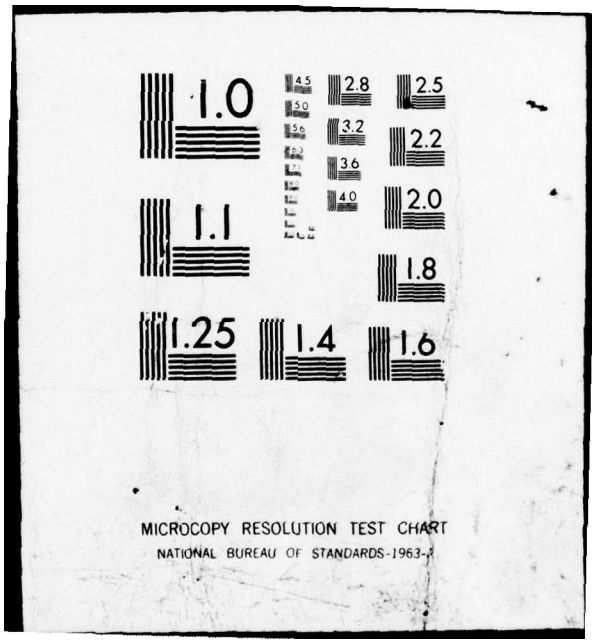
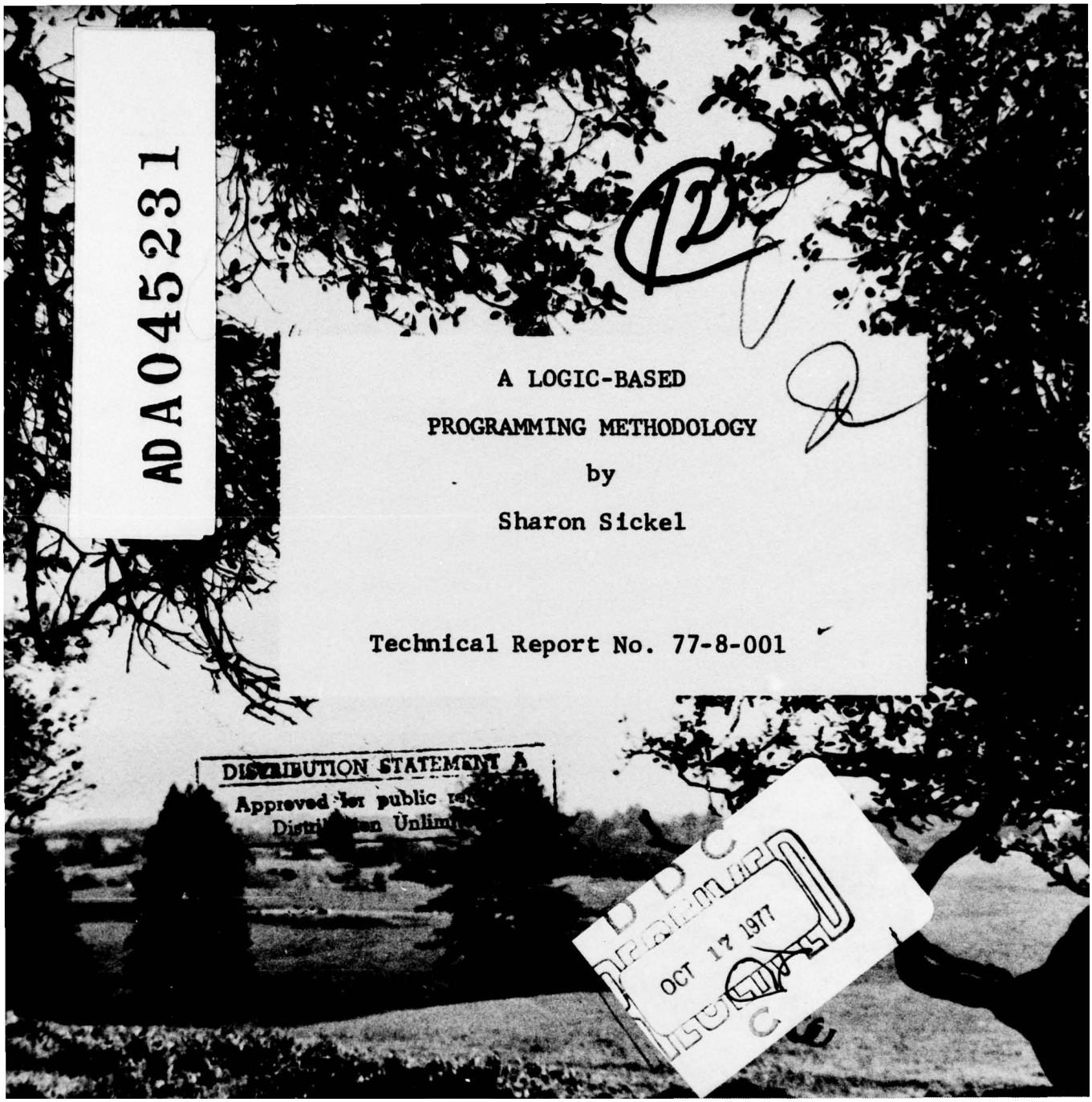
AD-A045 231 CALIFORNIA UNIV SANTA CRUZ INFORMATION SCIENCES F/G 9/2
A LOGIC-BASED PROGRAMMING METHODOLOGY.(U)
MAR 77 S SICKEL N00014-76-C-0681
UNCLASSIFIED TR-77-8-001 NL

