ADISSA: ARCHITECTURAL DESIGN OF INFORMATION SYSTEMS BASED ON STRUCTURED ANALYSIS

PERETZ SHOVAL

Computer and Information Systems Program, Department of Industrial Engineering and Management, Ben-Gurion University of the Negev, P.O. Box 653, Beer Sheva 84105, Israel

(Received 12 September 1986; in revised form 11 September 1987)

Abstract—ADISSA is a methodology for Architectural Design of Information Systems and Software, based on Structured Analysis. The architectural design includes: (a) design of a tree-structured menu system, which interfaces between the users and the system—viewed as the "external architecture" of the system; and (b) design of the transactions of the system, composed of various functions which are activated in response to user needs and various events in the universe of discourse—viewed as the "internal architecture" of the system. The menu tree and the transactions are tied up with the other major components of the system architecture: the inputs and outputs, and the database. ADISSA methodology is fully compatible and consistent with Structured Analysis, and therefore the stages of system analysis and design are integrated into one complete process.

Key words: Architectural design, data flow diagram, development methods, software engineering, structured analysis and design, transactions, user interface.

1. INTRODUCTION

In formation systems development a distinction is usually made between the stages of analysis and design. In carrying out these stages in a structured manner, various tools and techniques (termed software engineering) have been developed. See [1–6] for surveys of such techniques.

1.1. Structured analysis

Among the techniques for the analysis stage that will be dealt with specifically is Structured Analysis (SA) [7, 8]. SA is based on the use of hierarchical data flow diagrams (DFD) which define and describe the various functions to be performed by the system, the data stores within the system, the external entities which interface with the system, and the data flows connecting the above-mentioned elements. In SA hierarchical decomposition of both functions and data is performed: at the top level DFD the major functions of the system are defined, and then each of the major functions is subsequently decomposed and separately described in a more detailed DFD. Further decomposition of functions continues until a level of elementary functions is achieved. Every DFD details the component sub-functions of its "parent" and the data flows which connect it with the other components (i.e. other functions, data stores, external entities) in the DFD. Rules for decomposition and leveling of DFDs guide the analyst in obtaining a hierarchy of consistent DFDs. The other products of SA, besides DFDs, are:

—A data-dictionary (DD) which documents the details of the DFDs by giving structured and semantic definitions for data flows and data stores.
—A database schema, consisting of normalized record types [7], which are generated out of the data elements that flow in and out of the data stores (from and to the elementary functions). Access paths from the functions to the respective record types are defined.
—A structured description of the elementary functions (minispecs) which describes the process logic within each of the elementary functions in the DFDs using structured techniques such as decision trees, decision tables and pseudocode. These description are the basis for program design and coding in the subsequent stages of system development.

The products of SA constitute the specifications of the system and are used in the next stage of development—system design.

1.2. Structure design

During the design stage the analyst uses the requirements defined during the analysis stage to create the blueprints of the proposed system. In system design a distinction is usually made between two stages: architectural design (also termed preliminary design, strategic design, or subsystem design), and detailed design. In the architectural design stage the system's gross organization is specified in terms of a hierarchically structured collection of modules, each of which plays some well-defined role in the delivery of the system's functional capabilities. In addition to the delineation of the modules, their interfaces and interactions are defined [9]. According to [4] architec-
tural design is an attempt to develop software beginning from the top down, as based on information flow or structure, determined from requirements. That design results in the definition of interfaces among internal software elements and external system elements, functional descriptions of each software element, data structural details, and special packaging considerations.

One of the most widely used techniques for structural design is Structured Design (SD) [10–12], that converts the target document obtained from SA into a modular description of the system. SD measures the modularity of the system according to the principles of cohesion (the strength of functional association of elements within a module) and coupling (the measure of interdependency between modules). The objectives of SD is to create system structures in which the modules have high cohesion and low coupling.

Structured Charts (SC), the primary tool for structured design, show the partitioning of the system into modules, their hierarchy and the interfaces among the modules. In general, design starts with an evaluation of a DFD defining the problem/requirement. From that DFD, an initial SC is derived and then redefined. The logic in the logical modules that make up the SC is specified, and then modules are packaged, producing a set of physical modules and programs that can be coded during the implementation stage.

1.3. Combining SA with SD

Two methods were proposed for mapping a DFD to a SC: transformation analysis and transaction analysis. Transformation analysis, which deals with the conversion of DFDs for sequential processes, maps the DFD on a structure that allocates control to input, processing, and output along three discretely factored module hierarchies. Transaction analysis, dealing with DFD conversion for independent processes, maps the DFD onto a structure which allocates control to a substructure that acquires and evaluates a transaction; another substructure controls all potential processing actions based on the transaction [4]. When DFDs portray both sequential and independent processes, both methods of analysis are employed.

Combining the methods for analysis and design is vital for successful development of an information system. The solution provided by [7] and [8] suffers, however, from severe limitations. The SD method and its SC tool were originally used by [10] to develop programs, and not to design systems, hence "SD is not a comprehensive system design technique" [12, p. 6]. A primary step in combining the two methods is to create an initial SC from a DFD, but DFDs play only a minor role in SD design, as stated in [12]:

"...the data flow diagram need not be drawn at all... In fact, observation shows that most people find there is little need to draw these diagrams for more than the first few programs after learning the technique. This is more a description of a thinking process to use in finding the top of a program than of an actual mechanical process where the diagram must be drawn" (p. 148).

