SOFTWARE DESIGN VALIDATION TOOL

Loren C. Carpenter

Leonard L. Tripp

Boeing Computer Services, Inc.
Seattle, Washington

KEYWORDS AND PHRASES

Design Validation, Top Down Design, Structured Programming, Multilevel Modeling, Transition Diagram, Design Tree, Data Parcel, Design Expression

ABSTRACT

DECA is a computer program which is used in conjunction with a top-down dominated design methodology. The program organizes, validates, and produces a document depicting the design of a software system. The use of DECA significiantly enhances the quality of the software design. The quality of the design in turn significantly benefits the quality of the implemented software system.

INTRODUCTION

The construction of a reliable software system is predicated upon the existence of a sound design. The performance of the software industry to date indicates that adequate software designs are not commonly prepared.

A method is presented which significantly enhances the quality of a software design. The features of the method are:

- A sound theoretical basis.
- The ability to describe small to very large systems.
- A medium for communication between designers and requestor.
- All sequential processes can be described.
- A description of the design retrievable at any level of detail.
- Management of complexity is possible.
- Design can be described incrementally.
- Understandable by both programmer and non-programmer.
- A systematic format for program documentation.
- Decisions on design can be made at a point in the development process where they can be evaluated. This moves decisions from the code production phase to the design phase.
- Economical design evaluation possible.
- No inherent limitations placed on design by expression method.
- Design validation accomplished by consistency checks and by formalized testing procedures.
- Cost effective implementation of automated method feasible.

The theoretical basis for DECA is partially summarized in Burner (1973) which in turn relies on Zurcher and Randell (1968) for multilevel modeling and Parnas (1969) for transition diagrams.

TOP-DOWN DESIGN

The initial task of a designer of a software system is to determine a set of user requirements and how to satisfy them. In a top-down approach a software system is defined by levels (or successive refinements). The first or top level must be designed to satisfy the set of user requirements. Subsequent steps of the design process define the second and lower levels of the design. At each step the design progresses one level at a time. Every component (or element) on a level must be specified before the design process advances to the next level. This continues until the lowest level components have been designed. Proceeding a level at a time insures that the requirements of the previous level are satisfied before going on to lower levels. When the set of components on the lowest level has been designed, the entire system is designed with the confidence that it will satisfy the defined set of user requirements. The top-down design process reveals the architecture of the software system early in the design phase.

DESIGN EXPRESSION

The static structure of a software system is determined by relationships of containment. Processes contain data and other smaller processes. The dynamic structure of a system is determined by the temporal aspects of the system's processes. Decisions controlling the sequence of process execution and data manipulation are part of the dynamic structure. Software design expression is concerned with the static structure of a system, the dynamic structure of a system, and the relationship between them. Both the static and the dynamic aspects of a system must work harmoniously and as specified if the system is to perform as intended.

Three concepts have been identified as being adequate for the design expression of a software system. The three are a design tree, a transition diagram and a data parcel.

A tree (or design tree) is the natural structure of a hierarchically described system. The circles in Figure 1 are called nodes and the circle at the top is called the root node. In the tree of a hierarchical system, the nodes correspond to system components and the lines indicate a relationship of containment. The root node is said to be the top level of the design, and the nodes immediately below it are said to be on level one. Nodes two lines removed from the root node are on level two, and so forth. Since

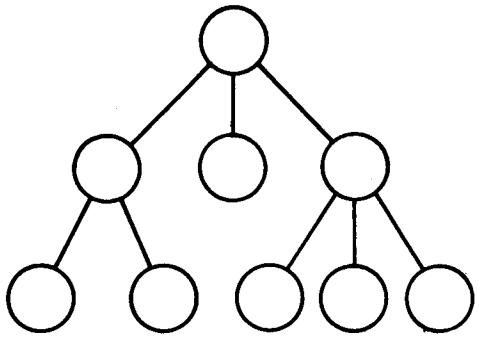


Figure 1

the lines specify containment, there are no "cross-links" and each level forms a complete partitioning of the root node, or system. The design tree depicts the vertical structure of the software system. However, it does not contain all the necessary information about the horizontal structure of the system.

