General Disclaimer

One or more of the Following Statements may affect this Document

- This document has been reproduced from the best copy furnished by the organizational source. It is being released in the interest of making available as much information as possible.

- This document may contain data, which exceeds the sheet parameters. It was furnished in this condition by the organizational source and is the best copy available.

- This document may contain tone-on-tone or color graphs, charts and/or pictures, which have been reproduced in black and white.

- This document is paginated as submitted by the original source.

- Portions of this document are not fully legible due to the historical nature of some of the material. However, it is the best reproduction available from the original submission.

Produced by NASA STI Compliance and Distribution Services

SOFTWARE DESIGN PROCEDURES*

Roger R. Bate
Advanced Software Technology Department
Texas Instruments Incorporated
Dallas, Texas

Abstract

The production of computer software, like hardware, requires tools, procedures and standards which increase productivity and quality. Our research has led to an improved methodology for evolutionary computer system development which includes software tools, e.g., a software library management system, a configuration processor, process design programming languages, a multiprocessor simulation system, and an automated documentation system. Procedures and standards for requirements specification, design methods, testing and quality assurance are necessary parts of a good methodology. An integrated set of software tools and appropriate procedures developed has demonstrated reduced costs and improved ability to predict software status milestones.

Introduction

Since 1972 Texas Instruments has been involved in the development of new methods for production of software for real-time systems, both under government contract, primarily with the Ballistic Missile Defense Advanced Technology Center [7, 10, 13], and under internally funded programs [6, 14]. This work has led to the development of tools, procedures and standards which allow the creation of software at lower cost and with more confidence that it will perform as required.

The key force driving the requirements for tools and procedures is the characteristics of what we call evolutionary computer system development [7, 10], an extension of techniques of top-down design [4, 17] emphasizing careful requirements definition followed by a top-level design of the computation to be performed, a selection or design of proper computer configuration, followed by an evolutionary software design which evolves software modules from stubs through models to final analytic code. Throughout evolutionary design the software is tested in a simulated system environment to assure conformance with functional and performance requirements. There is emphasis upon techniques of software validation to guide dynamic testing and permit static evaluation of the code.

Evolutionary Computer System Development

Let's be more specific about how an evolutionary design proceeds (Figure 1) and the impact of such a design on requirements for software tools. Firstly, the "top of the problem" must be firmly established. Others have addressed the need for careful requirements definition and validation before design begins [2, 11]. It is highly desirable that requirements be kept by an automated documentation system (e.g., [5, 7]) with the sections finely indexed so that traceability of test and design features to the requirements which generate them can be maintained throughout development and maintenance phases. Such a cross reference will aid in changing the design to respond to the inevitable changes in requirements.

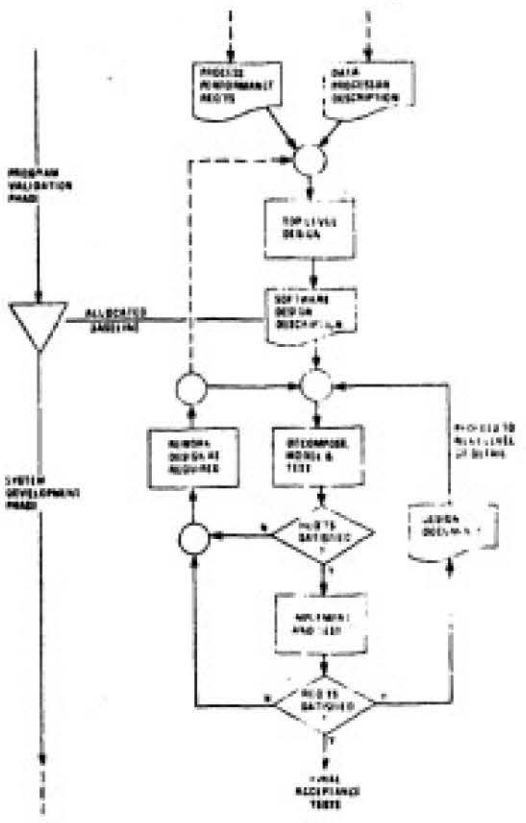


Figure 1. Overview of Process Design Methodology

Partitioning of system requirements into system functional components is the first step of design. This partitioning should be guided by trade studies and should attempt to simplify the interfaces between functional components. Description of the information which must flow between functions and the processing required of each function should be completed before design of that component is attempted. The specifications of the functional components become the requirements in the next stage of design. The partitioning process is then repeated until resulting subfunctions can be reasonably performed by a module which can be coded by one person and will run on one processor.

* This research was partially supported by Contract DAHC60-72-C-0156 from the Ballistic Missile Defense Advanced Technology Center. The findings of this report are not to be construed as an official Department of the Army position.

