Logic Programming for Software Verification and Testing

S. KHANNA
Syracuse University, P.O. Box 574, Syracuse, NY 13210-0574, USA

We propose a methodology for using logic programming to software verification and testing. The methodology is based on logic programming applications for the formation of decision-to-decision graph, path predicate evaluation and symbolic evaluation of output variables. We elaborate on an efficient software verification scheme which utilizes multiple dynamic theories in logic, organized as a tree structure. A technique to represent the symbolic environments as viewpoints of theories in logic, and an algorithm to locate the valid viewpoint of the leaf theories is presented. An Algol-like language is used to present our approach.

Received June 1989, accepted June 1990

1. INTRODUCTION

Software testing schemes involving numeric test cases\textsuperscript{16, 17, 18} are usually not thorough enough and require an excessive amount of programmer and computer time. We present in this report an alternative in the form of a symbolic execution scheme for conventional programming (Pascal like languages) utilizing logic programming.\textsuperscript{8, 13} We illustrate how a combination of path testing strategy and logic programming can be applied to program verification. The program is first converted to a decision-to-decision graph and then the different path predicates are evaluated using logic programming. The output variables are symbolically expressed in terms of input variables and numeric values. Symbolic evaluation is a useful aid for verifying a program's functional specifications. Determining the path predicates and the symbolic values of the variables which they produce can help verify whether or not a program meets its intended specifications. We also discuss how software verification can be performed with the help of multiple dynamic theories where the theories are organized in a tree structure. The conditional expressions in the if and while statements can produce a true or false value, and two child theories are dynamically created for each case. These theories do not store a complete symbolic environment or a path predicate at each node, since such a scheme will consume excessive memory space. Instead, we take advantage of an inheritance mechanism to pass the symbolic values for the parent theory to the descendent theories. We utilize the concepts of theories and viewpoints to aid us in the concise representation of symbolic environments and the variable-value pairs respectively.

The following definitions will help make the presentation more precise and a bit easier to follow.

\textit{Definition 1}: A \textit{program block} is a list of one or more statements such that if the first statement is executed, then all others are also executed. A block is also referred to as a node.

\textit{Definition 2}: A \textit{path} is a sequence of nodes (blocks) \(n_1, \ldots, n_k\) traversed during the program execution, such that \(n_1\) is the first node and \(n_k\) is the last node. Most programs have a large number of paths because of loop structures.

\textit{Definition 3}: In the \textit{path testing} strategy, each path in the program is traversed at least once. The logical conditions which the set of input variables must satisfy for a path to be traversed are called the \textit{path predicate}.

2. LOGIC PROGRAMMING FOR PROGRAM VERIFICATION

Our methodology for program verification involves performing lexical analysis on the source program to produce a list of tokens, parsing the tokens to form a decision-to-decision graph (dd-graph), and utilizing this graph to perform symbolic evaluation. The parsing task is easily achieved using Definite Clause Grammar\textsuperscript{8, 15} which is standard with any logic programming language. The dd-graph formation and symbolic evaluation is discussed in the next sections.

2.1. Formation of decision-to-decision graph

We now discuss how a decision-to-decision graph (dd-graph) can be formed for a given program. When we convert the program to its internal form during parsing, the program statements are stored as a list. Markers in the form of $N(N = 0, 1, 2, \ldots)$ are placed at the beginning of each program block. Thus, for the if statement, a marker is inserted in each of the \textit{then} and \textit{else} parts. For the while statement, markers are put at the beginning of the \textit{while} loop body and after the end of the loop. Markers are also inserted at the beginning and end of the program. The dd-graph also contains a list of all the variable-value pairs in the program. Each element of the list has the variable name and an initial value \textit{nil}.

\[
[(\text{var}_1, \text{nil}), (\text{var}_2, \text{nil}), \ldots, (\text{var}_N, \text{nil})]
\]

The \textit{nil} values are substituted by numeric or symbolic values during the execution of the dd-graph. This representation of the decision-to-decision graph aids in determining the path of execution on given input values, as well as in program verification using symbolic evaluation, as will be discussed in the next section.

