



Towards a measurement of ADT's design complexity

J.Y. Kuo & D.S. Morris

*Department of Electrical Engineering and
Computer Science, Stevens Institute of
Technology, Hoboken, New Jersey 07030, USA*

ABSTRACT

As software system have become more complex, the need for better abstraction mechanisms has led to the widespread use of *abstract data types* (ADT's). A measure of ADT design complexity is an important factor in describing, explaining, evaluating, and predicting key aspects of the development and maintenance of software systems that use them. This paper discusses a novel method for measuring the design complexity of an ADT through the process of proving its correctness. Then, the relative design complexity of ADT's is expressed in terms of the relative complexity of abstract-to-concrete representation mappings. Comparisons among the results of the developed complexity measure and the results of an experimental study conducted to investigate this issue are given.

INTRODUCTION

Over the past two decades, several program complexity metrics have been proposed [6]. Those that are widely known are software science [2], control flow complexity metrics [7, 1], and data flow complexity metrics [3, 8]. None of these complexity metrics have considered the complexity induced by the choice of different data structures, such as arrays and linked lists, and the effect of using various ADT's, such as stacks and trees, on the program design complexity. Motivated by the need for the software metrics that can model the complexity caused by the choice of different data structures, this paper

⁰This research was sponsored by the U.S. Army Research Office under Contract DAAL03-91-C-0034. The views, opinions, and findings contained in this paper are those of the authors and should not be construed as an official Department of the Army position, policy, or decision, unless so designated by other documentation.

386 Software Quality Management

discusses a novel method for measuring the design complexity of ADT's. This new method is based on a proof of correctness of the implementation of a data abstraction. Intuitively, a data abstraction correctness proof is the process of reasoning about the implementation of an ADT with respect to its specification.

In order to verify that an ADT implementation is correct, we must prove that the representation of this type in terms of data structures of other types is correct with respect to its specification. The relationship between the ADT and its concrete representation in terms of data structures can be expressed by a function called *representation mapping* function. Actually, this function captures the amount of knowledge needed to understand programs using the given data structure. Because of this, we argue that *the effect of data structures on the design complexity of ADT's is primarily due to the complexity of the mapping between the data structures used in the program and their corresponding ADT's*. As we shall see later, the design complexity of an ADT increases with the increase in the complexity of the mapping between the intended ADT and its data structure representation. In other words, the design complexity of an ADT can be measured by estimating the complexity of the representation mapping function.

THE CONCEPT OF DATA ABSTRACTION

Data abstraction is the conceptual approach of combining a data type with a set of operations on that data type. Just as procedural abstraction enables us to focus on *what* a procedure does without worrying about *how* it does it, data abstraction enables us to consider the data objects needed and the operations that must be performed on those objects without being concerned with unnecessary details.

In order to reason about programs using ADT's, it is necessary to understand the characteristics of ADT. The essence of ADT can be captured by the diagram of Figure 1 showing the relationship between the *concrete domain* objects manipulated by a conventional program procedures and the *abstract domain* objects that the program designer chooses to think about instead of the more detailed program objects. A mapping must be defined between the values of concrete objects and the values of the corresponding abstract objects. We call this the *representation mapping* function and denote it Rep_{type} for some data type.

An data type implementation is correct if its data objects are manipulated in such a way that the abstract objects to which they correspond appear to be transformed according to the abstract operations. That is, correct data type implementation uses the concrete data structures and procedures,

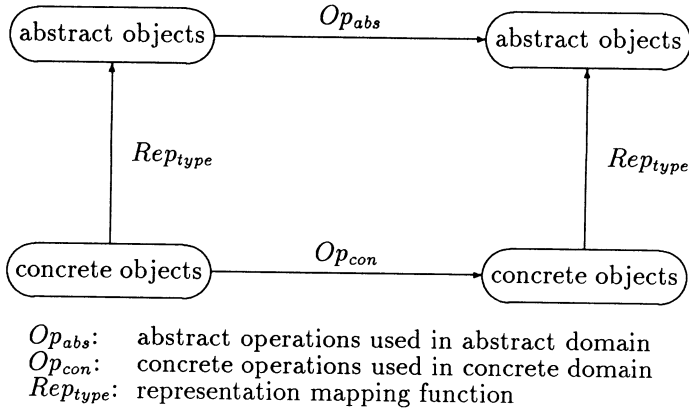


Figure 1: Relation between abstract and concrete domains

but in a way that properly mirrors the abstraction. In order to decide if this implementation is correct, we must show that the diagram in Figure 1 *commutes*. We say that the diagram *commutes* if and only if beginning in the lower left corner and passing in both possible directions gives the same result whenever the abstract data is defined, that is,

$$Rep_{type} \circ Op_{abs} \subseteq Op_{con} \circ Rep_{type} \quad (1)$$

where the symbol “o” stands for functional composition.

