SYSTEMS ARCHITECTURE

Rapid changes in information technology, uncertainty regarding future requirements, and increasing complexity of systems have led to the use of systems architectures as a key step in the systems engineering process. While hardware and software components of a system can change over time, the

J. Webster (ed.), *Wiley Encyclopedia of Electrical and Electronics Engineering*. Copyright © 1999 John Wiley & Sons, Inc.

underlying architecture remains invariant. This allows graceful upgrading of systems through the use of components from many manufacturers whose products conform to the architecture. The use of systems architectures has been very effective in telecommunication systems, in software development, and in computers. It is now being extended to large-scale information systems.

Two basic paradigms are available for designing and evaluating systems and architectures: the structured analysis and the object oriented approaches. Both require multiple models to represent them and both lead, through different routes, to executable models. The latter are appropriate for analyzing the behavior of the architecture and for evaluating performance prior to system implementation.

Systems architecting has been defined as the process of creating complex, unprecedented systems (1,2). This description fits the computer-based systems that are being created or planned today, whether in industry, government, or academia. The requirements of the marketplace are ill-defined and rapidly changing with evolving technology making possible the offering of new services at a global level. At the same time, there is increasing uncertainty as to the way in which they will be used, what components will be incorporated, and the interconnections that will be made. Generating a system architecture as part of the systems engineering process can be seen as a deliberate approach to manage the uncertainty that characterizes these complex, unprecedented systems.

The word architecture derives from the Greek word architecton which means master mason or master builder. The architect, now as then, is a member of the team that is responsible for designing and building a system; then the systems were edifices, now they are computer-based and software intensive. Indeed, the system architect's contribution comes in the very early stages of the systems engineering process, at the time when the operational concept is defined and the conceptual model of the system is developed. Consequently, the design of a system's architecture is a top-down process, going from the abstract and general to the concrete and specific. Furthermore, it is an iterative process. The process of developing an architecture in response to requirements (that are ill-structured because of multiple uncertainties) forces their re-examination. Ambiguities are identified and resolved and, when inconsistencies are discovered, the requirements themselves are reformulated.

One thinks of system architectures when the system in question consists of many diverse components. A system architect, while knowledgeable about the individual components, needs to have a good understanding of the inter-relationships among the components. While there are many tools and techniques to aid the architect and there is a well-defined architecture development process, architecting requires creativity and vision because of the unprecedented nature of the systems being developed and the ill-defined requirements. For detailed discussions on the need for systems architecting, see Refs. 1–7.

Many of the methodologies for systems engineering have been designed to support a traditional system development model. A set of requirements is defined; several options are considered and, through a process of elimination, a final design emerges that is well defined. This approach, based on structured analysis and design, has served the needs of systems engineers and has produced many of the complex systems in use today. It is effective when the requirements are well-defined and remain essentially constant during the system development period. However, this well-focused approach cannot handle change well; its strength lies in its efficiency in designing a system that meets a set of fixed requirements.

An alternative approach with roots in software systems engineering is emerging better able to deal with uncertainty in requirements and in technology, especially for systems with long development time and expected long life cycle during which both requirements and technology will change. This approach is based on object oriented constructs. The problem is formulated in general terms and the requirements are more abstract and, therefore, subject to interpretation. The key advantage of the object oriented approach is that it allows flexibility in the design as it evolves over time.

DEFINITION OF ARCHITECTURES

In defining an architecture, especially of an information system, the following items need to be described. First, there are processes that need to take place in order that the system accomplish its intended functions; the individual processes transform either data or materials that flow between them. These processes or activities or operations follow some rules that establish the conditions under which they occur; furthermore, they occur in some order that need not be deterministic and depends on the initial conditions. There is also need to describe the components that will implement the design: the hardware, software, personnel, and facilities that will be the system.

This fundamental notion leads to the definition of two architectural constructs: the functional architecture and the physical architecture. A *functional architecture* is a set of activities or functions, arranged in a specified partial order that, when activated, achieves a set of requirements. Similarly, a *physical architecture* is a representation of the physical resources, expressed as nodes, that constitute the system and their connectivity, expressed in the form of links. Both definitions should be interpreted broadly to cover a wide range of applications; furthermore, each may require multiple representations or views to describe all aspects.