To understand the concept of a design tree and multilevel top down design consider the following example:

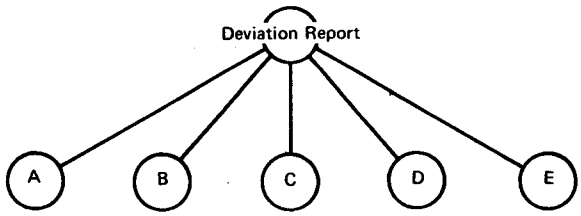
Produce from the company employee masterfile a report containing the average salary and age of all employees and the salary and age deviation of each employee.

Input	Process	Output
Employee Masterfile	Calculate Average Salary	Average Salary
	Calculate Average Age	Average Age
	Calculate Salary Deviation	Salary Deviation for Each Employee
	Calculate Age Deviation	Age Deviation for Each Employee

As a first step of design expression we will obtain a decomposition of the whole process into a set of basic subprocesses.

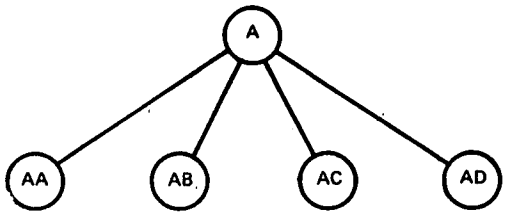
Subprocess	Description
A	Process Masterfile (read, edit, store table)
B	Compute Averages
C	Produce Report (deviations for each employee)
D	Write Error Message
E	Exit

The corresponding design tree would look like:



The next step of the design tree development would be to decompose each of the subprocesses. To illustrate this, node A will be decomposed.

Subprocess	Description
AA	Obtain Valid Record
AB	Store Record
AC	Sum Fields
AD	Exit



The decomposition of the nodes B through E (if required) would complete the design tree for this level.

The horizontal structure must define the interconnections between the components on each level, as the design proceeds from the top down. A simple notation to describe the horizontal relationships is a transition (or state) diagram.

Transition diagrams are constructed for each level of the design tree. Construction begins by viewing each node in the design tree as a state of the system. The diagram is a specification of the various conditions that cause a transfer of control between the states of the system. Transitions are dependent upon inputs to an individual state and without inputs (to a state) or a condition change, the system will not act. A name for a state is structured to indicate its position in the design tree. The name is composed of "syllables" separated by a delimiter. The number of syllables indicates the level number. The syllable components indicate in which branch of the tree the state is located.

A transition diagram is considered complete when the following information is given for each transition.

1. Origin state for the transition (from state).
2. Destination state for the transition (to state).
3. Description of the condition under which the transition occurs.
4. Inputs associated with the transition.
5. Data space changes associated with the transition.

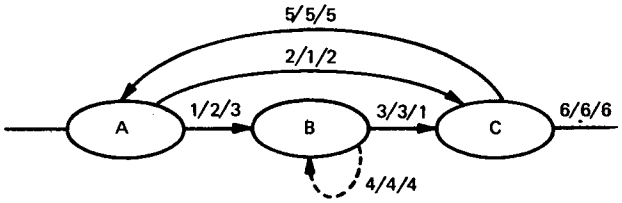


Figure 2

The graphical form of a transition diagram is given in Figure 2. The transition lines correspond to control statements in the code and the nodes represent one or more non-control statements. The i/j/k notation associated with the transitions denotes a reference to the condition, input, and data space change descriptions, respectively. The table form of the transition diagram is illustrated in Figure 3.

From	To	Condition	Input	Data Space Change
*A	B	1	2	3
*A	C	2	1	2
B	C	3	3	1
B	B	4	4	4
C	A	5	5	5
C	*	6	6	6

Where the asterisks (*) denote entrance and exit

Figure 3

The transition diagram will describe an arbitrary program. To ensure that the software system being designed will be properly structured a set of conventions was defined. The conventions restrict the set of diagrams that can be used in a design. First, to produce proper programs (one entry and one exit) it is necessary to consider that the transition logic occurs outside each node and each node is entered at one point and exited at one point (see Figure 4). However, in drawing transition diagrams, graphical clarity is increased if the arcs emanate from anywhere on the circumference of a node, as shown in Figure 5.

