its *triggers* are on, and all of the processes associated with IPFs that immediately precede it have entered state C.

C, the *completed* state. Process p_i is in state C if the computation it performs has been completed. It remains in this state until all of the processes associated with the IPFs it immediately precedes enter state A. The transition from state A to state C is the only state transition locally determined.

The local transition function δ_l specifies the global transition function δ_M as follows:

$$\hat{q} = \delta_M(\hat{q}) \Leftrightarrow \text{For all } p_i \in Z$$
$$\dot{q}_i' = \delta_l(q_i, N_i, T_i) \qquad (3)$$

Briefly, Eq. (3) asserts that $\delta_M(\hat{q})$ is the configuration obtained from $\hat{q}$ by applying the local transition function δ_l to each $p_i \in Z$, with neighborhood relation N_i, and trigger-set T_i (for details, see Ref. 2).

The LIPN model describes the time evolution of the state of an information-processing structure. As such, it constitutes a basis on which the performance of proposed designs of the software system can be assessed for various possible sequences of activation of IPFs. Throughput and response-time characteristics of specific designs can then be assessed as to their ability to meet the information system's quantitative and user-interface requirements.

A valid and acceptable design can be considered for construction and implementation. In fact, several proposed designs can be found acceptable in the sense defined above. The selection step allows the designer to choose one that favors specified objectives. In the case of conflictive objectives, adequate trade-offs must be obtained by the design selected for construction and implementation.

An important concern in our approach is flexibility of design. The idea is that the software designer must be able to synthesize and evaluate a sufficient number of alternatives, so that adequate trade-offs between design options can be obtained, in order to achieve specific design objectives. A design-independent specification of

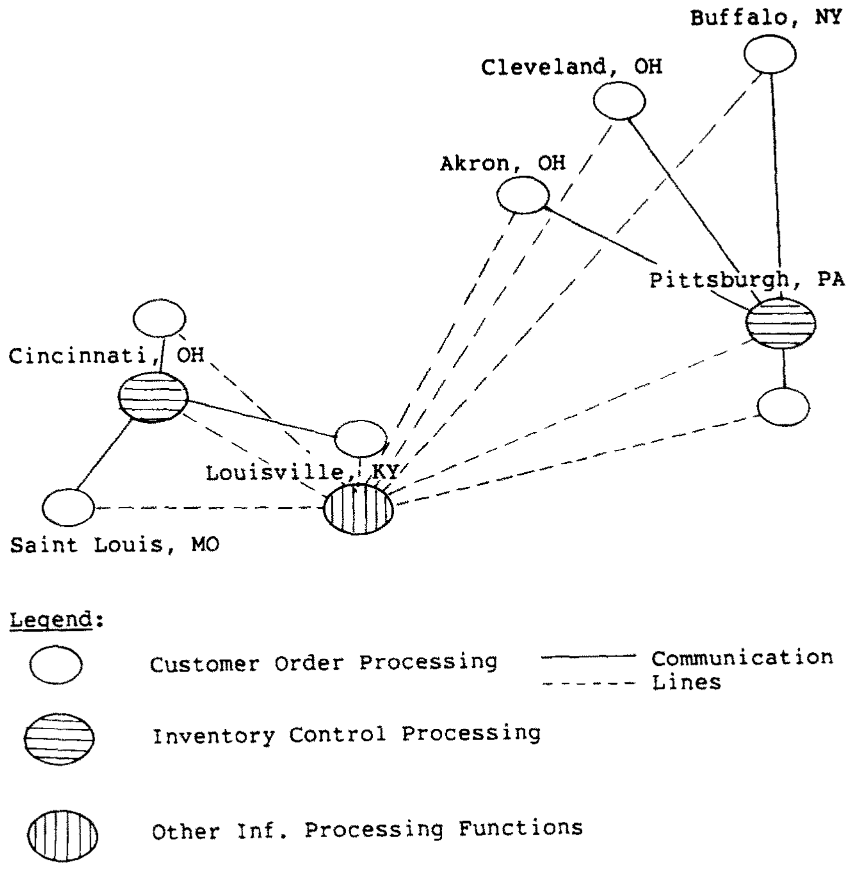


Figure 5. Some aspects of a design alternative. The diagram shows the allocation of information-processing functions to functional components (order processing and inventory control) of the software system, and the resulting geographical distribution. More detailed options on the four dimensions of the design space further specify the proposed design.

requirements is clearly an important prerequisite for this flexibility and, therefore, essential to our view of software design.