We use the following representation for the \textit{if} and \textit{while} statements in the decision-to-decision graph.

\[
\text{i}f \text{ statement } * \\
\text{[if,COND,$[$Ni|\text{THEN}]],[$Nj|\text{ELSE}]]}
\]

where COND specifies the conditional predicate of the \textit{if} statement and \textit{THEN} and \textit{ELSE} consist of the list of statements in the \textit{then} and \textit{else} parts respectively.
* while statement *
  [while, COND, [SN_i|LOS], SN_j]
where COND is the conditional predicate of the while statement, and LOS is the list of statements in the body of the while loop.

As an example, let us consider a program which performs integer division by successive subtractions. The reference branches in the program are illustrated with three markers.

/* PROGRAM 1 */
begin integer x, y, div, mod; $0
  div := 0;
  mod := x;
  while (mod $ y) do
    begin $1
      mod := mod $ y;
      div := div + 1;
    end;
$2
end;

The decision-to-decision graph for the above program is as follows:

2.2. Program verification with symbolic evaluation
Path predicate evaluation proceeds along a tree structure where the symbolic environments form the nodes and the conditional statements (branch predicates) form the edges between nodes. A branch predicate is evaluated in a current environment and leads to one or two new environments along a true branch, a false branch, or both. Both branches are followed if the result of the evaluation is not specific enough to yield true or false. Each node in the tree is tagged with the node number of the dd-graph which will be executed if that branch is followed. We start building the tree with the root node containing an empty path predicate. The path predicate corresponding to the left child of a node is obtained by a conjunction of the path predicate of the parent node and the branch predicate obtained by substituting the variables in the next condition (COND) of the dd-graph with their values in the symbolic environment represented.

![Diagram](https://example.com/diagram.png)

Figure 1. The tree structure for path predicate evaluation of program 2. The symbolic environment (SymEnv) and the node transversed in the dd-graph, after the execution of a block of statements corresponding to each branch predicate (BP) is shown.
by the parent node. The path predicate corresponding to the right child is similarly obtained except that the condition is replaced by its negation. The branch predicate in a while statement is evaluated only to the extent that the while body is executed 0 or 1 times. This avoids the possibility of an infinite number of nodes in the tree. The new symbolic environment for a child node is obtained by acquiring the symbolic environment of the parent node, then making the modifications according to the statements in the block following the conditional statement. A leaf node is reached when the node is tagged with the highest numbered node of the dd-graph. As an example, the symbolic environments tree structure for Program 2 below is illustrated in Figure 1.

/* PROGRAM 2 */
/* INPUT VARIABLES X, Y, Z */
begin integer P, Q, R, X, Y, Z;
S0 R := X; P := Y; Q := Z+1;
while R < P do
begin
S1 R := R+1;
if P > Q then
S2 P := P - Q
else
S3 Q := Q - 1
end;
S4 end.

The conjunction of branch predicates from the root node to a leaf node form a path predicate. The tags on the nodes from the root to a leaf node show the path executed by the program for the evaluated path predicate. As an example, we have the following path predicate (before simplification) corresponding to LEAF 2.

(X < Y) and (Y <= Z+1) and (X+1 >= Y)
Path: 0 → 1 → 3 → 4

Symbolic execution has been investigated by several researchers. The proposed methodologies evaluate the symbolic value of output variables along the corresponding path predicates. However, the techniques used suffer from the disadvantage that the values in the symbolic environments have to be passed along every branch of the program. Figure 1 shows the different symbolic environments (SymEnv) created while evaluating Program 2.

2.3. Symbolic evaluation using logic programming

Symbolic evaluation with the aid of logic programming can be performed by maintaining all the variable-value pairs in a substitution list.

