Predicate Logic for Software Engineering

David Lorge Parnas, *Senior Member, IEEE*

Abstract—**The interpretations of logical expressions found in most introductory textbooks are not suitable for use in software engineering applications because they do not deal with partial functions. More advanced papers and texts deal with partial functions in a variety of complex ways. This paper proposes a very simple change to the classic interpretation of predicate expressions, one that defines their value for all values of all variables, yet is almost identical to the standard definitions. It then illustrates the application of this interpretation in software documentation.**

Index Terms—**Formal methods, predicate logic, partial functions, software documentation, tabular expressions.**

I. Introduction

PROFESSIONAL engineers can often be distinguished from other designers by the engineers' ability to use mathematical methods to describe and analyze their products. Although mathematics is not commonly used by today's programmers, many researchers are developing mathematical methods that are intended for use in software development. We hope that these methods will do for software engineering what differential and integral calculus did for other areas of engineering. The shared basis of all these proposals is mathematical logic. In the future, a solid understanding of logic will be essential for anyone who hopes to be recognized as a software engineer.

In [8], we have shown how the contents of key computer systems documents can be defined in terms of mathematical functions and relations. We also reminded our readers that (1) functions and relations can be viewed as sets of ordered pairs, (2) sets can be characterized by predicates and described by logical expressions, (3) predicates can be represented in a more readable way using multidimensional (tabular) expressions whose components are logical expressions and terms, and (4) the meaning of these tables can be defined by rules for translating those tables into more conventional expressions. A complete discussion of these tabular expressions can be found in [6]. The most recent illustration of their use can be found in [9].

In our approach to software development, it is essential to have a precise meaning for logical expressions, one that unambiguously yields a value of ***true*** or ***false*** for every assignment of values to the variables that appear in an expression. Our documents represent predicates on the observable behavior of programs. If we want to know whether an observed behavior satisfies a specification, we want a definite "yes" or "no," not "maybe."

Because our goal is to make a change in industrial practice, we have to pay attention to the size (and perceived complexity) of the expressions. We have had extensive experience in the use of mathematical methods working with industrial practitioners (e.g. [7]). It is clear that practitioners do not want to use methods that require them to use many symbols to say simple things. They will not read expressions that are lengthy or deeply nested. If we tell them that the increased complexity is necessary, "in order to be formal," most will reject the concept of formality. Rather than follow burdensome rules, they will take short-cuts, inventing ad hoc abbreviations at will, if a formalism requires that they write down a lot of conditions, variable definitions, etc. that do not seem to carry much information. These "on-the-fly" inventions are often ambiguous and cannot be the basis of sophisticated support tools. A full, formal definition of a logic that permits concise expressions is a prerequisite for practical use.

The most conventional formal interpretations of logical expressions (e.g. [5]) assume that all functions are *total*, i.e. defined on a domain that includes all possible values of their arguments. Those interpretations are not intended to deal with *partial* functions, functions whose value has not been defined for certain values of the arguments. Under conventional interpretations, a logical expression that includes partial functions will have a defined value only if the values assigned to the arguments of each function are within that function's domain; in other words, the predicate described by the expression is partial. Such interpretations are of limited usefulness when describing software because we frequently use partial functions to describe the behavior of programs. This paper proposes an interpretation of predicate expressions that is as close as possible to the standard interpretations but makes all predicates total.

Fig. 1 contains a simple example that illustrates the problem that motivates this work. More interesting, software-related examples will be found later in this paper.

II. The Structure of This Paper

Section III discusses the goals of this paper and compares them with the goals of other papers on the subject. Section IV reviews the definitions of some basic concepts. Section V defines the class of expressions that we call predicate expressions. Section VI then gives precise meaning to these expressions by associating each with a set of n-tuples. The

Manuscript received October 1992; revised June 1993. Recommended by N. G. Leveson. This work was supported by the Government of Ontario through TRIO, and by the Government of Canada through NSERC's Research Grant program.