| OF |
AD
A045231

END
DATE
FILMED
11-77
DDC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A


ADA045231
A LOGIC-BASED
PROGRAMMING METHODOLOGY
by
Sharon Sickel
Technical Report No. 77-8-001
DISTRIBUTION STATEMENT A
Approved for public re
Distribution Unlim
DDC
OCT 17 1977

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: TR-77-8-001 (14)	2. GOVT ACCESSION NO. / 3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): A LOGIC-BASED PROGRAMMING METHODOLOGY (6)	5. TYPE OF REPORT & PERIOD COVERED: Technical rept. (9)
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): Sharon Sickel (10)	8. CONTRACT OR GRANT NUMBER(s): N00014-76-C-0681 (15)
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Information Sciences, University of California, Santa Cruz, California 95064	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: Office of Naval Research, Arlington, Virginia 22217 (11)	12. REPORT DATE: Mar[illegible] 77 / 13. NUMBER OF PAGES: 8 (12) 10p.
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Office of Naval Research, University of California, 553 Evans Hall, Berkeley, California 94720	15. SECURITY CLASS. (of this report): Unclassified / 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution of this document is unlimited. It may be released to the Clearinghouse, Department of Commerce, for sale to the general public.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Logic programming, executable program, data abstraction, program synthesis

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This paper describes a method of program construction that combines some contributions in structured programming, program verification and program synthesis. This method has start-to-finish continuity within mathematical logic.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102 LF 014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

410350

A LOGIC-BASED PROGRAMMING METHODOLOGY

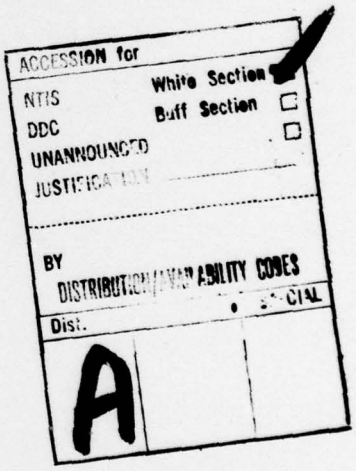
Sharon Sickel

Information Sciences

University of California

Santa Cruz,

California



Research supported by Office of Naval Research Contract # 76-C-0681

Introduction We propose a method of program construction that combines some of the contributions in structured programming, program verification and program synthesis. The new method has the advantage of start-to-finish continuity within a well-understood formal system.