ORIGINAL PAGE IS POOR

Copyright © American Institute of Aeronautics and Astronautics, Inc., 1977. All rights reserved.

At the bottom level of the system design some perspective of the whole software is desirable in order to identify possibilities for common or standard subroutines (i.e., pure "top-down" design is insufficient). Usually it will be necessary to estimate size, timing, and development costs for system modules before detailed design is complete in order to establish realistic milestones for further progress through the software development cycle. The milestones should be tracked and compared to actual progress in order to give continuous management visibility. It will probably be desirable to code the modules in a high order language on a machine with adequate capacity and cross-compilation facilities to the system's target architecture and debugging tools in order to make good estimates. More will be said about this software "breadboard" later.

This process of evolutionary design can be carried out in one of the "structured" programming languages (e.g., Pascal-based languages [9, 13, 14]), by expressing the lower, as yet unspecified functional components as "stubs," procedures with empty bodies. In this way the evolving design can be run through a compiler for syntax checking at any time [7]. A form of evolving simulation can be provided by inserting clock update and memory utilization statements in the bodies of the stubs, to estimate the performance of the system.

Software Tools

In order to support evolutionary computer system development, we have defined requirements for and developed a number of software tools. Many of these tools have been used to develop real-time ballistic missile defense software for the Texas Instruments Advanced Scientific Computer as a demonstration of their effectiveness in reducing software costs [7, 10].

A central component of any software development system is a Software Library Management System (SLMS) which stores such things as the requirements for the computer system, source modules, object modules, documentation, input data, output data and configurations and their attendant change notices. It is noteworthy that the SLMS is useful in all stages of the software development cycle. These items of information are tied together in a wide variety of relationships, which can be interpreted by user defined operations. In addition, attributes of each of the entities such as status and consumption of resources can be stored. The clerical task of keeping track of all the various versions of the modules which are characteristic of evolutionary design is also assigned to the SLMS. Many of the other software tools access the Software Library Management System in making their reports about the status of the program. Configuration processors which construct various versions of the entire computer software system use relationships and attributes defined in the library. Other approaches to functions of the SLMS include those given in [3, 5, 8].

The Software Library Management System (Figure 2) manipulates a data structure which is a directed graph with labelled nodes and edges. Each node has associated with it a set of attributes with values. Because these values can include integers, strings, flags, nodes and files as well as lists constructed of these types it is possible to express very general characteristics and relationships between SLMS-managed entities (nodes). Both system and user-defined procedures may manipulate the base data structure. The flow of commands through the SLMS is shown in Figure 3.

SOFTWARE LIBRARY MANAGEMENT SYSTEM

- DATA STRUCTURE
  - A DIRECTED GRAPH WITH LABELED NODES AND EDGES.
  - ASSOCIATED WITH EACH NODE IS A LIST OF ATTRIBUTE-VALUE PAIRS.
    - AN ATTRIBUTE IS A USER DEFINED IDENTIFIER.
    - A VALUE IS ONE OR A LIST OF CONSTANTS OF GIVEN TYPE.
    - SOME TYPES ARE: INTEGER, STRING, FILE, NODE, BOOLEAN.
- OPERATIONS ON SLMS STRUCTURE
  - SYSTEM-DEFINED STATEMENTS.
  - USER-AND SYSTEM-DEFINED PROCEDURES.
  - EXTENSIVE TYPE CHECKING.
  - PERMIT COLLECTING FILES WHICH SATISFY SOME USER-DEFINED CRITERIA.

Figure 2.

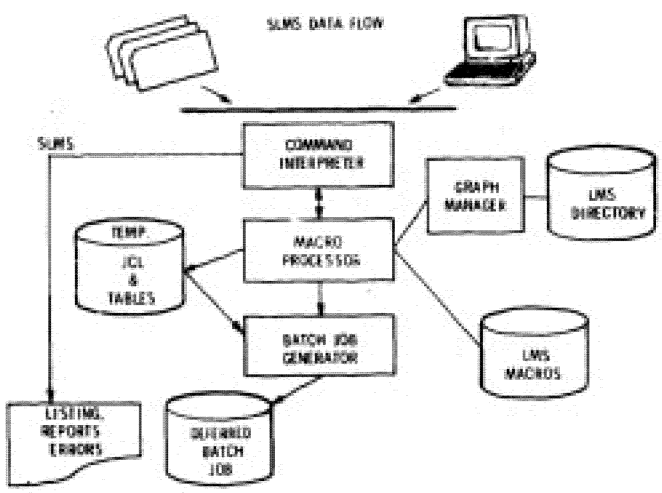


Figure 3.