The author is with the Telecommunications Research Institute of Ontario (TRIO), Communications Research Laboratory, Department of Electrical and Computer Engineering, McMaster University, Hamilton, Ontario, Canada L8S 4K1.

IEEE Log Number 9211725.

Assume that $\sqrt{\ }$ represents a function that is defined on a domain containing only non-negative real numbers. A software designer may write boolean expressions such as:

$$((x>0) \wedge (y=\sqrt{x})) \vee ((x\leq 0) \wedge (y=\sqrt{-x})) \qquad (1)$$

to describe the behaviour of a computer program. The writer's intent seems to be to write an expression that is equivalent to:

$$y=\sqrt{|x|} \qquad (2)$$

The usual rules for evaluating such expressions require evaluation of all of the functions and relational operators before application of the logical operators to the truth-values that result. For every value of x other than 0, some component of expression (1) is undefined. With the standard interpretation of logical operators, which are defined only for two-value logics, the value of (1) is not defined except when x is assigned the value 0.

Those who think in terms of programming languages might consider replacing (1) with:

$$((x>0) \rightarrow (y=\sqrt{x})) \wedge ((x\leq 0) \rightarrow (y=\sqrt{-x})) \qquad (3)$$

because implications look like conditional statements. However, (3) is exactly equivalent to (1).

Fig. 1. Motivating Example.

formal definition uses basic set theory and assures that the logical connectives have the expected behavior. The first paragraphs of Section VI explain the implications of the formal definition, and some readers may prefer to skip the formal definition on first reading. Section VII illustrates the way that we use this logic and explains its advantages in these applications. In Section VIII we offer some observations about reasoning, e.g. formula simplification, formula comparison, and formal proof, with this interpretation.

III. Comparison With Other Work

The problem illustrated in Fig. 1 is well known. There is a huge and complex literature on the subject of logic with partial functions. A discussion of recent work on this subject from a computer science point of view is given in [1]. Another, more philosophical, survey and analysis can be found in [3].[1] In fact, long before computer scientists became interested in this problem, it was discussed by most of the great philosophers and mathematicians who wrote about logic. It is not the purpose of this paper to duplicate the discussions of alternatives that are found in the literature. Our goals are more pedestrian. This paper presents a specific proposal and illustrates its use. The proposal presented is close to Farmer's proposal (in [3]) but the presentation can be simpler because of our intended application and audience. The proposal made in this paper is one of a class of proposals called "disconcerting," and not given serious consideration in [1]. Although the proposal will be controversial, we have found it to be useful.

This paper's treatment of the subject can be different from that in the papers cited because it is intended for a specific application. By limiting our attention to software engineering, we are able to limit our domains to finite sets of elements, n-tuples of elements drawn from a finite and fixed universe. We are able to evade many of the deeper philosophical issues that complicate other papers and to provide a formal definition that is very close to the classical definitions for logics that are limited to total functions.

[1] Insight into the complexity and extent of the literature on this subject can be obtained by comparing the references cited in these two surveys. They have few authors, and fewer papers, in common.

As our primary focus is on the use of mathematics for precise documentation of software and not on program verification, we present a formal semantics but do not present axioms or rules of inference for the logic. This allows a precise definition that is accessible to anyone with a knowledge of "naive" set theory [4].

Because of these simplifications, this paper will not satisfy everyone's needs and is not intended to do so. The logic proposed is not optimal for all applications. For example, when discussing the evaluation of the Boolean expressions that appear in programs, one may want to turn to three-valued logics. Those interested in the more general discussions will have to turn to the surveys included in the references.

Some proposals extend the conventional propositional and predicate calculus with new symbols, or change the meaning of the conventional ones. For example, it is common to define conditional versions of "$\wedge$" and "$\vee$" (e.g. the well-known "**cand**" [2]) to allow the use of such partial functions. These conditional operators are defined by describing an evaluation procedure, one that depends on the values of sub-expressions. Dijkstra's **cand**, and similar operators, are asymmetric. The value of the left operand determines whether or not the right operand will be evaluated. With such rules, sub-expressions cannot be properly understood outside of the context in which they appear. This is unfortunate because it means that lengthy expressions must be understood as a whole rather than piece by piece. In other approaches to the problem conventional two-valued logic is replaced by a three-valued logic in which the third value is understood as "undefined." Three-valued logics were deemed to be unsuitable for our applications, for reasons discussed earlier.