[(Name1, Sym1),
(Name2, Sym2), ..., (NameN, SymN)]

Each element of the substitution list has the variable name followed by its current symbolic value. Initially, the symbolic value of an input variable is the input variable itself, and the symbolic value of other variables is nil. When an assignment statement is encountered, the symbolic value of the variable on the left hand side of the assignment is updated in terms of the symbolic value of the variables on the right side. This involves a lookup of the variable in the substitution list, and the replacement of its value by the new one. A portion of the old substitution list is duplicated during this process which results in the creation of a new substitution list. The substitution lists corresponding to each environment have to be passed down to the new environments created as the execution of the dd-graph proceeds. The variables in the conditional expression of the if and while statements (COND in our dd-graph), are updated using the substitution list. The conditional expression then produces the branch predicate. In the while loop, variables in COND are symbolically updated every time around the loop. The new path predicates are just the conjunction of the old path predicates and the branch predicate before each node. To evaluate a procedure call, we symbolically substitute the actual parameters for the formal parameters. The body of the procedure may consist of while, if or assignment statements. The procedure body is symbolically evaluated in terms of these statements, as discussed above, and the symbolic value of the variables in the formal parameter list is passed back to the calling procedure. We illustrate the above concepts with the predicate symbolic_evaluate in Prolog.

Explanation of Variables
OE Present Program Variables.
NE New Numeric Environment of Program Variables.
IE Intermediate Numeric Environment of Program Variables.
OSE Present Symbolic Environment of Program Variables.
NSE New Symbolic Environment of Program Variables.
ISE Intermediate Symbolic Environment of Program Variables.
OCOND The path predicate prior to the execution of current statement.
NCOND The path predicate after the execution of current statement.
ICOND Intermediate path predicate.
COND The conditional expression in the if or while statement.
(Same branch predicate)
SCOND The branch predicate with variables replaced by their symbolic values (OSE).

symbolic_evaluate((while, COND, SLIST),
OE, NE, OSE, NSE, OCOND, NCOND):
expr(Result, OE, COND, []),
((Result = false, !), NE = OE, NSE = OSE,
symbolic_negated_branch_predicate(COND,
OSE, SCOND),
conjunct_predicate(OCOND, SCOND, NCOND)),
Result = true,
symbolic_branch_predicate(COND, OSE, SCOND),
conjunct_predicate(OCOND, SCOND, ICOND1),
symbolic_evaluate(SLIST, OE, IE, OSE, ISE, ICOND1, ICOND2),

352 THE COMPUTER JOURNAL, VOL. 34, NO. 4, 1991
symbolic_evaluate([while, COND, SLIST],
  IE, NE, ISE, NSE, ICOND2, NCOND)).

symbolic_evaluate([if, COND, THEN, ELSE],
  O, NE, OSE, NSE, OCOND, NCOND):-
  expr(Result, O, COND, []),
  (Result = true, !,
   symbolic_branch_predicate(COND, OSE, SCOND),
   conjunct_predicate(OCOND, SCOND, ICOND),
   symbolic_evaluate(THEN, O, NE, OSE, NSE, ICOND, NCOND));
  Result = false,
  symbolic_negated_branch_predicate(COND, OSE, SCOND),
  conjunct_predicate(OCOND, SCOND, ICOND),
  symbolic_evaluate(ELSE, O, NE, OSE, NSE, ICOND, NCOND)).

symbolic_evaluate([VAR, ':=', EXP], O, NE, OSE, NSE, OCOND, OCOND):-
  expr(Value, O, EXP, [VAR]),
  change_environ(O, [VAR, Value], NSE),
  symbolic_evaluate(SymbolicValue, OSE, EXP, []),
  change_environ(OS, [VAR, SymbolicValue], NSE).

symbolic_evaluate([H | T], O, NE, OSE, NSE, OCOND, NCOND):-
  symbolic_evaluate(H, O, IE, OSE, ISE, OCOND, ICOND),
  symbolic_evaluate(T, IE, NE, ISE, NSE, ICOND, NCOND).

symbolic_evaluate([], O, NE, OSE, OSE, OCOND, OCOND).

symbolic_evaluate([$|SLIST], O, NE, OSE, NSE, OCOND, NCOND):-
  symbolic_evaluate(SLIST, O, NE, OSE, NSE, OCOND, NCOND).

conjunct_predicate([], SCOND, SCOND).
  conjunct_predicate(OCOND, [], OCOND).

conjunct_predicate(OCOND, SCOND, NCOND):-
  append(OCOND, [and], T),
  append(T, SCOND, NCOND).