Before even attempting to develop these representations, the operational concept must be defined. This is the first step in the architecture development process. An *operational concept* is a concise statement that describes how the goal will be met. There are no analytical procedures for deriving an operational concept for complex, unprecedented systems. On the contrary, given a set of goals, experience, and expertise, humans invent operational concepts. It has often been stated (1) that the development of an architecture is both an art and a science. The development of the conceptual model that represents an operational concept falls clearly on the art side. A good operational concept is based on a simple idea of how the over-riding goal is to be met. For example, "centralized decision making and distributed execution" represents a very abstract operational concept that lends itself to many possible implementations, while an operational concept such as the "client-server" one is much more limiting. As the architecture development process unfolds, it becomes necessary to elabo-

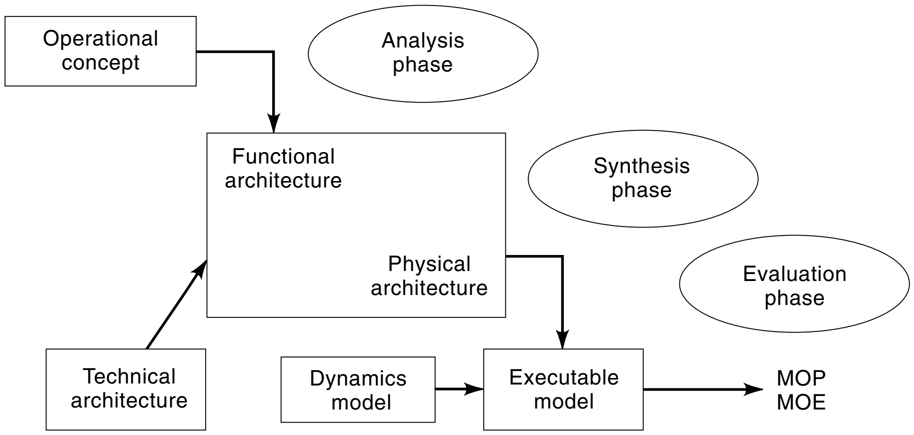

Figure 1. The three phase process of architecture development.

rate on the operational concept and make it more specific. The clear definition and understanding of the operational concept is central to the development of compatible functional and physical architectures.

Analogous to the close relationship between the operational concept and the functional architecture is the relationship between the technical architecture and the physical one. A *technical architecture* is a minimal set of rules governing the arrangement, interaction, and interdependence of the parts or elements whose purpose is to ensure that a conformant system satisfies a specified set of requirements. It provides the framework upon which engineering specifications can be derived, guiding the implementation of the system. It has often been compared to the building code that provides guidance for new buildings to be able to connect to the existing infrastructure by characterizing the attributes of that infrastructure.

All these representations are static ones; they consist of diagrams. In order to analyze the behavior of the architecture and evaluate the performance characteristics, an executable model is needed. An executable model is a dynamic model; it can be used to analyze the properties of the architecture and it can also be used to carry out simulations. Both methodologies, whether structured analysis based or object oriented based, become rigorous when an executable model is derived and the condition is imposed that all information contained in that model must be traced back to one or more of the static diagrams. This dynamic model of the architecture is called the operational-X architecture where the X stands for the executable property.

The architecture development process can be characterized as consisting of three phases: the *analysis* phase in which the static representations of the functional and physical architectures are obtained using the operational concept to drive the process and the technical architecture to guide it; the *synthesis* phase in which these static constructs are used, together with descriptions of the dynamic behavior of the architecture (often referred to as the *dynamics* model), to obtain the executable model of the architecture, and the *evaluation* phase in which measures of performance (MOP) and measures of effectiveness (MOE) are obtained. This three phase process is shown schematically in Fig. 1.