The interpretation for predicate expressions proposed here neither specifies the order of evaluation nor introduces new symbols into the logic. All logical connectives retain their familiar meanings. Instead of changing the meaning of the connectives, we restrict the set of primitive predicates. A side effect of our restriction is that some common relational operators cannot be primitive in our logic. However, relational operators with the expected properties can be constructed from the primitive operators. We return to this issue in Section VIII.

Another approach to the problem of partial functions has been the introduction, as part of the basic definitions, of bounded quantification, i.e. quantification on limited domains. We have experimented with this alternative and concluded that it complicates the expressions more than the approach presented in this paper. Bounded quantification can be introduced as an abbreviation in the logic defined in this paper if desired, but when introduced as an abbreviation, bounded quantification does not make a substantive change in the properties of the logic. We discuss this alternative further in Section VII.

Some researchers assume that what some people call the "axiom of reflection" (namely, that for all functions f, the value of $(f(x) = f(x))$ must always be *true*) is essential. This intuitively appealing rule seems fine but leads to further deliberations about whether $f(x) = f(y)$ should be *true* when x and y are distinct but both outside of the domain of f. In the approach presented in this paper, we have

consciously abandoned reflection as a universal property. In this interpretation, for any primitive relation, "=", $f(x) = f(x)$ would be ***false*** if x is not in the domain of f. It is possible to define a nonprimitive equivalence relation that satisfies the stated "axiom," but we do not believe that doing so is useful. We return to this issue in Section VIII.

IV. Basic Definitions

This section explains some well-known mathematical concepts used in the sequel, namely tuple, simple tuple, relation, function, predicate, characteristic predicate, domain, argument, and range.

(A) Tuples

We assume the existence of a finite[2] set of values called U. U must include the truth-values (represented by "***true***" and "***false***"), a distinguished value (represented by "$*$"), and all other values of interest.

A *simple tuple* is an ordered list of one or more members of U. A *simple n-tuple* is an ordered list of n members of U. We make no distinction between a simple 1-tuple and a member of U.

A *tuple* is an ordered list of one or more simple tuples. An *n-tuple* is a tuple containing n elements, each of which is a simple tuple. A 1-tuple whose element is a simple n-tuple is the same as that simple n-tuple.

When representing specific tuples, we separate the elements with commas and enclose tuples in <brackets> to make their structure clear. For example, "< *true*, *false*, *true*, *false* >" represents a simple 4-tuple, and "<< *true*, *false* >, *true*, *false* >" represents a 3-tuple that is not a simple 3-tuple. The strings "*true*," "< *true* >," and "<< *true* >>" all represent the same simple 1-tuple and member of U.

S^k is the set of all simple k-tuples; S^1 is U. S is the union of $S^1, S^2, \cdots, S^u$; u is the length of the longest tuple needed to apply the semantic model developed below.[3]

T^k is the set of all k-tuples. T^1 is S. T^k includes S^k. T is the union of $T^1, T^2, \cdots, T^u$.

(B) Relations, functions, predicates, and characteristic predicates

We define a *relation* to be a set of tuples, a subset of T. If the set consists entirely of pairs (2-tuples), we call the relation a *binary relation*. The set of values that appears as the first element of a pair in a binary relation is called the *domain* of that relation. The set of values that appears as the second element of a pair is called the *range* of that relation.