symbolic_branch_predicate([\VarCOND|RestCOND], OSE, [SymVal|RestSymVal]):-
  symbolic_branch_predicate(VaRCOND, OSE, SymVal),
  symbolic_branch_predicate(RestCOND, OSE, RestSymVal).

symbolic_branch_predicate(Var, [[Var, SymVal]|TailOSE], SymVal):-!.

symbolic_branch_predicate(Var, [[VAR1, _]|TailOSE], SymVal):-
  symbolic_branch_predicate(Var, TailOSE, SymVal).

symbolic_branch_predicate([], OSE, []).

symbolic_expr(SymVal, SymEnv) :-
  symbolic_expr2(Val1, SymEnv),
  (symbolic_expr1(Val1, SymVal, SymEnv); empty, SymVal = Val1).

symbolic_expr1(Val1, SymVal, SymEnv) :-
  addop(A), symbolic_expr2(Val2, SymEnv),
  (SymVal2 = [Val1, A, Val2], symbolic_expr1(SymVal2, SymVal, SymEnv); empty, SymVal = SymVal2).

symbolic_expr2(SymVal, SymEnv) :-
  symbolic_expr4(Val1, SymEnv),
  (symbolic_expr3(Val1, SymVal, SymEnv); empty, SymVal = Val1).

symbolic_expr3(Val1, SymVal, SymEnv) :-
  mulop(M), symbolic_expr4(Val2, SymEnv),
  (SymVal2 = [Val1, M, Val2], symbolic_expr3(SymVal2, SymVal, SymEnv); empty, SymVal = SymVal2).

symbolic_expr4(Val1, SymEnv) :-
  identifier(Ident), [lookValue(Ident, Val, SymEnv)].

symbolic_expr4(Val1, SymEnv) :-
  leftrap(L), symbolic_expr(Val, SymEnv),
  rtrap(R).

empty(E, E).

The predicate symbolic_evaluate takes the decision-to-
decision graph and the current numeric and symbolic
environments of the variables. Every conditional predi-
cate in an if or while statement is evaluated using the
expression evaluator, and the set of statements of the
corresponding reference branch are symbolically
evaluated. The symbolic and numeric environments are
changed for variables to which a new value is assigned.
The predicate expr evaluates the numeric value of an
arithmetic or boolean expression, given the current
numeric environment (OE). The predicate symbolic_expr
evaluates the symbolic value of an arithmetic expression
on the right hand side of the assignment statement with
the help of the current symbolic environment. The
predicate symbolic_branch_predicate substitutes the pre-
sent symbolic environment (OSE) in the conditional
expression (COND) of the if or while statement, to
obtain the symbolic branch predicate (SCOND). The
predicate symbolic_negated_branch_predicate substitutes
the current symbolic environment (OSE) in the negated
conditional expression, to obtain the symbolic branch
predicate (SCOND). This predicate is used when the
result of the evaluation of the conditional expression in
the if or while statement with the present numeric

THE COMPUTER JOURNAL, VOL. 34, NO. 4, 1991  353

23 CPJ 34
environment (OE) yields a false boolean value. The predicate conjunct Predicate performs a conjunction of the current path predicate (COND) and the symbolic branch predicate (SCOND) obtained from the conditional expression as discussed above. If the current path predicate (COND) is empty, then the new path predicate is the same as SCOND.

The clauses for symbolic evaluate have the following declarative meaning, in the order that they are listed above.

1. For the while statement, we evaluate the conditional expression (COND) with the current numeric environment (OE). If the result is false, then the symbolic environment is unchanged and the body of the while statement is not executed. Moreover, we substitute the current symbolic environment (OSE) in the negated conditional expression to obtain the symbolic form of the branch predicate (SCOND). Now we conjunct SCOND with the current path predicate (COND) to obtain the updated path predicate (NCOND).

If the result of conditional expression evaluation is true, then we substitute the current symbolic environment in the conditional expression to get the symbolic branch predicate (SCOND). Now the conjunction of SCOND and the current path predicate (COND) gives us the intermediate path predicate (ICOND). The list of statements (SLIST) in the body of the while is evaluated in the context of COND, OSE, and ICOND to produce intermediate symbolic (ISE) and numeric (IE) environments and another intermediate path predicate (ICOND2). Now recursively evaluating the while statement with IE, ISE and ICOND2 would give us the final symbolic environment (NSE) and the updated path predicate (NCOND).