The structured programming approach, developed in response to a concern over the reliability of software, provides a style that helps clarify the meanings of programs. This style advocates top-down refinement in both the control structure of the program, and in the structure of the data objects. Both are reflected in modern programming languages although perhaps not in as elegant and general form as we might wish. One specific disadvantage is that the programming language forces certain irrelevant engineering decisions (data encodings, fully ordered sequencing where a partial ordering is sufficient, etc.) during the problem-solving phase where it would be advantageous to deal only with abstractions. Data objects need only be understood in terms of their properties, and in terms of the relations between them and of the functions that act upon them. There is an extensive literature on this subject; for example see Software Specification and Design[Yeh 77].

We also need to state precisely the specification of a program and then to establish formally that the program carries out the task specified. Program verification starts with the program and builds a specification describing it. Program synthesis starts with a specification and builds a program to carry it out. In both, the predicate calculus provides the language in which to express the specification and the formal theory in which to carry out the proof of equivalence, but does not, in general, determine the construction of the program itself.

The method proposed here uses predicate logic for programming, but not as the executable form of the program (as contrasted to Prolog[Warren 76] where it is also executable). Process and data are describable but at a level of abstraction that is free from implementation issues. In particular the programming language, compiler and supporting hardware can be ignored. Once the program is correctly prepared in logic, the transition to an executable program is done by equivalence-preserving transformations in the first-order logic, or meta-logic.

This paper ties together several other works by the author[Sickel 76, Sickel and Clark 76, Sickel 77].

Overview Figure 1 shows the principal components of this approach, and their relative sequencing. The four forms of programs and the three mappings between them are defined in the following sections.

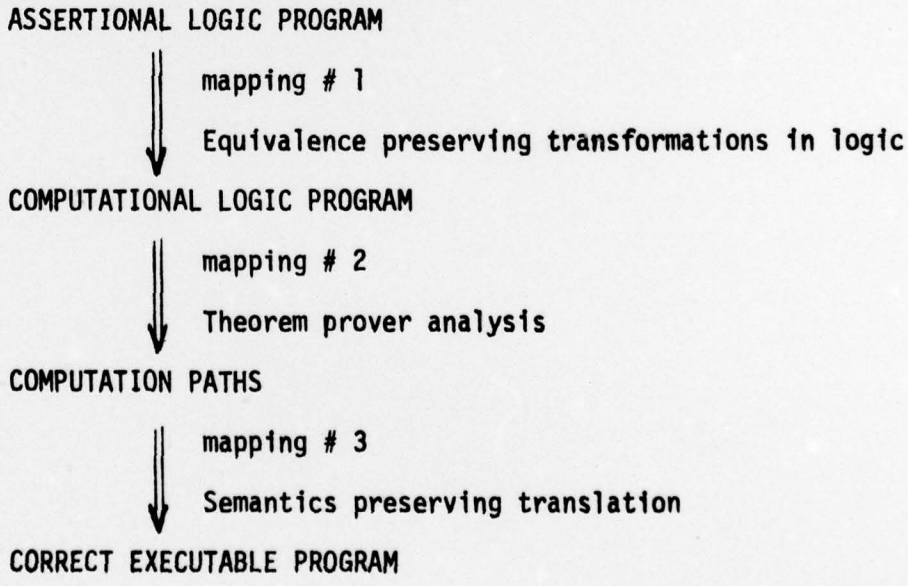


Figure 1. Program forms and their relationships.

An assertional logic program is any set of well-formed formulas of first-order predicate calculus that define a function. All data types and other functions used in this definition must either be primitives or have been previously logically specified. Example:

Given the predicate Member(v,S), meaning $v \in S$, define the predicate Subset(S, T), meaning $S \subseteq T$.

$$Subset(S,T) \Leftrightarrow (\forall v)[Member(v,S) \rightarrow Member(v,T)]$$

A computational logic program is an assertional logic program in which

1) all variables are implicitly universally quantified , and

2) all formulas have the form $A_1 \wedge A_2 \wedge \ldots \wedge A_n \rightarrow B$ where $n \geq 0$, the A_i's and B are positive literals, and B is optional, and

3) the A_i's are implicitly simpler subgoals than B. The particular intent here is to have the derivation of B subgoal driven and to avoid quantifier driven definitions. For example, the assertional logic program given above to determine the subset relation is not computational. The implicit computation is one of trying all elements of the universe to see that if they are in S, they are also in T. This is impossible for infinite domains, and undesirable in all cases. A computational form of Subset is:

$$Subset(\phi,S)$$

$$Member(v,T) \wedge Subset(S,T) \rightarrow Subset(v.S,T)$$

where v.S means $\{v\} \cup S$ with the proviso that $S \subset v.S$ (strict inclusion).