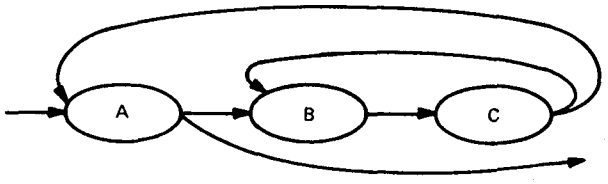


Figure 4

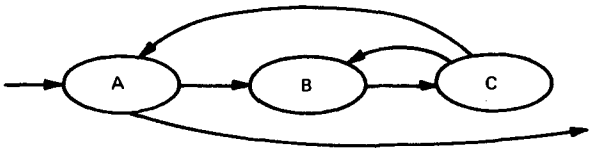


Figure 5

Proper block structure in the system design is accomplished by transition destination conventions. Each transition will occur exactly once in the design. Transitions are not refined as are states. The destination of a transition lies either within the transition diagram of the state containing the origin state of the transition or its destination is the parent state itself. Figure 6 illustrates these conventions.

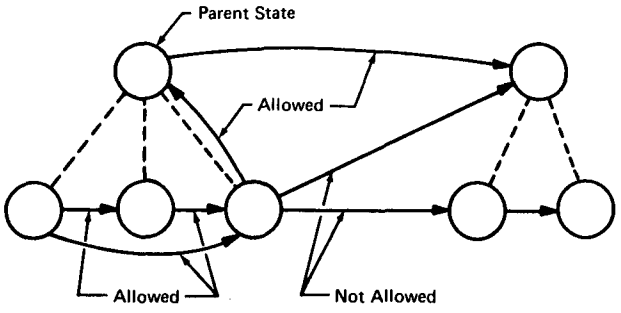


Figure 6

The transition diagrams corresponding to the three basic flowcharts of structured programming are illustrated in Figure 7. Since there is a one-to-one correspondence between flowcharts and transition diagrams, the representation theorem for flowcharts applies to transition diagrams. Any process which can be represented as a transition diagram can be represented as a combination of the basic transition diagrams. An algorithm for structuring a transition diagram equivalent to Bohm and Jacopini's for flowcharts has been derived in Carpenter, Redhed and Tripp (1974).

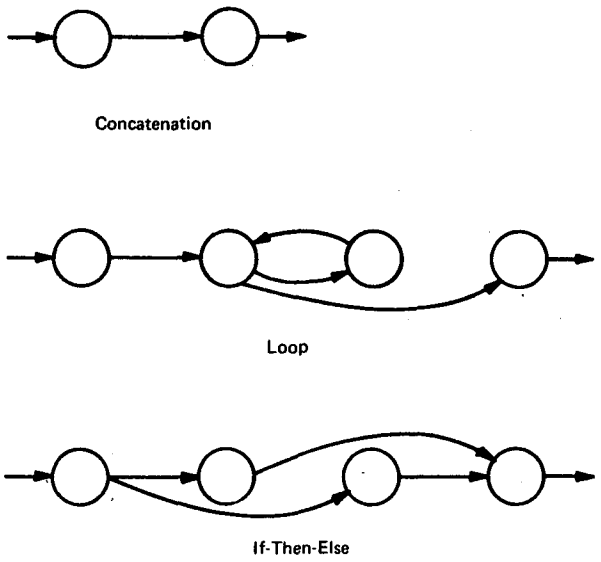


Figure 7

The level to level decomposition of the basic transition diagrams is illustrated in Figure 8. These decompositions are in their most general form.

