THE ASSERTION TABLE SYSTEM
FOR THE PL/CV2 PROGRAM VERIFIER

by

Dean B. Krafft[†]

TR78-337

Department of Computer Science
Cornell University
Ithaca, N.Y. 14853

[†]This research was supported in part by NSF grants MCS 76-14293 and MCS 77-08198.

THE ASSERTION TABLE SYSTEM
FOR THE PL/CV2 PROGRAM VERIFIER

by

Dean B. Krafft

78-337

Department of Computer Science
Cornell University
Ithaca, N.Y. 14853

Abstract

A system to implement the block structured storage of PL/CV2 assertions is described. The system allows certain simple logical deductions to be performed automatically. These include deductions involving propositional reasoning, associativity and commutativity of arithmetic operators, and reasoning about equality. The implementation is described at a conceptual level.

1. Introduction

The assertion table (AT) for the PL/CV2 program verifier system (described in [1]) is a set of routines and data structures which perform the following functions:

1) Block structured storage of proved PL/CV2 assertions.
2) Automatic deduction of certain obvious inferences from assertions stored in the table.

The AT has four external entry points, ASSERT, QUERY, BLOCKEN, and BLOCKEX. ASSERT declares in the table the truth of the assertion given as its argument. QUERY takes an assertion and returns true if it is in the table or if it can be automatically deduced from assertions in the table. BLOCKEN and BLOCKEX implement block entry and exit for the block structured storage of assertions (see [1] Sec. 2.4.5 for a discussion of the block structure of assertions in PL/CV2).

This paper provides a conceptual description of the functions performed by the AT. The data structures and their manipulations are fully described, but programming level details have not been included. For a technical description of the PL/I implementation of the AT see [2].

2. The Block Structured Storage of Assertions.

2.1. How Assertions Are Stored.

In this section we describe the data structures that represent assertions about the program. The motivation for these particular data structures, and some of the details of their use, will be discussed in the section on automatic deductions. In fact, every assertion is stored twice, once in each of two different data structures. These structures are the DAG and the AC table.

2.1.1 The DAG

All assertions are stored in a directed acyclic graph (dag) of expressions. We can envision creating this dag from a set of assertions by first creating the expression tree for each assertion in the set and then combining subtrees so that each unique subtree occurs exactly once. For example, given the assertions:

$$(X > Y) \ \& \ (Y > Z) \Rightarrow (X > Z)$$
$$(X > Y) \mid (Y > Z)$$

we give their corresponding expression trees in Figure 1 and the associated dag in Figure 2. The starred (*) nodes in the figure are the heads of proved assertions. To find an assertion in the dag given its expression tree matching progressively larger subtrees against the dag. The routines currently use a Hash-Cons technique ([6]) to find the appropriate operator node in the dag given its operands and the name of the operator.

The dag is used to implement automatic equality substitution (see Sec. 3.3). It is also used as a simple lookup table for assertions.

2.1.2. The AC Table

This is the associativity/commutativity table. It is used to implement automatic associativity and commutativity. It is simply a hash table in which assertions are stored in a special AC normalized operator prefix form. AC normalization is performed as follows:

1) If an operator is associative and if it appears as the major operator of an operand of itself, then the operator of higher precedence is given the operands of the operator of lower precedence. The operator of lower precedence is removed, and the arity of the remaining operator is corrected. Note that all associative operators are treated as arbitrary n-ary operators.
2) If an operator is commutative, then its arguments are sorted into lexicographic order.

For a discussion of which operators are considered associative and/or commutative, see Sec. 3.2.

2.2. How Block Structure Is Achieved

Conceptually we can think of block entry as simply causing a copy of both the dag and the AC table to be saved. The corresponding action on block exit is then simply to restore the saved copies. The actual implementation is, fortunately, slightly more efficient than this.

To implement block structure for the AC table we simply place new assertions on a stack. On block entry we push a special marker on the stack, and on block exit we just throw away the assertions from the top of the stack down to and including the block marker.