Deriving an initial SC from a DFD is not simple and straightforward; much depends on the skill and experience of the designer who has to exercise a good deal of judgement [13]. A DFD may be composed of combinations of transformation and transaction structures, making the generation of an initial SC complicated and difficult [3].

In SD, data flow diagrams are used to generate an initial SC only, and once this is accomplished the diagram is discarded. Improvements and refinements made in the SC, which may affect the DFD, are not fed back into the diagram. Actually, the "data flow diagrams" used in SD are not exactly the DFDs of the SA method since they refer only to functions and data flows, not to data stores and external entities. Moreover in SD all functions are assumed to be elementary; no distinction is made between general functions and elementary functions and no guidance is provided as to how to deal with hierarchical DFDs.

Another product of SA not considered in SD is the structured description of the elementary functions. This description serves as the basis for program design and coding, but the SD method makes no use of it. Another aspect of architectural design not considered in SA and SD is user system interface.

1.4. Advantages of ADISSA

From the preceding survey of SA and SD methods it is clear that their combination and handling of the transition from analysis to design stage is far from satisfactory. SA provides a set of products comprising the system specifications, most of which SD ignores. ADISSA was developed precisely to overcome this problem and to provide an architectural design method that is fully compatible with and a direct continuum of SA. In ADISSA the various products of SA are utilized and combined. A minor revision in the original DFDs makes architectural design possible. This involves the refinement of the DFD concept "external entity" by defining various types of entities that trigger the occurrence of events in the system's environment, thus influencing its design. This is in contrast to Ward's Transformation Schema method [14] which represents control and timing in the DFDs by adding new types of elements (e.g. data and control transformations, control-flows and event-flows) that actually change the original DFDs, making them difficult to generate and comprehend.

ADISSA adopts Wasserman and Stinson's [15] concept of interactive systems as consisting of three components: (a) user interface; (b) operations upon data objects; and (c) database. The analogous ADISSA components are: (a) a menu tree, the external architecture of the system, from the user's point
of view; (b) transactions, the internal architecture; and (c) a database schema, consisting of normalized record types.

ADISSA regards an information system as performing transactions, each consisting of one or more modules that perform specific functions in response to various types of events in the system environment. Transactions may be initiated by users who input data to or receive output from the system, and who access the system via menus. Transactions may also be triggered on a predefined, temporal basis, or by stimuli/conditions sensed in real-time, or to handle communication to and from other systems. A transaction may involve database activities (retrievals and updates) and input/output activities. As it will be shown, all these components are derivable from the hierarchy of DFDs.

The basic concepts and components of ADISSA are discussed in Section 2, introducing menus, transactions, and triggering events. This discussion sets the ground for a detailed description of ADISSA methodology in Section 3, followed by an example (in Section 4) that employs the methodology. Section 5 summarizes and discusses directions for further research.

2. BASIC CONCEPTS AND COMPONENTS OF ADISSA

2.1 Menus

In online interactive systems there exist mainly two types of user interfaces [3]: user-initiated, suitable for experienced users; and computer-initiated, for non-experienced users. These two types of user interface can be incorporated: menus (of the computer-initiated type) can be combined with a command-driven interface (of the user-initiated type). Such a combination enables a casual user to gain access to the system via menu screens, and a skilled user to skip the menus and gain access to desired functions using direct commands. Since a command-driven interface can be derived from a menu-driven one, the latter is here of primary concern.

In large scale systems, the number of menu screens required to contain the quantity of possible selections are organized into a hierarchy called a menu tree. A user (at least a casual one) first accesses the "root" menu screen, which presents the major selections, i.e. the major functions of the system. Once the user selects a line in the menu, the system presents the next level menu which outlines more specific functions. This process of "browsing" continues until the user selects a specific activity, which is then performed.

In a menu screen there may be lines which are expandable and others which are not. A selection line causes a more specific menu screen to be presented, and a terminal line activates specific procedures in the system. Obviously, a menu tree will not be balanced and there may be both selection lines and terminal lines in the same menu screen and at different levels in the hierarchy. To accommodate experienced users who wish to bypass menu tree browsing, the system provides direct access to terminal lines.

The menu tree, from a user's point of view, is the external architecture of the system, and should be designed rationally and systematically if it is to function as a proper guide for users. In ADISSA, menus are derived almost automatically from DFDs, yielding a menu tree that is a sub-set of the tree of DFDs. Menu lines in the various menu screens are derived from functions which connect to external entities (through data flows). Of these, general functions create selection lines in the menus, whereas elementary functions establish terminal lines. Other functions within the DFDs (which are not connected to external entities) generate no lines in the menu system.

2.2. Transactions

As said, ADISSA regards an information system as performing transactions. A transaction consists of elementary functions which are directly chained to each other via data-flows and of data stores and entities which are connected to these functions. All transactions together form the internal architectural of the system. The objective is to identify the component transaction of the system, and for every transaction to identify the causes of its activation, its component elementary functions, the order in which these functions are executed, the input and output operations involved in the transactions, and the data records accessed by it to update or retrieve data.

