PUC

MONOGRA 1974

Series: Monographs in Computer Science and Computer Applications

Nº 4/74

ON THE SYNTHESIS OF PROGRAMS AND RELIABILITY

by

Carlos Jose P. Lucena

Computer Science Department - Rio Datacenter

Pontifícia Universidade Católica do Rio de Janeiro
Rua Marquês de São Vicente, 209 — ZC-20
Rio de Janeiro — Brasil

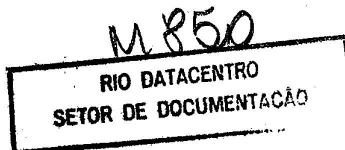
ON THE SYNTHESIS OF PROGRAMS AND RELIABILITY *

Carlos Jose P. Lucena
Associate Professor
Computer Science Department

DIVISÃO DE INFORMAÇÕES
código/registro 0925
data 9/4/25

Series Editor: Prof. S. M. dos Santos

M850



October /1974

* This work was first published in the Proceedings of the "Seventh Asilomar Conference on Circuits, Systems and Computers. - Pacific Grove - California - USA Mov. 1973.

ABSTRACT

This paper discusses the problem of synthesis of reliable programs. Reliability requires a good approximation of the solution of the problem of correspondence between a program and its flowchart. Reliability also calls for the ability of program strutures to fail-softly, to permit simple modification, and to permit large-scale composability.

1. INTRODUCTION

This paper will be dealing with the problem of prescribing synthesis rules for the construction of programs in such a way that the synthesis procedure guarantees a reliable final product. The word reliable will be used in this text with a double meaning. We will use the expression "reliable programs" both in the sense of the correspondence between a program and its flowchart and in the sense of a program possessing a property called generality . Moreover, we will be concerned with the development of large programming systems.

The two established approaches to this problem are generally called methods for proving the correctness of programs and methods of structured programming. Without eleborating on the nuances of the existing formulation of these methods, we will try to summarize them in the following way:

(1) Methods of proving program correctness. Steps :

(a) Synthesis of the program

(b) Synthesis of a set of assertions about the program involving program variables

(c) Informal or formal proof of equivalence between the program and its corresponding assertions

(2) Method of structured programming. Steps:

(a) Synthesis of the program according to a set of synthesis rules that encompass a form of informal proof of correctness

(b) The program is "correct".

A number of strong limitations prevent universal application now of either method. Some of them can be found in a very complete paper by Elspas et al. [ELS72]. Regarding the practability of the methods of proving program correctness, some comments pertain to the fact that the writing of assertions is a process which is as open and as creative as the writing of the program itself. Others refer to the limitations of informal proofs and the complexity of the automatic ones. Increasing activity of researchers gives hope of significant developments in this vital research.

The general attitude found in the methods of structered programming is likely to guide the development of the area of software construction for quite some time. Nevertheless, although thie methodology has already contributed to the discipline of the area, many open problems related to its current formulation are waiting for solutions [LUC73]. They can be summarized in the following way:

(1) As they exist at present, the recommended synthesis rules for structured programming cannot, in general, be followed unambiguously.

(2) Errors do occur in structured programs, and their recovery is not straighforward [PAR72].

(3) As defined at present, the method conflicts with the objective of program generality discussed in Section 3.

This paper describes some of the features of the DPTD System which aims at the integration of the actions of Design, Programming, Testing, and Debugging for the production of reliable software systems. The basic systems' feature consists of a formalism for program design that models a rigorous concept of modularity and enhances a systematic approach to testing

and error recovery.

The DPTD System can be briefly defined by the pair (R,E). R is a set of rules for program synthesis for reliability (in the sense presented before). These rules include consideration of the need for testing and debugging of the program during the synthesis process. E is a programming system in a special form of representation (see Section 3) and provides aids for systematic testing and debugging.

The system is in an early phase of design. Most of its basic concepts are being exercised (through the use of simulation of most of its features) for the synthesis of some application programs, including a Management Information System. The aspects to be emphasized in this paper are the techniques for stepwise composition of programs that will preserve the property of generality.

2. DESIGN METHODOLOGIES

We shall not review here the basic philosophy and techniques of structured programming. That can be found in a series of Dijkstra's papers [DIJ68, DIJ69, DDH72].