6. SYNTHESIS AND EVALUATION OF DESIGN ALTERNATIVES

In this section, I shall illustrate the synthetic view of design by looking at typical options the designer faces at each step of the process. I shall also characterize some of the interdependencies that exist between options on the various dimensions.

6.1 Synthesis of Design Alternatives

The logical requirements of the information system each branch (see Table 1), corporate offices are located in Cincinnati, and inventory-control centers operate in Pittsburgh and Cincinnati. Some aspects of a possible design option at the system-architecture level are shown in Figure 5. Assume, further, that IPFs are assigned to high-level components, and programs as indicated in Figure 6. This assignment clearly preserves the hierarchy relation σ_H. The definition of the interfaces of the various modules must, of course, satisfy relations σ_I, σ_O, as implied by Figure 2. Precedence and trigger relations must also be preserved.

Features defined on the applications dimension, however, are ultimately implemented through related options on the other dimensions. For this reason, the acceptability of a given design will depend to a great extent on the choices made at these other dimensions.

It is important to notice, here, that decisions on a given dimension are always likely to affect those concerning any other dimension. Decisions concerning network topology and degree of distribution, for example, have usually a strong impact on the type of hardware required for the implementation of such features. Similarly, the resources provided by the applications-support dimension, which include programming languages, data base management systems (DBMS), program libraries, utility programs, application packages, etc., are constrained to a great extent by choices made at the hardware/firmware and the systems-software dimensions. Instances of the relationship between the applications and the applications-support dimensions include constraints imposed on the type of representation of objects and programming-language

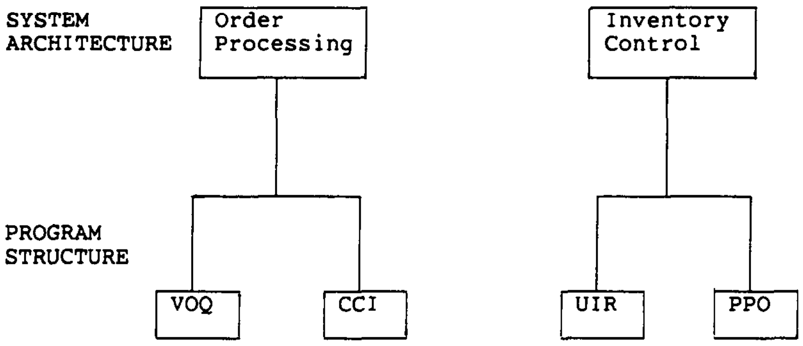


Figure 6. Aspects of an option on the applications dimension. The hierarchy chart shows a particular assignment of IPFs to computer programs and high-level components of the software system. Order processing and inventory control are high-level components shown at the system-architecture level. At the program-structure level, the figure shows IPFs VOQ, CCI, UIR, and PPO, each assigned to individual programs.

constructs used in the specification of the underlying processes. We can also mention the DBMS, file structures and access methods, and I/O capabilities supported by the installation as aspects of the hardware, systems software, and applications-support dimensions that directly influence the program-structure level of the applications dimension. Some features of program structures, for instance, depend on capabilities of the systems-software dimension, such as support of concurrency and distributed processing, and the inter-process-communication capabilities allowed by the operating system. Clearly, the interdependence between design options on the various dimensions is made more explicit with this kind of representation. This, of course, makes the analysis of trade-offs between design options easier.

