Switching and Programming

A. VAN WIJNGAARDEN
University of Amsterdam

MR 50

In switching theory, much attention has been paid to the analysis and simplification of circuits and systems and to properties of networks. The objective has been to provide network structures using rather simple components (e.g., relays and diodes in series or parallel). In the programs for automatic computers, similar structures are found, although on another scale. These programs consist of sequences of statements performing certain operations and connected by transfers of control into a complicated network. Executing the statements means moving along the paths of the network and setting conditions (assigning values to Boolean expressions to determine which transfers shall take place, i.e., which route to take at each node of the net). Just as in the case of switching circuits, seemingly completely different structures may be more or less the same functionally, and the problem of simplification arises immediately.

Of course, there is the question which of two equivalent programs is the simplest. There will be little doubt that replacing $x \vee \sim x$ by **true** constitutes a simplification, but in more complicated cases the answer to this question depends on the type of logical building blocks assumed to be available.

Measuring the complexity of a switching circuit in diodes or relays in some prescribed way produces a norm for comparison of different forms, although other criteria, such as equal loading of elements, may prevail under certain circumstances. In programs one would normally use criteria of length or duration, although other criteria, such as generality or provision for easier ways of checking, might be important. Or some building blocks of more complex structure may be available at the same cost as more elementary blocks, simply because they have already been made. Thus, availability of a five-input AND-circuit at the same cost as two single diodes may change the optimum form of a circuit considerably. Also, if a procedure is available to compute a function of two arguments, then it may be advantageous to use that procedure, even if the second argument is always zero, rather than to write a new and simpler procedure. In the following we shall mainly use the length of the text as the criterion for simplicity.

Of course the structure of a program is much more complicated than that of a binary switching circuit, and here we can only point out some problems, instead of giving definite rules for simplifying programs. But even a minimal attempt such as this can be justified, because the subject is becoming more and more interesting from the standpoint of automatic programming.

Reprinted from SWITCHING THEORY IN SPACE TECHNOLOGY edited by Aiken and Main.
Published by Stanford University Press.

As our language for expressing the program elements, we choose ALGOL 60 [2]. The generality of its expressions permits us to deal concisely with general situations, and guarantees absolute machine independence. Moreover, we can give a precise meaning to some concepts that are difficult to describe in any other way.

1. Simplification of Identities

If we take the length of the text literally as the criterion of simplicity of a program, we can simplify many programs directly by replacing long identifiers with short ones. This is true but so obvious that we shall assume it has been done already, and we can then focus on those parts of the text which convey information.

It is natural to start by investigating the simplification of Boolean expressions that may occur in the program. These may occur as constituents of more complicated Boolean expressions, such as the right-hand side of an assignment statement whose left-hand side contains only Boolean variables, in an if clause, or as actual parameters in a procedure statement or function designator. For instance, the relation $x = y$ might occur in $x = y \vee y > z$; $b := x = y$; **if** $x = y$ **then** u **else** v; $P(u, v, x = y)$.

As long as these Boolean expressions contain only logical values, logical operators, and identifiers of variables of type Boolean, we can apply all the techniques that have been developed in logic or switching theory for their simplification. When they contain relations, however, the situation demands a more careful investigation.

Let us start with the simplest possible expression containing a relation of that type, namely the relation $x = x$, and ask whether this might be simplified by replacing it with **true**. This now holds true only under certain restrictions on the meaning and interpretation of x. First of all, the truth of $x = x$ does not by itself imply the truth of $y = y$, since properties associated with the letter x are not necessarily associated with the letter y.

Let us then describe more carefully what we want to know by asking whether an identity, as defined by

$$\langle \textit{identity} \rangle ::= \langle \textit{expression 1} \rangle = \langle \textit{expression 1} \rangle ,$$

can be replaced by **true**. Here, as in the ALGOL 60 report [2], sequences of characters in the bracket $\langle\ \rangle$ represent metalinguistic variables whose values are sequences of symbols. The extension is that of the numbered metalinguistic variable, namely

$$\langle \langle \textit{letterstring} \rangle - \langle \textit{unsigned integer} \rangle \rangle ,$$