DESIGN COMPLEXITY OF DATA REPRESENTATION

From the proof technique stated in Hoare’s notable paper on correctness of data representations [4], we know that in order to prove that the data representation is correct, we must establish a representation mapping function between the abstract domain in which the abstract program is written and the domain of the concrete representation of data structure. In other words, *the representation mapping is the most crucial part in proving a data representation*.

In general, the proof of data representation requires more effort as the representation mapping functions become more complicated; and the representation mapping functions become more complex as the complexity of the relationship between the physical data structures used in the program and their corresponding ADT’s increases. Since the relative proof difficulty implies relative design complexity, we argue that the effect of data structures on the design complexity of ADT’s is primarily due to the complexity of the representation mapping between the data structures used in the program

388 Software Quality Management

and their corresponding ADT's. The design complexity of a data representation increases with the increase in the complexity of the mapping between the intended ADT and its data structure representation. In other words, *the design complexity of a data representation can be measured by estimating the complexity of the representation mapping function.*

Now we wish to estimate the design complexity of data representation directly from the representation mapping functions. For this purpose, we need to make the following assumptions. Since the mapping functions of scalar types such as *integer*, *real*, *boolean* can be defined by $Rep_{scalar}(x) = x$, any representation mapping function is identity on scalar types. In this case, proving of such representation mapping functions is trivial. Therefore, in order to measure the complexity of this representation, we could assume the weight of $\|Rep_{scalar}\|$ to be one unit where $\| \cdot \|$ stands for the relative complexity of some representation mapping function.

Considering the effort needed for proving array of scalar type, record of scalar type, and pointer of scalar type respectively, we have observed that in general the representation mapping of pointer of scalar type is more intellectually difficult than that of array of scalar type because the *alias* problem occurs while constructing the representation mappings of pointer type; and the amount of effort required to reason about the representation mappings of array of scalar type and record of scalar type are roughly the same since they both use the *selector* (or *subscripting*) operators. Therefore, the following relation

$$\|Rep_{scalar}\| \leq \|Rep_{array}\| \simeq \|Rep_{record}\| \leq \|Rep_{pointer}\| \quad (2)$$

holds.

Since the weight of $\|Rep_{scalar}\|$ is equal to one unit, it is possible that $\|Rep_{array}\|$ and $\|Rep_{record}\|$ carry the same weight, which is 2 units. Based on these assumptions, we might pick 3 units as the weight for $\|Rep_{pointer}\|$. These weights are very reasonable because they not only satisfy the relation given in (2) but also match the relative difficulty of explaining (or comprehending) the representation mappings associated with these elementary data types.

In simple cases such as the above elementary data types, it may be relatively easy to come up with a representation mapping function directly, but this will not always be the case. In general, the representation mapping functions become more complicated as the complexity of the relationship between the physical data structures used in the program and their corresponding ADT's increases.

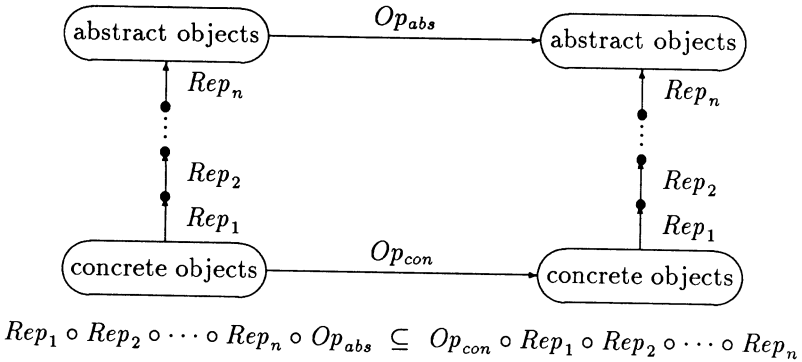


Figure 2: Extended commutative diagram for multilevel mapping

In order to cope with more complicated data types, we need to extend the current measure. The new method is based on the ability to organize the data into a multilevel data abstraction, in which representation mapping functions are used to characterize abstract data of one level in terms of abstract data at a lower level [10], so as to provide an efficient measure of the complexity of data representation. This new method, called *multilevel representation mapping*, formally constructs a representation mapping function in terms of levels of abstraction — each level of which can be represented by a single *scalar* or *elementary* data type.

