An application analyzer

by R. Ambrosetti
T. A. Ciriani
R. Pennacchi

An interactive tool, aimed at supporting the application user/analyst in specifying and analyzing a business area, is presented. The features of the tool, named the Application Analyzer/Experimental, are described both in their theoretical foundations and their actual implementation. A brief description of the architecture of the tool and its internal structure is given. A review of the main concepts of the application development area is also included. The follow-on of the prototype described here is the program offering known as System A.

Only 20 or 25 years ago, implementing software was synonymous with writing programs. That led computer scientists to focus their attention on the problem of defining algorithms and procedures in a complete and unambiguous way. It was a time in which technical meetings and symposia had, as an almost unique subject, "language," that is, a coding tool able to describe (as easily and as fast as possible) "how" to do something.

As time passed, user needs grew. A scenario that often resulted was humorously presented by E. Yourdon:[1]

> "The boss dashes in the door and shouts to the assembled staff: 'Quick, quick! We've just been given the assignment to develop an on-line order entry system by next month! Charlie, you run upstairs and try to find out what they want the system to do—and, in the meantime, the rest of you people start coding or we'll never get finished on time!'"

Such problems modified the scenario and, step by step, a software package tended to become a multilevel system, having a very complex nature, that modeled an organizational structure. Concurrently, increasing attention was given to specifying and designing the data processing model.

We are now in a kind of "renaissance" in which computer science is moving from its "ancient" time, and new concepts are becoming sound disciplines able to specify, analyze, and document the structure of a business area and the flow of information within it.

In the current literature, many terms are used in referring to the above activities. Examples are System or Process Analysis, Application Development, Business Area Analysis, etc. In this paper, the terms "application," "business function," and "information system" are synonymous, as are "function" and "process."

Even though no commonly accepted terminology exists, it is generally recognized that the following activities are required to develop an application:

1. Collect user requirements (exactly).
2. Build abstract systems[2] able to model (accurately) the user organization,[3] the data bases, and the network connecting data and processes.
3. Map these abstract systems onto data processing systems (efficiently).
4. Implement suitable software packages and physical data bases (correctly).

The words in parentheses emphasize that in each phase a specific aspect needs to be stressed to get the best result: *logic consistency* in requirements specification; *accuracy* and *completeness* in modeling;

efficiency in data processing systems design; and *correctness* in software implementation.

Many methodologies have been developed to address these challenges based on theoretical approaches like Structured Analysis (Yourdon-Constantine), Input-Output Data Structures (Jackson), Structured Systems Design (Warnier-Orr), Logic-Flow Techniques (HIPO), Data-Flow Networks (De Marco), Formalization (Jones), etc. It is beyond the scope of this paper to give a description of all these methodologies; general presentations[4,5] and detailed discussions[6–14] on them can be found in the current literature.

APAX is an interactive tool.

We shall limit our considerations of the methodologies because up to now not one of them has become dominant, and this situation is likely to continue in future years. Users have different reasons and needs, and, in addition, no technique is optimal in every case. Such a view is clearly demonstrated by the current literature and is supported by several authors.[15]

The *Ap*plication *A*nalyzer/E*x*perimental (APAX[16] as it is called hereafter) has been designed as a tool for specifying, analyzing, and documenting a business area, requiring no assumptions of predefined procedural schemes; it is a flexible tool open to any preferred user approach. APAX is essentially a series of visual display screens, each one interactively offering facilities (*operators*—listed in Appendix A) that support the user in analyzing the application, the component functions at any logical level, and the data.

APAX design perspective

In designing APAX, the following objectives were considered:

1. Support the user's "natural" view in specifying and analyzing a business area. By this approach, the user is not required to use formal languages nor to learn syntactic symbols that are foreign to his knowledge and experience.
2. Produce unambiguous machine-readable documentation for data processing experts. The greatest effort in implementing APAX was devoted to make it not only a tool for defining requirements and designing architectures, but also a bridge between the application analyst and the data processing expert.

APAX is therefore an interactive tool addressed to application users/analysts, for the first objective, and to data processing experts (in partnership with application users/analysts), for the second one.

In implementing APAX, highest priority was given to the functional characteristics related to generality, usability, and documenting capabilities.

Inherent generality means that no specific methodology of analysis is recommended or required: the user is allowed to execute top-down analysis or bottom-up synthesis in defining both processes and data structures and can perform his job by starting either from data or from functional considerations.

In order to support the different methodologies, the structure of the processes is shown in three ways to highlight the different aspects of the analysis: *data flow* (graphic display), *structured function tree* (indented list), and *logic flow* (pseudo-code-like representation). These three different types of results relate to the need of modeling a business area both as a set of data flows connecting couples of processes and as a sequence of processes connected to one another by logical relationships.

Error messages related to input/output inconsistencies are also given by APAX during the analysis.

To reach maximum usability, APAX was designed in such a way that the only competence needed relates to the application to be analyzed. Early experience has shown that a one-day training period consisting of a lecture and "hands-on" experience is sufficient to learn how to use APAX productively.

The first objective of the design of the screens with the use of color was to give the greatest amount of information to the user in the most immediate and understandable way.

For documentation purposes, APAX not only generates printed reports of any performed analysis, but

Figure 1 The data structure "Employee Register"

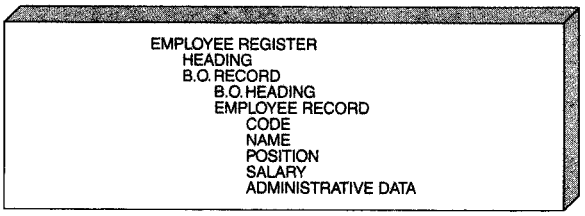
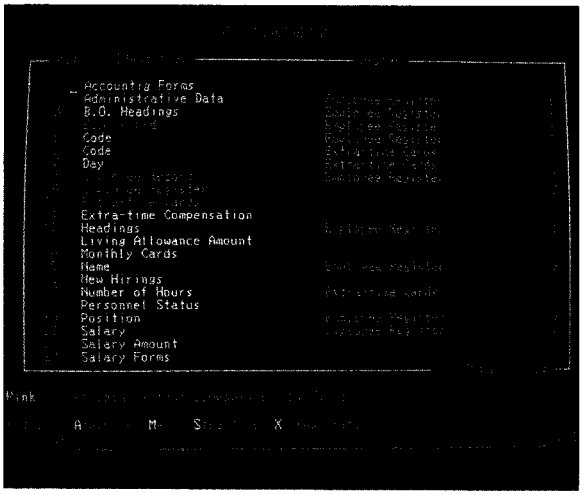


Figure 2 Data Universe of "Salary Application"



also allows the user to produce memos related to every process or every data structure involved in the application.