A *function* is a binary relation with one additional property: for any given simple tuple, x, in its domain, there is only one pair (x, y) in the function. If $< a, b >$ is in the function F, b is called the *value* of the function at a; we may write "$F(a)$" to represent b. A procedure determining the value of F at a is called an *evaluation* of F at a. If there is no pair $< a, b >$ in F, $F(a)$ is not defined (i.e. F cannot be evaluated) at a. Note that the domain and range of a function can include simple tuples, and it may make sense to write "$F(a, b)$," "$F(a, b, c)$," and "$F(F(a, b, c))$."[4]

We refer to functions whose domain is a proper subset of S as *partial functions* and functions whose domain includes all members of S as *total functions*. All of the functions that arise in software engineering are partial functions in the sense of this paper.[5]

A *predicate* is a function whose range contains no members other than ***true*** and ***false***.

For any set of simple tuples, X, the *characteristic predicate* of that set is a predicate whose domain is S, and whose value, for a simple tuple b, is ***true*** if and only if b is a member of X.

V. The Syntax of Logical Expressions

This section describes the class of expressions that is the subject of this paper. Readers will find our syntax very close to the standard ones. The semantics of these expressions is described in Section VI.

(A) Primitive functions and predicates

We assume that the strings $f_1, \cdots, f_k$ are the names of functions and that $R_1, \cdots, R_m$ are the names of characteristic predicates of sets of simple tuples. Viewing the functions as sets of pairs and the predicates as characterizing sets of simple tuples, we require that the distinguished member of U, $*$, not appear in any of the tuples in those sets.

(B) Terms

Expressions are constructed using a finite indexed set of mathematical variables, $x_1, \cdots, x_u$, and a finite set of constants, C. The constants are strings representing the members of U but no member of C represents $*$. For example, "***true***" is a constant that represents ***true***; "***false***" in an expression represents ***false***. The symbol "V" will be used to stand for a comma separated list of terms (see below). If all the elements of V are constants, V represents a simple tuple.

A *function application* is a string of the form $f_j(V)$. Nothing else is a function application. The elements of V are called the *arguments* of the function application.

A term is either a member of C, a variable, or a function application. Nothing else is a term.

(C) Primitive expressions

A *primitive expression* is a string of the form $R_j(V)$. Nothing else is a primitive expression. The elements of V are called the *arguments* of the primitive expression.

[2] We restrict ourselves to finite sets, not because it is strictly necessary to be so restrictive, but because they are all that we need for our application.

[3] u will be the size of the set of variables.

[4] We allow functions with varying "arity," both because it simplifies our definitions and because programmers often ask for such facilities.

[5] Because we allow functions of varying arity, our definition of "partial" must be weaker than the usual definitions. Usually, a function whose domain was S^k (for fixed k) would be considered total.

(D) Predicate expressions

All primitive expressions are *predicate expressions*. If P and Q are predicate expressions and x_k is a variable, then $(\forall x_k, P), (P), (P) \wedge (Q), (P) \vee (Q)$, and $\neg(P)$ are *predicate expressions*. There are no other predicate expressions, but extensions of this work could introduce additional propositional connectives and quantifiers (see Section VI-F below).

VI. The Meaning of Logical Expressions

This section defines the meaning of the expressions described in the previous section in such a way that if we evaluate a primitive expression, $R_j(V)$, and some elements of V include applications of partial functions, and the values of the arguments of any function are not in the domain of that function, the value of the primitive expression will be ***false***.

A full formal definition is important because experience has shown that the simple description in the previous paragraph is unclear for some. Some specialists have expressed doubt that a logic with this property can preserve the standard meaning of the logical connectives. Further, some implications of the informal description are not immediately obvious. However, some readers may wish to skip the definitions on first reading. The examples provided in later sections can be understood without using the detailed definition.

To define the meaning of these expressions, we will interpret each predicate expression as denoting a set, which we call its *denotation*. These denotations will be subsets of S^u, where u is the number of variables that may appear in the expressions whose meaning is being defined. Each simple u-tuple will be called an *assignment*.