representing a metalinguistic variable, which occurs in metalinguistic formulas and which is denoted by the same letterstring in the brackets $\langle\ \rangle$ but without the minus sign and the unsigned integer (under the condition, however, that whenever in a metalinguistic formula the numbered metalinguistic variable occurs more than once, it stands for the same [albeit arbitrary] value of the

corresponding metalinguistic variable). Examples of an identity are then

$$x = x, \quad y = y, \quad x + y = x + y, \quad read = read\,.$$

Such an identity cannot be replaced by **true** without more information. Consider, for example, the following partial program:

```
begin procedure  P(t); string t; ⟨code expressing that the basic symbols
                 of the string without 'and' are printed⟩;
integer procedure  Q(t); string t; ⟨code expressing that the number of
                 basic symbols of the string without 'and' is the value
                 of Q⟩;
if Q('x = x') > 2 then P('x = x')
end .
```

This ought to print: $x = x$, but if $x = x$ were replaced by **true** under P, it would print **true** instead, and if in addition $x = x$ were replaced by **true** under Q, it would print nothing at all.

Of course, when an expression occurs inside a string, it is usually not its value but the sequence of its constituent basic symbols that is important.

Let us then describe more carefully what we want to know by asking whether an identity not occurring inside a string can be replaced by **true**.

This is certainly not the case in general. Indeed, if the expressions occurring within the identity are Boolean expressions or designational expressions, the identity itself is not an expression, since the only meaningful occurrence of the relational operator $=$ is that in a relation, defined by

$$\langle relation\rangle ::= \langle arithmetic\ expression\rangle\langle relational\ operator\rangle \langle arithmetic\ expression\rangle\,,$$

which is a special case of a Boolean expression. In the cases mentioned above, the value of the identity is therefore undefined and hence it cannot be replaced by **true** without more information. Let us introduce the concept of a proper identity by

$$\langle proper\ identity\rangle ::= \langle arithmetic\ expression\ -1\rangle = \langle arithmetic\ expression\ -1\rangle\,.$$

Then we can describe more carefully what we want to know by asking whether a proper identity not occurring inside a string can be replaced by **true**.

This is not the case in general. One has to realize that the expression may contain a function designator, and that in the evaluation of that function designator a **go to** statement may be executed that defines its successor as a statement outside the procedure body so that the evaluation of the expression remains unfinished forever. For instance, the partial program

```
begin integer k;
    integer procedure x; if k < 0 then go to end else x := 1;
        procedure print (t); integer t; ⟨code expressing that the value
                of t is printed⟩; k := −1; if x = x then k := 1;
    end: print (k)
end
```

ought to print -1, but if $x = x$ were replaced by **true**, it would print 1.

Let us call an expression evaluable if it contains no function designator whose procedure declaration includes a **go to** statement leading out therefrom. Then we can describe more carefully what we want to know by asking whether a proper identity that does not occur inside a string and that compares two identical evaluable expressions can be replaced by **true**.

Again, however, this is not the case in general. The arithmetic expression may contain a function designator whose corresponding procedure depends for its operation on values which are altered by its own evaluation. For instance, the following useful procedure is of that type:

integer procedure *altsign* 1 (*b*); **Boolean** *b*;
begin *altsign* 1 := **if** *b* **then** 1 **else** −1; *b* := ∼*b* **end** .

Its value is in turn equal to 1 and to −1 if at least the value of the actual variable corresponding to *b* (say, *B*) is not changed between two occurrences of its identifier. In this example the variable *b* is the cause of this abnormal behavior, and one might think that by assessing the effect of the procedure upon its formal parameters, one could see whether the function designator is of this class. However, the variable *b* does not need to appear in the parameter list at all. It may be a non-local variable, as in the following procedure:

integer procedure *altsign* 2;
begin *altsign* 2 := **if** *b* **then** 1 **else** −1; *b* := ∼*b* **end** .

Or it may be an own local variable, as in the following:

integer procedure *altsign* 3;
begin own Boolean *b*;
altsign 3 := **if** *b* **then** 1 **else** −1; *b* := ∼*b*
end .

If we want a sequence of results to start with 1, the sequence of occurrences of the function designator may be preceded by *B* := **true** or *b* := **true** or **if** *altsign* 3 = 1 **then** *altsign* 3.