A computation path is a closed form expression that describes all proofs of a theorem. Green pointed out[69] that theorem proving can be used to compute answers. For example, if factorial(n) = x is represented as the predicate Fact(n,x) and the logic definition is:

$$Fact(0,1)$$

$$Fact(n,x) \rightarrow Fact(n+1, (n+1)*x)$$ ‡

---

‡This is, of course, assuming + and * as evaluable primitives which are assumed to be correct.

then we can refute

$$\overline{\text{Fact}}(5,y)$$

and in the process compute a value for y: factorial(5) = 120. Therefore, the computation paths also describe the sequence of operations constructing the output. In the case of Subset the form of the computation path is $d(cb)^*ca|e$ where a-e are shown in Figure 2. Each letter can represent a resolution between literals connected by the edge with that label. Then any refutations of $\overline{\text{Subset}}(A,B)$ can be described by e or $d(cb)^nca$, $n \geq 0$. To better understand this diagram see Sickel[76]. For a more program oriented description of this computation, see the next section.

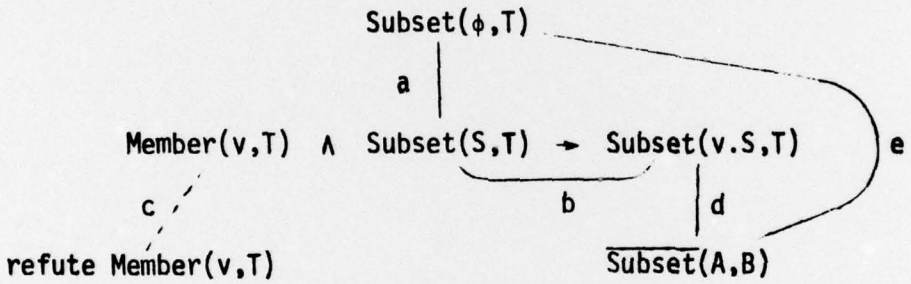


Figure 2

A correct, executable program is defined here to

(1) be expressed in a contemporary programming language, the semantics of which are formally defined, and

(2) be guaranteed to terminate, and

(3) have associated with it an assertional specification which the program is guaranteed to satisfy.

For the subset example, the associated assertional specification is

$$Subset(S,T) \leftrightarrow (\forall u)[Member(u,S) \rightarrow Member(u,T)]$$

and the program resembles the following

```
                logical procedure Subset(S,T)
e       |       if S = φ then TRUE
        |       else begin
d       |          Remove(v,S)
c       |          while Member(v,T) ∧ S ≠φ do
b       |             Remove(v,S)
c,a     |          if Member(v,T) ∧ S = φ then TRUE
                   else FALSE
                   end
```

The letters along the left-hand margin correspond to letters appearing in the computation path expression.

Mapping # 1. The mapping from assertional to computational form is within the predicate calculus and relies on its theorems and rules of inference (distributive, commutative, associative laws, modus ponens, etc.) It is described in detail by Sickel & Clark [77]. This process is partly automatable.

