Some Theorems We Should Prove

David Lorge Parnas

Telecommunications Research Institute of Ontario (TRIO)
Communications Research Laboratory
Department of Electrical and Computer Engineering
McMaster University, Hamilton, Ontario, Canada L8S 4K1

ABSTRACT

Mathematical techniques can be used to produce precise, provably complete documentation for computer systems. However, such documents are highly detailed and oversights and other errors are quite common. To detect the "early" errors in a document, one must attempt to prove certain simple theorems. This paper gives some examples of such theorems.

1 Introduction

In [4], 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 characterised 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 [3].

Our efforts have very pragmatic goals. We are not trying to provide mathematical proofs of program correctness; our goals are much more mundane. We wish to use mathematical methods to improve the quality of documentation in software systems. We believe, and have demonstrated using both practical and "academic: examples. ([1, 7, 3]) that we can provide mathematically precise documents that can be read by both programmers and properly prepared users.

Although we are not working on program verification *per se*, we believe that the ability to provide readable mathematical documentation is a prerequisite for regular practical use of mathematical methods in software development. It does no good to prove that a piece of software satisfies a specification, if that specification cannot be read, understood, and criticised by potential users or their representatives.

Although we are not trying to prove programs correct, we do have a need for theorem provers. The formulae in our tabular expressions must satisfy certain mathematical conditions. When we have used these tables in practice (e.g. [7]), we have found that the documents submitted for review often fail to satisfy those conditions; as a result the reviewers spent much too much of their time and energy checking for simple, application-independent, properties. This distracted us from the more difficult, safety relevant, issues and we felt that the preliminary checking should be done by a computer. Tools that check these tables must prove theorems, but theorems that are different from those that arise in program verification. The purpose of this paper is to formulate, but not prove, examples of those theorems. We would like to know which theorem provers or theorem proving support systems, are best able to deal with this type of theorem.

2 An introductory example

The example below describes a function in terms of a single real variable, x, applying a previously defined function, denoted by "$\sqrt{\ }$", which represents a function that is defined on a domain containing only non-negative real-numbers. The value of the function is a pair with two elements named y and z. The intent is to describe a function whose domain includes all real values. Each column of the table describes the value of the function in the subset of the function's domain that is characterised by the predicate expression in the column header. Each row in the table corresponds to an element of the pair; the rows are identified by the labels "y" and "z".

$x < 0$	$x = 0$	$x > 0$

H_1

y
z

H_2

$x + 2$	$x + 4.21$	$5.4 + \sqrt{x}$
$5 + \sqrt{-x}$	$x-4$	x

G

Figure 1: Arbitrary function of a single real represented by "x"

The first "theorem" that must be proven is given in Figure 2. It confirms our intent that the domain of the function described includes all real numbers.

$$(\forall\ x, (x < 0 \lor x = 0 \lor x > 0))$$

Figure 2: Domain Coverage Theorem for Figure 1

In addition, we wish to make sure that each column deals with a disjoint subset of the domain. This can be expressed by the theorem in Figure 3,

$$(\forall\ x, \neg((x < 0 \land x = 0) \lor (x < 0 \land x > 0) \lor (x > 0 \land x = 0)))$$

Figure 3: Disjoint Domains Theorem for Figure 1

Finally, since the function applied in this table is a partial function, we want to prove that there is a defined value for the function in each column. We introduce notation for referring to the domain of a partial function f, **domain**(f), (a predicate that characterises the domain of f) so that we can state the following two theorems...

$$x < 0 \Rightarrow \mathbf{domain}(\sqrt{-x})$$

Figure 4: Definedness theorem for Column 1 of Figure 1

$$x > 0 \Rightarrow \mathbf{domain}(\sqrt{x})$$

Figure 5: Definedness theorem for Column 3 of Figure 1

Each of these theorems is "obviously true", but they must be checked routinely when preparing these tabular descriptions of functions. "Proving" them requires knowing the definitions of each of the relations and functions that appear as well as knowing the characterisation of the domain of any partial functions. The functions used in these examples are familiar functions, but, in practice, designers define unfamiliar functions for their applications. Thus, it must be possible to add new functions and relations to the "vocabulary" of the prover. The users of these tables cannot be assumed to be mathematically sophisticated, or even rigorous. Thus, we would like the theorems to be formulated and verified automatically wherever possible.