We want to focus on the dynamics of hierarchical system development, and in particular on the modeling of "levels of abstraction".

We will start by pointing out the difference between partitioning a system into modules* in the sense of [PAR72b] and the organization of a system

*The word module in this section will be used in the sense in which it is used in the referenced literature.

in levels of abstraction. The partition into modules is assumed to consist of "analyzing the execution-time flow of the system and organizing the system structure around each major sequential task" [LIS72]. This strategy apparently conflicts with an approach where abstractions are first introduced in an order that follows the process of understanding the system by the designer.

Our following comments reflect some preliminary observations drawn from the experience we are gaining by comparing in practice the use of different design methodologies [LUT73].

When large systems are being considered, it seems that module decomposition in the sense described previously is the first idea that is formed about that systems. Based on this conjecture, we will proceed by illustrating how we can benefit from both the idea of modules and levels of abstraction.

In our proposal the idea i_0 in the first level of abstraction is defined by $i_0 = \{c_0, F_0\}$. In the definition the element c_0 is a set of control statements selected from the group of basic control mechanisms mentioned at the beginning of the section. F_0 is a family of functions expressed in words whose meanings are known at the respective level of abstractions. We want the functions in F_0 to be equivalent to a <u>first modularization</u> in the sense used in this section. Each function in F_0 will be expressed in ideas $i_1 = \{c_1, F_1\}$ and the tree-shape process will continue till a kth level that we will call <u>implementation level</u>. At this point, we want to make a few comments about the dynamics of this design approach.

(1) Supposedly level 0 was easy to specify through this strategy.

(2) The description of the functions (modules) will take place simultaneously so as to avoid the duplication of work mentioned before.

(3) An initial selection of modules can be revised in the following ways:

(a) The specification of the initial functions in $i_j = \{c_j, F_j\}$ can be reviewed after the first specification of a level j+1 and this revision can reflect back to the first level of the system.

(b) The systems modules can be reconfigured by the combinations of complete or partial definitions of functions.

The concept of an implementation level was created to make compatible the following requirements;

(1) The idea of the global evolution of a large system is desirable to avoid the problems of duplication of work and to allow the refinement mentioned before.

(2) The system must not grow to the point of becoming unmanageable by the main designer(s) (we are talking about large systems).

(3) The system has to be partitionable in a well-defined fashion for management purposes (team work).

Suppose now that instead of defining at each system's level the data structures in the form that they will actually be implemented, we use a language L that allows us to define implementation-independent data structures

(more will be said about L in the next section). We shall call k^{th} level an implementation level if it is such that to carry on the specification from that point the designer would actually have to get involved with implementation decisions. At this level the centralized design specification is stopped and the structured design is vertically partitioned with an eventual reconfiguration of modules. Graphically we would have the situation illustrated in Figure 1.

In the picture, $f_{1 \circ 2}$ indicates that the function f_j was composed with $f_2(f_1 \circ f_2)$ taking into consideration all the repercussions on the other levels. Observe that the implementation level can be attained at different stages of function specifications. The level k is the level in which no more functions can be expanded without concern for the implementation. Note that we can call L a language for module specification that responds to the need expressed by certain authors (e.g., [PAR72b]) for a formalized means of module specification. We proceed with this work concentrating on how programs may be composed to model a system described by the proposed design methodology. Our method of design so far did not consider the problem of data connections between modules. This will be a central topic in the next section.