The block structure situation for the dag is slightly more complex, but we solve it by a simple extension to the above method. We again have a stack, but this time it consists of pairs of a pointer to a node in the dag and a node to replace the current dag node on block exit. Each time a new node is added to the dag, we push a pointer onto the stack with a flag indicating that the node should be deleted on block exit. Each time we update a node, we save a copy of the old node on the stack. An update would occur, for example, when we mark the head of an existing subtree to show that the assertion it represents has been proved. On block entry we push a special marker on the stack, and on block exit we pop items off the stack up to the marker, deleting and replacing as needed. This method is fairly crude, since if a node is updated several times in a single block, several usless copies of it will be pushed on the stack, but it seems to be simple and effective.

3. Automatic Deductions

The interest of the AT lies almost entirely in the automatic deductions it can perform from previously proved assertions. These deductions fall into three general classifications:

1) propositional reasoning
2) associativity and commutativity
3) equality reasoning

Below we first discuss each classification individually, giving both the theoretical and practical constraints and outlining the actual implementation. We then discuss how automatic deductions from the different classes can be combined into a single automatic step. Note that for each class it will turn out that any number of automatic rules from that class can be combined into one automatic deduction.

3.1. Propositional Reasoning

3.1.1 Theoretical Constraints

Given an assertion in propositional logic, it is decidable whether or not that assertion follows from some group of assertions. Unfortunately, however, this problem can easily be shown to be NP-complete. Thus the best known algorithms to solve it take asymptotically exponential time to run. On a practical level there does not seem to be any straightforward algorithm which has good behavior in all but an identifiable and unlikely set of cases. Since we cannot allow automatic deductions involving the full set of propositional inference rules, we must decide what subset we can allow.

Within the natural deduction system of propositional logic we have constructed a polynomial time algorithm which automatically performs all propositional inferences but OR elimination, IMPLICATION introduction, and NOT introduction. The following rules, then, can be used to automatically deduce an assertion from assertions in the table:

AND introduction and elimination
OR introduction
IMPLICATION elimination
Two special cases of IMPLICATION introduction:
a) True conclusion
b) False hypothesis
EQUIVALENCE introduction and elimination
NOT elimination
FALSE introduction and elimination

3.1.2. Implementation

In our implementation we rely heavily on the normal form theorem for deductions. (For further discussion and a proof of the theorem see Prawitz [4]). We will make use of the following consequence of the theorem.

Given that an assertion Q can be deduced from a set of premises by an application of a sequence of atomic inferences $R=\{R_1, R_2, \ldots, R_n\}$, Q can also be deduced from the set of premises by a reordering of R having the form:

Eliminations from R
Introductions from R
An IMPLICATION elimination from R
Eliminations from R
Introductions from R
An IMPLICATION elimination from R
etc...

This is exactly how we proceed in our automatic propositional reasoning.

When a proposition P is initially asserted by ASSERT, we perform AND and EQUIVALENCE elimination on P. We then go through a list of all implications which have not yet been eliminated and query their hypotheses. If the query returns true, we recursively assert the conclusion and remove the implication from the list. In QUERY we perform AND, OR, limited IMPLICATION, and EQUIVALENCE introduction. NOT elimination (FALSE introduction) is handled as a subcase of IMPLICATION elimination. Every time $\neg P$ is asserted, we put P => False on the list of undischarged implications. When P becomes automatically deducible, False will be asserted. FALSE elimination is done in QUERY by keeping a flag indicating whether or not FALSE occurs in the table.

The actual processing of AND and EQUIVALENCE elimination in ASSERT is trivial. All the propositions we can infer are simply recursively asserted. For example, given the expression P & Q, ASSERT recursively asserts both P and Q.

The processing of the introductions in QUERY is equally simple. We just check for the existence of the necessary propositions. For example, if QUERY is given the expression P|Q|R, it recursively queries P, Q, and R and returns true if any one of them is true. For IMPLICATION introduction, if we are trying to introduce P => Q, we query Q and ¬P. If either is true, then the introduction is automatic.

The normal form theorem guarantees that the process described above will allow us to deduce in a single step the result of an arbitrary sequence of the propositional inference rules listed in section 3.1.1. The order in which these rules are actually applied will become significant in later sections when we consider the interaction of automatic propositional reasoning with the other classes of automatic deductions.

3.1.3. Some Examples

In the table below we give several examples of automatic propositional deductions possible with this system.