In other words, a representation mapping function can be entirely refined into a natural hierarchy of elementary data types. Then, the complexity of a representation mapping can be computed by combining the complexity of elementary type at each level of abstraction. For example, in Figure 2 Rep_{type} may be characterized by a certain number of levels of abstraction, that is,

$$Rep_{type} = Rep_1 \circ Rep_2 \circ \dots \circ Rep_n$$

where Rep_i is a representation mapping function of elementary data type and $1 \leq i \leq n$. Hence, the complexity of data representation of Rep_{type} will be the estimates of the composing mapping functions combined. Now, let us consider the following data type:

```

CLASS = array[1..60] of record
    id: Integer;
    score: Integer
end;
```

390 Software Quality Management

its representation mapping function is expressed as

$$Rep_{integer} \circ Rep_{record} \circ Rep_{array}$$

and the design complexity of its data representation can be calculated as

$$\|Rep_{array}\| \times \|Rep_{record}\| \times \|Rep_{integer}\|$$

Finally, the complexity of data representation, $\mathcal{P}_{type}$, for each abstract data type is defined as:

$$\mathcal{P}_{type} = \left\{ \begin{array}{ll} 1 & \text{if } type = \text{scalar type;} \\ 2 \times (\mathcal{P}_{subtype}) & \text{if } type = \text{array } [n..m] \text{ of } subtype; \\ 2 \times (\sum_{i=1}^n \mathcal{P}_{subtype_i}) & \text{if } type = \text{record} \\ & \text{field list}_1: subtype_1; \\ & \vdots \\ & \text{field list}_n: subtype_n \\ & \text{end;} \\ & \text{where } subtype_j \neq subtype_k \\ & \text{for } 1 \leq j, k \leq n \text{ and } j \neq k. \\ 3 \times (\mathcal{P}_{subtype}) & \text{if } type = \uparrow subtype; \end{array} \right.$$

For the sake of simplicity, we would like to relate the model of measuring data representation given above to a special kind of graph model called *data type tree*. A *data type tree* is an ordered, directed tree such that each node of the tree is associated with a data type and each node is presented in the form of $\bigcirc$, ∇ , $\triangle$, or $\diamond$ depending on the data type “stored” at the node. The following table gives a description of these symbols:

	Data type			
	<i>scalar</i>	<i>array</i>	<i>record</i>	<i>pointer</i>
Symbol	$\bigcirc$	∇	$\triangle$	$\diamond$
Weight	1	2	2	3

Suppose that there exists a node $\square$ such that

$$\text{node } \square \in \{ \bigcirc, \nabla, \triangle, \diamond \}$$

Then, the elementary data types can be illustrated by Figure 3.

Let $\mathcal{T}$ be a data type tree with designated root v_0 and nodes labeled $v_1, v_2, \dots, v_n$. Then, the value of node v_i , $\rho(v_i)$, is defined recursively as follows:

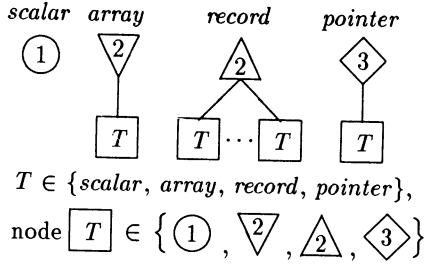


Figure 3: Tree diagrams for elementary data types

1. If node v_i is not a leaf of $\mathcal{T}$ and nodes $u_1, u_2, \dots, u_m$ are the children of node v_i , then the value of node v_i is

$$\rho(v_i) = \alpha(v_i) \times \left(\sum_{j=1}^m \rho(u_j) \right) \quad (3)$$

where $\alpha(v_i)$ is the weight associated with node v_i .

2. If node v_i is a leaf of $\mathcal{T}$, then the value of node v_i is

$$\rho(v_i) = 1 \quad (4)$$

Since node v_i has no children and the leaf must be a scalar type node with one unit weight, the value of node v_i is defined as 1.

Therefore, the complexity of data representation, $\mathcal{P}_{type}$, for each abstract data type can be defined as:

$$\mathcal{P}_{type} = \rho(v_0) \quad (5)$$

where v_0 is the root of $\mathcal{T}$.