STRUCTURED ANALYSIS APPROACH

The structured analysis approach has its roots in the structured analysis and design technique (SADT) that originated in the 1950s (8) and encompasses structured design (9), structured development (10), the structured analysis approach of DeMarco (11), structured systems analysis (12), and the many variants that have appeared since then, often embodied in software packages for computer-aided requirements generation and analysis. This approach can be characterized as a process-oriented one (12) in that it considers as the starting point the functions or activities that the system must perform. A second characterizing feature is the use of functional decompositions and the resulting hierarchically structured diagrams. However, to obtain the complete specification of the architecture, as reflected in the executable model, in addition to the process or activity model, a data model, a rule model, and a dynamics model are required. Each one of these models contains inter-related aspects of the architecture description. For example, in the case of an information system, the activities or processes receive data as input, transform them in accordance with some conditions, and produce data as output. The associated data model describes the relationships between these same data elements. The conditions that must be satisfied are expressed as rules associated with the activities. But for the rules to be evaluated, they require data that must be available at that particular activity with which the rule is associated; the output of the rule also consists of data that control the execution of the process. Furthermore, given that the architecture is for a dynamic system, the states of the system need to be defined and the transitions between states identified to describe the dynamic behavior. State transition diagrams are but one way of representing this information. Underlying these four models is a data dictionary or, more properly, a system dictionary, in which all data elements, activities, and flows are defined. The construct that emerges from this description is that a set of inter-related views, or models, are needed to describe an architecture using the structured analysis approach. The activity model, the data model, the rule model and the supporting system dictionary, taken together, constitute the functional architecture of the system. The term functional architecture has been used to describe a range of representations—from a simple activity model to the set of models defined here. The structure of the functional architecture is shown in Fig. 2.

At this time, the architect must use a suite of tools and, cognizant of the inter-relationships among the four models and the features of the tools chosen to depict them, work across models to make the various views consistent and co-

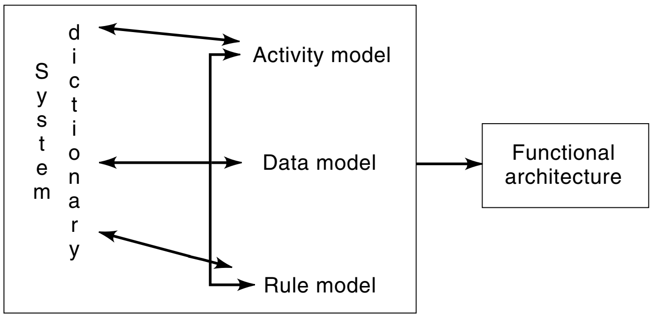

Figure 2. Structure of the functional architecture. The functional architecture contains an activity model, a data model, and a rule model. The three models must be in concordance with each other and have a common data dictionary.

herent, that is, to achieve model concordance. The architect must obtain a single, integrated system dictionary from the individual dictionaries produced by the various tools that generate the different views.

What a functional architecture does not contain is the specification of the physical resources that will be used to implement the functions or the structure of the human organization that is supported by the information system. These descriptions are contained in the physical architecture.

Activity Model

A method in wide use for the representation of an activity model is IDEF0 which has systems engineering roots; for its history, see (8). The National Institute of Standards and Technology (NIST) published Draft Federal Information Processing Standard #183 for IDEF0. IDEF0 is a modeling language for developing structured graphical representations of the activities or functions of a system. It is a static representation, designed to address a specific question from a single point of view. It has two graphical elements: a box, which represents an activity, and a directed arc that represents the conveyance of data or objects related to the activity. A distinguishing characteristic of IDEF0 is that the sides of the activity box have a standard meaning, as shown in Fig. 3. Arcs entering the left side of the activity box are inputs, the top side are controls, and the bottom side are mechanisms or resources used to perform the activity. Arcs leaving the right side are outputs—the data or objects generated by the activity. When IDEF0 is used to represent the process model in a functional architecture, mechanisms are not needed; they are part of the physical architecture.

Verbs or verb phrases are inscribed in the activity boxes to define the function represented. Similarly, arc inscriptions are used to identify the data or objects represented by the arcs. There are detailed rules for handling the branching and the joining of the arcs. A key feature of IDEF0 is that it supports hierarchical decomposition. At the highest level, the A-0 level, there is a single activity that contains the root verb of the functional decomposition. This is called the context diagram and also includes a statement of the purpose of the model and the point of view taken. The next level down, the A0 level, contains the first level decomposition of the system function and the interrelationships between these functions. Each one of the activity boxes on the A0 page can be further decomposed into the A1, A2, A3, . . . page, respectively.

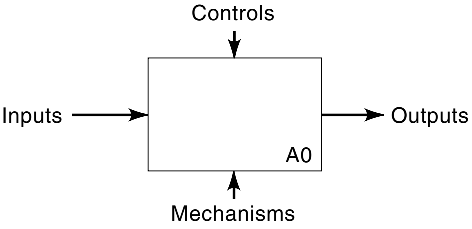

Figure 3. Box and arrow semantics in IDEF0.