DEDUCTION	RULES
P & Q	AND elimination
R	AND introduction
————	
Q & R	
P => (Q & R)	IMPLICATION elimination
P	AND elimination
————————	IMPLICATION introduction
¬Q => T	
(Q | R) => T	IMPLICATION elimination twice
P => ¬T	OR introduction
R => P	IMPLICATION elimination
R	FALSE introduction
————————	
'0'B	

3.2 Associativity and Commutativity

Implementing general associativity and commutativity is a fairly simple task. The problems, which will be discussed in later sections, occur when we attempt to combine it with the other automatic deduction rules. Basically, the data structure described in section 2.1.2 is the solution to the problem of implementation. As assertions are processed by ASSERT, they are inserted into the AC table in AC normalized form. When we wish to determine if an assertion being processed by QUERY can be deduced from existing assertions by general associativity and/or commutativity, we simply put the assertion into AC normalized form and see if the result exists in the AC table.

For the purposes of this system we consider the associative and commutative operators to be as follows:

Associative: + * & | ||

Commutative: + * & | = <=>

The problems of overflow and string truncation have been ignored.

Below we give examples of automatic deductions involving associativity and commutativity.

$$\frac{((X+5+Y)*Z)*W > 100}{Z*(W*(5+X+Y)) > 100}$$

$$\frac{(S \mid ((P \Rightarrow Q) \mid R)) \Rightarrow T}{(S \mid R \mid (P \Rightarrow Q)) \Rightarrow T}$$

3.3. Equality Reasoning

There are four equality rules which the system applies automatically. These are reflexivity, symmetry, transitivity, and equality substitution. We describe below the system designed to solve the most difficult of the problems, equality substitution. As we will see, this solution also provides an immediate and simple solution to the other three problems.

We should note that the equivalence operator (<=>) on boolean expressions is taken to imply that the two expressions are equal. Thus if P <=> Q, then P = Q, and all the automatic equality rules can be applied.

General equality substitution allows us to make the following type of deduction automatically:

$$P(t1)$$
$$t1 = t2$$

Here P(t) is an arbitrary assertion involving a term t. As usual, we can combine any number of steps of this type into one automatic deduction.

3.3.1. Theoretical Basis

The problem of general equality substitution given here is equivalent to the uniform word problem for finitely presented algebras discussed in [4]. In our implementation we make use of a modified version of the algorithm given in [4] as part of the proof that this word problem is polynomial time complete. As actually implemented, our algorithm has worst-case behavior n-squared. While there exist versions of the algorithm whose worst-case behavior is only n*((log n)-squared), these involve sufficient overhead that their average case behavior is much worse than that of the current implementation.

3.3.2 Implementation

We implement automatic equality substitution by means of the dag described in section 2.1.1. The basic idea is as follows. Any time two terms in the dag are asserted to be equal, we combine them into a single equivalence class, choosing one of them to be a "representative" of the class. Anywhere that either of the terms shows up as an operand, we replace that term by the representative of the new equivalence class. Now any time we look up an assertion in the dag, we always use the representative for the equivalence class containing any subterm we encounter. In most cases, this representative will simply be the term itself, but if that term was involved in an equality assertion, then the representative will be whatever term was chosen as the represnetative when the equivalence class was created. Note that we may have to follow a chain of representative pointers through the dag until we find the representative of the largest containing equivalence class. Such chains are created when we form the union of preexisting equivalence classes.

We should observe here that the directed graph resulting from the above operation may no longer be acyclic. In particular, asserting the equality X = X*1 may result in the * node referencing itself as its own left operand. This means that our equivalence classes are actually infinite. Using the above example, we could automatically deduce X = (X*1)*1, X = ((X*1)*1)*1, and so on. It turns out, however, that the introduction of cycles of this type does not adversely affect the overall algorithm.

There is one further complication which can occur in the process described above. When we replace operands by their representatives, we may create two expression subtrees in the dag for the same expression. This violates one of the constraints placed on the dag when it was defined. We resolve this problem by recursivley asserting the equivalence of the two terms involved, choosing one of them to be the representative, and proceeding as described above.

In figures 3 and 4 we illustrate the effect on the dag of equality assertions. In figure 3 we have made the following assertions:

```
P => ((X > Y) | (Z > W))
Q => ((X > Y) | (T > W))
```

Now we assert Z=T (or if Z and T are booleans, we could assert Z<=>T). Observe that this implies that the two expression subtrees for Z>W and T>W now represent the same expression. Thus we must recursively assert their equality. The final resulting dag is shown in figure 4. Dotted lines represent pointers to representatives of equivalence classes. In the figure we have chosen Z as the representative of the class {Z, T} and Z>W as the representative of the class {Z>W, T>W}. Note that operator/operand pointers have been moved from the original operand to the representative.

