State Constraints and Pathwise Decomposition of Programs

J. C. Huang
Department of Computer Science
University of Houston
Houston, Texas 77204-3475

ABSTRACT - A state constraint is a new programming construct designed to restrict the domain of definition of a program. It can be used to decompose a program pathwise, i.e., to divide the program into subprograms along the control flow, as opposed to divide the program across the control flow when the program is decomposed into functions and procedures. As the result one can now construct and manipulate a program consisting of one or more execution paths of another program. This paper describes the idea involved, examines the properties of state constraints, establishes a formal basis for pathwise decomposition, and discusses their utilities in program simplification, testing, and verification.

Index terms: state constraints, pathwise decomposition, program analysis, program testing, program decomposition, program simplification, program understanding

1. Introduction

In many problem areas, such as proving program-correctness, symbolic execution, and program testing, one often needs to deal with portions of a program associated with certain execution paths. Although conceptually it is useful to treat each of these components as a subprogram, there are no formalism or notational convention that can be used to represent it textually as such. In the literature, reference to a program component of that nature is usually made indirectly through a graphical description of the execution path, which rarely provides any insight into the problem at hand.

Introduced in the following section is a new programming construct, called a state constraint, that can be used to construct from a given program a subprogram with some of its execution paths. Such a subprogram can be systematically manipulated and simplified (Sec. 3).

A program can be decomposed into a set of subprograms by inserting different state constraints at different points in the control flow (Sec. 4). The process is called pathwise decomposition because the program is divided into subprograms along the execution paths. In the traditional method of decomposing a program into functions and procedures, the program is divided across the execution paths. Among all possible pathwise decompositions, the one that yields the symbolic traces [6] is of particular interest because the process can be readily automated (Sec. 5).

A major result of this development is that programs with complex logical structures can be systematically simplified. It can be accomplished by decomposing it pathwise into a set of subprograms, using the results presented in this paper to simplify them, and then recomposing the program from the simplified subprograms (Sec. 6). It can also be done directly by inserting state constraints that do not reduce the program into a subprogram (Sec. 7).

Finally, the concept of a state constraint not only provides a new mechanism for program decomposition and simplification, it also provides a unified framework for treating problems in program analysis, synthesis, testing, and verification as discussed in the concluding section.

The idea of pathwise decomposition is the main contribution of this paper. Theorems and corollaries in the following sections do not represent new results. They are mostly restatements of existing knowledge in new formalism designed to clarify the concept, and to make it possible to demonstrate the utility of pathwise decomposition with example programs.

2. State Constraints

Consider a restrictive clause of the form:

The program state at this point must satisfy predicate C, or else the program becomes undefined.

By program state here we mean the aggregate of values assumed by all variables involved. Since this clause constrains the states assumable by the program, it is called a *state constraint*, or a *constraint* for short, and is denoted by */\C*.

State constraints are designed to be inserted into a program to create another program. For instance, given a program of the form

Program 2.1: $S_1; S_2$

a new program can be created as shown below:

Program 2.2: $S_1; /\backslash C; S_2$

Program 2.2 is said to be created from Program 2.1 by constraining the program states to C prior to execution of S_2. Intuitively, Program 2.2 is a subprogram of Program 2.1 because its definition is that of program 2.1 restricted to C. Within that restriction, Program 2.2 performs the same computation as Program 2.1.

A state constraint is a semantic modifier. The meaning of a program modified by a state constraint can be formally defined in terms of Dijkstra's *weakest precondition*[*] [3] as follows. Let S be a programming construct and C be a predicate, then for any postcondition R,

Axiom 2.3: wp(*/*C;S, R) ≡ C wp(S, R).

A comparison of similar entities given below should help to sharpen the idea introduced above:

State constraint: The purpose is to restrict the domain of definition. The processor valuates its truth value. The program becomes undefined if it is evaluated to false.

Branch or loop predicate: The purpose is to alter the flow of control. The processor evaluates its truth value. If it is false, an alternative path is taken.

Constraint in the constraint programming languages [8]: The purpose is to define a relation that must be maintained throughout the program. The processor finds appropriate values of variables to satisfy the constraint. If it is found unsatisfiable, a programming error occurs.

Constraint in ADA [4]: It is used to specify further restrictions on the values that may be held by the variables, besides the restriction that the values must belong to the type specified by the type mark. The processor evaluates its truth value. If it is false, an exception is raised, but the definition of the program remains intact.

Constraint used in artificial intelligence: The purpose is to limit the values assumable by variables that occur in a description [2]. The processor evaluates its truth value. A false constraint signifies the inapplicability of the description.

---

[*] The weakest precondition of program S with respect to postcondition Q, commonly denoted by wp(S, Q), is defined as the weakest condition for the initial state of S such that activation of S will certainly result in a properly terminating happening, leaving S in a final state satisfying Q [3].

Assertion inserted in proving program-correctness [1] or dynamic assertion checking [9]: It is used to check the states assumed by the program, and does not modify the program in any way. The processor evaluates its truth value. The program is in error if it is evaluated to false.

Definition 2.4: Program S_1 is said to be *equivalent* to S_2 if wp(S_1, R) ≡ wp(S_2, R) for any postcondition R. This relation is denoted by $S_1 \Leftrightarrow S_2$.

Definition 2.5: Program S_2 is said to be a *subprogram* of program S_1 if wp(S_2, R) wp(S_1, R) for any postcondition R. This relation is denoted by S_1 S_2.

With these definition, one can now determine the relationship between any programs, with or without state constraints. For instance, consider Programs 2.1 and 2.2 again. Since wp(S_1;/\C;S_2, R) ≡ wp(S_1, wp(/\C;S_2, R)) ≡ wp(S_1, C wp(S_2, R)) ≡ wp(S_1, C) wp(S_1, wp(S_2, R)) ≡ wp(S_1, C) wp(S_1;S_2, R), it follows that wp(S_1;/\C;S_2, R) wp(S_1;S_2, R).

Thus, by Def. 2.5, Program 2.2 is a subprogram of Program 2.1.

Note that if C ≡ T, i.e., if C is always true, then wp(/\T;S, R) ≡ T wp(S, R) ≡ wp(S, R), and therefore, by Definition 2.4,

Corollary 2.6: /\T; S ⇔ S

That is to say, a state constraint will have no effect on a program if it is always true. On the other hand, if C ≡ F, i.e., if C is always false, then wp(/\F;S, R) ≡ F wp(S, R) ≡ F ≡ wp(/\F;S', R) for any S, S', and R, and therefore

Corollary 2.7: /\F; S ⇔ /\F; S'.

In words, any two programs are (trivially) equivalent if both are constrained by a predicate that can never be true.

Although in theory one can insert any constraints anywhere in a program to create a subprogram, it may not serve any purpose at all. The exact nature of constraints, and the points at which they are to be inserted, depend on the purpose to be served. If the purpose is to decompose a program into a set of simpler subprograms, the program should be constrained as discussed in the next two sections.