Examples of user-defined procedures to establish the structure of Pascal-like modules in a program are shown in Figures 4 and 5. The user defines his procedures in a language which is similar to Pascal but has primitive statements for manipulating the SLMS graph structure. For example, the assignment statement

```
A.LANGUAGE:=LANGUAGE
```

adds an attribute LANGUAGE to the node A (if it is not already present) and sets its value to the value of LANGUAGE, which is a parameter to

REPRODUCIBILITY OF THE ORIGINAL PAGE IS POOR

the function CREATE_PROC. The statement

```
ADD(B, A.CALLS)
```

adds the node B to the attribute CALLS of node A. A syntax check will assure that A.CALLS is a list of nodes and the relation CALLS will cause a user-defined dynamic check of the nodes A and B for suitability to be related by CALLS.

The Software Library Management System also provides facilities for recording information about the status of the various modules of a system. This information can be accessed by a software management information system which generates reports reflecting the current status of each module in the system. This can be used in connection with the management procedures described in the next section. Finally, configuration control and separate compilation are much aided by our configuration processor component of the SLMS, which can perform tasks like assembling a given version of a software "process," passing specified source modules in correct program format to an appropriate compiler, and collecting proper object modules for link editing or overlay construction.

CREATION OF A CONFIGURATION OF PASCAL MODULES

```
PROGRAM PG;
  :
  :
  PROCEDURE PA;
  :
  END;
  :
  PROCEDURE PB;
  :
  :
  PA;
  END;
BEGIN
  :
  :
  PA;
  :
  PB;
END.
```

```
VAR A, B, C: NODES;
BEGIN
  A := CREATE_PROC('PA', 'PASCAL', NIL,
                                LIB.PRJ.PA.1, NIL);
  B := CREATE_PROC ('PB', 'PASCAL', 'PA',
                                LIB.PRJ.PA.2, NIL)
  C := CREATE_PROC ('PG', 'PASCAL', 'PA', 'PB',
                                LIB.PRJ.PG.1, NIL);
  A.FATHER:=C;
  B.FATHER:=C;
END;
```

Figure 4.

Programming languages are, of course, the fundamental software tools. Suitability of programming language for one's application area and methodological approach is of the utmost importance.

In choosing a language which it will support a company should consider its ability to represent algorithms and control without frequent recourse to assembly language, the ease of expressing common operations, and its capability to convey to the usually neglected maintainer a clear understanding of the code is supposed to do. The capability of moving source language programs (and the compiler) from one computer to another is important for a company which deals with several computers. Transportability can be attained if great care is taken to produce a complete and rigorous defunction of the syntax and semantics of the language and thoroughly test the compilers on various machines to make sure the implementations do not violate the definition. We have found Pascal-based programming languages to be particularly amenable to real-time systems applications [6, 13, 14], and it is noteworthy that the new Department of Defense language for embedded computer systems applications will have a Pascal base [1].

```
FUNCTION CREATE_PROC(
    NAME, LANGUAGE:STRING;
    CALLS:LIST OF STRING;
    SOURCE:FILE;
    FATHER:NODE):NODE;

VAR A, B:NODE;
    S:STRING;

BEGIN
    CREATE(A); CREATE_PROC:=A;
    A.NAME:=NAME;
    A.LANGUAGE:=LANGUAGE;
    WHILE CALLS ≠ NIL DO BEGIN
        S:=FIRST (CALLS);
        CALLS:=REST(CALLS);
        FIND (B, NAME=S);
        ADD (B, A.CALLS) END;
    A.SOURCE:=SOURCE;
    ADD(A, FATHER.SUBPROCEDURES)

END;
```

Figure 5.
Definition of the Function CREATE_PROC

Well-supported compilers with adequate software libraries of standard procedures and with adequate debugging tools are another major consideration in the selection of a language to be used on a project. Disaster is lurking for a project which chooses a new language for which it must develop the compiler, unless it has both time and money to test the compiler thoroughly, provide adequate documentation and user training and construct a full set of related software development tools to make the new language safe and easy to use.

Several software tools for simulation, performance analysis, and documentation are worthy of mention here. We have developed an event-driven multi-facility time analysis simulator for performing simulations of distributed processing systems. This simulation has FORTRAN as its user interface language, provides a wide range of automatic default operation nodes with user override if desirable, and generates appropriate output reports at each stage of the simulation.

In order to assist with the testing and validation of software we have developed path analyzers of FORTRAN and extended Pascal programs

which can drive automated testing and performance measurements. Source level assertions [12] have also been incorporated into our translators.