Associated with IDEF0 is a data dictionary which includes the definitions and descriptions of the activities, listing and description of the inputs, controls, and outputs, and, if entered, a set of activation rules of the form "preconditions → postconditions." These are the rules that indicate the conditions under which the associated function can be carried out.

Data Model

The purpose of a data model is to analyze the data structures and their relationships independently of the processing that takes place, already depicted in the activity model. There are two main approaches with associated tools for data modeling: IDEF1x and entity-relationship (E-R) diagrams. Both approaches are used widely. The National Institute of Standards and Technology has published Draft Federal Information Processing Standard #184 in which IDEF1x is specified. There are many books that describe E-R diagrams: Sanders (14), Yourdon (15), McLeod (16).

IDEF1x (IDEF1 extended) is a modeling language for representing the structure and semantics of the information in a system. The elements of IDEF1x are the entities, their relationships or associations, and the attributes or keys. An IDEF1x model is comprised of one or more views, definitions of the entities, and the domains of the attributes used in the views.

An entity is the representation of a set of real or abstract objects that share the same characteristics and can participate in the same relationships. An individual member of the set is called an entity instance. An entity is depicted by a box; it has a unique name and a unique identifier. If an instance of an entity is identifiable with reference to its relationship to other entities, it is called identifier dependent. The box depicting the entity instance is divided into two parts: the top part contains the primary key attributes; the lower one the nonprimary key attributes. Every attribute must have a name (expressed as a noun or noun phrase) that is unique among all attributes across the entities in the model. The attributes take values from their specified domains.

Relationships between entities are depicted in the form of lines that connect entities; a verb or verb phrase is placed beside the relationship line. The connection relationship is directed—it establishes a parent–child association—and has cardinality. Special symbols are used at the ends of the lines to indicate the cardinality. The relationships can be classified into types such as identifying or non-identifying, specific and nonspecific, and categorization relationships. The latter, for example, is a generalization/specialization relationship in which an attribute of the generic entity is used as the discriminator for the categories.

Rule Model

In a rule oriented model, knowledge about the behavior of the architecture is represented by a set of assertions that describe what is to be done when a set of conditions evaluates as true. These assertions, or rules, apply to specific functions defined in the activity model and are formulated as relationships among data elements. There are several specification methods that are used depending on the application. They include decision trees, decision tables, structured english, and mathematical logic. Each one has advantages and disadvantages; the choice often depends on the way that knowledge about rules has been elicited and on the complexity of the rules themselves.

Dynamics Model

The fourth type of model that is needed is one that characterizes the dynamic behavior of the architecture. This is not an executable model, but one that shows the transition of the system state as a result of events that take place. The state of a system can be defined as all the information that is needed at some time t_0 so that knowledge of the system and its inputs from that time on determines the outputs. The state space is the set of all possible values that the state can take.

There is a wide variety of tools for depicting the dynamics, with some tools being more formal than others: state transition diagrams, state charts, event traces, key threads, and so on. Each one serves a particular purpose and has unique advantages. For example, a state transition diagram is a representation of a sequence of transitions from one state to another—as a result of the occurrence of a set of events—when starting from a particular initial state or condition. The states are represented by nodes (e.g., a box) while the transitions are shown as directed arcs. The event that causes the transition is shown as an arc annotation, while the name of the state is inscribed in the node symbol. If an action is associated with the change of state, then this is shown on the connecting arc, next to the event. Often, the conditions that must be satisfied in order for a transition to occur are shown on the arcs. This is an alternative approach for documenting the rule model.

System Dictionary and Concordance of Models

Underlying all these four models is the system dictionary. Since the individual models contain overlapping information, it becomes necessary to integrate the dictionaries developed for each one of them. Such a dictionary must contain descriptions of all the functions or activities including what inputs they require and what outputs they produce. These functions appear in the activity model (IDEF0), the rule model (as actions), and the state transition diagrams. The rules, in turn, are associated with activities; they specify the conditions that must hold for the activity to take place. For the conditions to be evaluated, the corresponding data must be available at the specific activity—there must be an input or control in the IDEF0 diagram that makes that data available to the corresponding activity. Of course, the system dictionary contains definitions of all the data elements as well as the data flows that appear in the activity model.