3. Subprogram Simplification

Presented in this section are three categories of equivalence relations showing how state constraints can be inserted into a program to reduce it to a simpler subprogram. The first indicates how to constrain a program so that the resulting subprogram will have a simpler logical structure; the second shows how the state constraints in a program can be manipulated and simplified. The third shows how to simplify long sequences of assignment statements, which often result from a repeated application of the first two.

Listed below are equivalence relations of the first category, the validity of which is immediately obvious.

Corollary 3.1: /\B; **if** B **then** S_1 **else** S_2 ⇔ /\B; S_1

Proof: By Axiom 2.3, wp(/\B; **if** B **then** S_1 **else** S_2, R)

≡ B wp(**if** B **then** S_1 **else** S_2, R)

≡ B (B wp(S_1, R) ¬B wp(S_2, R))

≡ B wp(S_1, R)

≡ wp(/\B; S_1, R). QED

Note: Most corollaries and theorems in this article can be simply proved in a similar manner, and thus are given without proof.

Corollary 3.2: /\¬B; **if** B **then** S_1 **else** S_2 ⇔ /\¬B; S_2

Corollary 3.3: /\$B_i$; **if** B_1 **then** S_1 **else if** B_2 **then** S_2 ...**else if** B_i **then** S_i ...**else if** B_n **then** S_n ⇔ /\$B_i$; S_i

Corollary 3.4: /\B; **while** B **do** S⇔/\B; S; **while** B **do** S

Corollary 3.5: /\¬B; **while** B **do** S ⇔ /\¬B

Corollary 3.6: /\B; **while** B **do** S ⇔ /\B; **repeat** S **until** ¬B

To demonstrate how these relations can be utilized to construct a subprogram with a simpler logical structure, consider the following Pascal program:

Program 3.7:

```
FUNCTION ctoi(VAR s : string; VAR i : INTEGER) : INTEGER;
  VAR
    n, sign : INTEGER;
  BEGIN
    WHILE (s[i] = BLANK) OR (s[i] = TAB) DO i := i + 1;
    IF s[i] = MINUS THEN sign := -1 ELSE sign := 1;
    IF (s[i] = PLUS) OR (s[i] = MINUS) THEN i := i + 1;
    n := 0;
    WHILE isdigit(s[i]) DO
      BEGIN
      n := 10 * n + s[i] - ORD('0');
      i := i + 1
    END;
    ctoi := sign * n
  END;
```

In light of the equivalence relations given above, one can produce a subprogram with simple logical structures by constraining the program states as shown below:

```
FUNCTION ctoi(VAR s : string; VAR i : INTEGER) : INTEGER;
  VAR
    n, sign : INTEGER;
  BEGIN
    /\ NOT((s[i] = BLANK) OR (s[i] = TAB));
    WHILE (s[i] = BLANK) OR (s[i] = TAB) DO i := i + 1;
    /\ NOT(s[i] = MINUS);
    if s[i] = MINUS THEN sign := -1 ELSE sign := 1;
    /\ NOT((s[i] = PLUS) OR (s[i] = MINUS));
    IF (s[i] = PLUS) OR (s[i] = MINUS) THEN i := i + 1;
    n := 0;
    /\ isdigit(s[i]);
    WHILE isdigit(s[i]) DO
      BEGIN
        n := 10 * n + s[i] - ORD('0');
        i := i + 1
      END;
    ctoi := sign * n
```

```
    END;
```

which can be simplified to

```
⇔   FUNCTION ctoi(VAR s : string; VAR i : INTEGER) : INTEGER;
      VAR
        n, sign : INTEGER;
      BEGIN
        /\NOT((s[i] = BLANK) OR (s[i] = TAB));
        /\NOT(s[i] = MINUS);
        sign := 1;
        /\NOT((s[i] = PLUS) OR (s[i] = MINUS));
        n := 0;
        /\isdigit(s[i]);
        n := 10 * n + s[i] - ORD('0');
        i := i + 1
        WHILE isdigit(s[i]) DO
          BEGIN
            n := 10 * n + s[i] - ORD('0');
            i := i + 1
          END;
        ctoi := sign * n
      END;
```

Note that Corollary 3.4 can be repeatedly applied to a loop construct to produce potentially an infinite number of subprograms. Subprograms of a loop construct will be discussed in detail later.

A subprogram usually can be simplified further if it does not contain any control statement at all. To illustrate, Corollary 3.5 is applied to the "while" statement above to reduce it to a loopless subprogram.

Program 3.8:

```
FUNCTION ctoi(VAR s : string; VAR i : INTEGER) : INTEGER;
  VAR
    n, sign : INTEGER;
  BEGIN
    /\NOT((s[i] = BLANK) OR (s[i] = TAB));
    /\NOT(s[i] = MINUS);
    sign := 1;
    /\NOT((s[i] = PLUS) OR (s[i] = MINUS));
    n := 0;
    /\isdigit(s[i]);
    n := 10 * n + s[i] - ORD('0');
    i := i + 1;
    /\NOT(isdigit(s[i]));
    ctoi := sign * n
  END;
```

This is one of infinitely many canonical subprograms (defined below) that can be produced from Program 3.7.

Definition 3.9: A subprogram is said to be *canonical* if it does not contain any control statement.

This kind of subprogram is of particular interest because it has the simplest logical structure.

Note that there are two state constraints at the beginning of Program 3.8. The meaning of two consecutive state constraints is given by the following relation, which is a direct consequence of Axiom 2.3.

Corollary 3.10: /\$C_1$;/\$C_2$;S ⇔ /\$C_1$ C_2;S.

Program 3.8 can thus be rewritten as

Program 3.11:

```
FUNCTION ctoi(VAR s : string; VAR i : INTEGER) : INTEGER;
  VAR
    n, sign : INTEGER;
  BEGIN
    /\NOT((s[i]=BLANK) OR (s[i]=TAB)) AND NOT(s[i]=MINUS);
    sign := 1;
    /\NOT((s[i]=PLUS) OR (s[i]=MINUS));
    n := 0;
    /\isdigit(s[i]);
    n := 10 * n + s[i] - ORD('0');
    i := i + 1;
    /\NOT(isdigit(s[i]));
    ctoi := sign * n
  END;
```

A state constraint not only directly constrains the program state at the point where it is placed, but also indirectly at other points upstream and downstream in control flow as well. Note that, in Program 2.2, predicate C is true if and only if wp(S_1, C) is true before execution of S_1. Thus, by constraining the program state between S_1 and S_2 to C, it also indirectly constrains the program state before S_1 to wp(S_1, C), and the program state after S_2 to R, where wp(S_2, R) ≡ C.

The *scope* of a state constraint, which is defined to be the range of control flow within which the constraint has an effect, may or may not span the entire program. A state constraint will have no effect beyond a statement that undefines, or assigns a constant value to, the variables involved. For instance, a state constraint like `x > 0` will have no effect on the program states beyond statement `read(x)` upstream, statement `return` downstream if `x` is a local variable, or statement `x:=4` upstream or downstream.