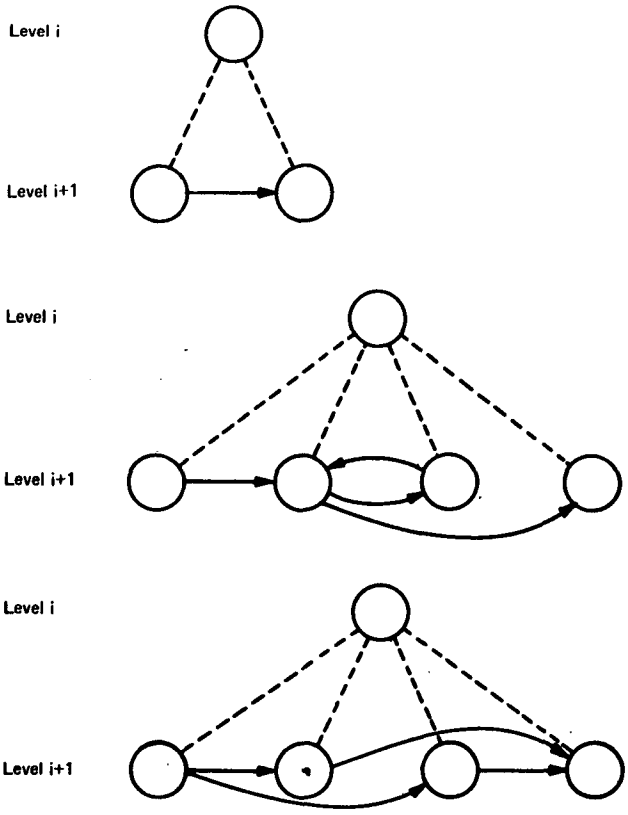
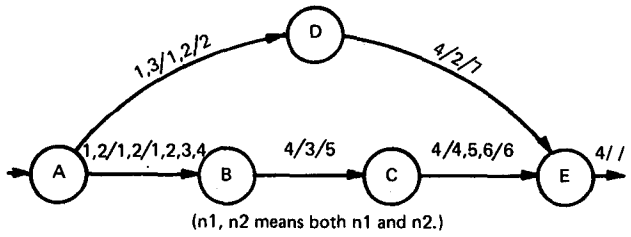


Figure 8

A graphical representation of the transition diagram for level one of the example is:



(n1, n2 means both n1 and n2.)

CONDITIONS

1 Masterfile Processed
2 Table Status OK
3 Table Status Not OK
4 Next in Sequence

INPUTS

1 Masterfile
2 Table Status
3 Total Age, Total Salary, Number of Employees Processed
4 Employee Table
5 Average Salary and Age
6 Number of Table Entries

DATA SPACE CHANGES

1 Complete Employee Table
2 Table Status
3 Totals of Age and Salary
4 Number of Entries in Table
5 Average Salary and Age
6 Report Line
7 Error Message

The above diagram would be input to DECA as:

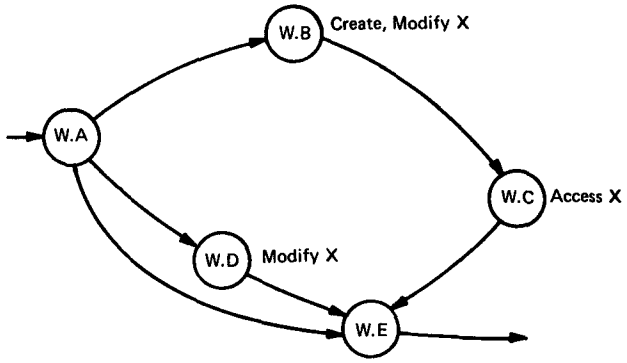
```
*  A  B  1,2/1,2/1,2,3,4
   A  D  1,3/1,2/2
   B  C  4/3/5
   C  E  4/4,5,6/6
   D  E  4/2/7
   E     4/ /
```

The transition diagram provides all the information needed in the design for control flow and data flow. A key requirement of software system design is the ability to express data requirements during design. To meet this requirement the concept of a data parcel was developed.