Basic concepts

Since there is no commonly accepted terminology in the area addressed by APAX, and since single terms often have multiple meanings, this section defines the principal concepts used in APAX. To better clarify these concepts, we shall refer to pictures of the related APAX screens obtained during the analysis of an application called "Salary Application."

In analyzing a business area, three main concepts are to be considered:

- *Data*
- *Functions*
- *Relationships* between data and functions

It is necessary to assign precise meanings to these terms to avoid misunderstandings when describing the APAX architecture and its related philosophy.

Data. In general terms, an application consists of a set of operations to identify, classify, store, and retrieve information (*input*), which is then processed to generate other information (*output*). In order to store and subsequently retrieve an item of information, an identifier (*name*) must be associated with it.

A *data element* is an elementary item of information which has a name and a value, e.g.,

(name) (value)
ADDRESS: '9045 Lincoln Boulevard, Los Angeles, CA 90045'

In real-life applications, information with complex structures must also be considered. Therefore, we define, in general, a *data structure* in the following way:

Data structure ≡ <Ordered set of data elements associated with a name>, <Ordered set of data structures associated with a name>.

This recursive definition lets us describe the very complex structures typical of real life. As an example, we can have

EMPLOYEE REGISTER: HEADING, B.O. RECORD
B.O. RECORD: B.O. HEADING, EMPLOYEE RECORD
EMPLOYEE RECORD: CODE, NAME, POSITION, SALARY, ADMINISTRATIVE DATA

Here, EMPLOYEE REGISTER, B.O. RECORD, and EMPLOYEE RECORD are defined as data structures, whereas HEADING, B.O. HEADING, CODE, NAME, POSITION, SALARY, and ADMINISTRATIVE DATA are data elements.

When a data structure (or data element) D_1 is part of another data structure D, a relation between D_1 and D exists. Such a relation is expressed by saying that D_1 is a *component* of D. In the above example, the data element NAME is a component of EMPLOYEE RECORD, and the data structure EMPLOYEE RECORD is a component of EMPLOYEE REGISTER.

In the following, the general term "data" is used to mean either data elements or data structures.

When, for a given D, no other D′ exists such that D is a component of D′, then D will be called the

master data. Referring to the above example, we see that EMPLOYEE REGISTER is a master datum, whereas NAME is not since it is a component of EMPLOYEE RECORD. For the same reason EMPLOYEE RECORD is not a master datum.

We shall call the *structure* of D the ordered set of the names of its components. The structure of the data EMPLOYEE REGISTER defined above is given in Figure 1.

Data can

- Belong permanently to the information system (in this case, called *files* or *system data*)
- Be temporarily passing through the system (called *transactions*)

We shall call the *Data Universe* U_D of the application A at the time T the set of all the data existing in A at that time. The set U_D will also be called the *status* of the system "Application A" at the time T.

An example of a Data Universe is shown in Figure 2. It gives not only a list of all the data but also auxiliary information: the files are marked by an "F"; the color pink identifies the names that refer to data elements (with no data components).

When a name identifies a component of a master datum, this master datum is written close to the name to avoid possible ambiguities generated by using the same names in different data structures.

Functions. We refer to the term *function* to indicate a process (storing, retrieving, and computing) acting on data in order to derive from them other data that can replace them completely or partially.

An application is composed of a set of functions (or *processes*), considered as the *first-level components* of the application itself, allowing us to obtain the final *output data* of the application from the initial *input data.*

Figure 3 shows the application "Salary," the first-level components of which are

1. Employee Register Up-dating
2. Salary Procedure
3. Accounting and Statistics
4. End-year Procedure

Of course, the same kind of decomposition can be repeated on each component of the application, giving rise to the *second-level components.* This process, repeated *n* times, leads to the definition of functions that are the *Nth-level components* of the application. Figure 4 shows the components of the function "Salary Procedure," which is a component of "Salary Application."

This sequence of steps is called, in the following, the *analysis* of the application, and each step is the analysis of a component function. The analysis ends

Figure 3 "Salary Application"; first-level components

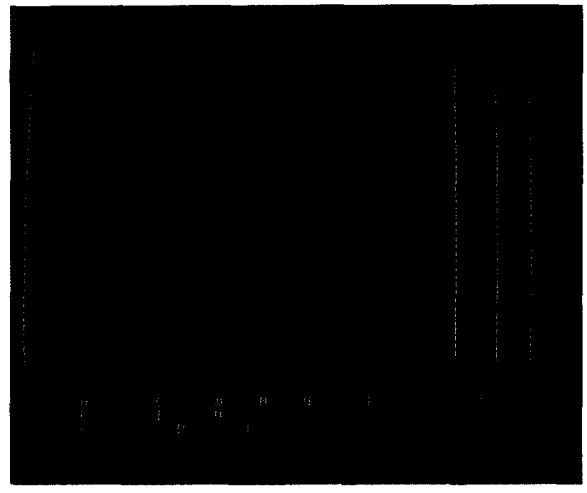


Figure 4 Function "Salary Procedure"; first-level components

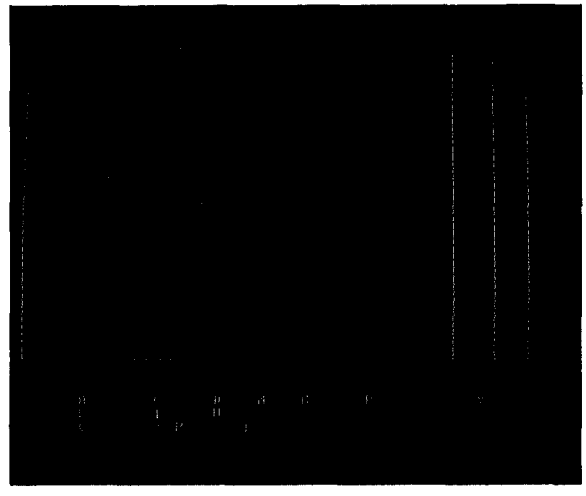
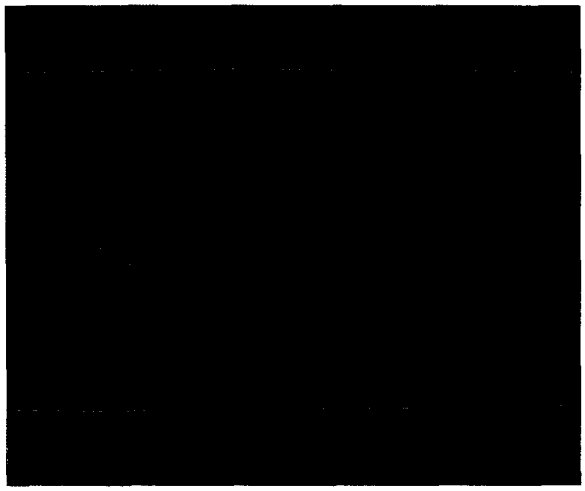