2. For the if statement, if the evaluation of the conditional expression (COND) yields a boolean true, we symbolically evaluate the statements in the then body (THEN) with the help of the intermediate path predicate (ICOND), which is obtained as discussed above. If the conditional expression evaluates to a boolean false, we symbolically evaluate the statements in the else body (ELSE), in the context of intermediate path predicate (ICOND), to produce the new symbolic (NSE) and numeric (NE) environments as well as the new path predicate (NCOND).

3. For the assignment statement, we evaluate the symbolic value of the expression (EXP) on the RHS with the aid of the current symbolic environment (OSE). The numeric and symbolic environments of the variable (VAR) on the LHS are updated with the help of the predicate change environ.

4. To accomplish symbolic evaluation for a list of statements, we symbolically evaluate the first statement to obtain the intermediate environments (IE, ISE) and the intermediate path predicate (ICOND). The rest of the statements are symbolically evaluated in the context of IE, ISE and ICOND to give us the new symbolic environment (NSE) and the new path predicate (NCOND).

5. Symbolic evaluation of an empty statements does not change the environments or the path predicate (COND).

The results of a symbolic evaluation of Program 2 from the previous section is shown below. The path predicates are shown on the left, while the symbolic values of the output variables for each possible path are depicted on the right. The symbolic output values for a given path predicate can now be matched against the functional specifications of the program to establish its correctness.

**PATH Predicate**

<table>
<thead>
<tr>
<th>Symbolic Value Of</th>
<th>R</th>
<th>P</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>( X &lt; Y ) and ( Y &gt; Z + 1 )</td>
<td>( X + 1 )</td>
<td>( Y - Z - 1 )</td>
<td>( Z + 1 )</td>
</tr>
<tr>
<td>( X &lt; Y ) and ( Y \leq Z + 1 )</td>
<td>( X + 1 )</td>
<td>( Y )</td>
<td>( Z )</td>
</tr>
<tr>
<td>( X \geq Y )</td>
<td>( X )</td>
<td>( Y )</td>
<td>( Z + 1 )</td>
</tr>
</tbody>
</table>

Thus, using the substitution list, we can evaluate the path predicates and the symbolic values of the output variables for each path in the program. However, this technique is not very efficient for programs with a large number of variables. Whenever the value of a variable needs to be updated (as a result of an assignment, for example) during the symbolic evaluation, the entire list may have to be scanned and a large portion of it duplicated. This process is repeated for every variable in every environment. To get around the inefficiency of the list searches and duplications, we propose a technique that uses multiple theories to store the values currently held in the substitution list. The viewpoints provide a fast lookup operation while the theories prevent the duplications through the inheritance mechanism.

### 3. PROGRAM VERIFICATION USING MULTIPLE THEORIES

Multiple theories can be used to simulate the symbolic environments. We organize the theories as a tree structure and establish a one to one mapping between them and the nodes of the tree structure for symbolic environments (observe the similarities between Figure 1 and Figure 2). Each variable-value pair in the symbolic environment is turned into a relation variable(value) in the corresponding theory. We use the viewpoints to hold the symbolic values of variables. The idea of a viewpoint is as follows: since a relation can exist in many theories with a slight variation from one theory to the other, we say that each theory has its own viewpoint on the relation. So the viewpoints of a theory on a certain relation consists of the clauses about that relation that are visible in that theory.

Since different environments may have different values for the same variable, the different theories representing these environments may have different viewpoints about the values of the variable. When a child theory is created, it inherits all the viewpoints from the parent theory about the values of the variables, except those in conflict with the new viewpoints introduced during the creation of the child theory. The new viewpoints are introduced as a result of changes in values of some variables when new statements are executed. Now the lookup process for values of variables becomes a simple call to variable (CurrentValue), while the update process involves the creation of a new theory where the viewpoint about the value of a variable is changed to variable(new Value). The new theory inherits the values of all the other variables from its parent theory. This elegant solution prevents the
Figure 2. Tree structure for symbolic evaluation with multiple theories in logic. At each node, we indicate the viewpoint on relations $r$, $p$, and $q$, and the path predicate (pp). A child theory inherits all the viewpoints from the parent theory, except for those newly introduced.

copying of variables between environments, as well as the expensive search for their values. Let's look at Figure 2 again for an example. From the viewpoint of theory T3, the value of $P$ is $Y$-$Z$-1 while from that of theory T5 it is $Y$. Theory T3 has its own viewpoint on the relation $p$, while theory T5 inherits its viewpoint from its ancestor theory T1.