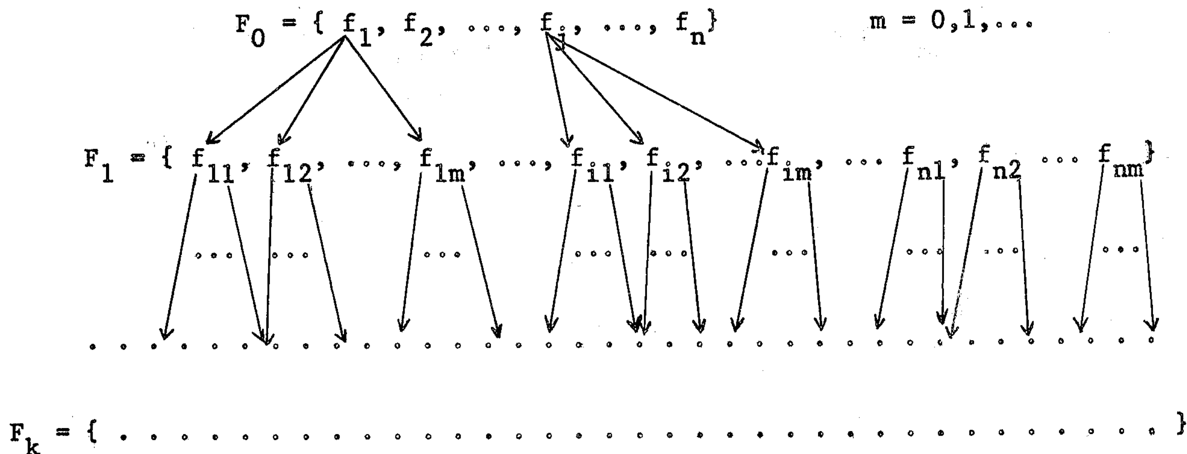


Figure 1. Module Reconfiguration

Since we will use the word module from now on in a more restricted sense, we shall refer in the sequel to what we called a module in this section as a subsystem (SS) of the general software system under consideration.

3. THE SYNTHESIS OF PROGRAMS FOR PROGRAM GENERALITY

In the preceding section, we dealt with the problem of synthesis of reliable programs in the sense that a systematic approach to design helps to build programs that maintain the programmer's original intent. In this section, we want additionally to guarantee that programming systems are engineered having in mind the unreliability of the computer systems in which they will perform. Other desireable software properties of computer systems such as changeability and large scale composability will also be discussed.

This section reflects our belief that programming languages (of the ALGOL variety) must be used with special caution and discipline by programmers developing a large programming system. The mention of special features which are not encountered in current programming languages are to be understood as being supplied by the systems environment.

We will be talking about program generality in essentially the same sense as J. Dennis [DEN68, DEN73]. Several of the ideas in this section were influenced by his work.

The following discussion will deal with procedures, programs, and classes of programs. Programs and procedures will be dealt with as functions.

With the purpose of emphasizing the order in which the synthesis process is to take place, we chose to talk about <u>programs</u> as being composed of <u>procedures</u>.

We shall use the term procedure in the sense of the ALGOL 60 definition. In the present context, a procedure could also be called a <u>sub-program</u>.

A procedure π can be characterized in terms of two logical propositions p_1 and p_2 such that if p_1 is true before execution of π then p_2 will be true after the execution.

The propositions p_1 and p_2 involve all variables used in π as well as components describing the state of all variables.

The variables in π can in turn be classified as <u>local</u> or <u>global</u>. A global variable would be any variable referenced in π such that not all uses of the variable are contained in π [WUSH73]. A local variable would have all its uses contained in π.

Procedures can be composed to any extent with other procedures to form more complex ones.

The notation

$$\pi_n \doteq \pi_0 \circ \pi_1 \circ \ldots \circ \pi_{n-1}$$

represents this composition. The reason for composing is to combine the effect to various procedures. The operator $\circ$ is realized through the control structures defined in Section 2 and the syntactic and semantic rules of ALGOL. The components of a composite procedures will be required to have unique names. We shall

be interested in formulating some practical conditions, to be incorporated in our synthesis rules, that will allow us to use a strong inference rule associated with the composition of procedures.

To state the rule we need only to deal with two procedures, say, $(\pi_o \circ \pi_1)$, since other sequences can be generated by nesting, like

$$(\pi_o \circ \pi_1^{\circ} \ldots (\pi_{n-1} \circ \pi_n)\ldots)))$$

We shall rewrite the characterization of a procedure in terms of p_1 and p_2 as [HOA69]

$$p_1\{\pi\}p_2$$

The inference rule of composition is then stated as

$$p_1\{\pi_o\}p_2 \text{ and } p_2\{\pi_1\}p_3 \text{ than } p_1\{\pi_o \circ \pi_1)\}p_3$$

In other words, we want to "prove" that a certain program works by constructing it through small correct units.

Having this desirable result in mind, we will discuss now some aspects of the composition of procedures that will play a fundamental role in the synthesis of programs for reliability.