Figure 5 Structure of "Salary Application"



when the last functions defined are such that no further decomposition is considered to be useful by the user.

The result of the analysis of a function is shown in Figure 5. We shall call it the *structure* of the function.

The similarity of this concept to the one defined above about data is evident.

The lowest-level components of an application will be called *primitive functions*. A primitive function in application A is a component of A (of any logic level) for which the user decides that no further decomposition is necessary. In the following, a primitive function will also be called a *block*.

A function F is defined as primitive in three cases:

1. Temporarily, when the analysis of A is not yet completed and the user prefers to postpone the analysis of F. We shall not examine this case in detail as an intermediate step of the analysis path.
2. When F has such a simple internal logic that it can be easily understood or coded/executed. In this case, the complete specification of F requires
   - The identification of input/output data
   - The specification of the input/output data physical attributes
   - An explicit definition of the set of conditions (if any) affecting its execution. This set is deduced from all the conditions assigned in the previous steps of analysis.
   - The description (by natural or formal language) of the internal logic.
3. When F is a *generalized function*, that is, a package of existing software which requires only the allocation of the proper data to its input/output ports and the explicit definition of the set of conditions (if any) affecting its execution.

The primitive functions play a primary role in the development of an application. In fact, when the analysis work is finished, the complete application is described by a structured set of blocks. Their implementation is equivalent to the implementation of the application itself.

The structure of a function integrated with logical conditions is called the syntax of the function.

From the above, it should be clear that the attribute "primitive" is not an intrinsic property of the function itself but depends on the stage of the analysis, the user's understanding at that time, the language used, etc. Also, any type of function (even a very complex one) for which a suitable software package is available can be defined as a primitive function.

Since the structure of a function includes all of the possible components (executable in all possible cases), it does not always coincide with the sequence of operations that are actually performed to get the final output data from the initial input data. This difference originated from the fact that the path followed can depend on the characteristics (or *predicates*) of some data.

If the components of a function are to be performed sequentially (only one path exists), those components are called "sequential functions." If only some of them are performed—the choice depending on the results of tests made on some data—different paths exist.

The structure of a function, completed by the specification of all conditions affecting the execution of its components, is called the *syntax* of the function.

As shown in Jackson's Structured Programming Theory,[9] multiple paths exist only in three cases:

1. When the execution of a component of F depends on predicates of some data (the component is a *conditioned function*).
2. When a component of F is conditioned by predicates opposite those of another predefined conditioned component F′ (the component is the *reverse function* of F′).
3. When a component of F is a sequence of identical functions to be executed on a set of data while given data predicates hold (the component is an *iterated function*).

When the function F has conditioned or iterated components, its syntax does not coincide with its structure, because in the component ordering, the different combinations of the conditional data predicates cause some "jumping" or "stopping" in the component sequence.

Those situations are represented in high-level programming languages by statements like

IF . . . THEN . . .
ELSE . . .

or

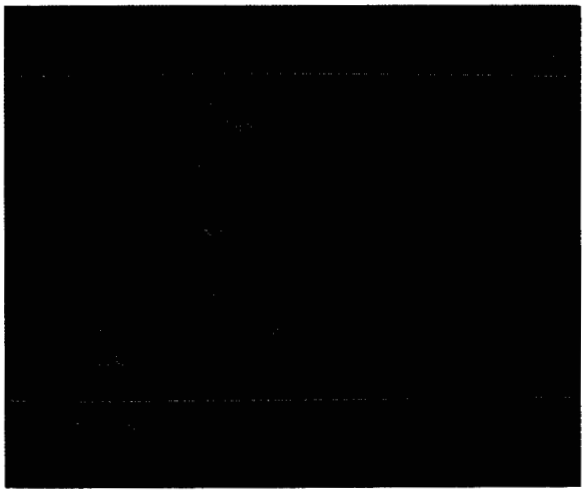
DO . . . WHILE

Summarizing, a sequential component is always executed. A conditional or reverse component is executed according to the predicates of a set of data. Defining a component as iterated is a short way to mean that it must be executed several times if an execution condition holds.

With use of pseudo-code symbols, the APAX screen shown in Figure 6 gives a view of the syntax of "Salary Application," automatically obtained from the information given by the user in the screen enabling the analysis.

By comparing Figures 5 and 6, we see the difference between the structure and the syntactic view of an application. The first one shows all the functions together with their components; the second one highlights the conditions affecting the logic flow of the application. In other words, the structure describes all the functions that are considered within an application at each level, whereas the syntactic view specifies when those functions are to be executed.

Figure 6 Syntactic view of "Salary Application"



Data/functions relationships. In the analysis of an application, each component function (regardless of its level) should have as input either initial input data or data coming from another previously executed function. In other words, the data come either from the initial status of the system or from *originator* functions. Similarly, it can be stated that data are used either by the final status of the system or by *receiver* functions.

From the above, it follows that the system "Application A" can be conceived either as a structured set U_F of functions (*Function Universe* of A) connected by a *data flow*, or as several Data Universes, each one generated from the preceding one through the execution of a sequence of functions (*logic flow*).

Let D_1 and D_2 be two items of data belonging to the Data Universe of an application A. In the data flow of A, two different cases can apply:

1. D_2 exists independently from D_1. (D_1 is not needed to compute D_2.)
2. D_2 can be obtained only when D_1 is known. (D_1 is needed to compute D_2.)