In all these cases we might still find out what the procedure does by inspecting its body. This would, however, present difficulty in a case like

integer procedure *altsign* 4; ⟨*code*⟩ ,

in which the body is expressed in some non-ALGOL language, although for its action *altsign* 4 might be equivalent to *altsign* 3.

Even if we were able to interpret the code, there might be a last source of values upon which the value of the function designator would depend, some outside source of information. For instance, *read* might be a function designator whose value is that of the next number read from a tape; this value would of course change after each reading of the tape.

An extreme case is provided at last by the function designator *random*, whose value is by definition unpredictable.

It is obvious that if these so-called function designators with side effects occur, in general our identity cannot be replaced by **true**. It must be emphasized that checking for the absence of side effects is often very difficult and can sometimes only be done during run time, so that perhaps the easiest way to find out whether or not the identity can be replaced by **true** is to evaluate it. Obviously, this does not help very much.

There is, however, an important class of expressions compared by an identity that can be replaced by **true**; these are to be called stable expressions, and are all those expressions whose value does not depend on the order of evaluation, provided they occur within one basic statement. These expressions include numbers, logical values, simple variables, subscripted variables if the subscript expressions are stable expressions, relations whose arithmetic expressions are stable expressions, and all expressions that can be generated with those as primaries, i.e., all expressions that can be generated without using a function designator. This subset of stable expressions is easily recognizable in the text. It is, however, unduly limited. We can distinguish in a procedure four categories of parameters: input parameters, output parameters, dummy parameters, and mixed parameters.

Input parameters are those non-local identifiers, and the actual parameters corresponding to those formal parameters, which appear only (1) in the value list or within the procedure body as variables in the right-hand expression of assignment statements, (2) in subscript expressions not within a function designator, or (3) as input parameters of function designators and procedure statements.

Output parameters are those non-local identifiers, and the actual parameters corresponding to those formal parameters, which occur only within the procedure body in the left-hand list of assignment statements or as output parameters of function designators and procedure statements.

Dummy parameters are those non-local identifiers, and the actual parameters corresponding to those formal parameters, which either do not appear in the value list and procedure body at all or appear only as designational expressions, procedure identifiers, or dummy parameters of function designators and procedure statements.

Mixed parameters are those non-local identifiers, and the actual parameters corresponding to those formal parameters, which do not belong to one of the three categories mentioned above.

If the function designators in an expression have no output and no mixed parameters, then the expression is stable. Also, if within a basic statement its output parameters and the two sets of the input parameters and the output parameters of itself and all function designators contained within it, and the mixed parameters one-by-one are disjunct, then all expressions contained within it are stable. Obviously, in this order it becomes more difficult but still possible to check whether the conditions are satisfied.

We can now define more carefully what we want to know by asking whether a proper identity that does not occur inside a string and that compares two identical, evaluable, stable expressions can be replaced by **true**.

There is still some difficulty in answering yes to this question. If the arithmetic expressions in the identity are of type **integer**, then the answer is yes. If they are of type **real**, however, it is a question of interpretation. Indeed the ALGOL 60 report [2] gives the following definition of the arithmetic of real quantities:

> Numbers and variables of type **real** must be interpreted in the sense of numerical analysis, i.e., as entities defined inherently with only a finite accuracy. Similarly, the possibility of the occurrence of a finite deviation from the mathematically defined result in any arithmetic expression is explicitly understood. No exact arithmetic will be specified, however, and it is indeed understood that different hardware representations may evaluate arithmetic expressions differently.

First of all, we might evaluate the left-hand expression by means of some hardware representation other than that of the right-hand side, since in a large computing system one might very well take advantage of the evaluability and stability of the expressions and evaluate them simultaneously on different component computers, which might work with different degrees of precision. But even if this is not the case, one might interpret the freedom specified above as including the possibility that performing the same computation twice, or even calling the same variable twice, might yield different results, or that the comparison of two values is an operation involving arithmetic and hence subject to inaccuracy (possibilities that are not excluded by the wording). Actually computation on an analog computer has these features. We might even write a program like the following:

```
begin integer i, j; real x; j := 0;
      for i := 1 step 1 until 1000 do
      begin x := random; if x = x then j := j + 1 end;
      for i := 0 step 1 until 1000 − j do ⟨computation⟩
end .
```

The assumption is made that a computation is repeated a number of times depending upon the quality of the computer used, which is determined by the program itself. This program would be spoiled completely if $x = x$ were replaced by **true**.