Example 1 Consider the following data type.

```

CLASS1 = array[1..60] of
  record
    name: array[1..20] of Char;
    id: Integer;
    test1, test2, test3: Real
  end;

```

392 Software Quality Management

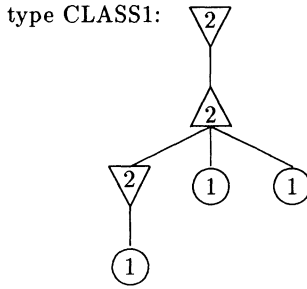


Figure 4: Tree diagram for data type “CLASS1”

Then, from the data tree illustrated in Figure 4, we have

$$\mathcal{P}_{class1} = 2 \times \left[2 \times \underbrace{\left(\underbrace{2 \times 1}_{array} + \underbrace{1}_{integer} + \underbrace{1}_{real} \right)}_{record} \right] = 16$$

DESIGN COMPLEXITY OF OPERATION’S IMPLEMENTATION

Since we usually implement the operations for an abstract data type in the form of procedures or functions, we can use any popular software metric for estimating program complexity to measure complexity estimation for implementation of operation. In this paper, we choose to use the complexity metric described in our earlier study [5] as the measure of the complexity of operation implementation.

Thus, the design complexity of operation implementation, $\mathcal{P}_{op}$, can be computed as

$$\mathcal{P}_{op} = \mathbf{P}_{int} + \mathbf{P}_{ext} \quad (6)$$

where

- $\mathbf{P}_{int}$ is the *internal complexity* which reflects the complexity of the individual program modules, disregarding their interactions with other program modules; and
- $\mathbf{P}_{ext}$ is the *external complexity* which reflects the complexity of the communications and interactions among the program modules.

To form $\mathbf{P}_{int}$ and $\mathbf{P}_{ext}$, we are searching for a combination of primitive software design complexity metrics which are useful, predictive, and objective.

More specifically,

$$\mathbf{P}_{int} = \sum_{k=0}^n \mathcal{P}_k \quad (7)$$

where

$$\begin{aligned} \mathcal{P}_k = & \sum_{j=1}^{|\mathbf{V}_k|} \left(\sum_{\substack{\forall \text{ statement} \\ s_{kji} \text{ in node } v_{k_j}, \\ s_{kji} \in T}} \left(w(s_{kji}) \times [(\delta(s_{kji}) \times \ell(s_{kji})) + 1] \right) \right. \\ & \left. + \sum_{\substack{\forall \text{ statement} \\ s_{kji} \text{ in node } v_{k_j}, \\ s_{kji} \in T}} \left((1 - \delta(s_{kji})) \times \beta(s_{kji}) \right) \right) \end{aligned} \quad (8)$$

and

$$w(s) = \begin{cases} c_{while} = 4 & \text{if } s \in \{\text{WHILE } B \text{ DO } S\} \\ c_{if} = 2 & \text{if } s \in \{\text{IF } B \text{ THEN } S, \\ & \text{IF } B \text{ THEN } S1 \text{ ELSE } S2\} \\ c_{seq} = 1 & \text{otherwise} \end{cases}$$

where the constants c_{while} , c_{if} , and $c_{seq} \in \mathcal{N}^+$,

$$\delta(s) = \begin{cases} 1 & \text{if } s \in \{\text{IF } B \text{ THEN } S, \\ & \text{IF } B \text{ THEN } S1 \text{ ELSE } S2, \\ & \text{WHILE } B \text{ DO } S\} \\ 0 & \text{otherwise} \end{cases}$$

$\ell(s)$ = nesting level of statement s , and

$\beta(s)$ = the number of binary/unary operators in statement s .

Note that c_{while} , c_{if} , and c_{seq} are the associated weights for WHILE-DO statement, IF-THEN-ELSE statement, and SEQUENCE statement, respectively, and are determined by the amount of effort required to explaining the formal proof of these statements.

And $\mathbf{P}_{ext}$ (as proposed by Oviedo [9]) is defined by

$$\mathbf{P}_{ext} = \sum_{k=0}^n \left(\sum_{j=1}^{|\mathbf{V}_k|} \left(\sum_{i=1}^{|U_{k_j}|} \mu(x_{kji}) \right) \right) \quad (9)$$

where $\mu(x_i)$ represents the number of available definitions of variable x_i in the set of variable definitions that reach node v_j , and U_j is the set of variables whose uses are locally exposed in node v_j .