The process of developing a consistent and comprehensive dictionary provides the best opportunity for ensuring concordance among the four models. Since each model has a different root and was developed to serve a different purpose, together they do not constitute a well integrated set. Rather, they can be seen as a suite of tools that collectively depict sufficient information to specify the architecture. The interrelationships among models are complex. For example, rules should be associated with the functions at the leaves of the functional decomposition tree. This implies that, if changes are made in the IDEF0 diagram, then the rule model should be examined to determine whether rules should be reallocated and whether they need to be restructured to reflect the availability of data in the revised activity model. A further implication is that the four models cannot be developed in sequence. Rather, the development of all four should be planned at the beginning with ample opportunity provided for iteration, because if changes are made in one, they need to be reflected in the other models.

Once concordance of these models has been achieved, it is possible to construct an executable model. Since the physical architecture has not been constructed yet, the executable model can only be used to address logical and behavioral issues, but not performance issues.

The Executable Model

Colored petri nets (17) are an example of a mathematically rigorous approach but with a graphical interface designed to represent and analyze discrete event dynamical systems. They can be used directly to model an architecture. The problem, however, is to derive a dynamic representation of the system from the four static representations.

The solution to this problem using the structured analysis models can be described as follows. One starts with the activity model. Each IDEF0 activity is converted into a petri net transition; each IDEF0 arrow connecting two boxes is replaced by an arc-place-arc construct, and the label of the IDEF0 arc becomes the color set associated with the place. All these derived names of color sets are gathered in the global declaration node of the petri net. From this point on, a substantial modeling effort is required to make the colored petri net model a dynamic representation of the system. The information contained in the data model is used to specify the color sets and their respective domains, while the rules in the rule model result in arc inscriptions, guard functions, and code segments.

The executable model becomes the integrator of all the information; its ability to execute tests some of the logic of the model. Given the colored petri net model, a number of analytical tools from petri net theory can be used to evaluate the structure of the model, for example, to determine the presence of deadlocks, or obtain its occurrence graph. The occurrence graph represents a generalization of the state transition diagram model. By obtaining the occurrence graph of the petri net model, which depicts the sequence of states that can be reached from an initial marking (state) with feasible firing sequences, one has obtained a representation of a set of state transition diagrams. This can be thought as a first step in the validation of the model at the behavioral level. Of course, the model can be executed to check its logical consistency, that is, to check whether the functions are executed in the appropriate sequence and that the data needed by each function are appropriately provided. Performance measures cannot be obtained until the physical architecture is introduced; it provides the information needed to compute performance measures.

Physical Architecture

To complete the analysis phase of the procedure, the physical architecture needs to be developed. There is no standardized way to represent the physical systems, existing ones as well as planned ones that will be used to implement the architecture. They range from wiring diagrams of systems to block diagram representations to node models to organization charts. While there is not much difficulty in describing in a precise manner physical subsystems using the terminology and notation of the particular domain (communication systems, computers, displays, data bases), a problem arises on how to depict the human organization that is an integral part of the information system. The humans in the organization can not be thought simply as users; they are active participants in the workings of the information system and their organizational structure that includes task allocations, authority, responsibility, reporting requirements, and so on, must be taken into account and be a part of the physical model description. This is an issue of current research, since traditional organizational models do not address explicitly the need to include the human organization as part of the physical system description.

Synthesis

Once the physical architecture is available, then the executable model of the architecture shown in Fig. 1 can be obtained. The process is described in Fig. 4 as the synthesis phase. The required inter-relationship between the functional and the physical architectures is shown by the bold two-way arrow. It is critical that the granularity of the two architectures be comparable and that the partitions in the hierarchical decompositions allows functions or activities to be assigned unambiguously to resources and vice versa. Once the parameter values and properties of the physical systems have become part of the data base of the executable model, performance evaluation can take place.