6.2 Evaluation and Selection of Design Alternatives

Valid alternatives must also satisfy quantitative and user-interface requirements in order to be considered candidates for implementation. For the sales and inventory system this means, for example, that customer orders must be verified and that the corresponding deliveries must be scheduled at the time they are placed. In addition to this, each sales branch must be able to handle the required volumes, and the specified user-interface requirements must be satisfied.

Throughput and response times can, in this case, be estimated for various combinations of line speed and processor capacities (for details of the calculations, see, for example, Ref. 14). These results can then be evaluated on the basis of the estimated frequency of generation of transactions at each location, and the corresponding response-time requirements.

Once a set of acceptable design alternatives has been defined and represented in the design space, a particular one can be selected for construction and implementation on the basis of specified objectives. This selection process must consider relevant trade-offs associated with each acceptable design. For example, a given alternative may show a greater degree of reliability in the communications component of the system at the expense of greater communications overhead. Another possible trade-off might involve a reduction of the communication load and a more efficient processing of customer orders by partitioning the customer-orders and customer-information parts of the data base by the sales branch. This partition could be done, for example, on the assumption that many customers normally operate on a single branch. Queries involving customers operating on various branches, and changes of the pattern of allocation of customers to branch represent, in this case, situations in which such a design decision might not be the best choice. The evaluation of performance aspects of software-system alternatives, and the evaluation of trade-offs between options, may become a highly specialized and, in some cases, very complex process. Nevertheless, the approach to design discussed here, based on a design-independent, formal specification of requirements, and enjoying the flexibility given by design independence in conjunction with this kind of representation is, in our opinion, an important step in the development of information systems tuned for the effective support of specific organizations.

7. REPRESENTATION OF SOFTWARE-SYSTEM STRUCTURES

Each of the alternative versions of the software system considered in the design step must be properly evaluated and their associated trade-offs analyzed. To facilitate this evaluation, relevant features of each alternative must be recorded and described conveniently. The basic idea is to record the design options characterizing each proposed version of the software system. The simplest form involves the use of a table in which rows correspond to specific design decisions, while columns correspond to dimensions of the design space. Thus, each row describes some aspects of a design alternative in terms of the options characterizing it on each dimension. Table 2 indicates possible design options for the sales and inventory system. The options so recorded must also be described in detail in a manner that

Table 2. Sample Design Decisions for the Sales and Inventory System

	Design Alternatives (options on each design dimension)			
Design option	Applications	Applications support	Systems software	Hardware
1	(See Figure 6)	VOQ and CCI performed centrally; data entered at each branch. COBOL PL/I	SNA (VTAM) CICS/VS	3270-type CRT
		Customer DB in Cincinnati. IMS/DB	OS/VMS SNA (VTAM) CICS/VS	IBM 370
		UIR and PPO performed at inventory centers. IMS/DB	OS/VMS SNA (VTAM) CICS/VS	IBM 370
2	(See Figure 6)	VOQ and CCI performed at each branch; local customer data base. COBOL PL/I.	DOS SNA (VTAM) CICS/VS	IBM PC
		Mainframe in Cincinnati	OS/VMS SNA (VTAM) CICS/VS	IBM 370
		UIR and PPO performed at inventory centers; inventory DB at centers.	OS/VMS SNA(VTAM) CICS/VS	IBM 370

facilitates their analysis and the evaluation of the trade-offs each offers with respect to specific objectives. There are, of course, many forms in which specific design aspects can be described. In a particular context, however, a particular form of representation may be more useful than others. Although structured charts, for example, convey conveniently the modular structure of a program, some forms of Petri nets are sometimes used for the description of interactive aspects of the user interface (see, for example, Ref. 15). In some cases, prototypes are built in order to facilitate the description and evaluation of design features which, otherwise, are difficult to represent and convey [16, 17].