(A) Evaluating terms for a given assignment

We define a mapping val, which associates pairs, comprising a term and an assignment, with members of S by the following rules. For a term, t, and assignment, A:

1) if t is a constant representing t' (a member of U), $val(t, A)$ is t',
2) if t is a variable, x_k, $val(t, A)$ is the kth element of the assignment A,
3) if t is a function application, $f_k(V)$, let
   a. n be the length of V,
   b. V_i be the ith element of V, and
   c. V' be the simple tuple $<val(V_1, A), \cdots, val(V_i, A), \cdots val(V_n, A)>$,

   and distinguish the following two cases:
   a. if V' is in the domain of f_k, $val(t, A)$ is $f_k(V')$,
   b. if V' is not in the domain of f_k, $val(t, A)$ is $*$.

(B) Evaluating primitive expressions for a given assignment

We define a mapping $tval$, which associates pairs, comprising a primitive expression and an assignment, with either ***true*** or ***false*** by the following rule.

For a primitive expression, $R_j(V)$, and assignment, A, let

- X_j be the set of simple tuples characterized by R_j,
- n be the length of V,
- V_i be the ith element of V, and
- V' be the simple tuple $<val(V_1, A), \cdots, val(V_i, A), \cdots val(V_n, A)>$,

and distinguish the following two cases:

1) If V' is in Xj, $tval(R_j(V), A)$ is ***true***.
2) If V' is not in $X_j, tval(R_j(V), A)$ is ***false***.

(C) The denotation of primitive expressions

For a primitive expression, p, the denotation of p is the set of all assignments, A, such that $tval(p, A)$ is ***true***.

(D) The denotation of predicate expressions

If A is an assignment, $A[k \rightarrow c]$ stands for an assignment, A', that is identical to A except that the kth element of A' is the member of U that is represented by c. If P and Q are predicate expressions,

1) the denotation of $(\forall x_k, P)$ is the set of all assignments, A, such that if c represents any value in U other than $*$, $A[k \rightarrow c]$ is in the denotation of P, and
2) the denotation of (P) is the denotation of P, and
3) the denotation of $(P) \wedge (Q)$ is the intersection of the denotations P and Q, and
4) the denotation of $(P) \vee (Q)$ is the union of the denotations of P and Q, and
5) the denotation of $\neg\ (P)$ is the set of all members of S^u that are not in the denotation of P.

(E) Satisfaction of an expression

The denotation of any predicate expression is defined above.

Expressions that denote the empty set are said to be $false$; those that denote all of S^u are said to be $true$. An expression, e, is said to be $satisfied$ by an assignment, A, if A is a member of the denotation of e.

(F) Notational conveniences

Existential quantification ("$\exists$") and implication ("$\Rightarrow$") can be introduced as abbreviations. "$(\exists x_k, P)$" can be written instead of "$\neg(\forall x_k, \neg(P))$". "$(P \Rightarrow Q)$" can be written instead of "$(\neg(P)) \vee (Q)$". It is usual to introduce operator precedence and eliminate many of the parentheses.

As most expressions include only a few variables, it is useful to describe sets of assignments by listing of the values of some variables and not specifying values for the others. For example, a list such as "$x_2 : 4, x_{24} : 96$" represents all assignments in which the second element is 4 and the 24th element is 96.

It is also convenient to introduce other variables (e.g.: cat, y), and conventional symbols representing the functions and relations. None of these conveniences would mean a substantive change in the interpretation of these expressions.

VII. Examples of the Use of This Logic in Software Documentation

This section illustrates the use of our logic by discussing some simple examples.

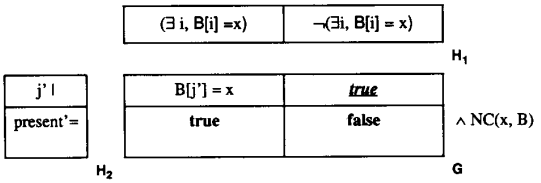

Fig. 2. Relational Description of a program that searches B for the value of x.

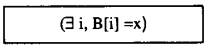

Fig. 3. Is the value of x to be found in B?

For increased readability, we have developed tabular representations of functions and use the logic described above within the tables. The meaning of the tables is the subject of [6]; here we discuss only the logical expressions that appear in the tables. These expressions are used to partition the domain of a relation (each partition corresponding to a column) and to describe the conditions that values must satisfy.