If digital computation is understood, we have

(i) if $E1$ and $E2$ are results of two identical computations, then $E1 = E2$;

(ii) a variable is a quantity that does not vary unless another value is assigned to it.

After these preliminaries, we can state that a proper identity that does not occur inside a string and that compares two identical evaluable stable expressions of type **integer**—or of type **real** if digital computation is assumed—can be simplified to **true**.

2. *Simplification of Relations*

Before we go on we shall assume, in order to avoid the same cumbersome wording over and over again, that from now on we are dealing with expressions

not occurring inside a string; that they are proper, evaluable, and stable; and that digital arithmetic is understood. We shall denote arithmetic expressions by $e1$, $e2$, $e3$.

Thus far we have dealt only with the relation operator $=$ and now we widen our relations by introducing the operators $<$ and $>$. If $e1 = e2$, can $e1 < e2$ be simplified to **false**? This cannot be said without more information. Actually, in one well-known computer system, both the relation $-0.0 = 0.0$ and the relation $-0.0 < 0.0$ hold. Disregarding those slips, we define as proper arithmetic an arithmetic in which the set of real numbers—i.e., numbers of type **real** or type **integer**—is ordered, so that exactly one of the relations $e1 < e2$, $e1 = e2$, $e1 > e2$ holds. This does not imply that $-0.0 = 0.0$, but it excludes such a case as the one mentioned above. We assume the arithmetic to be proper, and postulate the following axioms and definitions:

A1: $e1 = e1$,
A2: $+e1 = e1$,
A3: $(e1) = e1$,
A4: $e1 = e2 \equiv e2 = e1$,
A5: $e1 \neq e2 \equiv \sim e1 = e2$,
A6: $e1 \neq e2 \equiv e1 > e2 \vee e1 < e2$,
A7: $\sim(e1 > e2 \wedge e1 < e2)$,
A8: $e1 > e2 \equiv e2 < e1$,
A9: $e1 \geqq e2 \equiv \sim e1 < e2$,
A10: $e1 \leqq e2 \equiv \sim e1 > e2$,
A11: $e1 > e2 \wedge e2 > e3 \supset e1 > e3$.

This is as far as one can go without specifying the type of the expressions. One might think, for instance, that

$$e1 = e2 \wedge e2 > e3 \supset e1 > e3$$

might hold. This is certainly not the case. Indeed there is no objection to an arithmetic in which a relation like $er1 = ei1$ can hold. Here the letters r or i specify the type of the expression to be **real** or **integer**, respectively. For instance, $1.00_{10}3 = 1001$ and $1.00_{10}3 = 1002$ could very well hold and, since integers are dealt with exactly, certainly $1002 > 1001$. One would, however, find $1.00_{10}3 = 1002 \wedge 1002 > 1001 \supset 1.00_{10}3 > 1001$ against the supposition.

Actually the formulas with one or two equals signs run as follows:

A12: $e1 = er2 \wedge er2 > e3 \supset e1 > e3$,
A13: $e1 = er2 \wedge er2 < e3 \supset e1 < e3$,
A14: $e1 = er2 \wedge er2 = er3 \supset e1 = er3$,
A15: $e1 > ei2 \wedge ei2 = ei3 \supset e1 > ei3$,
A16: $e1 < ei2 \wedge ei2 = ei3 \supset e1 < ei3$,
A17: $e1 = ei2 \wedge ei2 = ei3 \supset e1 = ei3$.

This system enables us to simplify relations. For instance, if x, y, z are of type **real** and i and j of type **integer**, then

$$x < y \wedge (y > x \vee y > z) \equiv x < y ,$$
$$x < y \wedge y < z \wedge z = x \equiv \textbf{false} ,$$

but $x = i \wedge x = j \wedge i > j$ cannot be simplified.