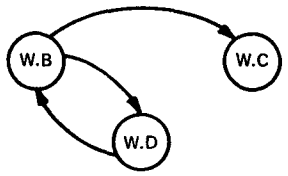
A data parcel is a named subset of the data space of the process being described. Its name is structured to indicate any hierarchical containment of parcels. Unlike a state name, its syllables may be several characters long. The description of a parcel consists of a prose explanation of its meaning and use, and syntactically formal components such as its format and location, and size and occurrence information.

Parcels are referenced by simply including their name (surrounded by brackets) in state and/or transition description text. In order to better formalize parcel activity, one or more of four parcel operators: create, access, modify and delete may immediately precede a parcel name.

The transition graph and parcel references together implicitly define a data parcel graph, for each data parcel. (Figure 9) Note that a data parcel graph represents all possible activity sequences for a parcel. It does not reflect static relationships like lists, rings, etc. Static relationships are described in the prose portion of a parcel description.



Transition Diagram for W



Parcel Graph for X

Figure 9

Parcels may be local, global, or external to a process. Global parcels are defined in some parent of the referencing process, while external parcels are defined in a parallel subtree of the system.

DESIGN VALIDATION

A key element in the production of reliable software is that there is a high degree of confidence that the software design is adequate. The design is validated by testing each level of the design. At level 1, testing is primarily conducted to satisfy the user requirements. Since later testing is done incrementally, from level to level, level 1 testing determines the quality of the software system. The testing falls into two broad categories:

Ensuring that the design works
Ensuring that the design satisfies the user requirements

The transition diagram is a vehicle to "walk through" the process simulating the system's actions. This becomes the basis for testing, to discover

Test 1 Having entered at the beginning, can the user terminate the process?
Test 2 Having entered at the beginning, can the user reach all system states?
Test 3 Being in any given state, can the user reach any other state?
Test 4 If all the paths which lead from one state to another are acceptable to the user.

Test 1, 2, and 3 have both necessary and sufficient conditions for acceptance. The necessary condition test establishes the necessary path exists, and these are mechanical. The test logic involves

constructing the adjacency matrix for the diagram and analyzing the corresponding reachability matrix. The sufficient condition test involves the examination of each input to see if the necessary conditions can be met to allow the required transition to take place. This is a manual test and should be performed by someone other than the designer. Test 4 is not easily done before the system is built. Using the transition diagram, judgments can be made about what is proper and what is comfortable. The testing at lower levels does not involve user requirements.

There are a set of design consistency checks which include:

1. All indicated inputs are available as specified in each transition description.
2. All data space changes are compatible and occur in the proper sequence.
3. Transition conditions can be satisfied from the available data.

Satisfaction of user requirements involves matching input and output specifications with the input and data space changes specified as external to the design process. These are documented in the transition diagram, and must be manually checked.

DESCRIPTION OF PROGRAM

The automatable aspects of the design methodology have been formalized and programmed. The current program, DECA III, is the third version of DECA, Design Expression and Confirmation Aid. Description of earlier versions is in Carpenter (1974). The program accepts a basically unordered line oriented description of a design tree, including its states, transitions, and data parcels. DECA makes extensive consistency and completeness tests on the tree, identifying errors and some questionable design practices. Then it produces a complete formatted listing of the design, with appropriate titles and page numbers in a form suitable for reproduction and publishing.

DECA is composed of five sequentially executed subprocesses. The first is a scanner which reads in a description of a design tree and its states, transitions, and data parcels. The scanner performs complete syntax checking of its input and prints all rejected lines with appropriate diagnostics. The primary function of the scanner is to append sort keys onto the accepted input records which are then passed on to the first sort.

The first sort performs all text ordering needed to produce the document. The first sort (and the second) is a standard system utility sort with user-written input and output interfaces.

Next comes the document printer. The document printer reads in, checks, and prints the ordered text, a state package at a time. A state package for a state consists of, for each substate of that state, the substate's structured name, its long name (up to sixty characters) and an optional body of prose describing in whatever language the user chooses the function of the substate. Following the substate descriptions is the transition table describing the transitions in this state package. Each transition entry is composed of an optional state package entry point flag, the structured name of the "from" state, the structured name of the "to" state if this is not an exit transition, and pointers to entries in the input, condition, and data space change tables for this state package. Pointers may be negated or combined with an "and" operator. The "or" operator was rejected since it introduces ambiguities.