Another view of this property is that exactly the same constraint on the program states in Program 2.2 can be affected by placing constraint /\wp(S_1, C) before S_1 or constraint /\C before S_2. To be more precise, S_1;/\C;S_2 ⇔ /\wp(S_1, C);S_1;S_2 if the scope of /\C is not terminated by S_1. In general, this relationship can be stated as follows.

Theorem 3.12: S;/\R ⇔ /\Q;S if Q ≡ wp(S, R).

This relation can be used repeatedly to move a constraint upstream, i.e., to constrain the program equivalently at a different point upstream.

On the other hand, given a program of the form /\Q;S, the same relation can be applied in other way to move the constraint downstream. In that case, what we need to do is to find a predicate R such that wp(S, R) ≡ Q. In accordance with the discussion in [5], this can be accomplished by using the relation R ≡ wp(S^{-1}, Q), where S^{-1} is a sequence of statements to be obtained from S and Q such that wp(S;S^{-1}, Q) ≡ Q.

Note that, if S;S^{-1} is a sequence of assignment statements, wp(S;S^{-1}, Q) ≡ Q. is true as long as an execution of S;S^{-1} will not alter the value of any variable that occurs in Q. Thus the required S^{-1} may be found by noting that, if S contains a statement that changes the value of a variable in Q, S^{-1} should contain a statement that restores the old value of that variable. S^{-1} does not exist if S assigns a constant value to a variable in Q [5].

From the above discussion we see that, for convenience in future applications, Theorem 3.12 can be alternatively stated as

Theorem 3.12a: S;/\R ⇔ /\wp(S, R);S,

or

Theorem 3.12b: /\Q;S ⇔ S;/\wp(S^{-1}; Q).

Corollary 3.10 and Theorem 3.12 belong to the second category of equivalence relations that one can use to combine and simplify the constraints placed in a program to make it more readable. For instance, by moving all state constraints in Program 3.11 to the top, it can be reduced to the one listed below:

```
FUNCTION ctoi(VAR s : string; VAR i : INTEGER) : INTEGER;
  VAR
    n, sign : INTEGER;
  BEGIN
    /\NOT((s[i]=BLANK) OR (s[i]=TAB)) AND NOT(s[i]=MINUS);
    /\NOT((s[i]=PLUS) OR (s[i]=MINUS));
    /\isdigit(s[i]);
    /\NOT(isdigit(s[i+1]));
    sign := 1;
    n := 0;
    n := 10 * n + s[i] - ORD('0');
    i := i + 1;
    ctoi := sign * n
  END;
```

One reason why the state constraints in a program can be simplified is that some state constraints are implied by the others, and thus can be eliminated. To be more specific, if two state constraints C_1 and C_2 are such that $C_1 \quad C_2$ then C_2 can be discarded because $C_1 \quad C_2 \equiv C_1$. For example, the first two state constraints in the above program are implied by the third. Hence the above program can be simplified to

Program 3.13:

```
FUNCTION ctoi(VAR s : string; VAR i : INTEGER) : INTEGER;
  VAR
    n, sign : INTEGER;
  BEGIN
    /\isdigit(s[i]) AND NOT(isdigit(s[i+1]))
    sign := 1;
    n := 0;
    n := 10 * n + s[i] - ORD('0');
    i := i + 1;
    ctoi := sign * n
  END;
```

Some state constraints may be eliminated in the simplification process because it is always true due to computation performed by the program. For example, the following program contains such a constraint.

```
x := 0;       ⇔   x := 0;       ⇔   /\0 + 1 <> 0;     ⇔     x := 0;
y := x + 1;       /\x + 1 <> 0;     x := 0;                 y := x + 1;
/\y <> 0;         y := x + 1;       y := x + 1
```

Definition 3.14: A state constraint is said to be *redundant* if it can be eliminated without changing the function implemented by the program. To be more precise, the constraint /\C in the program S_1;/\C; S_2 is redundant if and only if S_1;/\C; S_2 $\Leftrightarrow$ S_1;S_2.

The properties and possible exploitation of redundant constraints will be discussed in a later section.

As one might have observed in previous examples, moving state constraints interspersed in the statements to the same point in control flow often leaves a long sequence of assignment statements in the program. These assignment statements may be combined and simplified by using the three equivalence relations presented in the following.

Corollary 3.15: $x:=E_1; x:=E_2 \Leftrightarrow x:=(E_2)_{E_1 \rightarrow x}$

Here $(E_2)_{E_1 \rightarrow x}$ denotes the expression obtained from E_2 by substituting E_1 for every occurrence of x in E_2. For example, applying this corollary to the second and the third assignment statements of Program 3.13 yields

```
FUNCTION ctoi(VAR s : string; VAR i : INTEGER) : INTEGER;
  VAR
    n, sign : INTEGER;
  BEGIN
    /\isdigit(s[i]) AND NOT(isdigit(s[i+1]))
    sign := 1;
    n := s[i] - ORD('0');
    i := i + 1;
    ctoi := sign * n
  END;
```

Although in general two assignment statements cannot be interchanged, an assignment statement may be moved downstream under certain circumstances. In particular,

Corollary 3.16: If x_2 does not occur in E_1 then

$x_1:=E_1;x_2:=E_2 \Leftrightarrow x_2:=(E_2)_{E_1 \rightarrow x_1};x_1:=E_1$.

For example, by applying this relation to the above example it becomes

```
⇔     FUNCTION ctoi(VAR s : string; VAR i : INTEGER) : INTEGER;
          VAR
            n, sign : INTEGER;
          BEGIN
            /\isdigit(s[i]) AND NOT(isdigit(s[i+1]))
            n := s[i] - ORD('0');
            sign := 1;
            i := i + 1;
            ctoi := sign * n
          END;

⇔     FUNCTION ctoi(VAR s : string; VAR i : INTEGER) : INTEGER;
          VAR
            n, sign : INTEGER;
          BEGIN
            /\isdigit(s[i]) AND NOT(isdigit(s[i+1]))
            n := s[i] - ORD('0');
            i := i + 1;
            sign := 1;
```

```
        ctoi := sign * n
      END;
```

The purpose of an assignment statement is to assign a value to a variable so that it can be used in some statements downstream. Now if the above rule is used to move an assignment statement downstream past all statements in which the assigned value is used, the statement becomes redundant and thus can be deleted.

Definition 3.17: A statement in a program is said to be *redundant* if its sole purpose is to define the value of a data structure, and this particular value is not used anywhere in the program.

Obviously, a redundant statement can be removed without changing the computation performed by the program.

Corollary 3.18: If $x_1:=E_1;x_2:=E_2$ is a sequence of two assignment statements such that, by interchanging these two statements, $x_1:=E_1$ becomes redundant, then $x_1:=E_1;x_2:=E_2 \Leftrightarrow x_2:=(E_2)_{E_1 \rightarrow x_1}$.