The equivalence class representation described above maintains as its partition of the assertions the reflexive transitive closure of all assertions in the table. Thus to determine if two assertions are equal through reflexivity, symmetry, or transitivity of previously asserted equalities, we need only look them both up in the dag and see if they have the same representative. If so, then we can deduce equality, and if not, then we cannot.

3.3.3. Some Examples

Below we give several examples of automatic deductions involving the equality rules.

```
       X = 5 * Y + 3
         Z = X + 5
       -------------
     Z = 5 * Y + 3 + 5


((P | Q) => T) <=> (S & R)
     ((S & R) | Q) => V
----------------------------
(((P | Q) => T) | Q) => V
```

$$\frac{\begin{array}{c} X * 2 = Y + 8 \\ X * 2 = Z - 3 \\ (Z - 3) * 22 > 56 \end{array}}{(Y + 8) * 22 > 56}$$

In the example below A and B are FIXED arrays.

$$\frac{\begin{array}{c} \text{ALL I FIXED WHERE I>0 \& I<5, A(I) > B(I+1)} \\ \text{A=B} \end{array}}{\text{ALL J FIXED WHERE J>0 \& J<5, A(J) > A(J+1)}}$$

3.4. Combining Classes of Automatic Deductions

3.4.1. Propositional Reasoning and Associativity/Commutativity

Recall from section 3.1.2 the implementation of automatic propositional reasoning. Eliminations were performed in ASSERT and introductions in QUERY. Automatic associativity/commutativity (A/C) processing is performed only between the time a subexpression is actually asserted, i.e. after eliminations, and the time a subexpression is queried, i.e. before introductio: Thus, ignoring implication eliminations for the moment, the pattern is {Eliminations, A/C, Introductions}.

As we saw before, implication eliminations complicate the processing of deductions. If implication elimination is included in our deduction, we can get an extra round of introductions to establish the implication hypothesis and elimination: made possible by the conclusion. Thus the general form of legitimate combined deductions in regular expression format is:

{Eliminations, {A/C for subexpressions needed by introductions for the IMPLICATION elimination hypothesis, Introductions for the IMPLICATION elimination hypothesis, IMPLICATION elimination, Eliminations from IMPLICATION elimination conclusion}*, A/C, Introductions}.

(The star (*) represents zero or more repetitions of the subpart it follows.) This pattern will show up frequently in combined rules.

It is perhaps easier to remember the following special case of the above formula: {A/C, Propositional Reasoning}. We can use this form because there is no interference between the eliminations and the associativity/commutativity.

We give below an example of a deduction combining A/C and propositional reasoning which cannot be done automatically by the current system. The reason is that we cannot perform A/C on the result of an introduction.

$$\frac{P \mid Q}{P \mid R \mid Q}$$

The deduction below, however, which does not use A/C, is automatic (note that the parentheses are essential).

$$\frac{P \mid Q}{(P \mid Q) \mid R}$$

3.4.2. Propositional Reasoning and Equality

The situation here is almost exactly the same as in the case of associativity/commutativity described above. Once again the special processing for equality is done after eliminations and before introductions. With equality, however, eliminations do affect equality special processing. The simple special case, then, is {Eliminations, Equality, Introductions}. The general form of legitimate combined deductions is:

{Eliminations, {Equality, Introductions for IMPLICATION elimination hypothesis, IMPLICATION elimination, Eliminations from IMPLICATION elimination conclusion}*, Equality, Introductions}

3.4.3. Equality and Associativity/Commutativity

This is where things get interesting. The fact that the mechanisms by which automatic deductions involving these two classes are performed are completely separate might well give rise to suspicions that things are not going to be as easy here. These suspicions will be amply justified. We consider below several subparts of this problem and discuss the pertinent theoretical results.