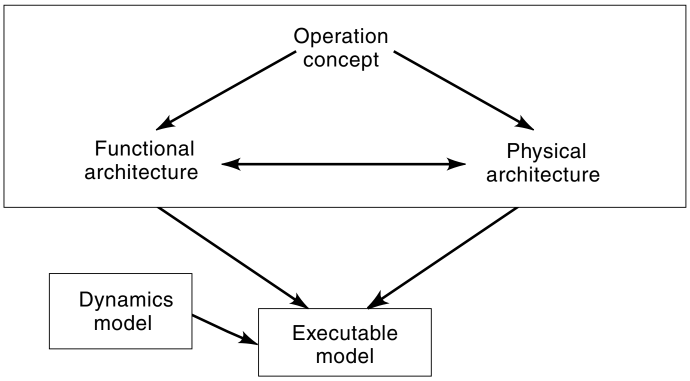

Figure 4. The synthesis phase. The executable model is obtained by assigning resources to functions and using the dynamics model to specify behavior.

Performance Evaluation

The executable model can be used both at the logical and behavioral level as well as the performance level. The latter requires the inclusion of the physical architecture. In one consistent architectural framework supported by a set of models, both requirement analysis, design, and evaluation can be performed. Furthermore, the process provides a documented set of models that collectively contain all the necessary information. Note that any changes made during the construction of the executable model must be fed back and shown in the static models.

Measures of performance (MOP) are obtained either analytically or by executing the model in simulation mode. For example, if deterministic or stochastic time delays are associated with the various activities, it is possible to compute the overall delay or to obtain it through simulation. Depending on the questions to be answered, realistic scenarios of inputs need to be defined that are consistent with the operational concept. This phase allows for functional and performance requirements to be validated, if the results obtained from the simulations show that the measures of performance are within the required range. If not, the systems may need to be modified to address the issues that account for the encountered problems.

However, the structured analysis approach is not very flexible; it cannot handle major changes that may occur during the development and implementation process. An alternative approach, that uses many of the same tools, has began to be used in an exploratory manner.

OBJECT ORIENTED APPROACH

This approach allows for the graceful migration from one option to another, in a rapid and low cost manner; it places emphasis on system integration rather than on doing one-of-a-kind designs.

The fundamental notion in object oriented design is that of an object, an abstraction that captures a set of variables which correspond to actual real world behavior (18). The boundary of the object that hides the inner workings of the object from other objects is clearly defined. Interactions between objects occur only at the boundary through the clearly stated relationships with the other objects. The selection of objects is domain specific.

A class is a template, description, or pattern for a group of very similar objects, namely, objects that have similar attributes, common behavior, common semantics, and common relationship to other objects. In that sense, an object is an instance of a class. For example, "air traffic controller" is an object class; the specific individual that controls air traffic during a particular shift at an Air Traffic Control center is an object—an instantiation of the abstraction "air traffic controller." The concept of object class is particularly relevant in the design of information systems, where it is possible to have hardware, software, or humans perform some tasks. At the higher levels of abstraction, it is not necessary to specify whether some tasks will be performed by humans or by software running on a particular platform.

Encapsulation is the process of separating the external aspects of an object from the internal ones; in engineering terms, this is defining the boundary and the interactions that cross the boundary—the black-box paradigm. This is a very natural concept in information system design; it allows the separation of the internal processes from the interactions with other objects, either directly or through communication systems.

Modularity is another key concept that has a direct, intuitive meaning. Modularity, according to Booch (19) is the property of a system that has been decomposed into a set of cohesive and loosely coupled modules. Consider, for example, the corporate staff, the line organization, and the marketing organization of a company. Each module consists of objects and their interactions; the assumption here is that the objects within a module have a higher level of interaction than there is across the modules.

In the context of object oriented design, hierarchy refers to the ranking or ordering of abstractions, with the more general one at the top and the most specific one at the bottom. An ordering is induced by a relation and the ordering can be strict or partial. In the object oriented paradigm, two types of ordering relations are recognized: aggregation and inheritance. Aggregation refers to the ability to create objects composed of other objects, each *part* of the aggregate object. The concept of aggregation provides the means of incorporating functional decompositions from structured analysis in the object oriented approach.

Inheritance is the means by which an object acquires characteristics (attributes, behaviors, semantics, and relationships) from one or more other objects (20). In single inheritance, an object inherits characteristics from only one other object; in multiple inheritance, from more than one object. Inheritance is a way of representing generalization and specialization. The navigator in an air crew inherits all the attributes of the air crew member object class, but has additional attributes that specialize the object class. The pilot and the copilot are different *siblings* of the air crew object class.