A transaction in the system is initiated, or triggered, by an event. Although the concept event is ignored by SA and not specified in DFDs, it can in fact be incorporated into DFDs. Transactions will be classified by the type of event that triggers them: (a) user event: (b) time event: (c) real-time event: (d) communication event: (e) internal event.

2.3. User events and user transactions

A user event in the system is caused by a user. A user may initiate a transaction usually by accessing the system via the menu tree. The data flows in the DFDs between external entities and functions identify user-transactions: a data flow from an external entity to a function signifies an event which causes a system user to trigger a transaction which inputs data, whereas a data flow to an entity signifies an event which causes a system user to trigger a transaction which retrieves and provides information. The entity signifies the cause of the event, but it is the user or operator of the system who operates the system to accomplish the transaction. In ADISSA we term an external entity user entity (UE).

A distinction is made between data flows connecting UEs with general functions and data flows connecting UEs with elementary functions. In the former, data flow does not indicate a specific transfer of data elements between a UEF and a function in the
system, but an aggregate of various potential transfers of data elements. In the latter, data flow signifies a specific transfer of data elements between a UE and a specific/elementary function. Hence, a *general function* connected to a UE creates a *selection line* in a menu screen, whereas an *elementary function* connected to a UE generates a *terminal line* that initiates the execution of a user transaction. Figure 1 depicts a schematic example of the above discussion.

It is assumed that in DFD-F, \( f_1 \) is a general function and \( f_2 \) is an elementary function (In all DFD figures, a circle denotes an elementary function; concentric circles denote a general function). Since both \( f_1 \) and \( f_2 \) are connected to UEs there should be two lines in the respective menu screen \( F \). Line (1) in the menu refers to data flow \( a \) (from UE1 to \( f_1 \)) and line (2) refers to data flow \( d \) (from UE2 to \( f_2 \)). Hence, line (1) is a selection line, whose selection causes another, more specific menu screen to be presented, whereas line (2) is a terminal line, whose selection causes a transaction to begin. That transaction involves, as shown in DFD-F: inputting of data from UE2, performing procedure \( f_2 \) and updating data store \( D_2 \). Selection of line (1) in menu \( F \) would cause the more specific menu screen \( F_1 \) to be presented. Since the general function \( f_1 \) was decomposed in DFD-F.1 into two elementary functions \( f_1.1 \) and \( f_1.2 \), each connected to a UE, the menu \( F_1 \) has two terminal lines, the selection of either causing a transaction to be activated. If line (1.1) is selected, another transaction is fired, the details of which can be “read” from DFD-F.1 in the figure (Determining the sequence of operations within transactions will be described later).

### 2.4. “Chain effect” within a transaction

A “chain effect” results when two (or more) elementary functions within a transaction sequentially activate each other: i.e. execution of the first function causes data to be transmitted to the second, thus activating it. The chain effect terminates when a function is not connected to a succeeding function, but to either a data store or an entity. In the former, the transaction terminated with a database update, whereas in the latter, the transaction terminates with an output to the user (the role of data stores and entities in transaction will be discussed later in more detail).

One of the elementary functions that constitute a transaction is the *trigger*, i.e. a function connected to the entity which is a cause of an event. If there are several functions in a transaction, the trigger function is not necessarily the first one to be executed. It may be located anywhere along the chain. If it is the first function along the chain, the chain effect can only proceed forward. If, however, its location is elsewhere (including the end of it), then it cannot be executed first: the other function(s) ahead of it have to be fired before. Figure 2 illustrates these two cases.

In the transaction triggered by UE1, the trigger \( f_1 \) is the starting function and the chain effect goes forward from \( f_1 \) to \( f_2 \). In the other transaction triggered by UE2, the trigger function \( f_4 \) is the last function on its chain, and cannot be executed before
Fig. 2. Two user transactions.

f3 is executed. This second transaction can be interpreted thus: UE2 needs some information (e.g. a report) so the user accesses the system through the menu tree (not shown in Fig. 2) and makes selections in menu screens. At some stage he selects a terminal line which refers to the data flow f from f4 to UE2. At this moment a transaction is fired, and performed in the following order: first, function f3 is activated, retrieves some data from data store D1, and then performs some internal procedure. As a result some data are transferred to f4, which is then executed, and the requested report is transmitted to UE2. The above is an informal description of a transaction; Section 3 will describe transactions in a structured manner.

A chain effect in a transaction is not always sequential. Sometimes there is a split in the chain, where a given function diverges to two or more functions. In such a case the chain effects may be selective, depending on the type of connection defined in the DFD between the data flows that emanate from the source function. The connection can be AND, meaning that the chain-effect continues in all directions, or OR, meaning that the direction in which the chain will be activated depends on conditions determined by the source function.

The greater the number of elementary functions chained, the longer and possibly more complex the transaction will be. A multi-function transaction may sometimes become quite complicated, particularly if it involves split chains, many input/output operations, several database updates and retrievals, or complex procedures within its elementary functions. Such transactions may be complicated for users (who may have to perform many I/O activities in a single transaction) as well as for the programmers (who have to code and then to maintain the complex programs). The chains of complex transactions can be cut by splitting it into separate, smaller transactions. This issue will be discussed in greater detail in Section 3.