One of the most unpleasant tasks which a software designer faces is the problem of writing adequate documentation for his system and keeping it up to date and in agreement with the actual state of the system. We believe that it is necessary to make that process as palatable and as easy as possible. For that reason we have provided a number of documentation aids which enable us to store, correct, and print in high quality form the total documentation of the software, both in the high level and any intermediate level of description which is desired.

Management Procedures

An important part of the software development methodology which we have adopted is the proper procedures for management and designers to follow in building the software and using the software tools which are available. We believe that it is extremely important to establish milestones for each component which can be tracked on a daily basis. For example, for one program we have established 23 steps through which each module goes and are recording this information in the management information system component for our SLMS (Figure 6). We produce reports periodically to indicate status and also to indicate the amount of effort required to complete the project. Management must understand the proper steps of software development and insist both that all steps are carried out in the proper order and that the achievement of each milestone be properly recorded.

We have also adopted a universal practice of building a software breadboard (Figure 7). This is a quick functional version of all the software processing to be done, written in a higher order language, usually on a very large machine (with cross compilation facilities to the system's target machine to aid critical module space estimates), so that the logical interrelationships of the functions and the behavior of the system can be assessed before the very expensive final software is undertaken. There is considerable value in having gone through such a software breadboard: the designers understand the system better and right answers are available to compare with those from final versions of software. Management must understand the value of the software breadboard and must schedule and budget for this step. All too often the natural tendency of managers is to assume that they can write software right the first time. Attempting to write in final form what eventually turns out to be the breadboard is expensive and virtually assures a cost and schedule overrun.

Finally, we have produced a software design guide [15] and software management standard [16], which are distillations of the experience we have had in developing software in the company. These documents give lists of do's and don't's in software management and development and record the current status of our software experience in a form which is suitable for use by future programs.

SAMPLE SOFTWARE DEVELOPMENT MILESTONES: FUNCTIONS/TASKS

STATUS	DESCRIPTION
F1	IDENTIFY TASKS/DATA FILES
F2	DEVELOP TASK STUBS/DATA FILE MODELS
F3	ALLOCATE TASKS TO COMPUTERS
F4	LOADING ANALYSIS VIA SIMULATION
F5	CODE INITIAL TASK SEQUENCING CONTROL LOGIC
F6	REFINEMENT OF TASK MODELS
F7	DEFINE DATA FILE ATTRIBUTES/ORGANIZATION
F8	ANALYZE CONTROL STRUCTURE/INTERFACES VIA SIMULATION
F9	DEFINE TEST PLAN FOR FUNCTION
F10	DEFINE REQUIREMENTS OF SUBTASK
F11	DEFINE CONTROL LOGIC FOR TASKS
F12	DOCUMENT DESIGN OF TASKS/REQUIREMENTS OF SUBTASKS/DATA FILES
F13	INTERNAL REVIEW
F14	CRITICAL DESIGN REVIEW
F15	DESIGN SUBTASKS
F16	DOCUMENT SUBTASK DESIGN
F17	370 UNIT TESTS
F18	990 UNIT TESTS
F19	370 LOOP TESTS
F20	COMPLETE SUBTASK CODE AND DOCUMENTATION
F21	990 LOOP TESTS
F22	990 SYSTEM TESTS
F23	COMPLETE DOCUMENTATION

Figure 6.

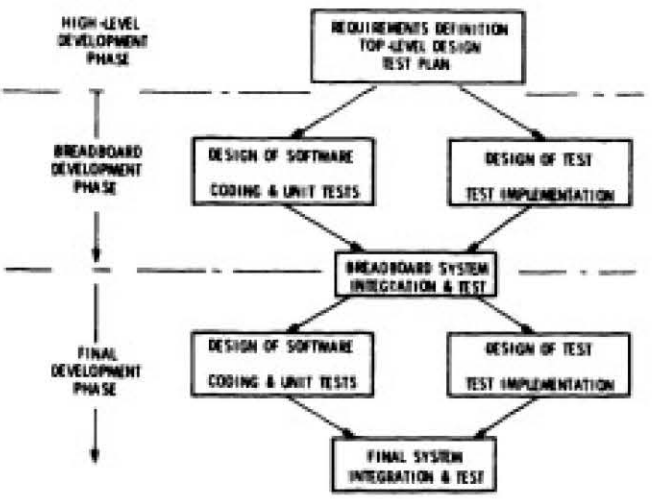


Figure 7.
Software Development With Breadboard

Concluding Remarks

In summary, we have discovered that successful software development depends not upon exotic methods, but more upon the application of good common sense in planning, establishing, and maintaining milestones and budgets and insistence upon the use of available software tools as a regular part of the activity of all software

people. The systematic application of a carefully prepared software methodology may be the key to the easing of the burgeoning "software crisis."