Def. 1 A composition of procedures is said to be carried out with syntactic non-interference if no changes need to be made to the components to obtain the final product.

Def. 2 The dependence of a procedure on another through a CALL will not preclude the testing of the calling procedure without the knowledge of

the called procedure (the expected ranges of output variables will be sufficient for the scheme of test to be applied).

Def. 3 The components of a composite procedure are said to have semantic contex-independence if the global test of the composite procedure can be accomplished through the sum of the individual tests of component procedures.

To reflect a few of the considerations that will prevail during program synthesis, we will present some very simple theorems accompanied by sketchy proofs.

Theorem 1: Syntactic non-interference can only be accomplished in general if global variables are not present among the components.

Proof: The existence of global variables would make possible compositions where the following would occur:

```
A:  Proc.
    <def. of x with meaning # 1>
    B:  Proc.
        <def. of x with meaning # 2>
        C:  Proc.
            <use of x with either meaning # 1 or # 2>
        end;
    end;
end;
```

The necessary renaming would be a change which is not allowed by Definition 1.

Theorem 2: Semantic context-independence cannot exist when global variables are present.

Proof: Let us call G the following generic procedure containing global variables.

```
G:  Proc (p1, p2, ..., pn)
<g1, g2, ..., gn>
............
end;
```

We want to test G exhaustively (or any simplified version of that) by exercising all its paths. After determining all the conditions in terms of p_1, p_2, ..., p_0 to exercise a given path, there is no way of discovering if a set of values g_1, g_2, ...,g_n would be compatible with the p's for exercising that path without inspecting other procedures to determine the ranges of variables. That violates Definition 3.

It is important to note that non-existence of global variables is only a necessary condition to avoid the fulfillment of Definition 3. For Definition 3 to work in all cases, it is additionally required that it be possible to determine the ranges of all input parameters of a procedure without reference to the context (this has to be provided for during the synthesis of the program).

Theorem 3: Semantic context-independence cannot exist in general when procedure parameters are passed through a "call-by-name" mode.

Proof: The application of the ALGOL replacement rule for parameters is such that if procedure p_1 calls procedure p_2 passing its parameters by name, the following would take place. There is no way through p_1 can predict the values of its variables that were used as parameters after the execution of p_2 without knowing a complete definition of p_2. This fact would prevent a complete independent test of p_1. Therefore, p_1 would not be semantically context-indepentent.

The above results illustrate how language features (global variables, call-by-name) can interfere with the desirable feature of program composability which is aimed at enabling the distribution of the programming task. We will continue by showing how language feature utilization can be further regulated.

Def. 4 We will call a composite procedure π expressed by

$$\pi_i = \pi_o \circ \pi_1 \circ \ldots \circ \pi_n$$

a proper program if the component procedures π_o, π_1 ... π_n have semantic context independence.

Def. 5 A composite program M', written as

$$M' = \pi_o \circ \pi_1 \circ \ldots \circ \pi_n$$

is called a semi-module if the program was composed with syntactic non-interference and semantic context-independence.

Definition 4 implies an idea of ordering in the synthesis process. A program π is not necessarily synthesized with syntactic non-interference. Therefore it is required that when eventual substitutions are taking place during composition, provision be made for the specification of ranges of global variables in the procedures where they do appear. As we saw , this last measure plus the independence of meanings will guarantee the requirement of semantic context-independence.

Definition 5 can in practice be achieved through independent compilation of the programs π.

We stressed the difference among procedures, programs and semi-modules based on composition criteria because we want those elements to play different roles in the process of program synthesis. They will act as models for the different levels of abstraction and components of levels of abstraction we discussed in the preceding section.

Observe that the programming process of coming from the procedure to the semi-module level has to be based on very fine design specifications. Even if the various programs π were written by different programmers, a very detailed description would be required as to how the data structure interfaces were to be handled. We would not like to use a semi-module to model the sub-systems as characterized in Section 2 because the same level of detail just mentioned for the semi-module would have to be applied on a very large scale.

We shall now introduce two definitions that will try to make more precise our ideas about the requirement that a software component should have to model a sub-system of a hierarchical system.