In the second case, there is a "logical precedence" relationship between D_1 and D_2. An example of logical precedence is shown in Figure 7, where "Employee Record," "Extra-time Cards," and "Tax Coefficient Table" are needed to obtain "Salary Forms." The two relations "to be component of . . ." and "to be precedent of . . ." define the logical structure of the set of all the data in an application.

Figure 7 "Salary Application"; I/O data of the function "Salary Procedure"

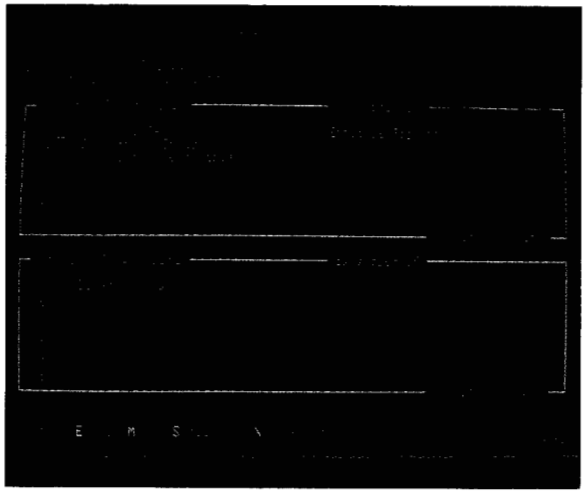


Figure 8 Application definition

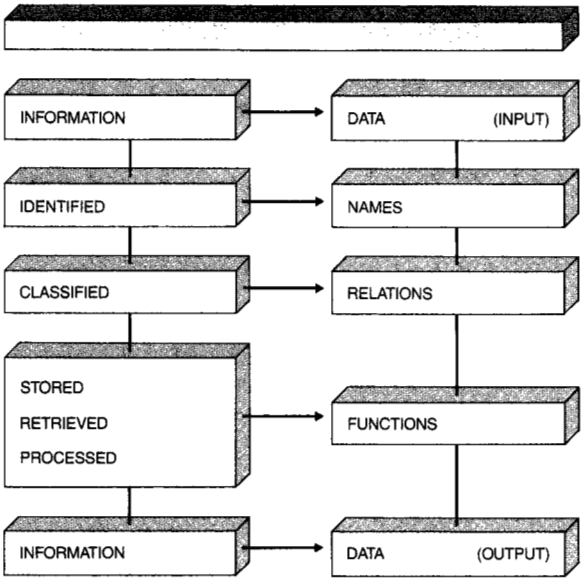


Table 1 Data and functions correspondence

Data	Functions
Data element	Primitive function
Data	Functions
Data components	Function components
Structure of data	Structure of function
Data predicates	Conditioned functions

Data and functions have a structural connection with the complete application, as shown by Figure 8.

Data/function duality. From the concepts defined in the preceding paragraphs, a correspondence appears, as shown in Table 1. This table suggests a kind of duality between data and functions, but a deeper analysis is needed to derive from it a more sound view of an application.

When an application has been completely specified, the result of the analysis is a system, the elements of which are data and functions connected by different types of relationships. A complete description of such a system leads to a description of all the elementary structures contained in it.

Let us examine two different cases.

1. Elementary structure DATA → FUNCTION → DATA
   This statement means to establish a relationship between two sets of data through a function (regardless of its logical level) that generates the second set from the first one. This approach leads to the methods that consider the functions as links between data (Jackson,[9] Warnier-Orr,[11] Myers,[12] etc.).
2. Elementary structure FUNCTION → DATA → FUNCTION
   This statement means to establish a relationship between two functions through a set of data (output of the first function and input to the second one). This approach leads to the methods that consider data as links between functions (Yourdon,[7] De Marco,[13] etc.).

In this way, the duality between these two elementary structures appears as the root of the duality between the different methods of application analysis.

To allow the user to apply the method that he prefers, in APAX both of these elementary structures are dynamically selected by the user, and their mutual consistency is automatically checked.

Flows within an application. The description of an application is generally given through two kinds of models:

1. By conceiving the application as a system having as its elements sequences of functions, each one generating data derived from other data. This model (logic flow) leads to the implementation of diagrams in which the functions, their order, and

the conditions of their execution are in full evidence. These diagrams are generally implemented by using graphical or semi-formal languages (HIPO diagrams, Petri networks, pseudo-code, etc.). Figure 6 is an example of logic flow in the form of pseudo-code.

2. By conceiving the application as a system in which a flow of data is defined starting from the initial input of the application. In this model, all the data are connected to the application final output by a network of transformations along which they pass (data flow).

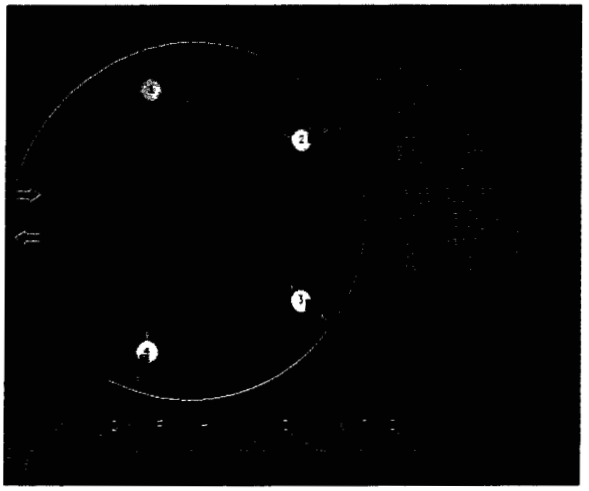
From our experience, graphic representation appears to be the clearest and the most useful way to document a data flow.

Figure 9 shows the same functional structure given by Figure 6 in the form of data flow. In it the big circle represents the "world" of the business area "Salary Application," the small circles numbered 1, 2, 3, 4 represent the component functions (the names of which appear at the top of the right side of the screen), and the data flows are represented by the arrows specifying the flow direction. The data identifiers on the right side of the screen allow a complete description of the connections among the component functions. The data coming from or going to the area external to the big circles are the input and output data of "Salary Application."

APAX is a series of visual display screens.

Logic flow and data flow of the same application are obviously connected to each other, but questions have been raised about the nature of their connection and their equivalence. We shall not examine this problem here, but clearly the above-defined duality ensures that the equivalence condition for the two models resides in the intrinsic self-consistency of the application (see Appendix B for more rigorous considerations).