Mapping # 2. Computational logic programs can be analyzed using automatic theorem proving techniques to yield the computation paths. If you wish to compute a function or establish a relation, negate the statement you wish to accomplish and use a theorem prover on the axioms to refute your negated goal. For example, refuting $\overline{Fact}(5,y)$ causes y = 120 to be computed. Refuting $\overline{Subset}(\{a,b,c\},\{c,d,a,b,e\})$ establishes the truth of its positive form. If we negate the most general form of the question, e.g. $\overline{Fact}(n,x)$ or $\overline{Subset}(A,B)$, then we can derive a schema for all proofs (and therefore computations) of

these relations. This schema is the result of a mapping from the axioms and negated theorem onto a grammar whose language is equivalent to the set of all proofs of the theorem from the given axioms. This mapping can be made automatically for all provable theorems in predicate logic [Sickel 77a]. A closed form for the language gives a closed form for the proof set.

Mapping # 3. Going from the computational path expression to the correct, executable program involves two major steps.

1. Represent the data in the target language. Prove that it satisfies the abstract definitions of the data types.

2. Construct the control part of the process by modeling the computation path. The components of the computation path have substitutions associated with them [Sickel 77a]. The substitutions can be used to generate invariants and to suggest constructs in the programming language whose semantics properly interface the invariants. The semantics of the target language must accurately reflect the actions of the compiler and include local hardware peculiarities.

To some extent these two steps are automatable[Sickel 77b].

Conclusions We have proposed a method of program construction. Programs are expressed in logic. The form of the programs encourages thinking at a high, abstract level. The resulting programs are portable in the first three forms, in that they are aimed at no particular target system. They can be transformed within the deductive system of logic to achieve efficiency, and they can be rewritten in a programming language for execution. Some of the steps can be automated. The resulting programs are more easily understood and their correctness is more credible.

References

1. Green, C., Theorem-proving by Resolution as a Basis for Question-answering Systems, Machine Intelligence, Vol. 4, American Elsevier, 1969.

2. Sickel, S., A Search Technique for Clause Interconnectivity Graphs, IEEE Transactions on Computers, Aug. 1976.

3. Sickel, S., Formal Grammars as Models of Logic Derivations, Proceedings of the International Joint Conference on Artificial Intelligence, Boston, Mass.,1977.

4. Sickel, S., and K. L. Clark, Predicate Logic: A Calculus for Deriving Programs, Proceedings of the International Joint Conference on Artificial Intelligence, Boston, Mass., 1977.

5. Warren, David, Implementation of the PROLOG language, University of Edinburgh, Department of Artificial Intelligence, 1976.

OFFICIAL DISTRIBUTION LIST

Contract N00014-76-C-0681

Defense Documentation Center
Cameron Station
Alexandria, VA 22314
12 copies

Office of Naval Research
Information Systems Program
Code 437
Arlington, VA 22217
2 copies

Office of Naval Research
Code 102IP
Arlington, VA 22217
6 copies

Office of Naval Research
Code 200
Arlington, VA 22217
1 copy

Office of Naval Research
Code 455
Arlington, VA 22217
1 copy

Office of Naval Research
Code 458
Arlington, VA 22217
1 copy

Office of Naval Research
Branch Office, Boston
495 Summer Street
Boston, MA 02210
1 copy

Office of Naval Research
Branch Office, Chicago
536 South Clark Street
Chicago, IL 60605
1 copy

Office of Naval Research
Branch Office, Pasadena
1030 East Green Street
Pasadena, CA 91106
1 copy

New York Area Office
715 Broadway - 5th Floor
New York, NY 10003
1 copy

Naval Research Laboratory
Technical Information Division
Code 2627
Washington, DC 20375
6 copies

Dr. A. L. Slafkosky
Scientific Advisor
Commandant of the Marine Corps (CodeRD-1)
Washington, D. C. 20380
1 copy

Naval Electronics Laboratory Center
Advanced Software Technology Division
Code 5200
San Diego, CA 92152
1 copy

Mr. E. H. Gleissner
Naval Ship Research & Development Center
Computation and Mathematics Department
Bethesda, MD 20084
1 copy

Captain Grace M. Hopper
NAICOM/MIS Planning Branch (OP-916D)
Office of Chief of Naval Operations
Washington, D. C. 20350
1 copy

Mr. Kin B. Thompson
Technical Director
Information Systems Division (OP-911G)
Office of Chief of Naval Operations
Washington, D. C. 20350
1 copy