On Constructing Executable Prototypes From Structured Analysis

Ren-Juh Lea* and Chyan-Goeti Chung†

*†Institute of Computer Science and Information Engineering
†Microelectronics and Information Science and Technology Research Center
National Chiao Tung University
Hsinchu, Taiwan, 30050, R.O.C.

Abstract
Structured Analysis (SA) is widely accepted for analyzing information systems recently. But the customers can view the intended system only through documents. It is apt to lead to misunderstanding. If the behavior of the intended system can be exhibited by rapid prototyping techniques, the analysts and customers can earlier attain a common opinion about the system specification. In this paper, a two-level model is used to construct executable prototypes from the results of SA. By this model, a prototype is constructed by a set of transactions representing the operation logic of system service functions and a set of data objects. For ease of writing and tolerance of incompleteness, the specification of transactions is in a procedure-like form, and the object operations are specified in a table-filling form. This specification scheme provides an adequate way for incremental extension of prototype functions. The procedure of deriving prototype specifications from SA results is presented. And a prototyping environment for the exhibition of prototype behavior is also described.

1. Introduction
Structured Analysis [1,2](abbreviated as SA) is widely accepted by software engineering professionals for analyzing information systems. Owing to the top-down characteristic and the graphical nature, it enables system analysts to visualize the intended system and to communicate with users much more easily than traditional methods. Unfortunately, before the intended system has been implemented, system analysts and customers can only talk about the envisioned system described in a pile of printed documents. It raises the issue of rapidly constructing executable prototypes from the results of SA. By rapid prototyping, the earlier feedback from the customers' and the system designers' experience with the workable version of the analyzed system can be used to adjust the system requirements and system design concepts before valuable time and effort have been invested in implementation [3,4,5]. Burns and Kirkham [6] have proposed a methodology to generate prototypical Ada programs from the data flow diagrams and the data dictionary of SA documents, but the primitive activities described in the mini-specs still have to be programmed in Ada; that will considerably increase the cost of prototyping. In this paper we present another approach to derive executable prototypes from the results of SA with the following emphasis.
(1) No implementation details have to be considered in prototype construction.
(2) No stubs (primitive functions) have to be designed and coded a priori in traditional programming languages.
(3) Prototypes should tolerate incomplete functions.
(4) Prototype specifications should be easy to read and understand.
(5) Prototype specifications should be easy to extend and modify.

2. Issues of Making SA Results Executable
The primary components of SA results include a set of data flow diagrams (DFDs), a set of mini-specs, and a data dictionary. Using SA, the analyzed system is hierarchically decomposed into processes and depicted by levels of data flow diagrams. For each process at the leaf of the hierarchy, a corresponding mini-spec is used to describe its operation of transforming the input data flows into the output data flows. All the data objects referred in the system are defined in the data dictionary, such as data flows, data elements, and files. The data flow diagrams and the mini-specs depict the activities responding to events of requests from the users [2]. Each event represents a service function provided by the system, and is triggered when the associated input data flows are supplied.

Since executability of prototype specification is a primary requirement for rapid prototyping, we need first to study the possibility of making SA results executable. Apparently, the SA results are not directly executable with the following reasons.
(1) It lacks necessary control flow information in the data flow diagrams; only the sequence of data flows is explicitly depicted.
(2) There is no formal (machine interpretable) semantics supported for mini-specs. Mini-specs are usually written in natural languages.

Nevertheless, it does not mean that SA results are not suited to construct executable prototypes. After investigating the SA
method, we have the following observations.

1. The leveled set of data flow diagrams can be expanded to a plane (single level) data flow diagram. For each system service function, there exists in the expanded data flow diagram a region of bubbles and the related data flows that compose the service function, as shown in Figure 1.

2. There is no control information explicitly represented in the data flow diagrams, but, for each service function, the sequence of data flows and the descriptions of mini-specs are highly related to the operation logic of that service function. The data flows determine the skeleton of the operation logic, and the descriptions of mini-specs determine the details.

3. In fact, the operation logic of a service function is mainly the control sequence of primary operations on data objects. The control sequence can be derived from the data flow sequence and mini-specs, as illustrated in Figure 2. The primary operations are referred in mini-specs; most of them are assumed to be understandable to the readers, so they have no explicit functional definitions in the documents.

4. If the specifications of data objects include the functions of their associated primary operations, the model of SA can thus contain all the necessary semantics for executing the specified system behavior (Figure 3).