3 More Advanced Examples

The example in Section 2 illustrates the meaning of our tabular expressions and the way that "theorems" are derived from such tables. In more advanced examples we want to prove the same general theorems, but the expressions become more complex. The primary source of new problems is the use quantification over finite sets.

3.1 Array Search Example

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

Figure 6 specifies the behaviour of a program that must search the array B, looking for an element whose value is the same as the value of the program variable x[1]. To describe the behaviour 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 *final* values of j and present (denoted by "j' " and "present' ") in both cases. In the first row, we state a predicate that j' must satisfy. Note that if the value of x cannot be found in the array, any value of j will satisfy the specification. In the next row, we provide a term whose value gives will be the value of present'. We further indicate that the variables x and B should not change (by writing "NC(x, B)"). [2]

$(\exists\ i, B[i] = x)$	$(\forall\ i, ((1 \le i \le N) \Rightarrow B[i] \ne x))$

H_1

j' |
present'=

H_2

B[j'] = x	***true***
true	**false**

$\wedge$ NC(x, B)

G

Figure 6: Relational Description of a program that searches B for the value of x

For Figure 6 to be proper, the two columns must be mutually exclusive. Further, we would like the domain of the function described to be the universe. This means that we would want the

[1] 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.

[2] NC(x,B) is our abbreviation for $x' = `x \wedge `B = B'$

formulae in Figures 7 and 8 to evaluate to ***true*** for any array B. . .

$$(\exists\ i,\ B[i] = x) \vee (\forall\ i,\ ((1 \le i \le N) \Rightarrow B[i] \neq x))$$

Figure 7: Domain Coverage Theorem for Figure 6

$$\neg((\exists\ i,\ B[i] = x) \wedge (\forall\ i,\ ((1 \le i \le N) \Rightarrow B[i] \neq x))$$

Figure 8: Disjoint Domains Theorem for Figure 6

It should be noted that the theorems in Figures 7 and 8 are not as obvious as they might appear. In the logic that we use (described in [2]), if both "=" and "≠" denote primitive relations, (i.e. one is not defined to be the complement of the other) they are *not* complementary. If i is not in the index set of B, both $B[i] = x$ and $B[i] \neq x$ will be ***false***. Thus one cannot apply the standard transformation in this case. We can simplify both the expressions, and the proofs, if we do not include "≠" in the set of primitive relations and define "≠" to be the complement of "=". If we do that all occurrences of the formulae: "$(\forall\ i,\ ((1 \le i \le N) \Rightarrow B[i] \neq x))$", can be replaced by "$(\forall\ i,\ B[i] \neq x)$". Alternatively, we could replace "$B[i] \neq x$" by "$\neg(B[i]= x)$" and get the same simplifications. The simplified theorems are shown in Figures 9 and 10. . .

$$(\exists\ i,\ B[i] = x) \vee (\forall\ i,\ \neg(B[i] = x))$$

Figure 9: Simplified Domain Coverage Theorem for Figure 6

$$\neg((\exists\ i,\ B[i] = x) \wedge (\forall\ i,\ \neg(B[i] = x))$$

Figure 10: Simplified Disjoint Domains Theorem for Figure 6

Column 1 of Figure 6 requires that prove that there will be a value of j' that satisfies the condition specified. This gives rise to the theorem of Figure 9. Since the expressions in the second

$$(\exists\ i,\ B[i] = x) \Rightarrow \textbf{nonempty}(\{j' \mid B[j'] = x\})$$

Figure 11: Definedness Theorem for Column 1 of Figure 6.

row are constants, we need not state the corresponding theorems.

3.2 Searching for a palindrome of length n in an array A with index set 1:N

Figure 12 shows the use of some additional mathematical functions. Although we could have avoided it, we use floor and integer division in the expression "$\lfloor n \div 2 \rfloor$". The Domain Coverage Theorem and the Disjoint Domain Theorem for Figure 12 are trivial as the header for column 2 is explicitly given as the complement of the header for column 1, The definedness theorem *should* be easy, but checking it is likely to be forgotten in practice. It is stated in Figure 13.