2.5. Time events and time transaction

A system, even if operating mainly in online mode, may include transactions that are performed on a predetermined temporal basis. This is typical to certain batch process, standard reporting, etc. Such transactions are initiated by time events, defined as events that activate a transaction at a predetermined point in time or time interval.

Time events are not treated in SA. In order to enable the system to handle time events it is necessary to add a new element to DFD notations: time entity (TE). TE will be represented in the DFDs as a triangle, and like a UE it will appear outside the frame of the DFD. Unlike in UEs, the data flow will always point from the TE to the function and will "carry" the unit of time which triggers the function. A TE will trigger a time transaction much as a UE triggers a user transaction.

It is possible to have "mixed" transactions, i.e. transactions that may be activated "automatically" at predetermined points in time (as time transactions), or "manually" at other points in time (as user transactions). Such transactions are identified by having both UEs and TEs connected to functions on their chain. An example for a "mixed" transactions is shown in Fig. 3. It involves functions f1 and f2 and can be triggered either automatically, by TE0 (at time t0) or by user UE1 (via the menu tree).

A mixed transaction imposes constraints on its implementation, because:

(a) if it involves a UE as source of data, activation of the transaction on a temporal basis requires that the input (from the UE) be prepared beforehand and ready to be acted upon by the transaction;

(b) if it involves a UE as destination of data (as in Fig. 3 above), activation of the transaction on a temporal basis would cause automatic production of output (e.g. reports).

2.6. Real-time events, communication events, and transactions

Two new types of external events that trigger transactions are presented here:

(a) real-time events and real-time transaction: In real-time systems there are devices that sense the environment and react accordingly. What is sensed is a real-time event and triggers appropriate reaction. The sensor/detector is termed a real-time entity (RTE), represented as a triangle in DFD (like a TE). The type of information that can be sensed by a RTE is carried by the data flow connecting it and a function in the DFD. Like a time transaction, a
real-time transaction is not accessible from the menu tree;
(b) communication events and communication transactions: a system may be connected to other systems via communication channels, enabling a message to arrive asynchronously from other systems. A communication event occurs when a message is received, and the arriving message activates a transaction. A communication entity (CE), also represented as a triangle in the DFD, triggers a communication transaction. The data flow connection a CE to a function in the DFD carries the type of message that is conveyed through the communication channel.

2.7. Internal events and internal transactions

An internal event occurs when an external event, assumed to trigger a specific transaction, actually causes the “invisible” execution of another. Two examples are:

—A user transaction accepts inputs (from the terminal). After a certain amount of input has been accepted (measured by that transaction), the transaction activates another transaction that performs, say, some computations on the input.

—A transaction adds records to a data store. At a certain moment, when the response time of the system falls below a certain level, another transaction is activated, performing file reorganization.

The examples show that one transactions may activate another if a certain status, or condition monitored by the first, is met. This means that an elementary function in the first transaction (that checks a condition) is necessarily chained to another function in the second transaction (triggered when the condition is met). In ADISSA terminology, these functions part of just one transaction which has two (or more) discrete, constituent parts: under certain conditions, only one part of the transaction is executed; under other conditions, the other parts may be executed too. An example is shown in Fig. 4.

Assume that the main task of the transaction in the figure is to accept data from UE1 and store it in D1 (a task that may be performed routinely). Function f2 in the transaction checks how many new records have been added to the data store. When it reaches a specified number, f2 triggers f3, which reads records from D1, performs some calculations and then updates D2. Note that the latter two functions perform a task which is “hidden” from the user; according to ADISSA terminology, however, all these functions belong to the same transaction.

3. ADISSA METHODOLOGY

Figure 5 provides an overview of ADISSA combined with SA, and includes seven stages. The starting stage, originally of SA, is (1) functional analysis, resulting in hierarchical DFDs and a data dictionary (DD). The figure indicates that this stage includes analysis of events and definition of entity types (UEs, TEs, RTEs, and CEs). The other SA stages, i.e. structured description of elementary functions (mini specs) and database schema design, do not occur until a later stage. Meanwhile the DFDs and DD are used to (2) design the menu tree, and (3) design the transactions of the system. As the dotted arrows indicate, there is feedback from stage (3) to stages (1) and (2), meaning that transaction design may
influence and change the specification and the initial menu tree. Next is (4) structured description of the transaction, replacing the original SA stage of structured description of elementary functions. The two subsequent stages, performed in an arbitrary order are (5) database schema design and (6) input/output design. Finally, there is (7) the DD of the architectural design.

The stages are discussed in detail in the rest of this section.

3.1. Analysis of events and definition of entity types (a step in Stage 1)

This stage extends and refines the SA method. Events are identified and the new entities (TEs, RTEs, and CEs) are incorporated within the DFDs and the DD. These entities are identified at the analysis stage, as are the functions and other elements of DFDs. In DFDs and the DD, the entities TE, RTE, and CE are treated similarly to UEs. The data flows between these entities and the functions “carry” the time interval (for TE), the information sensed (for RTE) or the message communicated (for CE), as explained in Sections 2.5 and 2.6 above.