![Figure 1. Regions of bubbles and data flows corresponding to the system service functions can be identified from the expanded DFD.](image1)

![Figure 2. Operation logic can be determined from the sequence of data flows and the description of mini-specs. (Ref. service function 2 shown in Fig. 1)](image2)

![Figure 3. Control sequence and object operation calls compose the operation logic of a service function.](image3)

From the above discussions, we recognize that to derive an executable model from SA results, the following two issues need first to be solved.

**Issue 1.** The operation logic for each system service function must be formally specified.

**Issue 2.** The functions of the primary object operations must be explicitly defined.

In the following, we will propose a model to solve the two issues.

### 3. Constructing Prototypes From SA Results by a Two-Level Model

The goal of rapid prototyping is to develop an executable model of the intended system early in the development process. To make prototypes executable, a formal scheme is needed to specify the functions of prototypes. One way to specify the prototype functions is by mathematical formulas, e.g., functional programming [7], logic programming [8], and algebraic equations [9]. Mathematical formulas support well established foundations for automatic interpretation, but it is difficult to write. The other way, from the experience of conventional programming languages, is by procedures and data declarations. Although they usually involve too detailed design considerations, we can avoid it by promoting the abstract level from procedures and data to transactions and objects [10]. A transaction represents the operation logic of a system service function. An object represents an abstract data object manipulated by the system. A transaction can be considered a macro operation comprising a set...
of primary object operation calls ordered by control sequences. So, the main component parts of a transaction are:

1. the operational control sequence — including sequential, conditional, and iterative control rules, and
2. the primary object operation calls — including the requests of changing objects' states and the queries for inquiring current status of objects.

From the concept of data abstraction, an object is characterized by its state and associated operations. Through the operations transactions can communicate with objects. To define the object operations in a computer interpretable way, there are two alternatives, one to describe the algorithms and the other to enumerate all the possible input/output pairs. The description of algorithms will involve design details; we need not think about so detailed at this stage. We neither need to define an object operation for all of its possible domain because only some significant cases in the domain are really concerned in prototype demonstrations. Specifying all its concerned input/output pairs serves this purpose. This allows incomplete specification and facilitates incremental extension of the object operations.

The problems of Issue 1 and Issue 2 addressed in the preceding section are now reduced to how to formally specify transactions and objects. So, if we have a scheme for specifying and interpreting transactions and objects, we will accomplish the initial goal of rapid prototyping from SA results. In the following, we first introduce a two-level model to conceptualize the transactions and objects, then the associated specification scheme will be described in the next section.

The concept of transactions and objects leads us to consider prototypes in two different views. One focuses on the system overall behavior, i.e., the operation logic of each system service function. The other concentrates on the system manipulating objects, i.e., the characteristics of data objects and their associated operations. So the prototype of an information system can be treated as an information base accessed by a set of transactions, as shown in Figure 4. Each transaction represents the behavior of a system service function, such as check out books, return books, add books to library, ... in a library system. The information base consists of data objects (or called objects for convenience) manipulated by the system, such as books, borrowers, etc. Associated with each object, a set of operations characterizes the behavior of that object. The transactions interact with the information base by invoking the operations of objects in the information base. For the convenience of management, the related transactions can be grouped into transaction-groups. For example, in a library prototype, the transactions may be grouped to book management, checking in/out books, queries, etc. Several transaction-groups may be further grouped into a larger transaction-group. These transaction-groups result in a hierarchy of tree structure with transactions located at the leaves of the tree structure. As shown in Figure 5, these transaction-groups and transactions form the upper level of the two-level model, and the objects manipulated in the system constitute the lower level of the model. More detailed description of this model can be found in [10].

This two-level model provides a foundation for three different kinds of abstractions. First, a prototype system is organized into a tree structure of transaction-groups, which manifests the structure of system service functions. Second, each transaction on the leaf of the tree structure expresses the behavior of a service function provided in the prototype. Third, the data entities manipulated in the system are modeled by objects in the scheme of data abstraction. These objects with their operations support an operational basis for exhibiting the behavior of the prototype. Separation of levels not only makes it easier to quickly construct a workable version of the intended system but also facilitates modifying and refining the prototype.
4. The Prototype Specification

Based on the two-level model, the specification of a prototype consists of three parts: (1) the transaction-group specifications, (2) the transaction specifications, and (3) the object specifications. They are briefly described in the following, and detailed description of the language can be referred to [11].