Both of the examples given below describe programs that deal with an array, B, with indices $1 \cdots N$. Like many others, we treat such arrays as partial functions whose domain consists of the integers $1 \cdots N$. The value of the array (partial function) is not defined for other values.

Fig. 2 documents the behavior of a program that must search the array B, looking for an element with value of the program variable x.[6] To describe the behavior of this program completely, we must distinguish two cases depending on whether or not there is such an element. The table describes the required properties of the values of j' and present' in each case. We further indicate that the variables x and B should not change (by writing "$NC(x, B)$").

The key predicate expression in Fig. 2 is that in Fig. 3.

A logic not designed for partial functions would leave the expression in Fig. 3 undefined because there are values of i for which $B[i]$ is not defined. Other logics, e.g. some of those that introduce a third value, would assign that third value to this expression whether or not the value of x could be found in B. Neither of these interpretations would be consistent with the intended meaning of this table. We want one, and only one, of the two expressions in the column headers to evaluate to ***true***. Other alternatives introduce bounded quantification, i.e. quantification over an explicitly described set, allowing expressions like "$(\exists i : 0 < i \leq n, B[i] = x)$" and "$(\forall i : 0 < i \leq n, B[i] = x)$."[7] The use of bounded quantification as a primitive concept could solve this problem, but the expressions would always be longer, more complex than Fig. 3. The complexity can become especially troublesome if the arrays in an expression do not have the same index set. Consider the expression in Fig. 4.

[6] In these tables, ***true*** and ***false*** are predicate values, while **true** and **false** represent the values of program variables. "|" is read "such that" and indicates that the value of the variable must satisfy a predicate given in the appropriate column.

[7] It is important *not* to define these as abbreviations for $(\exists i. 0 < i \leq n \wedge B[i] = x)$" and "$(\forall i. 0 < i \leq n \Rightarrow B[i] = x)$," respectively. Bounded quantification must be primitive.

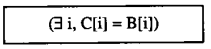

Fig. 4. Looking for matching elements in two arrays.

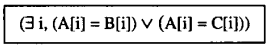

Fig. 5. Looking for common matching in three arrays.

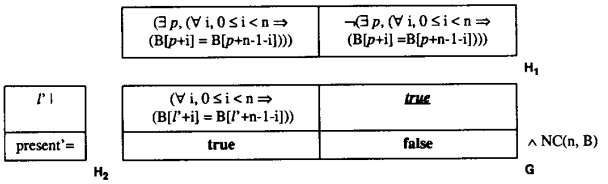

Fig. 6. Relational description of a program checking for palindromes.

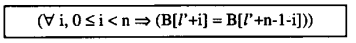

Fig. 7. Is there a palindrome of length n beginning at l'?

For this example, if we were depending on bounded quantification, the quantification would have to take place over the intersection of the index sets of B and C. Now, consider Fig. 5; if we were using bounded quantification, and the index sets of the three arrays A, B, and C were distinct but overlapping, the expression in Fig. 5 would have to be rewritten as the disjunction of two separate quantified expressions.

The logic proposed in this paper gives exactly the answers that would be wanted in such cases. When the value of i is outside the index set of either B or C, the value of $C[i] = B[i]$ is ***false***.

The slightly more complex example in Fig. 6 is introduced to show that we get the desired results when universal quantification is used. Fig. 6 would document a program that examines an array, B, looking for a palindrome of length n.

If there is such a palindrome, its presence and location are indicated by the values of present' and l'. If a palindrome is present, the value of l' must satisfy the expression in Fig. 7.

This expression gives the desired results even though the implication is evaluated outside the domain (index set) of B; that domain is characterized by the left-hand side of the implication. When the expression is evaluated outside of the index set, the left-hand side of the implication is ***false*** and the implication is ***true***. With universal quantification our interpretation requires an explicit statement of the domain of interest, but we do not need to introduce bounded quantification as a primitive concept.