3.4.3.1. Equality and Concatenation

The problem of automatic equality reasoning combined with automatic associativity of concatenation is just the uniform word problem for semigroups. Given an algorithm which solves all instances of this problem, we can solve the halting problem for arbitrary Turing machines as follows. Concatenated strings of symbols will represent instantaneous descriptions of a valid computation of the Turing machine. We use equality rules to specify the changes to a three symbol string whose center symbol is marked to represent the current position of the machine head. The allowed changes would correspond to legitimate transitions in the machine. Now we simply ask if the string representing the start state of the Tm equals the string representing the final state. If so, then there is a valid computation of the Tm from the start state to the final state, and the Tm will halt. If not, then the Tm doesn't halt.

We conclude that no algorithm solving all instances of the uniform semigroup word problem can exist, i.e. the problem is undecidable. Thus we are not going to be able to find an algorithm which will allow us to perform arbitrary automatic deductions involving both equality and associativity of concatenation.

3.4.3.2. Equality and Multiplication

Here we have the uniform word problem for commutative semigroups. This problem was recently shown to be exponential-space complete ([5]). Thus, although the problem is decidable, there is no feasible algorithm to solve it in general. It is fairly safe to say that there will never be a verification system which will automatically deduce every application of equality substitution and associativity and commutativity of multiplication.

3.4.3.3. What the AT Actually Does

Given problems like those described above, it is easy to see that the AT is not going to be able to do very much. Thus, in the current system it is not possible to combine equality and A/C except in one very special case. Each time we are about to perform an implication elimination, the hypothesis can be deduced by either equality or A/C. Thus if the sequence of automatic rule applications involved two implication eliminations, we could do the introductions for the hypothesis of one by A/C and the other by equality. (Section 3.4.4 below describes this situation in much greater detail). This is the only time that we can combine the two classes in one automatic deduction.

3.4.4. The General Rule for Allowable Automatic Deductions

The expression below summarizes the general rule. It is in regular expression format, where + indicates a choice among alternatives:

{Eliminations,{{Equality + A/C} for subexpressions needed by introductions for the IMPLICATION elimination hypothesis, Introductions for the IMPLICATION eliminations hypothesis, IMPLICATION elimination, Eliminations from the IMPLICATION elimination conclusion}*, {Equality + A/C}, Introductions}

It will probably avoid confusion for the reader of the program proof if only one class of automatic deduction is used in any automatic step unless the deduction is trivially obvious. Be sensible about your use of the automatic deduction facility.

3.4.5. Some Examples

Below we give examples illustrating some of the automatic deductions possible with this system. To the right of the deduction we list the classes involved.

```
(X > 3*Y) => Q                     Propositional Reasoning
X > Z+5 & Z+5 = 3*Y                Equality
-------------------
```

```
(P | Q | R) => X+5 > 13*Z              All Classes
(X+5 > W*2) => S                       (legitimately combined)
Y = W*2
13*Z = Y
P | R
-----------------------------
-S => T
```

```
(Q <=> R) => T                         Propositional Reasoning
Q & R
-----------------
T
```

```
3+Y >= 0 => Z < 8*W                    All Classes
X > 5 & Y+3 >= 0                       (legitimately combined)
W*S = 24
24 > Z => -(X > 5)
----------------------
'0'B
```

4. Future Directions

There are several ways in which the automatic deduction facilities of the assertion table could be expanded, and there are some absolute limitations as well. For example, it will never be possible to arbitrarily combine automatic deductions of the equality and associativity classes. Several areas for possible extensions include automatic introductions and eliminations of the universal and existential quantifiers, automatic arithmetic rules, and an improvement in the relationship between propositional reasoning and the other classes of automatic deductions.

When this system was being designed, we very seriously considered making available automatic deductions involving ALL elimination and SOME introduction. Both of these deductions could be done by using a pattern match facility that attempted to match the assertion being deduced against previous assertions. For example, in SOME introduction we would search for an existing assertion matching the one to be introduced where the bound variable in the new assertion could match any single term in the previous assertion. We decided at the time that it would be too difficult to implement this facility together with the others we actually implemented. A careful study of the problem, however, might well reveal some reasonably efficient implementation, making this facility a desirable addition.

Another desirable addition to the current system would be some form of automatic deduction for simple arithmetic. Certainly some kind of constant folding facility, allowing the system to automatically deduce, for example, 7 > 5, would be useful and straightforward to implement. At the moment, the user must invoke the arithmetic package for any arithmetic deduction at all. There might be some interesting problems involved in meshing an automatic arithmetic subsystem in with the existing classes of automatic deduction, particularly equality. This facility will probably be the next one added to the existing system.