Figure 9 Data flow of "Salary Application"



APAX architecture

Functional specifications. As was stated earlier, APAX is a series of visual display screens. Although the internals of APAX are strictly formal, from the user's view, APAX operates in a nonprocedural way and with no use of formalized languages. All data, functions, and relations involved in the description of an application are stored in a data base called the *repository* of the application.

Special care has been taken to maximize the efficiency and the usability of APAX. For example, color provides readable screens and is also used to give the greatest amount of information quickly and easily.

The focal point of this approach lies in the fact that the user builds up the application by defining its components: APAX allows at any time a comprehensive view of the structure and relationships of functions and data. Moreover, it checks the leveling consistency of the application in terms of those relationships.

All these operations are carried out at any logical level, down to the level at which the functions are defined as blocks, after which the analysis is complete.

Data base structure. Two kinds of basic objects are in the APAX repository: functions and data. These objects are represented by records of a direct access data base in which pointers define the links among

Figure 10 Data flow

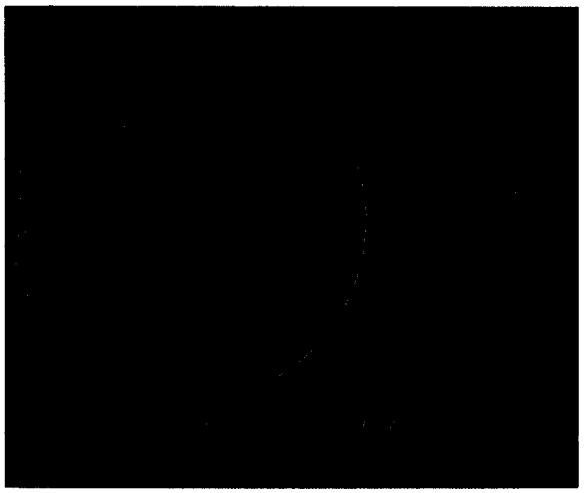
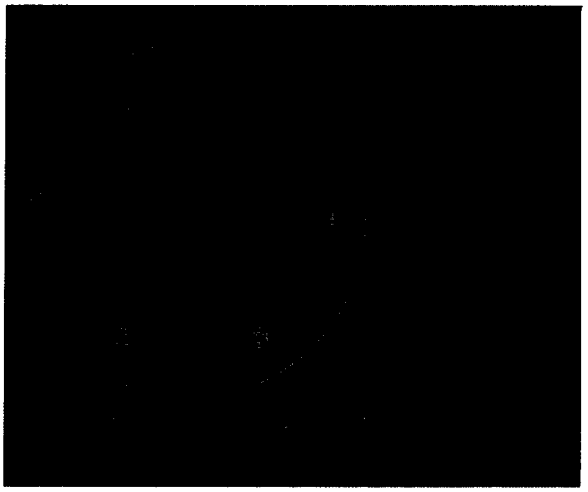


Figure 11 Data flow with function highlighted



the data base elements. The components of such a data base are as follows:

- *Repository anchor point record*—It contains the starting point of the application chains. Also stored here is the starting point of the available record chain in order to allow flexible maintenance of the repository space and re-use of erased records.
- *Application record*—Besides the application name and password, the application pointers are stored here. These pointers establish the correct connections to the Master Data List record, to the record of the first-level component functions, and to the record containing the application I/O data.
- *Function records*—This record contains the following information on each function: function description, random code (used by hashing tables), pointer to the function to which it belongs, list of the pointers to the records containing the component functions of the next level, and list of pointers to the records containing I/O data of the function itself.
- *Data records*—Besides the data description, the pointers to the functions using it and to the data involved in its structure are stored here.

Hashing tables. In APAX execution, checking is frequently required to determine if some "objects" (data or functions) are already included in the repository, either explicitly (e.g., when check operators are performed) or implicitly (e.g., when I/O data are defined and the system itself checks for their existence).

To reduce response time because of the frequency of these inquiries, a random technique[17] is used that reduces the number of scans in the repository. In this technique, a numerical random code, obtained directly from the name, is associated with each object. This numerical code locates a table position in which the key corresponding to the actual record is stored. A second table exists in which the keys of synonym records are stored.

The main operations that can be performed by the hashing tables are insertion, deletion, and inquiry.

Since the architecture of the APAX data base has a fully open and flexible design, APAX could be connected in the future with specific software products such as the Structured Query Language/Data System (SQL/DS).

APAX operators

APAX carries out an interactive dialogue through a series of "operators," each of which is implemented by actions performed through the display terminal. For the sake of classification, operators can be grouped in four families, depending on the "objects" on which they act: (1) operators on the whole application, (2) operators on each function, (3) operators on each data structure or data element, and (4) operators that cross.

The operators belonging to the first three groups permit the user to initiate, update, inspect, analyze,

Figure 12 Data flow with data element highlighted

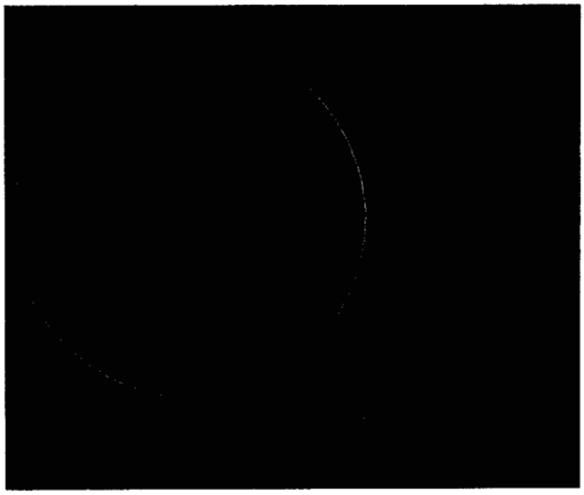
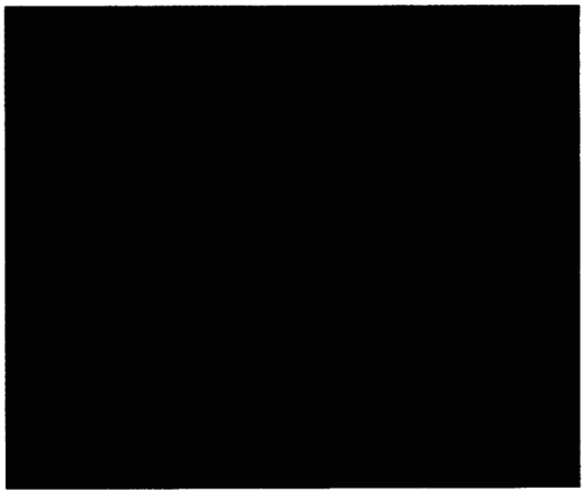


Figure 13 "Salary Application"; check against formal errors



or synthesize every component of function or data. The operators of the fourth group perform actions that relate to functions and data together.