For example, by interchanging the last two statements in the above program, the statement `sign := 1` becomes redundant, and therefore Corollary 3.18 can be applied to simplify the program to

```
FUNCTION ctoi(VAR s : string; VAR i : INTEGER) : INTEGER;
  VAR
    n, sign : INTEGER;
  BEGIN
    /\isdigit(s[i]) AND NOT(isdigit(s[i+1]))
    n := s[i] - ORD('0');
    i := i + 1;
    ctoi := n
  END;
```

In general, Corollary 3.18 becomes applicable when $x_2:=E_2$ is the last statement to make use of definition provided by $x_1:=E_1$. Corollaries 3.16 and 3.18 can be used to reduce the number of assignment statements in a program, and thus the number of steps involved in computation. The end result is often a simpler and more understandable program. For instance, the above program can be further simplified to the one listed below:

```
FUNCTION ctoi(VAR s : string; VAR i : INTEGER) : INTEGER;
  VAR
    n, sign : INTEGER;
  BEGIN
    /\isdigit(s[i]) AND NOT(isdigit(s[i+1]))
    n := s[i] - ORD('0');
    ctoi := n
    i := i + 1;
  END;

⇔   FUNCTION ctoi(VAR s : string; VAR i : INTEGER) : INTEGER;
    VAR
      n, sign : INTEGER;
    BEGIN
      /\isdigit(s[i]) AND NOT(isdigit(s[i+1]))
      ctoi := s[i] - ORD('0');
      n := s[i] - ORD('0');
      i := i + 1;
    END;

⇔   FUNCTION ctoi(VAR s : string; VAR i : INTEGER) : INTEGER;
        VAR
          n, sign : INTEGER;
```

```
BEGIN
  /\isdigit(s[i]) AND NOT(isdigit(s[i+1]))
  ctoi := s[i] - ORD('0');
  i := i + 1;
END;
```

4. Pathwise Decomposition

From the above discussion, it is obvious that a program can be decomposed into a set of subprograms by inserting different state constraints at different points in control flow. To distinguish this process from the traditional method of decomposing a program into procedures and functions, the present method is called *pathwise decomposition* because the program is divided *along* the control flow, whereas in the traditional method of decomposing a program into procedures and functions, a program is divided *across* the control flow.

To be able to speak of, and make use of, a set of subprograms produced by pathwise decomposition, it is necessary to introduce a new programming construct called a *program set*. The meaning of a program set, or a set of programs, is identical to the conventional notion of a set of other objects. As usual, a set of n programs is denoted by $\{P_1, P_2, ..., P_n\}$. When used as a programming construct, it describes the computation prescribed by its elements. Formally, the semantics of such a set is defined as

Axiom: 4.1: $wp(\{P_1, P_2, ..., P_n\}, R) \equiv wp(P_1, R) \quad wp(P_2, R) \quad ... \quad wp(P_n, R)$.

The choice of this particular semantics will be explained in detail at the end of this section. A program set so defined has all properties commonly found in an ordinary set. For instance, since the logical operation of disjunction is commutative, a direct consequence of Axiom 4.1 is that

Corollary 4.2: The ordering of elements in a program set is immaterial, i.e.,

$$\{P_1, P_2\} \Leftrightarrow \{P_2, P_1\}.$$

Furthermore, since every proposition is an idempotent under the operation of disjunction,

Corollary 4.3: $P \Leftrightarrow \{P\} \Leftrightarrow \{P, P\}$ for any program P.

In words, a set is unchanged by listing any of its elements more than once. A program set can be used just like a block of statements in program composition. Concatenation of program sets is defined similarly as concatenation of two ordinary sets. In particular,

Corollary 4.4: For any programs P, P_1, and P_2,

(a) $P;\{P_1, P_2\} \Leftrightarrow \{P;P_1, P;P_2\}$
(b) $\{P_1, P_2\};P \Leftrightarrow \{P_1;P, P_2;P\}$

Given below are a number of useful relations that can be derived from the semantics of a program set.

Corollary 4.5: If $P \Leftrightarrow P'$ then $\{P\} \Leftrightarrow \{P, P'\}$.

Corollary 4.6: If $P \Leftrightarrow P_1$ and $P \Leftrightarrow P_2$ then $P \Leftrightarrow \{P_1, P_2\}$.

Definition 4.7: A program is said to be *unconstrained* if it contains no state constraint at all, or if every state constraint in the program is redundant.

Definition 4.8: Two programs $/\backslash C_1;P_1$ and $/\backslash C_2;P_2$ are said to be *equivalently constrained* if and only if P_1 and P_2 are both unconstrained and $C_1 \equiv C_2$.

Theorem 4.9: If $P_1 \quad P_2$, and if P_1 and P_2 are equivalently constrained, then $P_1 \Leftrightarrow P_2$.

The following relations are useful in working with sets of constrained subprograms.

Corollary 4.10: $/\backslash C_1 \quad C_2;P \Leftrightarrow \{/\backslash C_1;P, /\backslash C_2;P\}$

Corollary 4.11: If $C_1, C_2, ..., C_n$ are n constraints such that $C_1 \quad C_2 \quad ... \quad C_n \equiv T$ then

$$P \Leftrightarrow /\backslash C_1 \quad C_2 \quad ... \quad C_n;P \Leftrightarrow \{/\backslash C_1;P, /\backslash C_2;P, ..., /\backslash C_n;P\}$$

The last corollary serves as the basis for decomposing a program into an equivalent set of subprograms. It can also be used to compose a new program from a set of subprograms, which is more general than any of its components. The most general program that can be obtained through this process is the one that does not contain any non-trivial state constraint at all.

Since braces and comma may have a different meaning in a real programming language, and since in practice program statements are written line by line from top to bottom, a program set of the form $\{P_1, P_2, ..., P_n\}$ may be alternatively written as

```
{{{
   P1
,,,
   P2
,,,
    .
    .
    .
,,,
   Pn
}}}
```

where triple braces and triple commas are used instead to avoid any possible confusion.

To clarify the concept and notations just introduced, consider Program 4.12 listed below, which can be decomposed into a set of two subprograms:

Program 4.12:

```
PROGRAM mc91f (INPUT, OUTPUT);
  VAR  x, y, z: INTEGER;
  BEGIN
    READ (x);
    y := 1;
    WHILE x <= 100 DO BEGIN x := x + 11; y := y + 1 END;
    WHILE y <> 1 DO
      BEGIN
        x := x - 10;
        y := y - 1;
        WHILE x <= 100 DO BEGIN x:=x+11; y:=y+1 END;
      END;
    z := x - 10;
    WRITELN ( 'z = ', z );
  END.
```

```
⇦     PROGRAM mc91f (INPUT, OUTPUT);
        VAR  x, y, z: INTEGER;
        BEGIN
          READ (x);
          y := 1;
          /\x <= 100 OR x > 100;                                        by 4.11
          WHILE x <= 100 DO BEGIN x:=x+11; y:=y+1 END;
          WHILE y <> 1 DO
            BEGIN
              x := x - 10;
              y := y - 1;
              WHILE x <= 100 DO BEGIN x:=x+11; y:=y+1 END;
            END;
          z := x - 10;
          WRITELN ( 'z = ', z );
        END.

⇦     PROGRAM mc91f (INPUT, OUTPUT);
        VAR  x, y, z: INTEGER;
        BEGIN
          READ (x);
          y := 1;
          {{{                                                           by 4.11
            /\x <= 100;
            WHILE x <= 100 DO BEGIN x:=x+11; y:=y+1 END;
          ,,,
            /\x > 100;                                                  by 3.5
          }}}
          WHILE y <> 1 DO
            BEGIN
              x := x - 10;
               y := y - 1;
               WHILE x <= 100 DO BEGIN x:=x+11; y:=y+1 END;
             END;
           z := x - 10;
           WRITELN ( 'z = ', z );
         END.

⇦   Program 4.13:
       {{{                                                              by 4.4
         PROGRAM mc91f (INPUT, OUTPUT);
           VAR  x, y, z: INTEGER;
           BEGIN
             READ (x);
             y := 1;
             /\ x <= 100;
             WHILE x <= 100 DO BEGIN x:=x+11; y:=y+1 END;
             WHILE y <> 1 DO
               BEGIN
                 x := x - 10;
                 y := y - 1;
                 WHILE x <= 100 DO BEGIN x:=x+11;y:=y+1 END;
               END;
             z := x - 10;
             WRITELN ( 'z = ', z );
           END.
       ,,,
```

```
PROGRAM mc91f (INPUT, OUTPUT);
  VAR  x, y, z: INTEGER;
  BEGIN
    READ (x);
    y := 1;
    /\ x > 100;
    WHILE y <> 1 DO
      BEGIN
        x := x - 10;
        y := y - 1;
        WHILE x <= 100 DO BEGIN x:=x+11;y:=y+1 END;
      END;
    z := x - 10;
    WHILE ( 'z = ', z );
  END.
}}}
```

There are two more corollaries that we need for discussion in a later section.

Corollary 4.14: /\C{P_1 P_2} ⇔ {/\C;P_1, /\C:P_2}

Corollary 4.15: {P_1;/\$C_1$ P_2;/\$C_2$} ⇔ {$P_1$; /\$C_1$, P_2;/\$C_2$}/\$C_1$ C_2

Finally, for those who are familiar with Dijkstra's work on the guarded commands [3], the general form of a program set

{/\$C_1$;$P_1$, /\$C_2$;$P_2$, ..., /\$C_n$;$P_n$}

may appear to be deceptively similar to Dijkstra's "IF" statement:

if $C_1 \rightarrow P_1 \| C_2 \rightarrow P_2 \| ... \| C_n \rightarrow P_n$ **fi**.

Are these two constructs equivalent? Note that, by Axiom 4.1,

Corollary 4.16:

wp({/\$C_1$;$P_1$, /\$C_2$;$P_2$, ..., /\$C_n$;$P_n$}, R)

≡ wp(/\$C_1$;$P_1$, R) wp(/\$C_2$;$P_2$, R) ... wp(/\$C_n$;$P_n$, R)

≡ C_1 wp(P_1, R) C_2 wp(P_2, R) ... C_n wp(P_n, R),

and according to the definition given in [3],

Corollary 4.17:

wp(**if** $C_1 \rightarrow P_1 \| C_2 \rightarrow P_2 \| ... \| C_n \rightarrow P_n$ **fi**, R)

≡ (C_1 C_2 ... C_n) (¢$_1$ wp(P_1, R)) (¢$_2$ wp(P_2, R)) ... (¢$_n$ wp(P_n, R)).

Therefore, {/\$C_1$;$P_1$, /\$C_2$;$P_2$, ..., /\$C_n$;$P_n$} is not equivalent to **if** $C_1 \rightarrow P_1 \| C_2 \rightarrow P_2 \| ... \| C_n \rightarrow P_n$ **fi** in general.

The reason for choosing the semantics defined by Axiom 4.1 instead of the Dijkstra's is to make the meaning of a program set as intuitive as possible.

In the present work it is appropriate to view a program set as a collection of tools, each of which can be used to do something. As such, if one added a program to the set, one naturally expects the capability of the resulting set to increase, or at least to remain the same.

The capability of a program increases if its weakest precondition becomes weaker for the same postcondition (i.e., the program can do the same for more input data), or if the same set of inputs satisfies a stronger postcondition (i.e., the program can do more computations for the same set of input data).

The semantics specified by Axiom 4.1 conforms to this view of a program set: the larger the set, the more versatile it becomes. In particular, if one added a new program P_2 to the program set $\{P_1\}$ to form the new set $\{P_1, P_2\}$, the weakest precondition of the resulting set becomes $wp(\{P_1, P_2\}, R) \equiv wp(\{P_1\}, R) \quad wp(P_2, R)$, which is weaker than (or equivalent to) $wp(\{P_1\}, R)$. This is always true regardless of the relation between P_1 and P_2.

For example, consider the singleton program set $\{/\backslash x > 2;\ y := 1\}$, which is capable of setting y to 1 if x is greater than 2. If a new program $/\backslash x > 2;\ y := 2$ is added to this set to form a larger program set $\{/\backslash x > 2;\ y := 1,\ /\backslash x > 2;\ y := 2\}$, then, by Corollary 4.16,

$$wp(\{/\backslash x > 2;\ y := 1,\ /\backslash x > 2;\ y := 2\},\ y = 1) \equiv x > 2 \quad F \equiv x > 2.$$

That is to say, the expanded set of programs still has the old capability to set y to 1 if $x > 2$.

What will happen if one adopts Dijkstra's definition and let the semantics of $\{/\backslash C_1;P_1,\ /\backslash C_2;P_2,\ ...,\ /\backslash C_n;P_n\}$ be identical to that of **if** $C_1 \rightarrow P_1 \,[]\, C_2 \rightarrow P_2 \,[]\, ... \,[]\, C_n \rightarrow P_n$ **fi** ? One immediate consequence is that

$$wp(\{/\backslash C_1;P_1,\ /\backslash C_2;P_2\}, R)$$

$$\equiv wp(\textbf{if } C_1 \rightarrow P_1 \,[]\, C_2 \rightarrow P_2 \textbf{ fi}, R)$$

$$\equiv (C_1 \quad C_2) \quad (C_1 \quad wp(P_1, R)) \quad (C_2 \quad wp(P_2, R))$$

$$\equiv C_1 \ \neg C_2 \ wp(P_1, R) \quad C_2 \ \neg C_1 \ wp(P_2, R) \quad C_1 \ C_2 \ wp(P_1, R) \ wp(P_2, R),$$

and thus

$$wp(\{/\backslash x > 2;\ y := 1,\ /\backslash x > 2;\ y := 2\},\ y = 1) \equiv F.$$

That is to say, by acquiring a new program, the expanded set of programs is no longer capable of setting y to 1 if x is greater than 2. In the present work, this is counterintuitive.

There are other fundamental differences between the constrained program $/\backslash C;P$ and the guarded command $C \rightarrow P$. The former is a compound statement, whereas the latter is only an expression which can be used to construct a statement. The constraint $/\backslash C$ is introduced as an analysis tool to facilitate analysis of a given program, whereas the guarded command $C \rightarrow P$ is introduced as a synthesis tool to facilitate specification of a program.