Corresponding to every path predicate is a leaf theory which holds the symbolic values for the output variables showing the results should the execution proceed along that path. To get back to our example of Program 2, the leaf theories T4, T6 and T7, represent the following viewpoints.

T4: $(x + 1)$
T6: $(y + 1)$
T7: $(z + 1)$

The argument value of the relations at the leaf theories give us the symbolic output value of the variables. Thus the leaf theories T4, T6, and T7 hold the symbolic values for the output variables $R$, $P$, and $Q$ for each of the three alternative path predicates, as shown in Table 1.

A theory is assigned a positive integer as its ID number. Theory ID's start with 1 for the root theory and increase by one for every new theory. A theory is represented by its ID number, its parent theory ID and a sequence start theory ID. The sequence start theory ST of a theory T, represented by ST(T), is the oldest ancestor theory of T such that the theory ID's of the theories between ST and T form a continuous sequence of consecutive integers. As an example, in Figure 2, ST(4) is 1 and ST(6) is 5. For the purposes of software verification we need to find the viewpoints of the leaf theories. Since every viewpoint carries the signature (ID)

<table>
<thead>
<tr>
<th>THEORY</th>
<th>PATH PREDICATE</th>
<th>SYMBOLIC VALUE OF</th>
</tr>
</thead>
<tbody>
<tr>
<td>T4</td>
<td>$(X &lt; Y)$ and $(Y &gt; Z+1)$ and $(X+1 \geq Y-Z-1)$</td>
<td>$X+1$ $Y-Z-1$ $Z+1$</td>
</tr>
<tr>
<td>T6</td>
<td>$(X &lt; Y)$ and $(Y \leq Z+1)$ and $(X+1 \geq Y)$</td>
<td>$X+1$ $Y$ $Z$</td>
</tr>
<tr>
<td>T7</td>
<td>$(X \geq Y)$</td>
<td>$X$ $Y$ $Z+1$</td>
</tr>
</tbody>
</table>
of the theory in which it originated, that signature can be matched against the ID’s of the theories along the branch for the leaf theory to the root theory. A viewpoint about a given relation R is valid in a particular theory T if it either carries the signature of T, or it carries the signature of one of the ancestors of T, and this viewpoint is the latest such viewpoint about R. Viewpoints on a particular relation are chained together from the latest to the earliest (Figure 3). The latest viewpoint is called FinalVP. The sequence start theory ID helps us in examining all the theories in a sequence simultaneously, instead of examining every theory along the branch. The following algorithm efficiently locates the viewpoints in a leaf theory L.

1. If the signature of the viewpoint FinalVP is equal to the theory ID of L or the root theory, then return FinalVP as the correct viewpoint in L and go to step 7.
2. If the signature of the viewpoint FinalVP occurs between the present theory L and ST(L), then return FinalVP and go to step 7. For example, suppose we have to determine the validity of the FinalVP viewpoint \( r(x+1) \), carrying signature 2, for the relation r in theory T4 of Figure 2. We can say that FinalVP is valid in T4 if FinalVP originated anywhere in the sequence from 1 (ST(4)) to 4. Even though there are two viewpoints in r in the sequence, corresponding to T2 and T1, FinalVP is still valid since FinalVP is the latest viewpoint in the sequence. Thus we do not have to check every theory in the sequence. We only have to ascertain that the signature of the viewpoint FinalVP lies between the ID’s of the theories in the sequence. This scheme makes the algorithm very fast.
3. If FinalVP originated after the theory L was created, then Final VP cannot be valid in L. We traverse the chain of viewpoints backwards until we find a viewpoint whose signature is less than the theory ID of L or we reach the last viewpoint in the chain. FinalVP is now the new viewpoint.
4. If the signature of FinalVP is greater than the theory ID of the current theory and we have no more viewpoints available in the chain, then L does not have a valid viewpoint on the given relation, and we quit by going to step 7.
5. While the sequence start theory of L, ST(L), is greater than the signature of FinalVP and ST(L) is not the root theory, we make the current theory L to be the parent of ST(L). In other words, if the viewpoint does not originate in the current sequence, we look for an earlier sequence along the branch.
6. Repeat steps 2 through 5 till a valid viewpoint is found or it is determined that none exists.
7. Exit.