	$(\exists\, l, (\forall\, i, 0 \le i < \lfloor n \div 2 \rfloor \Rightarrow A[l+i] = A[l+n-1-i]))$	$\neg(\exists\, l, (\forall\, i, 0 \le i < \lfloor n \div 2 \rfloor \Rightarrow A[l+i] = A[l+n-1-i]))$
l' |	$(\forall\, i, 0 \le i < \lfloor n \div 2 \rfloor \Rightarrow A[l'+i] = A[l'+n-1-i])$	***true***
present'=	***true***	***false***

H_1

H_2

$\wedge NC(n,A)$

G

Figure 12: Find a Palindrome of length l in A[1:n]

$(\exists\, l, (\forall\, i, 0 \le i < \lfloor n \div 2 \rfloor \Rightarrow A[l+i] = A[l+n-1-i])) \Rightarrow$ **nonempty**$(\{l' \mid (\forall\, i, 0 \le i < \lfloor n \div 2 \rfloor \Rightarrow A[l'+i] = A[l'+n-1-i])\})$

Figure 13: Definedness Theorem for Column 1 of Figure 12.

3.3 Looking for the longest palindrome in A [1:N], N>0

The program described by Figure 12 could be used in looking for the longest palindrome that can be found in an array. n' is to designate the length of this palindrome and l' will indicate a location where a palindrome of that length can be found. A specification of such a program is given in Figure 14. Note that maxel(x), where x is a non-empty set of integers, is a function whose value is the largest value in x. Even though we do not need to distinguish cases, the table format is useful. Because, this table has only one column, the Domain Coverage Theorem is as trivial as one can get. However, the Definedness Theorem could be interesting because it depends on our recognising that there will always be palindromes of length 1 in any array of non-zero length.

	true
l' |	$(\forall\, i, 0 \le i < \lfloor n' \div 2 \rfloor \Rightarrow A[l'+i]=A[l'+n'-1-i])$
n'=	maxel$(\{n \mid (n>0) \wedge (\exists\, l, (\forall\, i, 0 \le i < \lfloor n \div 2 \rfloor \Rightarrow A[l+i] = A[l+n-1-i]))\})$

H_1

$\wedge$ NC(A)

H_2

G

Figure 14: Finding the longest Palindrome in a non-empty array.

$n>1 \Rightarrow$ **nonempty**$(\{n \mid (\exists\, l, (\forall\, i, 0 \le i < \lfloor n \div 2 \rfloor \Rightarrow A[l+i] = A[l+n-1-i]))\})$

Figure 15: Definedness Theorem for Figure 12.

4 Concluding observations

Mathematicians reading this paper will find the theorems posed trivial; they are certainly shallow when compared to the theorems that mathematicians prove in published papers. However, they are more difficult than the majority of the theorems that arose in the documentation and inspection of the Darlington Nuclear Plant Shutdown Systems discussed in [7]. Most of the theorems that we had to check, were similar in nature to the ones discussed in Section 2. The scrupulously careful inspection resulted in about 40 kg. of such trivial tables. If these theorems can be proven automatically by today's theorem proving programs, we should be using those programs. If these theorems still require human intervention, perhaps the developers of theorem proving programs would like to turn their attention to this type of theorem.

5 Acknowledgements

This work was supported by the Government of Ontario, through TRIO, and by the Government of Canada through NSERC's Research Grant programme.

6 References

[1] Heninger, K.L., Kallander, J., Parnas, D.L., Shore, J.E., "Software Requirements for the A-7E Aircraft", NRL Memorandum Report 3876, United States Naval Research Laboratory, Washington D.C., November 1978, 523 pp.

[2] Parnas, D.L., "Predicate Logic for Software Engineering", *CRL Report 241,* McMaster University, TRIO (Telecommunications Research Institute of Ontario), February 1992, 8 pgs. To appear in IEEE Transactions on Software Engineering.

[3] D.L. Parnas, J. Madey, M. Iglewski, "Formal Documentation of Well-Structured Programs", *CRL Report 259,* McMaster University, TRIO (Telecommunications Research Institute of Ontario), September 1992, 37 pgs.

[4] D.L. Parnas, J. Madey, "Functional Documentation for Computer Systems Engineering (Version 2)", CRL Report 237, McMaster University, Hamilton Canada, TRIO (Telecommunications Research Institute of Ontario), September 1991,14 pgs.

[5] Elliot Mendelson, "Introduction to Mathematical Logic", Third Edition, Wadsworth and Brooks, Pacific Grove California (USA), 1987.

[6] D.L. Parnas, "Tabular Representation of Relations", *CRL Report 260,* McMaster University, TRIO (Telecommunications Research Institute of Ontario), October 1992, 12 pgs.

[7] D.L. Parnas, G.J.K. Asmis, J. Madey, "Assessment of Safety-Critical Software in Nuclear Power Plants", *Nuclear Safety.* vol. 32, no. 2, April-June 1991, pgs. 189-198.