Def. 6 A computation method is bound if its corresponding algorithm is determined by a set of data structure implementations.

Example: Write a program that finds the product of two matrices that are represented as two one-dimensional arrays.

Def. 7 A computational method is free if its corresponding algorithm can specify the way in which the data structure will be implemented.

Example: Write a program that finds the product of two matrices.

The computer algorithm will be free to use any internal representation that is convenient for the context in which it will perform.

We shall be looking now for concepts of syntatic non-interference and semantic context-independence applied to large composite programs modeled as semi-modules.

We called L in Section 2 a language which enables the definition of very general data structures and that forces no restriction on how those structures will be finally implemented on a memory system.

There are several examples in the literature of models of languages that could satisfy this definition [MEA67, CHI68, LAS68]. A set of data structures specified in such a language will be called L-defined.

At the present, we are conducting our experiments based on the notation proposed by Hoare in [DDH73]. This notation allows the specification of general structures built from the basic or unstructured elements (t ypes), defined by enumeration, ordered enumeration, and sub-ranges. Those base elements can be put together to form structures such as Cartesian Products, Discriminated Unions, Arrays, Powersets, etc.

We will call S a storage system where L-defined structures can be physically represented as sets. Again, it is not important to the developments of our concepts to describe any particular method now of representing sets in computers. Suppose for the moment that the scheme proposed in Set-L [SCH70] is the most appropriate. In general, this set theoretical data structure comprises in a notation similar to the one in [CHI68], the following components:

$$STDS = \{\tau, \beta, \theta, \eta, r\}$$

where

τ = collection of set operations

β = set of datum names

θ = collection of datum definitions, one for each datum name

η = collection of set names

r = collection of ser representations, each with a name in η

We will call I another storage system where the data definition facilities of a high level programming language ℓ (ALGOL, PL/I, etc) are to be implemented.

We will call F^{-1} a mapping function that will map the final state of a program recorded in its data structures and stored in I and S(F^{-1}: I→S). In practice this mapping function will be realized through calls to the system's environment where the program expressed in ℓ will be supposed to run.

Def. 8 A module of program module M is a composite program defined by the following 4-Tuple M = {M', F, F^{-1}, E} where:

M' is a semi-module
F and F^{-1} are the functions F:S→I and F^{-1}: I→S, respectively; and E is a systems environment that supports the existence of I, S, F and F^{-1}

Def. 9 A program module is said to have data generality or simply generality

Theorem 4: Composite program modules can always be built with syntactic non-interference.

Proof: The proof of this result is based on the fact that the environment E supports the elements I, S, F, and F^{-1}. Since the memory systems S act as standard interfaces the internal representation of each module is uniquely defined during composition by the pair of functions F and F^{-1} that are internally defined in the module. No changes are required in individual

modules when composition takes place.

Theorem 5: Program modules have semantic Context-independence

Proof: This result is proven simply by reference to the definition of the function F. If in all cases a module writer can build the function F, then the program can be tested independently.

Theorem 6: L-defined program modules allow the specification of free computational methods

Proof: The L definition implies the existence of the memory systems S. As the function of S is to defer the implementation decisions to when F is specified and as the specification of F does not take into consideration the form of representation adapted in S, we can say that a computation method specified for an L-defined modules is free.

There is now a need to discuss what is being gained in return for the inefficiency we propose to add to the programming system.

As was said in the beginning of this section, we want to discuss the sense in which program generality will contribute to the reliability of computer systems.

The justification is the following. Using a definition in DEN73B we will say taht "Reliability is the ability of a software system fo perform its function correctly in spite of failures of computer system components" Dealing with reliability in this sense involves considerations about the detection of failures and the loss of information that accompanies hardware failure. Most of the existing results in fault-tolerant computing [AVI71] are concentrated in the detection aspect. We conjecture that the

form of implementation we suggest for the concept of program generality is such that a software system composed of modules in the sense we have defined allows the definition of powerful methods for the retrieval of information associated with hardware failure. In fact, to restart the execution of a module M_i we only need the memory state of the memory system S produced by the function F_{i-1} associated with the module M_{i-1} besides a register of the fact that M_{i-1} precedes M_i controlwise. We envision a very effective utilization of this idea in a system where both the operating system and the application programs are structured in modules as defined in Definition 8; possibly supported by specially tailored hardware (e.g., associative memories for modules).