The input, condition, and data space change tables are next. Each entry is given an integer label which is used to sort the table. Entries may contain English prose, tables, or most any printer-representable figure.

It is vitally important in software design to understand the functional characteristics of the states or subprocesses, that is, what happens to the input. DECA produces, for each state package, two matrices showing the functional relationship between the input and both the conditions and data space changes. This helps the reader verify that sufficient data is available to detect some condition or to perform some data space change. Data parcels referenced in the state package are listed next, along with the corresponding sub-state names and mode of reference.

The document printer makes extensive local consistency checks to ensure against ambiguity and incompleteness. It also performs a test for strong connectivity on each transition diagram. We assume the strong connectivity of all higher level transition diagrams by artifically connecting the exit state of a state package to its entry state.

In addition to the design tree which is primarily a control structure, the document printer produces a second part of the document which contains all user-defined data parcel descriptions, and where and how those data parcels are used, including expected storage requirements.

The last task of the document printer is to produce a directory giving page numbers for each state name and data parcel.

The document printer also constructs records for the second sort whose function is to order information for the fifth part of DECA, the global checker.

The global checker performs those tests requiring information from more than one state package. It prints both the top and the bottom of the tree, identifying all states which have no parent, (there had better be exactly one), and all states which have not yet been refined. It identifies data parcels that are created and not used or used and not created.

Finally, the global checker will, using the entire tree, and estimates of the bandwidth of inter-state data communication (taken from data parcel and transition descriptions), perform a modularization algorithm. This results in a recommended packaging structure which minimizes the coupling between modules.

RESULTS AND DISCUSSION

DECA has been used extensively with The Boeing Company and Boeing Computer Services for the past two years. Applied initially to the design of the NASA Integrated Program for Aerospace Vehicle Design (IPAD) System, it has since been employed on the Wiring Information Release System, Online Planning, Engineering Data System, and several smaller special-purpose projects. Customers have been impressed with the method's ability to display solutions to their requirements. This has two aspects. First, the customer understands his own requirements better, and second, he is better able to understand the designer's proposed solution.

DECA is operational on both IBM 370 and CDC 6000 computer systems. It is highly cost effective. A camera-ready document master costs less than 10 cents per page. In addition, DECA allows a system design to be maintained online, enabling design

modifications to be made, checked, and distributed almost immediately.

CONCLUSION

The Design Methodology, assisted by DECA, contributes to the production of reliable software for the following reasons:

1. The product is visible prior to coding, hence code changes are minimized.
2. The design is validated. A great number of conventionally difficult, tedious, and error prone checks have been automatically performed.
3. The document corresponds to the code. In fact, the code is produced directly from the document.
4. Structured programming is an integral part of the methodology, with respect to both control and data structures.
5. The quality of the design is maximal for the available time and money.

Two improvements are forthcoming for DECA-III. Alternative control structure schema, such as flowcharts, and limited entry decision tables, will be supported. A partial code generation capability, for control structures only, will produce GO-TO free equivalents of transition diagrams.

REFERENCES

1. Burner, H. B., "An Application of Automata Theory to the Multiple Level Top Down Design of Digital Computer Operating Systems," PhD Thesis, Washington State University, February, 1973.

2. Carpenter, L. C., "DECA, Design Expression and Confirmation Aid," 1974 Computer Science Conference, Detroit, Michigan, February 1974.

3. Carpenter, L. C., D. D. Redhed and L. L. Tripp, "The Systematic Development of Computer Software," Boeing Computer Services, Seattle, Wash. August 1974.

4. Parnas, D. L., "On the Use of Transition Diagrams in the Design of a User Interface for an Interactive Computing System," Proc. 24th National Conference ACM 1969, pp. 379-385.

5. Zurcher, F. W. and B. Randell, "Iterative Multilevel Modeling – A Methodology for Computer System Design," Proceedings of IFIP Congress 1968, pp. 138-142.