It is interesting to note that

$$/\backslash C;\ \textbf{if } C_1 \rightarrow P_1 \,[]\, C_2 \rightarrow P_2 \,[]\, ... \,[]\, C_n \rightarrow P_n \textbf{ fi}$$

$$\Leftrightarrow \textbf{if } C \quad C_1 \rightarrow P_1 \,[]\, C \quad C_2 \rightarrow P_2 \,[]\, ... \,[]\, C \quad C_n \rightarrow P_n \textbf{ fi}.$$

5. Trace Subprograms

In light of Corollary 4.11, it is obvious that there are infinitely many ways to decompose a program into an equivalent set of subprograms. From the practical point of view, however, it is of particular interest to decompose a program into the following type of subprograms.

Definition 5.1: A subprogram of some program P is called a *trace subprogram* of P if it describes the computation performed by a single execution path in that program. It is defined for those and only those inputs that cause the associated path to be traversed in an execution.

For instance,

```
/\ i < 10;
/\ a[i] < 0;
sum := sum - a[i];
i := i + 1
/\ NOT(i < 10);
```

is a trace subprogram of the following program:

```
WHILE i < 10 DO
  BEGIN
    IF a[i] < 0 THEN sum:=sum-a[i] ELSE sum:=sum+a[i];
    i := i + 1
  END;
```

However, neither

```
/\ i < 10;
/\ a[i] < 0;
sum := sum - a[i];
/\ i > 5
i := i + 1
/\ NOT(i < 10);
```

nor

```
/\ i < 10;
IF a[i] < 0 THEN sum:=sum-a[i] ELSE sum:=sum+a[i];
i := i + 1
/\ NOT(i < 10);
```

is a trace subprogram of that program because the former represents only a portion of an execution path while the latter contains more than one execution path.

Note that any trace subprogram is a canonical subprogram (cf. Def. 3.9), or can be rewritten into one. This is so because, by definition, a trace subprogram may contain only one execution path. Such a program can be written without the use of any control statement. On the other hand, a canonical subprogram of a program is not necessarily a trace subprogram of that program because it may corresponds to only a portion of an execution path.

For any particular execution path in a program, the associated trace subprogram can be constructed directly from its source code by listing all statements on the execution path, in the order they are encountered, and constraining every control statement with the branch predicate which is evaluated to true at that point.

Alternatively, the trace subprogram of an execution path can be obtained from its symbolic trace. As defined in [6], the symbolic trace of an execution path is a linear list of all statements and branch predicates that occur on the execution path. A symbolic trace is transformed into a trace subprogram by rewriting every branch predicate therein as a state constraint. The trace subprogram of any execution path, like its symbolic trace, therefore, can be generated automatically through program instrumentation as described in [6].

Trace subprograms are of particular interest because, with proper tools, they can be produced and simplified readily. Given a program that is difficult to understand because of complex logical structure, one can often reduce the difficulty by understanding the program in terms of its simplified trace subprograms.

For example, consider the C program listed below:

Program 5.2:

```
main ()
{
  int  i, j, k, match;
       scanf("%d %d %d", &i, &j, &k);
       printf("%d %d %d\n", i, j, k);
       if (i <= 0 || j <= 0 || k <= 0) goto L500;
       match = 0;
       if (i != j) goto L10;
       match = match + 1;
  L10: if (i != k) goto L20;
       match = match + 2;
  L20: if (j != k) goto L30;
       match = match + 3;
  L30: if (match != 0) goto L100;
       if (i+j <= k) goto L500;
       if (j+k <= i) goto L500;
       if (i+k <= j) goto L500;
       match = 1;
       goto L999;
 L100: if (match != 1) goto L200;
       if (i+j <= k) goto L500;
 L110: match = 2;
       goto L999;
 L200: if (match != 2) goto L300;
       if (i+k <= j) goto L500;
       goto L110;
 L300: if (match != 3) goto L400;
       if (j+k <= i) goto L500;
       goto L110;
 L400: match = 3;
       goto L999;
 L500: match = 4;
 L999: printf("%d\n", match);
 }
```

The main body of this program can be decomposed into a set of 12 trace subprograms. By rewriting the program in terms of the simplified subprograms, it becomes

Program 5.3:

```
main ()
{
  int  i, j, k, match;

  scanf("%d %d %d", &i, &j, &k);
```

```
printf("%d %d %d\n", i, j, k);
{{{
  /\ (i <= 0) || (j <= 0) || (k <= 0)
  match = 4;
'''
  /\ (i > 0) && (j > 0) && (k > 0)
  /\ (i + j > k) && (j + k > i) && (i + k <= j)
  /\ (i != j) && (i != k) && (j != k)
  match = 4;
'''
  /\ (i > 0) && (j > 0) && (k > 0)
  /\ (i + j > k) && (j + k <= i)
  /\ (i != j) && (i != k) && (j != k)
  match = 4;
'''
  /\ (i > 0) && (j > 0) && (k > 0)
  /\ (i + j <= k)
  /\ (i != j) && (i != k) && (j != k)
  match = 4;
'''
  /\ (i > 0) && (j > 0) && (k > 0)
  /\ (j + k <= i)
  /\ (i != j) && (j == k)
  match = 4;
'''
  /\ (i > 0) && (j > 0) && (k > 0)
  /\ (i + k <= j)
  /\ (i != j) && (i == k)
  match = 4;
'''
  /\ (i > 0) && (j > 0) && (k > 0)
  /\ (i + j <= k)
  /\ (i == j) && (i != k)
  match = 4;
'''
  /\ (i > 0) && (j > 0) && (k > 0)
  /\ (i == j) && (i == k) && (j == k)
  match = 3;
'''
  /\ (i > 0) && (j > 0) && (k > 0)
  /\ (i + j > k)
  /\ (i == j) && (i != k)
  match = 2;
'''
  /\ (i > 0) && (j > 0) && (k > 0)
  /\ (j + k > i)
  /\ (i != j) && (j == k)
  match = 2;
'''
  /\ (i > 0) && (j > 0) && (k > 0)
  /\ (i + k > j)
  /\ (i != j) && (i == k)
  match = 2;
'''
  /\ (i > 0) && (j > 0) && (k > 0)
  /\ (i != j)&& (i != k) && (j != k)
  /\ (i + j > k) && (j + k > i) && (i + k > j)
  match = 1;
```

```
    }}}
    printf("%d\n", match);
}
```

Obviously, Program 5.3 is more understandable then Program 5.2 because its logical structure is much simpler.

Let f be a function defined on a set D. Usually a program p is designed to implement f by decomposing it into a set of n subfunctions, i.e., $f = \{f_1, f_2, ..., f_n\}$ such that $D = D_1 \quad D_2 \quad ... \quad D_n$ and f_i is f restricted to D_i, for all $1 \le i \le n$. If the program is correctly constructed, it must be composed of n trace subprograms, i.e., $p = \{p_1, p_2, ..., p_n\}$ such that each p_i implements subfunction f_i. The state constraints in p_i jointly define the input subdomain D_i, and the statements in p_i describe how f_i is to be computed.