3.2. Initial menu tree design (Stage 2)

The initial menu tree is designed by reviewing the hierarchy of the DFDs from the top to bottom. For every DFD that has functions connected to UEs, a menu screen is defined. Hence, the “root” menu screen includes a line for every function in the “root” DFD which is linked to a UE (At this stage all functions are probably general, through this need not necessarily be so). A “child” menu screen may be defined for “child” DFD (which details a general function in the higher level DFD), if that DFD involves UEs, thus prompting a line in the new screen to be defined for every function connected to a UE. If a UE is connected to a general function, a selection line is defined in the respective menu screen, and if a UE is connected to an elementary function, a terminal line is defined. To distinguish between selection lines and terminal lines the latter are marked “T”. At the end of the process the initial hierarchy of menu screens is obtained.

At this stage the lines in the menus are denoted with the ID-numbers and names of the involved functions. This is done for convenience only, and will be changed to meaningful line numbers and headings after the initial menu tree is revised.

During the course of menu tree design a situation may occur where a menu screen has one line only. This happens for DFDs in which only one function is connected to a UE. “Menu” is not applicable because there is only one selection. This one-line “degenerate” menu will eventually be eliminated from the menu-tree. Figure 6 gives an appropriate example.

From DFD- X the menu screen X is derived, with two lines for the two functions in the DFD, x 1 and x 2, each connected to one or more UEs. Function x 2 is assumed to be elementary, so line (x 2) in the menu is a terminal line, marked “T”. Function x 1 is assumed to be general, so it is decomposed into DFD-X1. In the DFD-X1, only function x 1.1 is connected to a UE, and therefore the respective menu screen X1 has one line only. This degenerate menu, marked with a diagonal line, will be eliminated from the menu tree.

It should be noted that a single line in a degenerate menu may be either a selection or a terminal line. It is a selection line if the involved functional is general. In the example shown in Fig. 6, function x 1.1 is general and therefore detailed in the separate DFD-X1.1. In that DFD the two functions x 1.1.1 and x 1.1.2 are connected to UE2, and therefore there are two lines in the respective menu screen X1.1. The situation in Fig. 6 is interpreted thus: during the course of his selections in the menu tree the user is presented with menu X; if line (x 1) is selected, the system does not actually present the degenerate menu X1. Rather, the next menu screen X1.1 is presented, with its two lines (x 1.1.1) and (x 1.1.2).

3.3. Transactions design (Stage 3)

The process of design is almost identical for all types of transactions, the only difference being in finding their triggers. The triggers of user transactions are identified at terminal lines in menu screens (marked “T”), whereas the beginning of other transactions are identified by reviewing the DFDs and searching for TEs, RTEs, or CEs connected to elementary functions.

A transaction is composed of the elementary function in a DFD, which is the trigger, and of other elementary functions which are chained one to the other (along the data flows). The design of a transaction involves, among other things, identification of the chained elementary functions; these may sometimes span to neighbouring DFDs. All the chained functions are traced when the transaction is designed until points are reached where the elementary functions are connected to data stores or to entities only. Recall that any function on a chain may be connected also to a data store or to an entity, and the transaction includes also these data stores and entities.

Data stores in transactions. A data store can function as a regulator of the size or complexity of a transaction. If, during the course of the design of a transaction, it is found that many elementary functions are involved, or it is judged by the designer as too complex, the transaction can be simplified by introducing appropriate data stores between chained functions. Such an intervention cuts the chain effect and hence divides the transaction into separate ones, thereby changing the DFD specification.

Figure 7 shows an example of such an intervention. Assume that all functions in DFD-X are elementary, and disregard momentarily data store D3. At first
one transaction is identified in DFD-X, consisting of six functions and involving one input (from UE1), two data store updates and one output (to UE2). This transaction is judged too complex, so to simplify it data store D3 is introduced between functions x3 and x4. Consequently, two separate transactions are obtained: one includes functions x1, x2 and x3; the other includes function x4, x5 and x6.

Since data stores play a significant role in transaction design and their location within the DFDs is important for determining the size and complexity of transactions, it is advisable to locate data stores between functions as early as possible in the hierarchy of DFDs, in order to avoid transaction that are too complex.

*External entities in transactions.* If more than one function along the chain is connected to a UE it would seem that more than one transaction is involved, because appropriate menu tree would have a line for every such function. It is already known,
However, that all the chained functions belong to the same transaction. This situation is shown in Fig. 8. Assuming that all the functions in the DFD are elementary, the respective menu screen would have 3 terminal lines, indicating 3 transactions.

The two terminal lines (x1) and (x3) in the menu actually trigger the same transaction because the three functions x1, x2, x3 are chained. Hence, there is inconsistency between the menu and the actual transaction. The problem can be solved in two ways:

(a) if the transaction is judged as too complex, it can be split by introducing a data store between two of its chained functions (as was shown in the previous example). This solution simplifies the transaction and has no effect on the menu because it already has two terminal lines indicating two separate transactions;

(b) if the transaction is judged as simple enough, there is no need to change it, and it is the menu screen that has to be changed: the separate terminal lines have to be united: the separation lines have to be united (and given a new ID-number and heading). This solution is shown in Fig. 8 above: instead of the two lines (x1) and (x2) in the initial menu screen, there is one line only [numbered (x9)] in the revised menu [the other line (x4) remains unchanged because it refers to the other transaction].