4.1. Transaction-Group Specification

The definition of a transaction-group is a menu containing all the service functions belonging to this group. A descriptive form is provided to specify all the pairs of service function name and the corresponding transaction name:

**MENU**

Transaction-Group-Name

(Service-Name_1) - (Transaction-Name_1);
(Service-Name_2) - (Transaction-Name_2);
...
END.

A menu containing Service-Names is displayed on the user interface when the transaction-group is activated. The Transaction-Name specified can be a transaction name or another transaction-group name. When one of the service names is selected by the user, the corresponding transaction (or transaction-group) will be executed (or be activated).

4.2. Transaction Specification

Due to operational characteristics of transactions, the most natural way to specify them is in a procedure-like form, which is similar to the mini-spec description, but we have a rigorous definition for the specification constructs. Similar to conventional programming languages, in transaction specification we use variables to denote data objects, and use assignments, conditions, loops, and other statements to describe the order in which the object operations are invoked. Two kinds of object operations can be called in transaction specifications:

1. state transition request — to change the state of an object:
   
   object-name ! transition-name(parameters),

2. query request — to inquire the status of an object:
   
   object-name ? query-name(parameters).

For example,

user1 ! borrowing(book1)

and

library ? has_book(book2)

are respectively a state transition request and a query request. The former asks for the object user1 to change its state to reflect that he has borrowed book1. The latter inquires of the object library whether it contains book2 in the library. The object operations called in transaction specifications are defined in object specifications.

4.3. Object Specification

Every object manipulated in the prototype is defined in an object specification by its state domain and associated operations.

A. State Representation

The state of an object is represented by its attribute constituents, each of which can be another object or a constant such as a number, a string of text, or a Boolean value. Two kinds of state representations are provided in this language, one for objects which are characterized by a fixed number of constituents, and the other for objects which are characterized by a varying number of constituents. The first one is represented by a tuple of constituents which determine the state of the object. For example, a book in a library system may have the attributes of book names, whether it is in the library, whether it is reserved, and the last borrower of this book. So the state of a book can be represented as

< Book_Name, In_Library, Reserve_Flag, Borrower >

where Book_Name is a character string for the book title, In_Library and Reserve_Flag are two Boolean values that represent whether the book is in the library and whether it is a reserved book, respectively, and Borrower is another object name for the last borrower of this book. For example, the tuple

< "Algebra", false, false, John >

represents one of the book's state. The number of attributes in the tuple will not be changed once the object is created.

The second state representation is by a sequence of varying number of objects and/or constants, named a trace. Examples for this kind of objects are sets, arrays, stacks, and queues. It is represented by

Object_1.Object_2.Object_3...Object_N

where the elements Object_1 through Object_N are linked by dots. A special symbol $\phi$ is used to represent the state which contains no object. For example, the initial state of a book file which does not contain any book can be represented as

$\phi$.

When the file is gradually inserted by books book1, book2, book3, and book4, its state becomes


B. Operations Definition

To allow the incomplete definition of object operations, a table-filling form defining an object operation as a relation mapping pieces of exemplified input arguments into exemplified
results is appropriate. There are two kinds of object operations, state transition operations and query operations. All state transition operations of an object are defined in a state-transition-table:

<table>
<thead>
<tr>
<th>Current-State_i</th>
<th>Operation_1</th>
<th>Operation_2</th>
<th>...</th>
<th>Operation_j</th>
</tr>
</thead>
<tbody>
<tr>
<td>Current-State_1</td>
<td>New-State_11</td>
<td>New-State_12</td>
<td>...</td>
<td>New-State_ij</td>
</tr>
<tr>
<td>Current-State_2</td>
<td>New-State_21</td>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Current-State_i</td>
<td>New-State_ij</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

When a state transition operation requested to an object, the state of this object becomes New-State_ij if the current state of the object matches Current-State_i and the requested operation matches Operation_j. Note that there may exist parameters for operations, so, for operation matching, in addition to matching of operation name, the corresponding parameters must also be matched. In each trial for matching, the unbound variable which appears both in Current-State and in Operations can be assigned any appropriate value to satisfy the matching. Once the matching is finished, all these variables are reset to be unbound. If the matching fails or there is nothing specified in the field New-State_ij, an error occurs.

