# 구조적 프로그램을 위한 소프트웨어 Tool 의 개발

°이 동춘* 김 성종** 김 창복** 신 인철** 이 상범**
*삼성종합기술원 **단국대학교 전자공학과

# Software Tool Development for Structured Programming

Dong Choon Lee*, Seong Jong Kim**, Chang Bok Kim**, In Chul Shin**, Sang Burm Rhee*
* Samsung Advanced Institute of Technology ** Dept. of Electronics, Dankook Univ.

Abstract

The purpose of this study is to develop a systematic tool which can reduce the abstractness of the structured programming disciplines by a system. The system is an interactive software which forces the user to think and write software in a hierarchical stepwise refinement fashion for an implementation of the software design. The program produced by the system has a program control structure and a logic flow which are very easy to recognize. The modification of a program is, therefore, easier to attain by altering the specifications of the modules involved. It is possible to reduce the programming errors because of those characteristics.

## I. Introduction

Structured programming may be summarized as building a software system in stages of hierarchical levels, at each level hiding out some of the physical details to successive lower levels which is sometimes termed as top-down design or levels of abstraction. It suggests a certain set of disciplines and guidelines to be followed in the development of softwares. Most of the guide lines and disciplines are abstract in nature, which is not a surprise since it needs to be appled in a wide range of softwares and on variety of languages.

The structured programming can be disseminated for its many well taken advantages over conventional programming techniques. Yet, with the lack of precise and less abstract guidelines, the structured programming may well be misinterpreted or may result in a bad software design. It may not be claimed that we can do away with the abstract nature of the structuredness of a software, but we may reduce the abstractness of the structured programming disciplines by a system which forces a programmer to follow certain essential set of rules, such as, stepwise refinement disciplins, using subprogram in the devlopment stages.

## II. System Design

As mentioned in the previous section, we need a systematic way to implement a structured sofware hierarchically. Hierarchical top-down

design (program design) is mainly based on the idea of levels of abstraction which become levels of modules in a program. If we use the levels of abstraction, we can call it a modular design. The main idea of our system is based on the modular design. We can divide our system into four programs which perform four functions:

1) Accept -- specify modules.
2) Merge -- merge the lower level module specification into the higher level modules.
3) Enter -- accepts input conditions for transition from a module to another.
4) Genpro -- generates Pascal program.

Developing complete specifications of a module at once is difficult in the early stages of a design. It will be easier for a complex operation to build specifications in a hierarchical form leaving a certain amount of details to be refined later. It is defined as a compound module with an abstract meaning which wil be refined.

The specification for a module which contains high level specificaations are called "base module specifications." The corresponding refinement for a compound module is called a "refinement module specification." Furthermore, a module in base module specification which has to be refined is called the "target module."

A. Producing the Modules

In the proposed system, a program called "accept" is developed which interacts with the user for accepting the specifications of each module. Processes in a module can be represented by their states and transitions. Each state represents the current condition of the process, which is a progress of exchanging messages (program control) between modules (procedures). In each module, there are some processes to be executed before the program control is transferred to another module. A state can be classified as either a wait state, a loop state, or a default transition state.

In a wait state, an initializing sequence of processing will be done. The input conditions will be checked to decide the module to which the program control has to be transferred. If there are no conditions for transition, the process will remain in the wait state until an input condition meets the condition for transition.

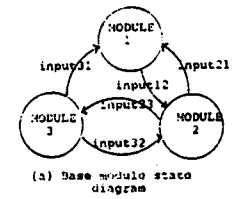
In a loop state, the necessary sequence of processing will be done repeatedly until an input condition is satisfied. if input condition is satisfied, the program control will be transferred to another module according to the input condition.

In a default transition state, the transition to another module, which is specified by default transition, will be done if every input condition does not match. Regardless of the module specified in the default transition, appropriate transition will be accomplished if there are any matching input conditions. Here, all input conditions are represented as mnemonic for convinience. It will then be changed into boolean, numerical values, or characters according to the necessity. Refinement module

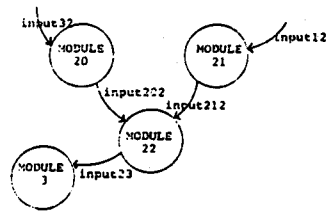
specification will follow the same steps as base module specification. There are no limitations on repeating the refinement step until it meets the goals of a module. The condition of program controls will be indicated later and then be stored in another file for easy modifications. In that manner, if a programmer wants to modify the types of numerical values of the program control, all he has to do is change the specification table instead of changing the program itself. Therefore, the functions of lower-module will not be affected by the changes in the program controls since those controls flow from the higher-module to lower-module.