Thus, the number of execution paths in a program is equal to the number of input subdomains partitioned by the program. It does not necessarily grow exponentially with the number of conditional statements in the program as commonly believed. For instance, there are 14 "if" statements in Program 5.2. Theoretically, there could be as many as $2^{14} = 16384$ execution paths, while in fact there are only 12.

The fact is that most real programs have relatively few feasible execution paths unless a loop construct is involved.

When the program contains a loop construct, it can be decomposed into a potentially infinite number of trace subprograms. Fortunately, a loop construct can be recomposed from a small number of its trace subprograms as described in the next section. A loop construct can therefore be simplified by decomposing it into a number of trace subprograms, simplifying the subprograms, and then recomposing the loop construct from the simplified subprograms.

6. Program Recomposition

How can a loop construct be recomposed from a finite number of its trace subprograms? The reason is very simple. Consider a loop construct of the form **repeat** S **until** ¬B, whose trace subprograms are formed by concatenating the trace subprograms of S. If S is loop-free then it is composed of a finite number of trace subprograms. Therefore, all information needed to recompose S, and hence **repeat** S **until** ¬B, can be found in a finite subset of the trace subprograms of the loop. If S contains loop constructs, the above argument can be recursively applied to the inner loop.

This fact can be precisely stated as follows.

Theorem 6.1: If **repeat** S **until** ¬B $\quad S_1;/\backslash B;S_2;/\backslash B;S_3; ... ;/\backslash B;S_n;/\backslash \neg B$ then

repeat S **until** ¬B $\quad$ **repeat** $\{S_1, S_2, S_3, ..., S_n\}$ **until** ¬B and

repeat $\{S_1, S_2, ..., S_n\}$ **until** ¬B $\quad S_1;/\backslash B;S_2;/\backslash B; ... ;S_{n-1};/\backslash B; S_n;/\backslash \neg B.$

Theorem 6.2: If **repeat** S **until** ¬B $\quad$ **repeat** $\{S_1, S_2, ... S_n\}$ **until** ¬B and if $\{S_1, S_2, ..., S_n\}$ as a whole is unconstrained then

repeat S **until** ¬B ⇔ **repeat** $\{S_1, S_2, ... S_n\}$ **until** ¬B.

For example, the following trace subprogram is produced by instrumenting Program 4.12 and then having it executed with x = 97.

Program 6.3:

```
READ (x);
y := 1;
/\ x <= 100;
```

```
x := x + 11;
y := y + 1;
/\ NOT(x <= 100);
/\ y <> 1;
x := x - 10;
y := y - 1;
/\ x <= 100;
x := x + 11;
y := y + 1;
/\ NOT(x <= 100);
/\ y <> 1;
x := x - 10;
y := y - 1;
/\ x <= 100;
x := x + 11;
y := y + 1;
/\ NOT(x <= 100);
/\ y <> 1;
x := x - 10;
y := y - 1;
/\ x <= 100
x := x + 11;
y := y + 1;
/\ NOT(x <= 100);
/\ y <> 1;
x := x - 10;
y := y - 1;
/\ NOT(x <= 100);
/\ NOT(y <> 1);
z := x - 10;
WRITELN ( 'z = ', z );
```

Applying Theorem 6.1 to the above subprogram yields

```
READ (x);
y := 1;
/\ x <= 100;
REPEAT x := x + 11; y := y + 1 UNTIL x > 100;
/\ y <> 1
REPEAT
  {{{
    x := x - 10;
    y := y - 1;
    /\ x <= 100
    x := x + 11;
    y := y + 1;
    /\ NOT(x <= 100);
  ,,,
    x := x - 10;
    y := y - 1;
    /\ NOT(x <= 100);
  }}}
UNTIL y = 1;
z := x - 10;
WRITELN ( 'z = ', z );
```

which can be readily simplified to

Program 6.4:

```
⇔       READ (x);
        y := 1;
        /\ x <= 100;
        REPEAT x := x + 11; y := y + 1 UNTIL x > 100;
        REPEAT
            {{{
              /\ x > 99;
              /\ x <= 110;
              x := x + 1;
            '''
              /\ x > 110;
              x := x - 10;
              y := y - 1;
            }}}
        UNTIL y = 1;
        z := x - 10;
        WRITELN ( 'z = ', z );
```

Given below are some other relations that can be used to recompose a program from its subprograms.

Corollary 6.5: if B **then** S_1 **else** S_2; ⇔ {/\B;S_1, /\ ¬ B;S_2}.

Corollary 6.6:

if B **then** S_1
else if B_2 **then** S_2
.
.
.
else if B_n **then** S_n
else S_{n+1};

⇔ {{{
/\ B_1;
S_1;
‘’’
/\ ¬ B_1 B_2;
S_2;
‘’’
.
.
.
‘’’
/\ ¬ B_1 ¬ B_2 ... ¬B_{n-1} B_n;
S_n;
‘’’
/\ ¬B_1 ¬B_2 ... ¬B_n;
S_{n+1};
}}}

For example, Corollary 6.6 can be applied to recompose Program 5.2 from the simplified subprograms in Program 5.3. The result is a much simpler program listed below:

Program 6.7:

```
main ()
{
  int  i, j, k, match;

  scanf("%d %d %d", &i, &j, &k);
  printf("%d %d %d\n", i, j, k);
  if (i <= 0 || j <= 0 || k <= 0
     || i+j <= k || j+k <= i || k+i <= j) match = 4;
  else if (i == j && j == k && k == i) match =3;
```

```
  else if (i == j || j == k || k == i) match = 2;
  else match = 1;
  printf("%d\n", match);
}
```

7. Redundant Constraints

As demonstrated in the preceding sections, inserting certain state constraints into a program often allows the resulting program to be simplified. The program so produced, however, is generally a subprogram of the original program. To simplify a program, therefore, one needs to simplify a set of its subprograms, and then recompose the program from the simplified subprograms. Now if the inserted constraint is redundant (see Def. 3.14), the resulting program is equivalent to the original one. Therefore, one can insert redundant constraints to simplify a program directly. Given below is a set of relations that can be used for this purpose.

Corollary 7.1: If B C then $S_1;/\backslash B;S_2 \Leftrightarrow S_1;/\backslash B;/\backslash C;S_2$.

Corollary 7.2: For any assignment statement of the form x := c, where c is a constant, x := c ⇔ x := c; /\(x = c).

Corollary 7.3:

if B **then** S1 **else** S2 ⇔ **if** B **then begin** /\B;S1 **end else begin** /\ ¬B ;S2 **end**.
⇔ **if** B **then begin** /\B;S1 **end else begin** S2 **end**.
⇔ **if** B **then begin** S1 **end else begin** /\ ¬B ;S2 **end**.

Corollary 7.4:

(a) **while** B **do** S ⇔ **while** B **do** S; /\¬ B.
(b) **while** B do S ⇔ **while** B **do begin** /\ B; S **end**.