The object modeling technique (OMT) of Rumbaugh et al. (21) requires three views of the system: the object view, the functional view, and the dynamic view. The object view is represented by the object model that describes the structure of the system—it is a static description of the objects and it shows the various object classes and their hierarchical relationships. The functional view is represented in terms of data flow diagrams, an alternative to IDEF0, that depict the dependencies between input data and computed values in the system. The dynamic view is represented in terms of state transition diagrams. While these three views are adequate for object oriented software system design, they are not sufficient to represent an architecture and answer user's questions. As in the structured analysis approach, an executable model is needed to bring them all together and to provide a means for performance evaluation.

The Object View

The object view presents the static structure of the object classes and their relationships. The object view is a diagram that is similar to the data model, but in place of the data entities there are object classes. An object class is depicted by a box divided into three parts: the top part contains the name of the class; the second part contains the attributes (they are the data values held by all the objects in the class); and the third part contains the class operations. These are the functions or transformations of the class that can be applied to the class or by it.

The lines connecting the object classes represent relationships. These relationships have cardinality (one to one, one to many, etc.). In addition to the generalization and inheritance relationships, the lines also represent associations: they show how one class accesses the attributes or invokes the operations of another.

The Functional View

The *functional view* consists of a set of data flow diagrams that are analogous to the activity models in structured analysis. A data flow diagram, as used in the object modeling technique, depicts the functional relationships of the values computed by the system; it specifies the meaning of the operations defined in the object model without indicating where these operations reside or how they are implemented. The functions or operations or transformations, as they are often called, are depicted by ovals with the name of the transformation inscribed in them, preferably as a verb phrase. The directed arcs connecting transformations represent data flows; the arc inscriptions define what flows between the transformations. Flows can converge (join) and diverge (branch). A unique feature of data flow diagrams is the inclusion of data stores which represent data at rest—a data base or a buffer. Stores are connected by data flows to transformations with an arc from a store to a transformation denoting that the data in the store is accessible to the transformation, while an arc from a transformation to a store indicates an operation (write, update, delete) on the data contained by the store. Entities that are external to the system, but with which the system interacts, are called terminators or actors. The arcs connecting the actors to the transformations in the data flow diagram represent the interfaces of the system with the external world. Clearly, data flow diagrams can be decomposed hierarchically, in the same manner that the IDEF0 diagram was multileveled.

While data flow diagrams have many strengths such as simplicity of representation, ease of use, hierarchical decomposition, and the use of stores and actors, they also have weaknesses. The most important one is the inability to show the flow of control. For this reason, enhancements exist that include the flow of control, but at the cost of reducing the clarity and simplicity of the approach.

The Dynamics View

The dynamics view in OMT is similar to the one in structured analysis—state transition diagrams are used to show how events change the state of the system. The rules that govern the operations of the system are not shown as an independent model, but are integrated in the dynamics model.

A final construct that describes the trajectories of the system using events and objects is the event trace. In this diagram, each object in the object view is depicted as a vertical line and each event as a directed line from one object to another. The sequencing of the events is depicted from top to bottom, with the initiating event as the topmost one. The event traces characterize behaviors of the architecture; if given, they provide behavioral requirements; if obtained from the executable model, they indicate behavior.

The Executable Model

The three views, when enhanced by the rule model embedded in the state transition diagrams, provide sufficient information for the generation of an executable model. Colored petri nets can be used to implement the executable model, although the procedure this time is not based on the functional model. Instead, the object classes are represented by subnets that are contained in "pages" of the hierarchical colored petri net. These pages have port nodes for connecting the object classes with other classes. Data read through those ports can instantiate a particular class to a specific object. The operations of the pages/object subnets are activated in accordance with the rules and, again, the marking of the net denotes the state of the system.

Once the colored petri net is obtained, the evaluation phase is identical to that of structured analysis. The same analytical tools (invariants, deadlocks, occurrence graphs) and the same simulations can be run to assess the performance of the architecture.

UML

More recently, the Unified Modeling Language (UML) has been put forward as a standard modeling language for object-oriented software systems engineering (22). The language incorporates many of the best practices of industry. What is particularly relevant to its future extension beyond software systems-to-systems engineering is the inclusion of a large number of diagrams or views of the architecture. It includes use cases which describe the interaction of the user with the system, class diagrams that correspond to the object view in OMT, interaction diagrams which describe the behavior of a use case, package diagrams which are class diagrams that depict the dependencies between groups of classes, and state transition diagrams. The proposed standardization may provide the necessary impetus for developing system architectures using object-oriented methods.