VIII. Concluding Observations

The meaning of expressions like the one presented in Fig. 1 can be defined in terms of well-understood set-theoretic operations. As a result, the logical connectives have properties analogous to the corresponding set theoretic operators, and the proposed definition is consistent with the intuitive meaning of these operators. It is not necessary to introduce either a third

$$(y = \sqrt{x}) \vee (y = \sqrt{-x}) \quad (4)$$

Fig. 8. Simplified version of (1).

$$(\forall x_1, (\forall x_2, \neg(x_1 = x_2) \vee (f_1(x_1) = f_1(x_2)))) \quad (5)$$

Fig. 9. "axiom" of reflection, which does not hold in this interpretation.

value or conditional operators in order to deal with partial functions.

Some researchers have proposed avoiding the problem of partial functions by avoiding the concept of function completely. It is possible to work entirely with relations and not use the "$f(x)$" notation. If F is the characteristic predicate of the function f, one can replace each use of "$y = f(x)$" with "$F(x, y)$". However, engineers have found the use of functional notation to be very valuable, and we are reluctant to discard it. One nice property of our proposal is that it gives exactly the same results that one would get if one avoided functions by using the corresponding relations.

Not only is our introductory example (1) fully defined using this interpretation, so is the simplified form in Fig. 8.

This form, in which there are no "guarding" expressions, has exactly the same denotation as (1) and (2). The interpretation of logical formulae presented here allows us to simplify many expressions substantially. Obtaining the most compact readable formulation possible is essential if these notations are to be used to describe real programs.

Extensive discussions of axioms and rules of inference for logics similar to the one described here can be found in [1], [3] and the papers that they reference. Many of the usual axioms apply only to functions that are total. For example, we often assume that the expression in Fig. 9 evaluates to ***true*** for any function, f_1. However, (5) is equivalent to ***true*** if, and only if, the domain of f_1 includes all values in $U - \{*\}$[8]. If x_1 is outside of the domain of $f_1, f_1(x_1) = f_1(x_1)$ would have the value ***false***.

Some expressions that are normally assumed to represent complementary predicates would not do so in our interpretation if the relations are included in the set of primitive relations. For example, if both "$>$" and "$\leq$" are primitive, "$\sqrt{x} > \sqrt{y}$" would not denote the complement of the denotation of "$\sqrt{x} \leq \sqrt{y}$"; both evaluate to ***false*** when either x or y are assigned negative values. We can define two $nonprimitive$ ordering relations that are complementary by defining **one** of them to be true if both of the primitive relations are false. This would be an arbitrary choice and probably not useful.

It should be noted that our definitions do not treat equality different from any of the other relations used in the expressions. Equality would be included in $\{R_1, \cdots, R_m\}$, and should be defined to be the smallest symmetric, transitive, reflexive, binary relation on its domain; the domain should be $U - \{*\}$. If this definition is used, the expression "$\sqrt{a} = \sqrt{a}$" cannot be replaced by "***true***" if U includes negative values. If U is the set of real numbers, "$\sqrt{a} = \sqrt{a}$" can be replaced by "$a > 0$," which characterizes the domain of the function applied in the expression.[9] Because this is contrary to our habitual assumptions and could lead to careless errors, the properties of the functions that we use must be stated precisely. Conventional simplification rules, and hence some automatic simplifiers and verifiers, must be either modified or used with caution; they are often based on the implicit assumption that functions are total.

The interpretation proposed here can be simpler than some proposed elsewhere because some of the complexities of dealing with partial functions have been kept out of the general interpretation; the complexity will reappear in the axiomatic definitions of the functions actually used. Simplification has also been obtained by insisting that all primitive predicates evaluate to ***false*** whenever one or more of their arguments is not defined. We believe that these are the proper decisions because (1) keeping the logic simple is essential to practical application, (2) the assigned meanings are consistent with intuitive interpretations, and (3) the formulae that result are relatively simple for cases arising frequently in our use of the logic.

[8] "-" denotes set difference.