B. Merging of Modules

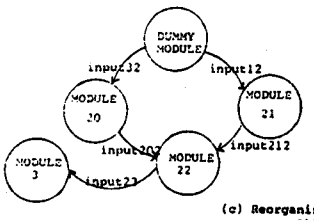
A program called "merge" is developed for accepting both the name of a file for base module specification and the name of a file for the refinement module specification. It will also merge them automatically to generate a merged file. Two files are merged into one file physically, but the base module specification will be logically compounded with the refinement module specifications. This merging process could be repeated as many times as needed until the final (lowest) module specifications ar derived. When we merge two files, we may have the problem of introducing multiple transition into the separate modules of the refinement module structure. For example, suppose we have a base module specification consisting of three modules, and a refinement module specification consisting of three modules for module 2 (target module) shown in Fig. 1. When we merge those two together, we have two input conditions, input 12 and input 32. These conditions are from other modules to module 2 which has to be refined. We need, therefore, to introduce a dummy module which is added to the beginning of the refinement module structure in order to keep the structure of refinementt module specification identical with the structure of base module specification. (see Fig. 1 (c)") By doing that, all transitions enter the dummy module and then it will be removed in a later step. Table 1 is an example of dummy module specification.




(a) Base module state diagram




(b) Refinement module state diagram




(c) Reorganized refinement module state diagram

Fig. 6. State diagram for dummy module.

TABLE 1

DUMMY MODULE SPECIFICATION

| | | |
|---|---|---|
| dummy (module name) | | |
| begin sub<br>empty<br>end sub | | Subprocess Part |
| input 12<br>input 32 | module 21<br>module 20 | Module Transition Part |
| end module | | |

C. Entering Boolean Condition

A program "enter" is developed for accepting the actual values of input conditions for each module of the lowest (final) module specificattion, finding out every minterm covered by the input mnemonic and storing those information in a table form termed as "condition specification."

D. Generating the Program

For generating the program, a program called "genpro" is developed. "Genpro" generates the Pascal program with the final module specification and the condition specification in a way that each module is programmed as a procedure, and the transfer of program control from a module to another is attained through procedure calls. To maintain an idea of top-down development of a design, it is effective to make the procedure calls from a present procedure. We possibly have, therefore, a problem of infinite procedure calls. But in our system, there is a capability of having infinite procedure calls. Because "genpro" preprocesses the final module specifications and removes the calls from within the each procedures and places them in a table in a main program. The program control will be driven by the table so that user's specifications for refinement will not be affected. But "genpro" translates the specifications into an executable form.

## III. Conclusion

An interactive software system, which forces the user to think and write software in a hierarchical stepwise refinement fashion, has been brought into activity to furnish the developer with a systematic tool for a implementation of the software design. Our system is adaptable to a mini-computer environment. The program produced by the system has a program control structure and a logic flow which are very easy to recognize at a glance. The modification of a program is, therefore, easier to attain by altering the specifications of the modules (procedures) involved. It is possible to reduce the programming errors because of the above mentioned characteristics, also.

Though our system does not intend to convert the pseudo code completely into the Pascal code, it accommodates a tool for a systematic specification and implementation of a softwaare design. Implementation of the system in perrsonal computer and enhancement of the system with more functions can be defined as further research areas.

## REFERENCES

[1] E. F. Miller and G. E. Lindamood, "Structured programming : Top-down approach," Datamation, Vol. 19, No. 12, Dec. 1973.

[2] E. J. Dijkstra, "Notes on structured pro gramming," Technische Hogeschool Eindhoven, Report No. EWD-248, 70-WSK-0349, April 1970.

[3] J. K. Hughes and J. I. Michton, A structured approach to programming, Prentice Hall, New Jersey, 1977.

[4] N. Wirth, "On the composition of well-structured programs," Computing Survey, Vol. 6, No. 4, pp. 247-259, Dec. 1974.