All query operations of an object are defined in a query-table:

<table>
<thead>
<tr>
<th>Current-State_i</th>
<th>Operation_1</th>
<th>Operation_2</th>
<th>...</th>
<th>Operation_j</th>
</tr>
</thead>
<tbody>
<tr>
<td>Current-State_1</td>
<td>Return-Exp_11</td>
<td>Return-Exp_12</td>
<td>...</td>
<td>Return-Exp_ij</td>
</tr>
<tr>
<td>Current-State_2</td>
<td>Return-Exp_21</td>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Current-State_i</td>
<td>Return-Exp_ij</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Like the matching principle of state-transition-table, in query-table, when Current-State_i and Operation_j are matched, the expression specified in the field Return-Exp_ij is evaluated and returned. A specification example of object book in a library system is shown in Figure 6.

![Figure 6](image_url)

Figure 6. An example of object specification. The identifiers starting with a lower case letter, such as 'undefined', 'empty_list', 'yes', and 'true', are constants. B, L, and A are variables representing Book_id, In_library, Last_borrower, and Authors, respectively.

5. Prototyping Procedure

Based on the specification scheme presented above, the steps of prototyping from SA results (including DFDs, mini-specs, and a data dictionary) are discussed as follows.

Step 1. List the service functions of the prototype system. The service functions correspond to the external events requested from the users in the data flow diagram. Each service function corresponds to a transaction; it must be identified by a transaction name.

Step 2. Organize the service functions into related groups. Organize the service functions listed in Step 1 into a hierarchy of transaction-groups. This hierarchy can help the system users to overview the system functional structure, and also help the maintenance of the prototype system.

Step 3. Decide in the expanded DFD a corresponding region for each service function. For each service function, allocate in the expanded DFD a region (sub-DFD) by tracing the data flows and the bubbles that compose the service function. If this DFD is created by the principles of essential analysis [2], the bubbles in the region have no data flow going directly into bubbles in other regions. For each region, it can only communicate with the files, external data sink/source, and the system users.

Step 4. Identify the data objects manipulated in the system. The data objects used in the system can be identified either from the original system requirement or from the contents of the data dictionary. They often denote the significant data entities in the application domain. For example, book, borrower, book file, and borrower file are significant data objects in the library system.

Step 5. Identify the operations associated with each data object. We can determine the operations for each data object by exploring the mini-specs of all system service functions to find what primary operations are referred, and to determine which object each operation is associated with. Name these operations and determine the parameters of each operation.

Step 6. Complete the object specification for each data object. Each data object is defined in an object specification. For each data object, the state representation form is first determined. Then, the state transition operations and the query operations are defined in a state-transition-table and a query-table, respectively. Shown in Figure 6 is an example for defining the
Step 7. Derive the operation logic for each service function. The operation logic of a system service function is partly determined from the sequence of data flows in the sub-DFD region, and partly determined from the description of mini-specs for bubbles. Figure 7 shows an example of deriving the operation logic of checking out a book in a library system from the partial data flow diagram and the corresponding mini-specs.

Step 8. Complete the transaction specification for each service function. Each system service function is formally defined by translating its operation logic to transaction specification. Figure 8 shows the transaction specification for the transaction check-out derived in Figure 7.

Step 9. Run the prototype. The prototype specification (including menus obtained from Step 2, object specifications from Step 6, and transaction specifications from Step 8) can then be executed on the prototyping environment described in the next section.

Step 10. Amend the prototype specification. The errors occurring in the execution of the prototype specification can be updated by returning to Step 3, 5, or 7. The corresponding flaw in the SA document can also be recovered.

6. The Prototyping Environment

In this section, we discuss the design of a prototyping environment (named RSPS) for processing the specifications and supporting a running environment for prototype behavior exhibition. Shown in Figure 9 is the system architecture of RSPS.

The RSPS consists of two subsystems: a specification preprocessor and a running environment. When a prototype specification, containing transaction specifications and object specifications, enter the system, the preprocessor (consisting of a transaction specification preprocessor and an object specification preprocessor) first translates each part of the specification into an intermediate form, and stores them in Transaction Data Base and Object Data Base, respectively. At this stage, some static errors such as syntactic errors, definition inconsistency, and define/use anomaly can be detected and informed. RSPS provides two ways to input the specifications. The batch mode reads the specifications from a file, and the interactive mode facilitates the designer to add or modify the specifications interactively on the screen.