394 Software Quality Management

OVERALL MEASURE OF ADT'S COMPLEXITY

From the discussion carried out in the previous sections, we know that the overall design complexity of ADT's, P_{adt} , is composed of two parts:

P_{type} the design complexity induced by the choice of different representations of the ADT in terms of data structures of other types.

P_{op} the design complexity caused by the implementation of each operation.

Thus,

$$P_{adt} = \sum_{\forall type} P_{type} + \sum_{\forall op} P_{op} \quad (10)$$

EXAMPLE AND EVALUATION

In this section, we apply the developed complexity measure to a set of programs used in Bastani's experimental study. These programs implement a stack *Push* operation presented in three procedures each using a different data structure (i.e. array, linked list, and combination of array and linked list). After analyzing the ADT's complexity of these programs, we will then evaluate our measure of relative complexity of ADT by comparing the results with the experimental results from other Bastani's study.

Example 2 *In this example, the abstract data type STACK is represented by the array data structure.*

```
const Max = 100;
type STACK = record
    top: Integer;
    elements: array[1..Max] of Real
end;
procedure Push (var S: STACK; x: Real);
node # 1    begin
2          if (S.top >= Max) then
3            Error('stack is full')
4          else
            begin
                S.top := S.top + 1;
                S.elements[S.top] := x
            end
5          end;
```

By using the developed metric, the complexity of this data type is:

$$\mathcal{P}_{type} = 6$$

and, the internal design complexity is:

$$\mathbf{P}_{int} = 10$$

Then, the external design complexity of Push is

$$\begin{aligned} \mathbf{P}_{ext} &= \sum_{j=1}^5 du_j \\ &= 3 \end{aligned}$$

Hence, using Equation 6, the complexity of operation Push is

$$\mathcal{P}_{op} = \mathbf{P}_{int} + \mathbf{P}_{ext} = 13$$

Therefore, the design complexity of abstract data type STACK using the array data structure is calculated as:

$$\mathbf{P}_{adt} = \mathcal{P}_{type} + \mathcal{P}_{op} = 19$$

Example 3 In this example, we consider the abstract data type STACK represented by the linked-list data structure.

```

type STACK = ^Stack_Node;
Stack_Node = record
    data: Real;
    next: STACK
end;
procedure Push (var S: STACK; x: Real);
var temp: STACK;
node # 1    begin
2          new(temp);
          temp^.data := x;
          temp^.next := S;
          S := temp
3          end;
```

Similarly, the relative complexity of data representation for this abstract data type STACK is:

$$\mathcal{P}_{type} = 12$$

The internal design complexity is

$$\mathbf{P}_{int} = 6$$

396 Software Quality Management

and, the external complexity of Push is

$$\begin{aligned} P_{ext} &= \sum_{j=1}^3 du_j \\ &= 2 \end{aligned}$$

Hence, the complexity of operation Push is

$$P_{op} = P_{int} + P_{ext} = 8$$

Therefore, the design complexity of abstract data type STACK using the linked-list data structure is calculated as:

$$P_{adt} = P_{type} + P_{op} = 20$$

Example 4 In this example, the abstract data type STACK is represented by the combination of array and linked-list data structures.

```

const Size = 10;
type STACK = ^Node;
      Node = record
                top_index: Integer;
                item: array[1..Size] of Real;
                next: STACK
            end;
procedure Push (var S: STACK; x: Real);
var temp: STACK;
node # 1      begin
2              if (S^.top_index >= Size) then
3                  begin
                    new(temp);
                    temp^.next := S;
                    S := temp;
                    S^.top_index := 0
                end;
4              S^.top_index := S^.top_index + 1;
              S^.item[S^.top_index] := x
5              end;

```

Likewise, the relative complexity of data representation for the abstract data type STACK is

$$P_{type} = 24$$

And,

$$P_{int} = 13$$

$$P_{ext} = 6$$

Hence, the complexity of operation **Push** is

$$\mathcal{P}_{op} = \mathbf{P}_{int} + \mathbf{P}_{ext} = 19$$

Finally, the design complexity of abstract data type **STACK** using the array and linked-list data structures can be computed as:

$$\mathbf{P}_{adt} = \mathcal{P}_{type} + \mathcal{P}_{op} = 43$$

COMPARISON WITH THE EXPERIMENTAL RESULTS

In this section we describe the experiment conducted by Iyengar and Bastani to determine the effect of data structures on the logical complexity of program and then compare our results with theirs. Their hypothesis tested in the experiment was that the representation of advanced data structures in terms of simpler data structures can increase the complexity of a program if the mapping relationship between the two data structures is obscure.