This type of solution which unites lines in the menu screen may have side effects: if the initial menu screen has only terminal lines which are to be united, the revised menu screen becomes degenerate. Assume, for example, that in Fig. 8 the transaction which involves function x4 does not exist, so the original menu includes line (x1) and (x3) only. In the revised menu these will be united into a single terminal line, and therefore that menu becomes degenerate.

The revised menu tree. We have seen how the initial menu tree may be affected during transactions design. At the end of stage (3) the menu is finalized as follows:

(a) all degenerate menu screens are discarded from the tree. If the single line in such a menu is a terminal line (marked "T") then the appropriate selection line in its parent menu screen is marked "TQ" to indicate that it is a quasi-terminal line, the selection of which triggers a transaction;

(b) line i.d.-numbers and their headings are changed in order to make them more meaningful to the users. In particular, the headings of selection lines in the various menu screen should express the semantics of the options that are suggested, and the headings of the terminal lines should express the meaning of the transactions "behind" those lines.

The menu tree obtained enable various users/operators of the system to access it and activate the required transactions. The creation of a menu tree, as we have seen, is based on the functional structure of the system, as expressed in the hierarchy of DFDs. By analogy to the database arena, we say that the menu tree is a "conceptual schema" of the user interface. From this schema one can derive sub-schemata, i.e. sub-menu trees, for various user groups/types. The division of a menu tree into sub-menu trees may become necessary in large scale systems with many user types who would be shown only those portions of the menu tree that involve transactions they are allowed to operate; the rest of the menu tree would be "hidden" to them. This study does not detail the method by which sub-menu trees can be derived and adjusted; this will be reported separately. Other possible changes to the structure of the menu tree (such as the addition of "help" or "browse" functions) or the implementation of additional interface methods (such as commands) are not covered. These kinds of changes and additions may take place in later stages of system development.

3.4. Structured description of transactions (Stage 4)

This stage replaces the original SA stage of structured description of elementary functions, in which the process logic of each of the elementary functions is described separately and independently. Describing the process logic of single (elementary) functions is not always possible or meaningful because elementary functions are not isolated—they are performed as part of transactions. The problem is not merely to
describe the internal logic of a function, but also the interrelationships among the functions, including the data store- and the I/O-related activities. The description of process logic in ADISSA is therefore at the level of transactions. A two-level method for describing transactions is proposed: the top level includes the "skeleton" of the transactions and the

The above primitives are used along with the standard primitives of structured programming which enable one to express sequential, conditional (if...then...else), or repetitives (do while) activities included in the skeleton description.

The following is a possible top-level description of a transaction shown in Fig. 9.

```
Begin user transaction "x"
Input from UE1:a
Read from D1:b
Execute Function x1
If (condition) then
Move from Function x1 to Function x2:c
Execute Function x2
Write to D2:e
else
Move from Function x1 to Function x3:d
Read from D1:f
Execute Function x3
Output to UE2:g
End transaction.
```

3.5. Database schema design (Stage 5)

This stage is deferred until transaction design has been achieved because possible changes in the DFDs during transaction design may involve data stores and data flows. No change is proposed here to the SA method for database design (which created a database of normalized record types). After the schema is designed, it is "bound" to the top-level description of the transaction: for every "Read" or "Write" line in the top-level descriptions (referring to a data store) the bottom-level description details the corresponding record types which are to be accessed, and the fields(s) through which this will be accomplished.

3.6. Input/output schema design (Stage 6)

For every "Input" or "Output" line in the top-level description of a transaction, the bottom-level includes a definition of the type/media, and a sketch or prototype of that I/O. This can best be accomplished using prototyping tools, e.g. forms and reports generators. The whole collection of I/Os designed for all the transactions is termed I/O schema, by analogy to the concept schema in database design.
3.7. Data dictionary of ADISSA (Stage 7)

Earlier, in stage (1) the need to update the DD with details on the new entity types was discussed. Now, in the final stage, the DD is extended to include the new products of ADISSA. The menu tree dictionary includes details of all screens and their lines. Details of a line include the original ID-number (i.e. the appropriate function number), the new/final number and heading, and the “T” or “TQ” marks for terminal lines and quasi-terminal lines, respectively. The transactions dictionary detail, for every transaction, the type of transaction, elementary functions, data stores, record types, inputs and outputs, and the top-level description of the transaction.

4. SAMPLE APPLICATION OF ADISSA

In this section we present an example for the application of ADISSA in development of an information system for a plumber. Appendix A includes five DFDs of the system (because of space limitations, the DFDs have been simplified and some details omitted; due to the nature of the application, no real-time or communication entities are evident).

Figure 10 shows the initial menu tree derived from the DFDs. At this stage there are 6 menu screens. The line numbers and headings are those of the corresponding functions in the DFDs from which the lines have been derived.

The 15 terminal lines, marked “T”, indicate that there initially are 15 user transactions. In addition, there are 2 time transactions (one identified in DFD-3, where a TE is connected to elementary function 3.3; the other is in DFD-3.1, where a TE is connected to 3.1.2).