As an example, let us find the viewpoint on the relation \( p \) in theory 6 (Figures 2 and 3). The signature of FinalVP of \( p \) is 3. The sequence start theory of 6, ST(6), is 5. Since ST(6) is greater than the signature of FinalVP, we make the current theory, L, to be the parent of ST(6), i.e., theory 2 (Step 5). Next, we note that the signature of FinalVP of \( p(3) \) is greater than the theory ID (2) of the current theory. We, therefore, traverse the chain of viewpoints backwards (Step 3). The signature of the new FinalVP of \( p \) is 1. Now we observe that the signature of FinalVP (1) lies between the theory ID (2) of the current theory and its sequence start theory (1). Hence, we return the viewpoint, \( p(y) \), as the viewpoint valid in theory 6 (Step 2). As a second example, suppose we wish to find the viewpoint on the relation \( p \) in theory 4. Since the signature of FinalVP of \( p(3) \) is between 1 (ST(4)) and 4, we return FinalVP \( p(y-z-1) \) as the correct viewpoint in theory 4 (Step 2). In this case a valid viewpoint is found in one step.

For the purposes of software verification with multiple theories in logic, a meta-level\(^{6,15}\) extension of Prolog, called MetaProlog\(^{1,4,5}\) is very useful. In MetaProlog a new theory is created from an existing theory with the help of the \texttt{addto} and \texttt{dropfrom} predicates.

\[
\text{addto}(\text{OldTheory, Clauses, NewTheory})
\]
\[
\text{dropfrom}(\text{OldTheory, Clauses, NewTheory})
\]

NewTheory inherits all the viewpoints from the parent theory, OldTheory, except for those newly introduced with Clauses. The newly introduced viewpoints are either modifications (additions or deletions) of viewpoints in OldTheory, or viewpoints on relations about which OldTheory had no knowledge.

Symbolic evaluation with multiple theories can be accomplished in parallel. For a node in the tree structure where two alternatives along two child theories are possible, symbolic execution can fork into two parallel executions: one following the alternative along the left child theory and the other along the right child theory. The state of symbolic computation and the path constraints at the parent theory are inherited by the two child theories. The two child theories proceed independently from this stage, building on the computation state which exists at the parent theory.
4. CONCLUSIONS

A software verification scheme with multiple theories in logic, is more efficient than a scheme utilizing a single-
theory database. The methodology incorporating mul-
tiple theories consumes considerably less space, since it
avoids unnecessary duplication of symbolic environments
at each node in the symbolic environments tree structure.
Moreover, with efficient viewpoint representation and
search, the verification process is a lot faster. Instead of
examining every theory in the branch from the current
to the root theory, we look at all the theories in a
particular sequence simultaneously. Thus the order of
execution time of the algorithm depends on the number of
sequences and not the number of theories along the
branch.

Acknowledgement

The author would like to thank Hamid Bacha for some
useful discussions on symbolic execution with the help of
multiple theories. His contribution to parts of section 3
of the paper is gratefully acknowledged.