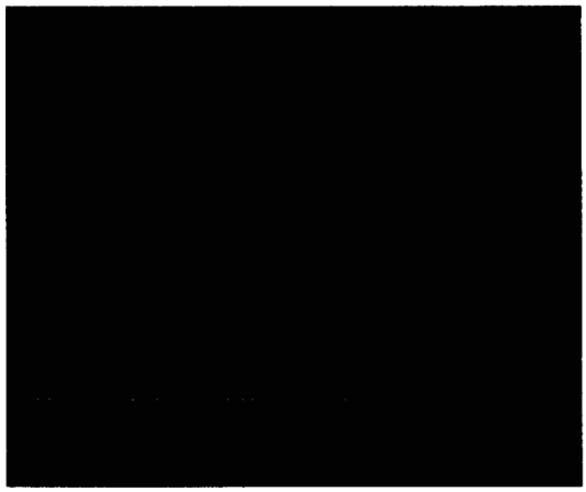
In designing the operators, special care has been taken in minimizing all the user actions that are likely to generate errors. In particular, copying actions have been avoided as much as possible. For instance, when two functions have the same input (or output) data, it is possible to assign automatically the input (or output) of the first function to the second one. This is shown in Figure 7, where the Copy operator can be applied through the functional key PF9.

Another operator designed to avoid copying errors is the facility "Pick" appearing in the Action line of the screen in Figure 3. With it, any data name appearing in the Data List (Data Universe) shown in Figure 2 can be automatically assigned as input (or output) of the analyzed function just by writing "i" (or "o") close to it.

Display of Data Flow Network is another operator, which produces the screen seen in Figure 10. Similar to it is the operator for "zooming in" on a component data flow. It allows the display of the inner data flow of every component function appearing in the initial network.

To improve the readability of the network, a *highlighting* facility has been implemented to show all the data flowing through any selected function (Figure 11), or to the data flows containing a selected data element (Figure 12).

Figure 14 "Salary Application"; diagnostic messages



Three operators have been specifically implemented for checking purposes: Check Formal Errors, Diagnostic Message List, and Automatic Correction. They permit detection of any formal inconsistency in the data flow of every function, and diagnostic messages specify where the inconsistencies are. Depending on the nature of an error, a particular type of correction is proposed to the user, and, if accepted, is automatically performed. Figures 13 and 14 show the screen "Check on Data Flows" and a sample of diagnostic messages.

A complete list of APAX operators is given in Appendix A.

Concluding remarks

It is commonly recognized that the application modeling area is one of the challenges of today. APAX is a tool that can be used by the user/analyst in concert with the application analyst in this difficult job. In APAX we have tried to build a tool that is easy to use and "neutral," as much as possible, to the different methodologies existing in this area.

The APAX conceptual starting point was general system theory and set theory rather than computer science. The aim of this approach was to obtain a structure broad enough to be able to analyze and specify general systems and then to apply it to those special systems that represent business areas.

As a result, we obtained a general flexible tool on which several improvements have been possible without major difficulties. It is hoped that the characteristics of generality and flexibility will enable APAX to connect to other software packages through bridges able to transfer functions and data. Synergistic effects are expected from such connections so as to improve the user's productivity. Furthermore, the use of new graphic facilities will provide better usability and enhanced dialogues.

Acknowledgment

An early version of the APAX prototype was shown to people involved in application analysis. From them, we received constructive criticism and suggestions that we used in the second prototype. We thank all of them. Particular thanks are due to Sue Smith of the IBM National Accounts Division Headquarters and to Marilyn Parker of the IBM Los Angeles Scientific Center for their extremely effective support.

Appendix A

Following is a list of the APAX operators.

- Operators on the complete application
  1. Nominate New Application
  2. Cancel Application
  3. Rename Application
  4. Assign Password
  5. Start Analysis (Editing)
  6. Display I/O Data of Application
  7. Define I/O Data of Application
  8. Cancel I/O Data of Application
  9. Write User Comments
  10. Restore User Comments
  11. Save User Comments
  12. Display/Print User Comments

- Operators on each function
  1. Display Components
  2. Define Components (Analyze Function)
  3. Cancel Components
  4. Update Components
  5. Display I/O Data
  6. Define I/O Data
  7. Cancel I/O Data
  8. Copy I/O Data from Another Function
  9. Display Function Tree
  10. Display Function Pseudo-Code (single level)
  11. Display Function Pseudo-Code (nested up to given depth)
  12. Function Synthesis (Move Function)
  13. Write User Comments (Memo) on Functions
  14. Write User Comments (Restore)
  15. Write User Comments (Save)
  16. Display User Comments
  17. Display Alphabetic List of All Functions

- Operators on each data
  1. Nominate Master Data
  2. Cancel Master Data
  3. Update Master Data
  4. Data Synthesis (Hang Data)
  5. Display List of Master Data
  6. Display Structure of Data
  7. Define Structure of Data (Analyze Data)
  8. Update Data
  9. Cancel Data
  10. Display Alphabetic List of All Data
  11. Write User Comments (Memo) on Data
  12. Write User Comments (Restore)
  13. Write User Comments (Save)
  14. Display User Comments

- Cross operators
  1. Check Formal Errors
  2. Diagnostic Message List
  3. Automatic Correction
  4. Display of Data Flow Network (for given function)
  5. Display of Data Flow Network (zooming in on a component)
  6. Display of Data/Function Cross-Reference
  7. Put Data (from data list) as I/O of a Function

Below is a brief description of some operators, the

effects of which are not immediately clear from the above list of names.

Application of I/O Data. Initial and final data of the application are entered, changed, or erased.

Master Data List. The master data of the application are defined and handled. This operator is connected to the following related operators: Analysis of Data, Cross-Reference, Structure of Data, Data Synthesis (Hang), and User Comments. The Hang operator allows data to be allocated as components of another data structure (bottom-up method in data definition).

Analysis of Data. Used to define, insert, or change the components of a data structure.