DFD-1 includes 2 user transactions, identifiable in menu screen 1 (Fig. 10). One, dealing with the preparation of purchase orders, is activated by the system user/operator (say, the plumber himself, or a clerk, but not the supplier who is only a recipient of the output). The transaction “behaves” as follows: from time to time, the plumber initiates the preparation of purchase order. To do so, he triggers the transaction using the menu tree. At the beginning, function 1.1 reads the active orders from D3 (where it finds the parts and quantities needed for new jobs). Then it reads from D1 the quantities in stock. Consequently the function determines what and how much has to be purchased, and prepares a purchase order (which will be used by the plumber at the supplier's). The details of the purchase order are also written to data store D5. The following is a top-level description of the transaction (space precludes the inclusion of a bottom-level description):

Begin transaction “1.1”
Read from D3 (active orders)
Read from D1 (parts in stock)
Execute Function 1.1 (prepare purchase orders)
Output to UE1 (purchase orders)
End transaction.

4. SAMPLE APPLICATION OF ADISSA

In this section we present an example for the application of ADISSA in development of an information system for a plumber. Appendix A includes five DFDs of the system (because of space limitations, the DFDs have been simplified and some details omitted; due to the nature of the application, no real-time or communication entities are evident).

Figure 10 shows the initial menu tree derived from the DFDs. At this stage there are 6 menu screens. The line numbers and headings are those of the corresponding functions in the DFDs from which the lines have been derived.

The 15 terminal lines, marked “T”, indicate that there initially are 15 user transactions. In addition, there are 2 time transactions (one identified in DFD-3, where a TE is connected to elementary function 3.3; the other is in DFD-3.1, where a TE is connected to 3.1.2).

DFD-1 includes 2 user transactions, identifiable in menu screen 1 (Fig. 10). One, dealing with the preparation of purchase orders, is activated by the system user/operator (say, the plumber himself, or a clerk, but not the supplier who is only a recipient of the output). The transaction “behaves” as follows: from time to time, the plumber initiates the preparation of purchase order. To do so, he triggers the transaction using the menu tree. At the beginning, function 1.1 reads the active orders from D3 (where it finds the parts and quantities needed for new jobs). Then it reads from D1 the quantities in stock. Consequently the function determines what and how much has to be purchased, and prepares a purchase order (which will be used by the plumber at the supplier's). The details of the purchase order are also written to data store D5. The following is a top-level description of the transaction (space precludes the inclusion of a bottom-level description):

Begin transaction “1.1”
Read from D3 (active orders)
Read from D1 (parts in stock)
Execute Function 1.1 (prepare purchase orders)
Output to UE1 (purchase orders)
End transaction.
The other user transaction in DFD-1, triggered by line 1.2 of menu screen 1 (Fig. 10), is somewhat more complex. It is activated by the user/plumber after an actual purchase from a supplier, and involves the following: the plumber enters the details of the purchase from a supplier, and then function 1.2 finds and reads the details of the appropriate purchase order from D5, thus activating two functions; function 1.3 updates D1 with the new parts actually purchased, and function 1.4 updates the supplier’s account in D2. The AND connector between the above two functions implies that both are fired. Since the activities within a transaction are performed in a sequence, it is determined in this case that function 1.3 is activated first, then function 1.4. The top-level description of the transaction is:

```
Begin Transaction “1.2”
  Input from UE1
  Read from D5
  Execute function 1.2
  Move from Function 1.2 to Function 1.3
  Execute Function 1.3
  Write to D1
  Move from Function 1.2 to Function 1.4
  Execute Function 1.4
  Write to D2
End transaction
```

The next example deals with a transaction triggered by lines (2.3), (2.4) and (2.5) in the initial menu screen 2 (Fig. 10). In reviewing the respective DFD-2, it is seen that functions 2.3, 2.4 and 2.5 are chained, so it is clear that they belong to the same transaction. The transaction is judged simple enough not to be split, and therefore the three lines in the menu screen are combined into one. This can be seen in Fig. 11, which shows the revised menu screen. From the transaction it can be seen that the activation of its functions 2.4 and 2.5 depend on a condition detected in the preceding function 2.3. The transaction is initiated by the plumber when he gets a reply about a job proposal from a customer. After entering the customer’s reply, function 2.3 reads the details of the proposal from D3 (for verification). Then, if the reply is a rejection of the proposal, function 2.3 activates function 2.6, which just updates D3 (say, deleting the proposal) and the transaction terminates. If, on the other hand, the reply is an approval of the proposal, function 2.4 is activated; this function asks the user to enter details of first payment, it marks this fact in D4, and then fires function 2.5 which opens a customer account in D4 and prepares a receipt to be sent to the customer. The following is a top-level description of the transaction:

```
Begin Transaction “2.3–2.6”
  Input from UE3
  Read from D3
  Execute Function 2.3
  If (reply = “approval”) then
    Move to Function 2.4
    Input from UE3
    Execute Function 2.4
    Write to D3
    Move to Function 2.5
    Execute Function 2.5
    Write to D4
    Output to UE3
  else
    Move to Function 2.6
    Execute Function 2.6
    Write to D3
  End-If
End Transaction.
```

In this sample system, there are more cases in which an initial menu screen includes more than one triggering line for the same transaction. One such case is the transaction triggered by lines 3.1.3 and
3.1.4 (in Fig. 10). This transaction is of interest not only because of the inconsistency with menu screen, but also because it is a mixed transaction: it can be triggered manually, whenever the plumber decides to check the accounts receivable or to send reminders; additionally it is triggered automatically at the end of every month. At any rate, the transaction is performed as follows: function 3.1.2 is executed first (no matter what triggered the transaction) and checks the balance of a customer’s account in D4. The following cases are possible: (a) if the balance is greater than some amount and/or is of some “age”, function 3.1.3 is fired to prepare a reminder; (b) if function 3.1.3 finds that the debt is small or new, it does nothing else, and the transaction terminates; (c) if the balance is zero, function 3.1.4 is fired to close the customer’s account in D4 and to prepare an end-of-job report. The top-level description of this transaction is not given.