A group of graduate students in computer science from the University of Houston and Louisiana State University were selected as subjects in the experiment. Then, these students were given a set of programs which are identical to the programs used in Example 2 – Example 4.

In order to estimate the relative complexity of the ADT's, Iyengar and Bastani tested for comprehension by requiring the students to find and mark up to two embedded errors in each program. This is a reasonable criterion since it is usually the first step in program maintenance. The students were also asked to note the time (in minutes) taken by them to detect the errors in the program. Following the experiment, the tests were scored to determine how many errors were correctly detected by each student. For the purpose of the study, they were scored 0, 1, or 2, as being the number of correct answers. Table 1 summarizes the results of the experiment.

From the experiment, Iyengar and Bastani conclude that the stacks represented by the combination of array and linked-list data structures is significantly more difficult to comprehend than either the array representation of stacks or the linked-list representation of stacks; and the linked-list representation of stacks is more difficult than the array representation of stacks, but the difference between them does not reach significance.

Recall from the previous example, the relative complexities of abstract data type **STACK** implemented by the array, linked-list, and combination of array and linked-list data structures are **19**, **20**, and **43** respectively. It is easy to observe that the results produced by our measure highly correlate

398 Software Quality Management

Data structure	Test site							
	University of Houston				Louisiana State University			
	Scores		Time		Scores		Time	
	Mean	Std	Mean	Std	Mean	Std	Mean	Std
<i>Array</i>	1.310	0.891	3.828	1.311	1.889	0.333	3.664	1.144
<i>Linked list</i>	1.103	0.939	2.103	0.860	2.000	0.000	5.500	1.936
<i>Array and linked list</i>	1.034	0.865	5.414	2.113	1.333	0.866	16.278	4.711

Table 1: Results of the experiment for the stack Push operation

with the results produced by the experiment. At this point, we feel very confident that the techniques we have developed yield a reasonable measure of relative complexity of ADT's.

CONCLUSIONS

On the basis of our discussion carried out in this paper, we conclude that the relative complexity of an abstract data type can be measured by estimating the complexity of the representation mapping. We have developed the method, based on the multilevel representation mappings, for measuring design complexity of abstract data types. In addition, the data representation can be express in the data type tree model from which the relative complexity of the data type can be determined directly. And, in order to detect these aspects of design complexity, a complexity metric based on a program flow graph scheme with an intellectual difficulty model is developed to express software design complexity in terms of program control flow and data flow. We also evaluated the developed complexity measure by comparing the results of the measure with the results of an experimental study conducted to investigate this issue.

References

- [1] Bill Curtis. The measurement of software quality and complexity. In Alan J. Perlis, Frederick G. Sayward, and Mary Shaw, editors, *Software Metrics: An Analysis and Evaluation*, chapter 12, pages 203–224. The MIT Press, 1981.
- [2] Maurice H. Halstead. *Elements of Software Science*. Elsevier North-Holland, 1977.

- [3] Sallie Henry and Dennis Kafura. Software structure metrics based on information flow. *IEEE Transactions on Software Engineering*, SE-7(5):510–518, September 1981.
- [4] C. A. R. Hoare. Proof of correctness of data representations. In David Gries, editor, *Programming Methodology: A Collection of Articles by Members of IFIP WG2.3*, chapter 19, pages 269–281. Springer-Verlag, 1978.
- [5] Joseph Y. Kuo. *A Software Complexity Measure Based on Program Proof Complexity*. PhD thesis, Stevens Institute of Technology, Hoboken, New Jersey, March 1993.
- [6] H. F. Li and W. K. Cheung. An empirical study of software metrics. *IEEE Transactions on Software Engineering*, SE-13(6):697–708, June 1987.
- [7] Thomas J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, Dec 1976.
- [8] Everald E. Mills. Software metrics. Technical Report SEI-CM-12-1.1, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, Dec 1988.
- [9] Enrique Oviedo. *Control Flow, Data Flow and Program Complexity*. PhD thesis, State University of New York at Buffalo, Buffalo, New York, 1983.
- [10] Lawrence Robinson and Karl N. Levitt. Proof techniques for hierarchically structured programs. In Raymond T. Yeh, editor, *Current Trends in Programming Methodology: Program Validation*, chapter 7, pages 173–196. Prentice-Hall, 1977. Vol. 2.