3. *Simplification of Relations Containing Arithmetical Operations*

The next step consists of introducing actual arithmetic operations. Again, proper arithmetic is supposed to satisfy a set of axioms, some of which are

$$e1 + e2 = e2 + e1 ,$$
$$e1 > e2 \supset e3 + e1 \geqq e3 + e2 ,$$
$$e3 + e1 > e3 + e2 \supset e1 > e2 .$$

These three axioms can be used to simplify, for instance,

if $x + y > y + z$ **then** (**if** $x > z$ **then** 1 **else** 2)
else if $x > z$ **then** 3 **else** 4

into

if $x + y > y + z$ **then** 1 **else if** $x > z$ **then** 3 **else** 4.

4. *Simplification of Statements*

Next we turn our attention to syntactical units of a higher level than expressions, i.e., to statements.

The separator $:=$ which is actually partly an arithmetic operator deserves special attention. Indeed there is no reason to assume that the evaluation of expressions of type **real** is performed with the same precision as that with which the results are remembered. Usually the expressions are evaluated in higher precision than that of the variables, so that the assignment usually includes a round-off (or chopping in more primitive machines).

This means that the sequence

$$x := y \times z; \quad i := \textbf{if } x = y \times z \textbf{ then } 1 \textbf{ else } 2$$

cannot be simplified into

$$x := y \times z; \qquad i := 1.$$

On the contrary, $i := 2$ is a much better guess.

However, proper arithmetic is supposed to include the requirement that assignment be well defined and idempotent, so that when $v1$ and $v2$ are variables of the same type, the two axioms hold true:

A18: $v1 := v2 := e1 \supset v1 = v2$,
A19: $v1 := v2 \supset v1 = v2$.

They enable us, for instance, to simplify

$$x := y := z \times z; \quad i := \textbf{if } x = y \textbf{ then } 1 \textbf{ else } 2$$

into

$$x := y := z \times z; \quad i := 1 .$$

5. *Simplification of Procedures*

At last we turn our attention to the bigger portions of a program, such as procedures. Obviously the possible gain by simplification is greatest here, but it seems difficult to point out any substantial recognizable points of attack. Although something can be said about passing on parameters from one procedure to the next, which is again a somewhat simply ordered structure, really important simplification is dealt with by mathematical rather than logical methods. Also, the requirement of absolute equality of results may become academic. Also, it is not obvious whether it is improper to treat the assignment to the procedure identifier of a function designator inside the procedure body like assignment to a variable, i.e., including a possible loss of precision. In many classical programming schemes the function designator would be handled by means of a subroutine that included round-off. On the other hand, in the ALGOL 60 compiler made by E. W. Dijkstra and J. A. Zonneveld for the ELECTROLOGICA X1 computer [1] the function designator does not include the loss of precision. However, in neither case is this logically conditioned, and one may have subroutines that compute a double-length result or modern procedures that round off, but it does make a difference. Consider, for instance, the following two procedures for computing the sum for k from a to b of the expression fk:

```
real procedure sum (k, a, b, fk); value b;
                integer k, a, b; real fk;
    begin       real s;
                s := 0; k := a;
            L:  if k ≦ b then begin s := s + fk; k := k + 1; go to
                L end;
    end         sum := s
```

and

```
real procedure sum (k, a, b, fk); value a, b;
                integer k, a, b; real fk;
                if a ≦ b then begin k := a; sum := fk + sum (k, a + 1,
                b, fk) end
                else sum := 0.
```

Formally, we ought to say that the first procedure cannot be simplified into the second one, since the arithmetic result is not necessarily equal. But since this difference would presumably also be to the advantage of the second procedure, it can hardly be said to be a reasonable objection.

References

[1] Dijkstra, E. W., Ein ALGOL 60 Uebersetzer für die X1, *Mathematik-Technik-Wirtschaft*, 1961, **8**(2, 3).

[2] Naur, P., ed., Report on the algorithmic language ALGOL 60, *Numerische Mathematik*, 1960, **2**, 106–37; *Acta Polytechnica Scandinavica* (Math. and Comp. Mach. Ser. No. 5), No. 284, 1960. *Comm. Assoc. Comp. Mach.*, 1960, **3**, 299–314.