SUMMARY

The problem of developing system architectures, particularly for information systems, has been discussed. Two main approaches, the structured analysis one with roots in systems engineering, and the object oriented one with roots in software system engineering, have been described. Both of them are shown to lead to an executable model, if a coherent set of models or views is used. The executable model, whether obtained from the structured analysis approach or the object oriented one should exhibit the same behavior and lead to the same performance measures. This does not imply that the structure of the colored petri net will be the same. Indeed, the one obtained from structured analysis has a strong structural resemblance to the IDEF0 (functional) diagram, while the one obtained from the object oriented approach has a structure similar to the object view. The difference in the structure of the two models is the basis for the observations that the two approaches are significantly different in effectiveness depending on the nature of the problem being addressed. When the requirements are well defined and stable, the structured analysis approach is direct and efficient. The object oriented one requires that a library of object classes be defined and implemented prior to the actual design. If this is a new domain, there may not exist prior libraries populated with suitable object classes; they may have to be defined. Of course, with time, more object class implementations will become available, but at this time, the start-up cost is not insignificant. On the other hand, if the requirements are expected to change and new technology insertions are anticipated, it may be more effective to create the class library in order to avail the systems engineering team with the requisite flexibility in modifying the system architecture.

BIBLIOGRAPHY

1. E. Rechtin, *Architecting Information Systems,* Englewood Cliffs, NJ: Prentice-Hall, 1991.
2. E. Rechtin and M. Maier, *The Art of Systems Architecting,* Boca Raton, FL: CRC Press, 1996.
3. E. Rechtin, The art of systems architecting, *IEEE Spectrum.,* **29** (10): 66–69, 1992.
4. E. Rechtin, Foundations of systems architecting, *J. NCOSE,* **1**: 35–42, 1992.
5. D. N. Chorafas, *Systems Architecture and Systems Design,* New York: McGraw-Hill, 1989.
6. A. P. Sage, *Systems Engineering,* New York: Wiley, 1992.
7. A. H. Levis, *Lecture notes on architecting information systems,* Rep. GMU/C3I-165-R, Fairfax, VA: C3I Center, George Mason Univ.
8. D. A. Marca and C. L. McGowan, *Structured Analysis and Design Technique,* New York: McGraw-Hill, 1987.
9. E. Yourdon and L. Constantine, *Structured Design,* New York: Yourdon Press, 1975.
10. P. Ward and S. Mellor, *Structured Development of Real-time Systems,* New York: Yourdon Press, 1986.
11. T. DeMarco, *Structured Analysis and Systems Specification,* Englewood Cliffs, NJ: Prentice-Hall, 1979.
12. C. Gane and T. Sarson, *Structured Systems Analysis: Tools and Techniques,* Englewood Cliffs, NJ: Prentice-Hall, 1978.
13. A. Solvberg and D. C. Kung, *Information Systems Engineering,* New York: Springer-Verlag, 1993.
14. G. L. Sanders, *Data Modeling,* Danvers, MA: Boyd & Fraser, 1995.
15. E. Yourdon, *Modern Structured Analysis,* Englewood Cliffs, NJ: Yourdon Press, 1989.
16. R. McLeod, Jr., *Systems Analysis and Design,* Fort Worth, TX: Dryden, 1994.
17. K. Jensen, *Coloured Petri Nets,* New York: Springer-Verlag, 1992.
18. A. P. Sage, Object oriented methodologies in decision and information technologies, *IEEE Trans. Syst., Man, Cybern.,* **SMC-19**: 31–54, 1993.
19. G. Booch, *Object-oriented Analysis and Design,* Redwood City, CA: Benjamin/Cummings, 1994.
20. E. V. Berard, *Essays on Object-oriented Software Engineering,* Englewood Cliffs, NJ: Prentice-Hall, 1993.
21. J. Rumbaugh et al., *Object-oriented Modeling and Design,* Englewood Cliffs, NJ: Prentice-Hall, 1991.
22. M. Fowler and K. Scott, *UML Distilled,* Reading, MA: Addison-Wesley, 1997.

ALEXANDER H. LEVIS
George Mason University