[9] The primitive predicates can be used to construct other predicates if desired. For example, it is possible to define $E(a, b)$ to be $(a = b) \vee (\neg((a = b) \vee (a \neq b)))$.

ACKNOWLEDGMENT

I am grateful to Professors M. Iglewski, J. Madey, A. Kreczmar, W. Lukaszewicz, J. Zucker, P. Gilmore, M. van Emden, J. Ludewig, and Dr. J. McLean for helpful comments on earlier drafts of this paper. Careful reading and provocative remarks by Ramesh Bharadwaj, Philip Kelly, Yabo Wang, and Delbert Yeh were also very helpful. Several of the referees helped me to explain why I chose to add one more paper on this subject to the already immense literature.

REFERENCES

[1] J. H. Cheng and C. B. Jones, "On the usability of logics which handle partial functions," in *Proc. Third Refinement Workshop*, C. Morgan and J. Woodcock, Eds. Heidelberg Germany: Springer-Verlag, 1991.
[2] E. W. Dijkstra, *A Discipline of Programming*. Englewood Cliffs N.J.: Prentice-Hall, 1976
[3] William F. Farmer, "A partial functions version of Church's simple theory of types," *J. Symbolic Logic*, pp. 1269–1291, Sept. 1990.
[4] P. R. Halmos, *Naive Set Theory*. New York: Van Nostrand Rheinhold, 1960.
[5] E. Mendelson, *Introduction to Mathematical Logic*, Third Ed. Pacific Grove, CA: Wadsworth and Brooks, 1987.
[6] D. L. Parnas, "Tabular representation of relations," CRL Report 260, McMaster University, TRIO (Telecommunications Research Institute of Ontario), Oct. 1992.
[7] D. L. Parnas, G. J. K. Asmis, and J. Madey, "Assessment of safety-critical software in nuclear power plants," *Nuclear Safety*, vol. 32, no. 2, pp. 189–198, Apr.-June 1991.
[8] D. L. Parnas and J. Madey, "Functional documentation for computer systems engineering (version 2)," CRL Report 237, McMaster University, Hamilton Canada, TRIO (Telecommunications Research Institute of Ontario), Sept. 1991.
[9] D. L. Parnas, J. Madey, and M. Iglewski, "Formal documentation of well-structured programs," CRL Report 259, McMaster University, TRIO (Telecommunications Research Institute of Ontario), Sept. 1992.

David Lorge Parnas (SM'92) was born February 10, 1941. He received the Ph.D. in electrical engineering from Carnegie-Mellon University, and an honorary doctorate from the ETH in Zurich, Switzerland.

He is a Professor in the Department of Electrical and Computer Engineering at McMaster University in Hamilton, Ontario, Canada. He is a member of the Communications Research Laboratory and Principal Investigator for the Telecommunications Research Institute of Ontario. He has been a Professor at the University of Victoria, British Columbia, the Technische Hochschule Darmstadt, and the University of North Carolina at Chapel Hill, North Carolina. He has also been a member of the faculty at Carnegie-Mellon University and the University of Maryland, as well as holding nonacademic positions with Philips Computer, the United States Naval Research Laboratory in Washington, D.C., and the IBM Federal Systems Division. At NRL he instigated and led the Software Cost Reduction Project, where he developed and applied software technology for avionics systems for several years. He has advised the Atomic Energy Control Board of Canada on the use of safety-critical real-time software at the Darlington Nuclear Generation Station. He has authored more than 150 papers and reports, and his fields of interest include precise abstract specifications, real-time systems, safety-critical software, program semantics, language design, software structure, and synchronization. He seeks to find a "middle road" between theory and practice, emphasizing theory that can be applied to improve the quality of products.

Dr. Parnas won the ACM "Best Paper" award in 1979 and a "Most Influential Paper" award from the International Conference on Software Engineering in 1991. Deeply concerned that computer technology be applied to the benefit of society, Dr. Parnas was the first winner of the "Norbert Wiener Award for Professional and Social Responsibility." He was recently elected a Fellow of the Royal Society of Canada.