Finally, we might wish to consider the problem of improving the relationship between propositional reasoning and both equality and associativity/commutativity. There may well be theoretical bounds on doing this efficiently, and in any case it is almost certain to be a messy problem. It would be very desirable, however, to be able to improve on the currently grotesque form of the general rule for combining automatic deductions.

The question of future directions is both interesting and important. Program verifiers will only become practical when they can provide sufficient automatic facilities so that writing the proof is little if any more difficult than writing the program itself. The individual who designs such a system will have made an indelible mark on the science of computing.

Acknowledgements: The author is indebted to R.L. Constable, S.D. Johnson, and M.J. O'Donnell for many helpful suggestions.

FIGURES

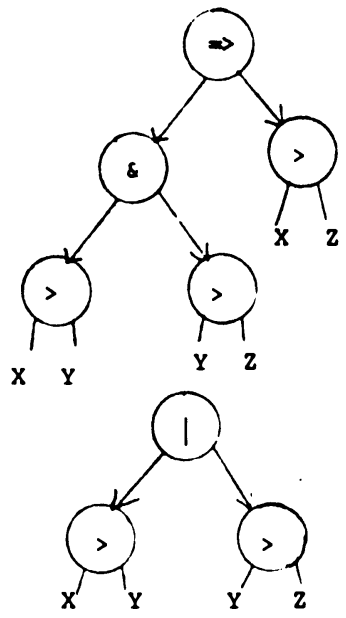

FIGURE 1

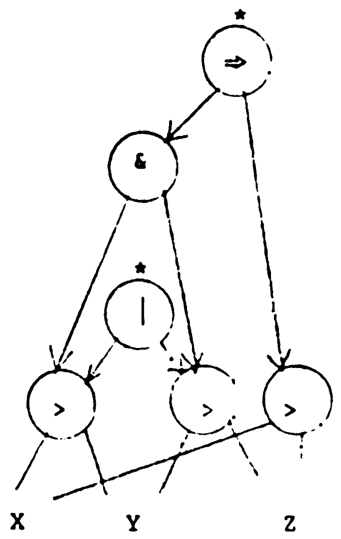

FIGURE 2

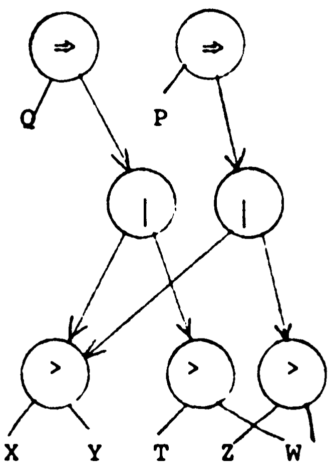

FIGURE 3

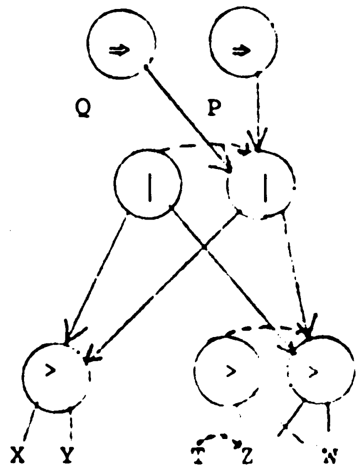

FIGURE 4

REFERENCES

[1] Constable, Robert L., Scott D. Johnson. PL/CV2 Program Verifier Reference Manual, Cornell Computer Science Department, March 1978.

[2] Krafft, Dean B. Assertion Table System Program Manual, Cornell Computer Science Department, to be published.

[3] Prawitz, D. Natural Deduction, Almqvist & Wiksell, Stockholm, Sweden, 1965.

[4] Kozen, Dexter. Ph.D Thesis, Cornell University Department of Computer Science, May 1977.

[5] Cardoza, E., R. Lipton, A.R. Meyer. "Exponential-Space Complete Problems for Petri Nets and Commutative Semi-groups", Proceedings of the Eighth Annual ACM Symposium on the Theory of Computing, ACM, New York, May 1976.

[6] Goto, E. "Monocopy and Associative Algorithms in Extended Lisp", University of Tokyo, Japan, May 1974.