Structure. The structure of data is shown, emphasizing the position within the structure of the master data to which it belongs.

Analysis of a Function. Allows the definition of the components of a function (or of the complete application). Through this operator, an application is analyzed following a top-down technique. Another operator (Move) can be used to allocate previously defined functions as the components of another (bottom-up synthesis).

Function I/O Data. This operator creates a relation between a function and a set of data, defining the set as input or output of the function. To simplify this assignment, the operator Copy is provided. Through Copy, the I/O data of another specified function are copied, avoiding the risk of using incorrect names.

Check Formal Errors. Any formal inconsistency in the data flow of a given function is detected. Diagnostic messages specify where the inconsistencies are. Automatic correction is proposed to the user, and, if accepted, performed.

Function Structure (Tree). The structure of a function is displayed as an indented list of all the components (up to the deepest level) of the function itself.

Function Syntax (View). The syntax of the function is described using the standard form of pseudocoding. Comments specifying the starting and ending points of each component are automatically added on request.

Data Flow Network. The data flow in a given function is displayed in graphic form. The function components are represented as circles and the data flows as vectors (arrows) connecting them. A full description of all the data and function names is given. A "highlighting" facility is also provided to better identify data, flows, and components on the screen. The facility for zooming in on a component appearing in the network is included.

User Comments. The user may insert in the documentation nonformatted comments related to any function or data structure.

Printed Reports. At any time of the analysis, the user can automatically generate output reports. The list of all the available output reports can be handled for printing, change, or deletion purposes.

Appendix B

Intuitive considerations have developed on data, functions, and their role in a business area analysis. These concepts are more rigorously specified here. Table 2 defines the symbols used in this discussion.

The data/function duality. Let Δ and Λ be a set of data and a language, respectively. We shall call function in Λ an operator

$$F : \Phi \subset \Pi(\Delta) \rightarrow \Pi(\Delta)$$

such that F is fully described by a finite number $n(F)$ of Λ statements.

$\Pi(\Delta)$ is here the power set of Δ, according to the fifth axiom of Zermelo-Fraenkel Set Theory.[18]

When $n(F) = 1$, F is said to be a primitive function in Λ.

Table 2 Symbol definition

Symbol	Definition
$A \cup B$	Union of set A and set B
$=$	Defined as
$A > B$	A follows B
$A < B$	A precedes B
$x \in A$	Element x belongs to set A
$A \subset B$	The set A is a subset of B
$\alpha : A \rightarrow B$	The operator α has domain A and range B
$\Rightarrow$	Implies
$\Pi(A)$	Power set of the set A
$\neg$	Not

When $n(F)$ is "large" (what "large" means is arbitrarily assumed by the user), it is convenient (or necessary) to describe it in Λ by a set of routines F_1, F_2, . . ., F_m, each of which is still a function (components of F).

In the concepts specified above ("data" and "functions"), two couples of operators have been implicitly defined. The first couple is

$$I_D : U_F \rightarrow \Pi(U_D)$$

$$O_D : U_F \rightarrow \Pi(U_D)$$

where $I_D(F)$ is the set of the input data and $O_D(F)$ is the set of the output data of F.

The second couple is

$$I_F : U_D \rightarrow \Pi(U_F)$$

$$O_F : U_D \rightarrow \Pi(U_F)$$

where $I_F(D)$ is the set of the functions having D as input and $O_F(D)$ is the set of the functions having D as output.

The two couples (I_D, O_D), (I_F, O_F) specify the duality between data and functions because their definitions can be deduced one from the other simply by interchanging the subscripts D and F. Therefore, all the concepts that are invariant in front of this duality can be translated from a set of functions into a set of data and vice versa. For example, the input and output data of a function are transformed by duality into the *cross-reference* of data, that is, the sets of functions that have them as input or output.

Flows in a function. Exploiting the duality, the control flow, and the data flow within a function can be better specified.

To better note the relationships between data flow and control flow, some more definitions are needed. We shall say that a function F″ *follows* a function F′ $(F'' > F')$ when

$$I_D(F'') \times O_D(F') \neq 0$$

Similarly, a data structure D″ *follows* a data structure D′ $(D'' > D')$ when

$$I_F(D') \times O_F(D'') \neq 0$$

Those definitions can be extended to the concept of *chain* linking functions or data. A sequence of functions $(F_i, (i = 1, \ldots, n)$ is linked by a chain when

$$F_{i+1} > F_i, \quad i = 1, \ldots, n-1.$$

In a similar way a sequence of linked data is defined.

Now let A be a business function having the Data Universe U_D and the Function Universe U_F, and let f be any component of A. From our general point of view, to *analyze* A means to define an operator

$$C : U_F \rightarrow \Pi(U_F)$$

and therefore a control flow model of f is given when C is given together with a partial ordering relationship ρ on C(f). (C(f) identifies the set of the components of f.)

We shall use the symbol

$$CF(f) \equiv (C(f),\rho)$$

to represent the control flow within the function f.

By applying the already noted duality between functions and data, we can give a similar definition for the data flow within f. When a function f is defined, input and output data are assigned to f and to its components. That means that in general an operator is defined on U_F which associates to any f a subset of U_D whose elements are all the data involved in the execution of f. Let such operator be

$$D : U_F \rightarrow \Pi(U_D)$$

Therefore, a data flow model of f is given when D is given together with a partial ordering relationship λ on D(f).

We shall use the symbol

$$DF(f) \equiv (D(f),\lambda)$$

to represent the data flow within the function f.

Since CF(f) and DF(f) are two different models of the same function f, it is evident that a connection should exist to avoid a basic inconsistency in the f definition. This connection exists through the two couples of operators I_D, O_D, and I_F, O_F already defined. In fact, given the operators C, I_D, and O_D, the elements of the set of data D(f) should exactly be (for consistency reasons) the elements of the union of the sets $I_D(g)$ and $O_D(g)$ when g is either f or one of the f components.

Formally, the relationship between the operators C and D is given by

$$D(f) = \cup(I_D(g) \cup O_D(g)) \text{ being } g \in C(f) \cup \{f\} \qquad (1)$$

The above condition is necessary for the consistency of the two models of f, but it is not enough because no ordering considerations have been involved in it until now. To complete the consistency conditions, the partial ordering relations ρ and λ also should be connected in some way.