8. SUMMARY AND CONCLUSIONS

A synthetic approach to the design of information-systems software is presented in which the design process is viewed as a search on the space of possible structures. The abstraction-synthesis methodology of information-systems development facilitates this synthesis through the identification of basic components of information needs and information-system requirements in the abstraction phase. Design is viewed as a process in which alternative versions of the software system are synthesized from components drawn from combinations of compatible options in a four-dimensional space of software-system structures. These are the applications, applications-support, systems-software, and hardware/firmware dimensions, which constitute a design space in which design alternatives can be represented and analyzed. This view of design takes advantage of a design-independent specification of *logical, quantitative,* and *user-interface* requirements of information systems obtained in connection with the abstraction-synthesis methodology. First, logical requirements, defined in terms of the LIPS/LIPN formalism [2], are used for the verification of the validity of software-system components generated on the applications dimension. Valid designs are then evaluated as to their acceptability, that is, their ability to satisfy quantitative and user-interface requirements. An acceptable design that adequately satisfies specified design objectives is then selected for construction and implementation. It is suggested that the consideration of this space facilitates the analysis of interdependencies between design options and, consequently, the evaluation of complex trade-offs between design alternatives. The synthetic approach to the design of information-systems software discussed here depends, for its success, on an enhanced flexibility of software design. To achieve this flexibility, the design-independent specification of information-system requirements is of critical importance.

REFERENCES

1. R. Kampfner, A systems-oriented framework for the analysis and design of information systems, in *The Relation Between Major World Problems and Systems Thinking,* Vol. II, (G. E. Lasker, ed.), Intersystems, Seaside, California, 1983, pp. 707–713.
2. R. Kampfner, Formal Specification of Information-Systems Requirements, *Information Processing and Management* 21(5), 401–414 (1985).
3. R. Kampfner, A Hierarchical Model of Organizational Control for the Analysis of Information-Systems Requirements, TR CSC-86-001, Department of Computer Science, Wayne State Univ., 1986.
4. D. T. Ross, Structured Analysis (SA): A Language for Communicating Ideas, *IEEE Trans. Software Eng.* SE-3(1), 16–34 (1977).
5. T. De Marco, *Structured Analysis and System Specification,* Prentice-Hall, Englewood Cliffs, New Jersey, 1979, pp.
6. C. Gane and T. Sarson, *Structured Systems Analysis:*

Tools and Techniques, Prentice-Hall, Englewood Cliffs, New Jersey, 1979, pp.

7. W. Stevens, G. Myers, and L. Constantine, Structured Design, *IBM Syst. J.* 13(2), 115–139 (1974).
8. R. E. Fairley, *Software Engineering Concepts,* Mc-Graw-Hill, New York, 1985, pp.
9. M. Brodie, On the development of data models, in *On Conceptual Modelling,* No. 2 (M. Brodie, J. Mylopoulos, and J. Schmidt, eds.), Springer-Verlag, New York, 1984.
10. J. Mylopoulos, P. A. Bernstein, and H. T. K. Wong, A Language Facility for Designing Interactive Data Base Intensive Applications, *ACM Trans. Data Base Syst.* 5(2), 185-207 (1980).
11. V. Berzins, M. Gray, and D. Naumann, Abstraction-Based Software Development, *Commun. ACM* 29(5), 402-415 (1986).
12. R. Wiener and R. Sincovec, *Software Engineering with Modula-2 and ADA,* John Wiley and Sons, New York, 1984, pp.
13. G. Booch, *Software Engineering with ADA,* Benjamin/Cummins, New York, 1983, pp.
14. U. Black, *Data Communications, Networks, and Distributed Processing,* Reston Publishing Company, Inc., Reston, Virginia, 1983, pp.
15. A. Borgida, J. Mylopoulos, and H. K. T. Wong, Generalization/specialization as a basis for software specification, In *On Conceptual Modelling,* (M. Brodie, J. Mylopoulos, and J. Schmidt eds.), Springer-Verlag, New York, 1984.
16. ACM SIGSOFT, Rapid Prototyping Workshop, April 1982, published as *ACM Software Eng. News,* 17(5), Dec. 1982.
17. R. E. A. Mason and T. T. Carey, Prototyping interactive information systems, *Commun. ACM* 26(5), 347–354 (1983).