The running environment consists of five primary modules. (1) Input/Output Interface is designed for: (a) showing menus to users, (b) accepting the selections from users and sending the selected events to Transaction Manipulator, (c) reading the input data from users, and (d) displaying the output messages.

(2) Transaction Manipulator is an interpreter of transaction specifications. When a transaction is selected to run, Transaction Manipulator first fetches the corresponding procedure (in intermediate form) of the transaction from Transaction Data Base, and then interprets the procedure. When an object operation is encountered, Transaction Manipulator sends the operation request to Object Manipulator and waits for a response.
TRANSACTION check_out
EXTERNAL INPUT book_file, borrower_file, user
EXTERNAL OUTPUT book_file, borrower_file, user
Book_id <- user ? input("book id")
if (Book ? is_borrowed)
  user ! output("Book has been checked out"); exit
Borrower_name <- user ? input("borrower name")
Borrower <- borrower_file ? get_borrower(Borrower_name)
if (Borrower ? over_limit)
  user ! output("Has borrowed too many books"); exit
Borrower ! borrow(Book_id)
Book ! borrowed_by(Borrower_name)
END.

Figure 8. Transaction specification of the service function "check_out".

Figure 9. The prototyping environment.

(3) Object Manipulator is a table look-up mechanism that accepts the requests of object operations from Transaction Manipulator and searches for the proper results in state-transition-tables or query-tables stored in Object Data Base. The current states of objects are inquired from Object Manager. The Object Manipulator returns the results to Transaction Manipulator after a query request is finished, and sends a new state to Object Manager when a transition request is finished.

(4) Object Manager is a program which manages the states of objects stored in the working memory. The functions of Object Manager include creating a new object, deleting an object, and changing the state of an object.

(5) Execution Monitor supports the facilities for supervising the execution of prototypes. It contains the functions of tracing the execution history of a transaction, setting break points for checking the status of objects, and many other symbolic debugging kits.

A prototype is started to run by showing the main menu containing all the top level transaction-groups on the user interface. One of them can be selected by the user and activated. Then a new menu is shown again. By following the selection sequence, once a transaction is selected, the Transaction Manipulator will be triggered to execute it by loading and interpreting the corresponding intermediate form (a procedure) in the Transaction Data Base. During execution, any request of object operation encountered will be sent to the Object Manipulator, which determines the result by looking up the state-transition-table or query-table associated with the designated object. If it is a state transition request, a new state will be set for this object through the Object Manager, and if the request is a query, a proper result will be returned. After the execution of a transaction is finished, the last displayed menu will be activated again and wait for the selection of another transaction or returning to the preceding menu.

7. Conclusions and Future Works

The early construction of executable prototypes in the stage of system analysis will significantly improve the reliability and productivity of software development. SA is a widely accepted method for analyzing information systems. But, insufficiency of necessary control information description and lacking of formal primary operation definitions make it impossible to directly generate executable prototypes from SA results. In this paper we propose the use of a two-level model with the associated specification scheme to overcome these two problems. The approach offers the following advantages.

(1) Prototypes can be directly constructed from the intermediate results of SA; it makes prototyping coincide with the course of system analysis.

(2) The active components (i.e., the operation logic of system service functions) and the passive components (i.e., the characteristics of data objects) are separately specified in two appropriate description manners; that makes prototype specifications simpler and easy to read. Besides, there are no hard mathematical formulas in our prototype specifications.

(3) Primary object operations are defined in a table-filling form that avoids the necessity of implementing them in conventional programming languages. The table-filling form permits incomplete specification and facilitates incremental refinement.

By inheriting from the main theme of SA, in this paper, we
put emphasis on specifying the functional behavior of software prototypes and pay little attention to the description of user interface, e.g., screen format design. Since user interface plays an important role in prototyping, we will extend the specifications of prototypes for user interfaces in the future. Currently, a first version of the prototyping environment for executing prototype specifications has been designed and implemented by C language with an IBM PC/AT [12]. Practical examples (such as a simplified library system, an inventory management system, etc.) have been developed and tested on the prototyping environment, and the results show the feasibility and effectiveness of the proposed approach.

For larger information systems, it is still a complicated work to derive prototype specifications from a pile of data flow diagrams and mini-specs, and a long list of data dictionary. We will study to extend the RSPS capabilities to help deriving and maintaining prototype specifications.

References