The relation λ included in the above data flow definition allows us to decide, given d' and $d'' \in D(f)$, if $d' < d''$ ($d'' < d'$); that means that d' is needed to get d'' (or vice versa). This implies that any function having d' as output cannot follow (according to the ρ relation) any other function having d'' as input.

Therefore, for the consistency of the two models CF(f) and DF(f), the following relationship should hold:

$$f' \rho f'' \Rightarrow (\neg E d', d'') \mid ((d'' \lambda d'), (d' \in I_F(f')), (d'' \in O_F(f''))) \quad (2)$$

Summarizing, when the consistency relationships of (1) and (2) hold, CF(f) and DF(f) are two isomorphic models of f. The term "isomorphic"[19] is used here as it is in the First-Order Predicate Logic (here applicable since both the Data and Function Universes are necessarily finite).

Since the operators implemented in APAX essentially deal with the definition of the components of any function and of the data that are input or output, a proof of the validity of such an approach is equivalent to the proof that those operators lead to the definition of two isomorphic models of the application.

To this purpose, the following theorem is established:

Assume that the sets U_D, U_F and the operators C, I_D, O_D are as defined above; assume also that the relations ρ and λ are specified by chaining the components of a function and all the data involved in them (see preceding paragraph); then the relation ρ is a partial ordering if and only if such is also the relation λ.

Proof. Let ρ be a partial ordering relation. If λ is not, a couple of the data d', d'' should exist such that

$$d' \lambda d'' \quad \text{and} \quad d'' \lambda d' \quad (3)$$

The first means that two functions, f' and f'', exist so $d'' \in O_D(f'')$ and $d' \in I_D(f')$ and they are connectable by a function chain. That implies $f' \rho f''$. But starting from the second one of (3), we can state in the same way that $f'' \rho f'$, which is impossible since ρ is a partial ordering relation.

In a similar way, it can be proved that if λ is a partial ordering relation, the same ρ should be. The theorem is therefore proved.

From the above theorem it follows:

Corollary. If the analysis of a function f has been carried out (regardless of the data flow or control flow method) and its results imply a partial ordering of the data or of the functions involved, then the two models CF(f) and DF(f) are isomorphic.

But the basis on which APAX is implemented is an analysis that ensures a partial ordering on U_F or U_D. Therefore, the above corollary gives a proof of the validity of such an approach.

Cited references and notes

1. E. Yourdon and L. L. Constantine, *Structured Design*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1979).
2. R. Pennacchi, "Principles of an abstract theory of systems," *International Journal of Systems Science* **3**, No. 1, 1–11 (1972).
3. M. M. Parker, *Enterprise Information Analysis: Cost-Benefit Analysis of Information System Using PLS/PSA and the Yourdon Methodology*, IBM Corporation, Los Angeles Scientific Center Report G320-2716 (1982); available through IBM branch offices.
4. W. W. Cotterman et al., *System Analysis and Design: A Foundation for the 1980's*, Elsevier North-Holland, Inc., New York (1981).
5. L. L. Beck and T. E. Perkins, "A survey of software engineering practice: Tools, methods, and results," *IEEE Transactions on Software Engineering* **SE-9**, No. 5, 541–561 (1983).
6. D. T. Ross and K. E. Schoman, Jr., "Structured analysis for requirements definition," *IEEE Transactions on Software Engineering* **SE-3**, No. 1, 6–15 (1977).
7. Yourdon and Constantine, *op. cit.*, p. 4.
8. W. P. Stevens, *Using Structured Design*, John Wiley & Sons, Inc., New York (1981).
9. M. A. Jackson, *Principles of Program Design*, Academic Press, Inc., New York (1975).
10. V. J. Crandall, *Data Structured Systems Development Methodology*, Ken Orr and Associates, Inc., Topeka, KS 66607 (1982).
11. K. T. Orr, *Structured Systems Development*, Yourdon, Inc., New York (1977).
12. *HIPO—A Design Aid and Documentation Technique*, GC20-1851-0, IBM Corporation (1974); available through IBM branch offices.
13. T. De Marco, *Structured Analysis and System Specification*, Yourdon, Inc., New York (1978).
14. C. B. Jones, *Software Development: A Rigorous Approach*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1980).
15. Beck and Perkins, *op. cit.*, section entitled "Discussion and Conclusion," p. 553.

16. APAX is our writing of the ancient Greek word ἅπαξ, which means "all together" or "once at a time." It comes from an old Dorian verb πάγνυμι, meaning "to build something by bringing together different pieces."
17. R. L. Obermarck and R. K. Treiber, *Practical Uses of Hashing for Main Storage Searching*, Research Report RJ-3483, IBM Corporation, Research Division, 5600 Cottle Road, San Jose, CA 95193 (1983).
18. R. Rogers, *Mathematical Logic and Formalized Theories*, North-Holland Publishing Co., Amsterdam (1971), Chapter VII.
19. *Ibid.*, Chapter III.

Reprint Order No. G321-5229.

Rodolfo Ambrosetti *IBM Italy, Program Product Development Center, Viale Oceano Pacifico 73, 00144 Rome, Italy.* Mr. Ambrosetti joined IBM in 1977 with a particular interest in the fields of application development and mathematical programming. Since 1981 he has participated in the activity of the Advanced Technology Group at the PPDC in Rome, mainly in the area of application development techniques. Mr. Ambrosetti received a degree in mathematics from Rome University in 1973.

Tito A. Ciriani *IBM Italy, Program Product Development Center, Viale Oceano Pacifico 73, 00144 Rome, Italy.* Mr. Ciriani joined IBM Italy in 1963. He managed the IBM Pisa Scientific Center up to 1972, and after 1975 he was responsible for scientific relations for the IBM Italian Scientific Centers. In 1979, he joined the Program Product Development Center in Rome. At present he is in the Advanced Technology Group, where he is involved in experiments on advances in graphics and on application development methodologies. Mr. Ciriani received his university degree in electrical engineering from Genoa University in 1962.

Renato Pennacchi *IBM Italy, Program Product Development Center, Viale Oceano Pacifico 73, 00144 Rome, Italy.* Mr. Pennacchi joined IBM Italy in 1954. After a variety of assignments, he became manager of the IBM Italy Scientific Centers in 1969. In 1980 he joined the Program Product Development Center, where he is responsible for the Advanced Technology Group. His experience has been with system theory and application development methodologies. Mr. Pennacchi received his university degree in mathematics from Rome University in 1949.