REFERENCES

1. H. Bacha, Meta-Level Programming: a Compiled Ap-
proach. Proc. of 4th International Conference on Logic
Programming, Melbourne, Australia (1987).
of 5th International Conference on Logic Programming,
TR88-1, School of Computer and Information Science,
Syracuse University, Syracuse, NY 13244. (1988).
4. K. A. Bowen and R. Kowalski, Amalgamating Language
and Metalanguage in Logic Programming, Logic Pro-
gramming, edited K. L. Clark and S. A. Tarnlund, Aca-
5. K. A. Bowen and T. Weinber, A Meta-level Extension of
Prolog, Proc. of the 1985 IEEE Symposium on Logic
Programming, Boston, pp. 669–675.
6. K. A. Bowen, Meta-level programming and knowledge
(1985).
7. L. A. Clarke, A system to generate data and symbolically
execute programs. IEEE Trans. Software Eng., SE-2,
8. W. F. Clocksin and C. S. Mellish, Programming in Prolog,
9. A. Colmerauer, Metamorphosis Grammars, Natural
language communication with computers. Lecture Notes
in Computer Science no. 63, pp. 133–189, Springer,
Heidelberg.
10. William Howden, Reliability of the path analysis testing
strategy. IEEE Transactions on Software Engineering,
11. William Howden, Symbolic Testing and the DISSECT
Symbolic Evaluation System. IEEE Transactions on
12. J. C. King, Symbolic execution and program testing,
13. R. Kowalski, Predicate Logic as programming language,
14. L. Sterling and E. Shapiro, The Art of Prolog, Chapter 16,
15. A. Takeuchi and K. Furukawa, Partial Evaluation of
Prolog Programs and its Application to Meta-programming,
16. Walker et al., Knowledge Systems and Prolog, Addison
17. E. J. Weyuker and T. J. Ostrand, Theories of program
testing and the applications of revealing subdomains. IEEE
Transactions on Software Engineering, SE-6, (3), 236–246
(1980).
18. L. J. White and E. I. Cohen, A Domain Strategy for
Computer Program Testing, IEEE Trans. on Software

Book Review

L. R. Henderson and A. M. Mumford
The Computer Graphics Metafile
Butterworth Scientific Ltd., Guildford. £45.00.
ISBN 0 408 02680 4
This is the second book in a planned series on
computer graphics standards, following ISO
Standard for Computer Graphics by Arnold
And Duce, and it is intended as an introduction
and supplement to the CGM standard ISO
8632-1:1987. It is, however, very much a
reference work, of particular relevance to
graphics software engineers and support
groups that are becoming involved with
implementing and maintaining the CGM
standards, and who therefore need the expert
help the authors give if they are to find their
way round a difficult-to-read standards docu-
ment. It is also relevant to managers deciding
whether the CGM is relevant to a particular
application of computer graphics, who may
then wish to move on to consider implemen-
tation details. Although it is well written and
presented, it is still not a book to be
recommended as a light, bedtime read.

The book is written in five overlapping
parts. Part 1 explains how a computer graphics
metafile can be used for storing and com-
unicating pictures that have perhaps been
generated by a user of GKS, GKS3D, PHIGS
or some other standard, and explains how
the new standard relates to them. It is as though
the graphic output commands from one of the
earlier standards can be directed to storage in
the CGM format, as an alternative destination
to a plotter or some interactive graphics device.

There are illustrative examples of some
applications which led to the activity giving
rise to the new standard, and a section on
Finding your way round the Standard’
document itself. Part 2 moves on to look at the
CGM in further detail, with a chapter listing
the detailed elements involved, while part 3
gives a very detailed look at the CGM
encodings, and is a genuine attempt to add to
the information in the standard and to share
the authors, experience of implementations
with others embarking upon a similar in-
volvement at either the expert user or
implementor levels.

Part 4 addresses a number of implementa-
tion details and suggests how an appropriate
subset of the metafile elements can be chosen
to match a specific application – though it is
again emphasised that the metafile is con-
cerned with the storage of pictorial infor-
mation, rather than the application – and
how a CGM can be generated from GKS, GKS3D or
PHIGS.

Finally part 5 outlines some proposals for
discussion by the appropriate standards
workers for future elements, including new
capabilities in both 2D and 3D, and a final
chapter adds, for the sake of completeness,
‘details of the new Standard Addendum to
GKS which adopts CGM elements and
encodings to produce an audit trail metafile’.
In summary, this is a useful introduction
and reference work for those likely to become
professionally involved with the new Com-
puter Graphics Metafile Standard.

M. L. V. Pitteway
London