The last transaction we discuss involves functions 3.2.1 and 3.2.2. In this case there is an inconsistency because the initial menu screen 3.2 (Fig. 10) includes two lines that trigger the transactions. If we decide that the transaction is complex, it can be split into two (say, by eliminating the data flow which chains the two functions, thus creating one transaction to handle the reminders, and another—to handle payments). If this solution is adopted, there is no need to change the menu screen. In our example, it is decided not to split the transaction. As one transaction, it can be activated by the plumber when a reminder arrives from a supplier, or when he decides to initiate a payment. At any rate, function 3.2.1 is executed first to check the supplier’s balance, and the result of this check determines whether or not function 3.2.2 is fired (the description of this transaction is not shown). Note that if this solution is adopted and additional change will occur in the menu tree: since initially there are only two lines in the menu screen 3.2 (Fig. 10), once combined into one, the menu screen becomes degenerate and is eliminated from the menu tree; therefore line 3.2 in the parent screen is marked “TQ”.

Figure 11 shows the revised menu tree of this example. It includes 5 screens only and indicates that there are 10 user transactions, two of which are mixed user- and time-transactions.

5. SUMMARY

ADISSA methodology successfully incorporates SA methodology with innovative refinements to combine the two major stages in system development: analysis and design. Not only does ADISSA accomplish this in a unified manner, but it also fulfills such well-established principles in software engineering as top down approach, stepwise refinement, and modular design.

It is important to reiterate how critical the correct and precise execution of the stage of analysis is. It becomes more crucial now because the success of architectural design, and hence the success of the rest of the stages, depend on successful system analysis. Once a system is analyzed and specified, the next stage of architectural design becomes a mere structured and technical process, aside for some steps in transactions design, in which judgement and intuition are still needed.

The steps of ADISSA are so well-structured and algorithmic that it is possible to develop software tools to perform most of them automatically. Development of a set of tools supporting ADISSA is already underway and includes the following modules:

(a) *DFD Drawer*: a graphical tool that draws and updates a hierarchy of DFDs (following our enhanced notation);

(b) *DFD Analyser*: analyses the correctness of single DFDs and the consistency between leveled DFDs;
(c) Data Dictionary: handles the data dictionary of the DFDs and the other components;

(d) Menu-Tree Designer: creates automatically the initial menu tree; later on, in an interactive mode, assists the designer to update the initial menus to form the revised menu tree;

(e) Transaction Designer: an interactive system that identifies transactions and their components, updates the DFDs according to changes in transactions, and finally creates a top-level description of the transactions;

(f) Database Designer: ADDS is a system that automatically creates a database schema out of a conceptual scheme [16]. Currently it is a "stand-alone" system; it is our intention to combine it with the ADISSA tools, such that "Read/Write" lines in the transaction descriptions will be tied up with the appropriate record types/relations defined in the database schema that is created by ADDS. Details on the above tools will be provided in separate report.

The ADISSA method described in this article was tested on several data-processing, online-oriented projects, and is currently being used in more projects. The method is being taught in system analysis and design courses and professional seminars. We intend to expand the implementations of ADISSA in a variety of large-scale projects, and we anticipate that this will be enhanced once the above mentioned tools become operational.

Acknowledgement—This work was supported in part by a grant from the Israel Council for Research and Development.

REFERENCES


APPENDIX A

DFDs of the Plumber Information System

DFD-0

1. Purchases from suppliers
   - Bills and receipts
   - Purchases
   - Active orders
   - Bills and receipts
   - Stock
   - Parts
   - Parts needed
   - Prices, stock
   - Proposals, approvals/rejections

2. Proposals and job orders
   - Proposals, requests, approvals/rejections
   - Customer details, prices
   - Proposals, receipts (for tax payment)
   - Payments
   - Reminders, payments

3. Accounting and reporting
   - End-of-month
   - Payments, balances
   - Transactions
   - Reminders, payments
   - Customer's accounts
   - D2 Supplier's accounts
   - D1 Parts

UE1 Supplier
UE2 Plumber
UE3 Customer
UE1 Supplier
UE2 Plumber
UE4 Accountant
UE3 Customer
UE1 Supplier
TE
DFD - 3.1

3.1.1 Accept payments

3.1.2 Check accounts receivable

3.1.3 Prepare reminders

3.1.4 Prepare end-of-job reports

DFD - 3.2

3.2.1 Check accounts payable

3.2.2 Prepare payments

UE1 Supplier

UE2 Plumber

UE3 Customer