Two very powerful characteristics of such a system would be changeability and large-scale composability. In fact, the modification in system's modules would be performed without the existence of side effects. The modular structure would also be a handy feature to be used in connection with a library of easily integratable programs. As soon as a good taxanomy of programs is achieved, we can think in terms of the production of a big system being partly reduced to the composition of existing program modules.

ACKNOWLEDGEMENT

We would like to thank Professor G. Estrin for his criticisms and important contributions to this present work.

REFERENCES

[ELS72] Elspas, B. et al. "An Assessment for Proving Program Correctness", Computing Surveys, Vol. 4 No 2, pp.97-147, June 1972.

[LUC73] Lucena, C. "Modeling, Measurement and Software Reliability", Internal Momerandum # 115, Computer Science Department UCLA, April 1973.

[EMU70] Estrin, G., R. Muntz, and R. Uzgalis, "Modeling, Measurement, and Computer Power", AFIPS Conference Proceedings, Vol. 40, pp. 725, 737, 1970.

[DIJ68] Dijkstra, E. "A Constructive Approach to the Problem of Program Correctness", BIT 8, 1968.

[DIJ69] Dijkstra, E. "Notes on Structured Programming", Technische Hogeschool Eindhoven, 1969.

[DDH72] Dahl, O., E, Djikstra, and C. Hoare. Structured Programming, Academic Press, 1972.

[PAR72a] Parnas, D., "Response to Detected Errors in Well-Structured Programs", Computer Science Department, Carnegie-Mellon University, July 1972.

[PAR72b] Parnas, D., "On the Criteria fo Be Used in Decomposing Systems into Modules", Communications of the ACM, Vol. 15, No. 12, pp. 1053 a 1058, December 1972.

[LIS72] Liskov, B., "A Design Methodology for Reliable Software Systems", AFIPS Conference Proceedings, Vol. 41, pp. 191-199 1972.

[DEN68] Dennis, J., "Programming Generality, Parallelism and Computer Architecture", IFIP Conference Proceedings, pp. C1-C7, 1968.

[DEN73a] Dennis, J., "The Design and Construction of Software Systems", Advanced Course on Software Engineering, Springer-Verlag, 1973.

[DEN73b] Dennis, J., "Modularity", Advanced Course on Software Engineering, Springer-Verlag, 1973.

[HOA69] Hoare, C., "An Axiomatic Basis for Computer Programming", Communications of the ACM, Vol. 12 No. 10, pp. 576-583, October 1969.

[WSH73] Wulf, W. and M. Shaw. "Global Variables Considered Harmful", SIGPLAN Notices, February 1973.

[MEA67] Mealy, G. "Another Look at Data", AFIPS Conference Proceedings, Vol. 31 pp. 525-534, 1967

[CHI68] Childs, D. "Description of a Set-Theoretic Data Structure" AFIPS Conference Proceedings", Vol. 33, pp. 557 - 564, 1968.

[LAS68] Laski, F. "The Morphology of Pre-An Essay in Meta-Algoritms, "Machine Intelligence 3, American Elsevier, 1968

[SCH70] Schwartz, J. "Set Theory as a Language for Program Specification and Programming", New York University, 1970

[AVI71] Avizienis, A. "Fault-Tolerant Computing: An Overview," Computer January/February 1971

[BAL71] Balzer, R. "PORTS - A Method for Dynamic Interprogram Communica tion and Job Control", AFIPS Conference Proceedings, Vol. 33, pp. 485 - 489, 1971

[WES73] Weissman, L. and G. Stacey. "An Interface System for Improving Reliability of Software Systems", Proceedings of the IEEE Symposium on Computer Software Reliability, April 1973.

[SEN73] Senko, Altman, Astrahan, Fehder, "Data Structures and Accessing in Data-Base Systems," IBM Systems Journal", Vol. 12 nº 1, 1973

[LUT73] Lucena C., and G. Tompkins, "An Experience with Methodologies for the Design of Programming Systems", this Proceedings.

[LCS73] Lucena, C. I. Campos, and G. Simon. "On the Power and Limitations of Automatic Methods for Program Testing," to appear.