Corollary 7.5:
(a) **repeat** S **until** ¬B ⇔ **repeat** S **until** ¬B; /\¬B.
(b) /\B; **repeat** S **until** ¬B ⇔ /\B; **repeat** /\B; S **until** ¬B.

Theorem 7.6: If (Q B) wp(S, Q) then

(a) /\Q; **while** B **do** S ⇔ /\Q; **while** B **do** S;/\Q.
(b) /\Q; **while** B **do** S ⇔ /\Q; **while** B **do begin** /\Q;S **end**.

Theorem 7.7: If C wp(S, Q) and (Q B) wp(S, Q), then

(a) /\C; **repeat** S **until** ¬B ⇔ /\C; **repeat** S **until** ¬B; /\Q.
(b) /\C; **repeat** S **until** ¬B ⇔ /\C; **repeat** S;/\Q **until** ¬ B.

Theorem 7.8: If C B and $wp(S^{-1}, B) \equiv R$ then

/\C; **repeat** S **until** ¬B ⇔ /\C; **repeat** S **until** ¬B; /\R.

Theorem 7.9: If C wp(S, Q) and (Q B) wp(S, Q), and $wp(S^{-1}, Q \quad B) \equiv R$ then

/\C; **repeat** S **until** ¬B ⇔ /\C; **repeat** S **until** ¬B; /\R.

Corollary 7.10: **repeat** S; /\C **until** ¬ B ⇔ **repeat** S; /\C **until** ¬B; /\C.

Corollary 7.11: /\C; **repeat** S; /\C **until** ¬B ⇔ /\C; **repeat** /\C; S; /\C **until** ¬B.

Note that predicate Q in Theorems 7.6, 7.7, and 7.9 is known as a *loop invariant* in the field of proving program correctness. The utility of these relations can be demonstrated by applying them to the middle segment of the first subprogram of Program 4.13 as shown below:

Program 7.12:

```
        y := 1;
        /\ x <= 100;
        WHILE x <= 100 DO BEGIN x := x + 11; y := y + 1 END;
        WHILE y <> 1 DO
          BEGIN
            x := x - 10;
            y := y - 1;
            WHILE x <= 100 DO BEGIN x := x + 11; y := y + 1 END;
          END;

⇔       y := 1;
        /\ y = 1;                                                    by 7.2
        /\ x <= 100;
        REPEAT x := x + 11; y := y + 1 UNTIL x > 100;
        WHILE y <> 1 DO
          BEGIN
            x := x - 10;
            y := y - 1;
            WHILE x <= 100 DO BEGIN x := x + 11; y := y + 1 END;
          END;

⇔       y := 1;
        /\ x <= 100;
        REPEAT x := x + 11; y := y + 1 UNTIL x > 100;
        /\ x > 100;                                                  by 7.5
        /\ x <= 111;                                                 by 7.8
        /\ y > 1;                                    by 7.7 (y>1 is a loop invariant)
        WHILE y <> 1 DO
          BEGIN
            x := x - 10;
            y := y - 1;
            WHILE x <= 100 DO BEGIN x := x + 11; y := y + 1 END;
          END;

⇔       y := 1;
        /\ x <= 100;
        REPEAT x := x + 11; y := y + 1 UNTIL x > 100;
        /\ x > 100;
        /\ x <= 111;
        /\ y > 1;
        /\ y <> 1;                                                   by 7.1
        WHILE y <> 1 DO
          BEGIN
            x := x - 10;
            y := y - 1;
            WHILE x <= 100 DO BEGIN x := x + 11; y := y + 1 END;
          END;

⇔       y := 1;
        /\ x <= 100;
        REPEAT x := x + 11; y := y + 1 UNTIL x > 100;
        /\ x > 100;
        /\ x <= 111;
```

```
        /\ y <> 1;                                          predicate y>1 is no longer needed
        REPEAT                                                                      by 3.6
            x := x - 10;
            y := y - 1;
            WHILE x <= 100 DO BEGIN x := x + 11; y := y + 1 END;
        UNTIL y = 1;

⇔       y := 1;
        /\ x <= 100;
        REPEAT x := x + 11; y := y + 1 UNTIL x > 100;
        /\ x > 100;
        /\ x <= 111;
        /\ y <> 1;
        REPEAT
            x := x - 10;
            y := y - 1;
            {{{                                                     by 4.11 and 3.6
              /\ x <= 100;
              REPEAT x := x + 11; y := y + 1 UNTIL x > 100;
            '''
              /\ x > 100;
            }}}
        UNTIL y = 1;

⇔       y := 1;
        /\ x <= 100;
        REPEAT x := x + 11; y := y + 1 UNTIL x > 100;
        /\ x > 100;
        /\ x <= 111;
        /\ y <> 1;
        REPEAT
            {{{                                                                     by 4.4
              x := x - 10;
              y := y - 1;
              /\ x <= 100;
              REPEAT x := x + 11; y := y + 1 UNTIL x > 100;
              /\ x > 100;                                                           by 7.5
            '''
              x := x - 10;
              y := y - 1;
              /\ x > 100;
            }}}
        UNTIL y = 1;

⇔       y := 1;
        /\ x <= 100;
        REPEAT x := x + 11; y := y + 1 UNTIL x > 100;
        /\ x > 100;
        /\ x <= 111;
        /\ y <> 1;
        REPEAT
            {{{
              /\ x > 100;                                           by 7.11 and 4.4
              x := x - 10;
              y := y - 1;
              /\ x <= 100;
              REPEAT x := x + 11; y := y + 1 UNTIL x > 100;
              /\ x > 100;
```

```
        '''
          /\ x > 100;                                                    by 7.11 and 4.4
          x := x - 10;
          y := y - 1;
          /\ x > 100;
        }}}
    UNTIL y = 1;

⇔   y := 1;
    /\ x <= 100;
    REPEAT x := x + 11; y := y + 1 UNTIL x > 100;
    REPEAT
        {{{
          /\ x > 100;
          x := x - 10;
          y := y - 1;
          /\ x <= 100;
          x := x + 11;                                                   by 3.4
          y := y + 1;
          WHILE x <= 100 DO BEGIN x := x + 11; y := y + 1 END;
          /\ x > 100;
        '''
          /\ x > 100;
          /\ x > 110;                                                    by 3.12
          x := x - 10;
          y := y - 1;
        }}}
    UNTIL y = 1;

⇔   y := 1;
    /\ x <= 100;
    REPEAT x := x + 11; y := y + 1 UNTIL x > 100;
    REPEAT
        {{{
          x := x + 1;                                                    by 3.16 and 3.18
          /\ x > 101;                                                    by 3.12
          /\ x <= 111;                                                   by 3.12
          WHILE x <= 100 DO BEGIN x := x + 11; y := y + 1 END;
          /\ x > 100;
        '''
          /\ x > 110;                                                    by 7.1
          x := x - 10;
          y := y - 1;
        }}}
    UNTIL y = 1;

⇔   y := 1;
    /\ x <= 100;
    REPEAT x := x + 11; y := y + 1 UNTIL x > 100;
    REPEAT
        {{{
          x := x + 1;
          /\ x > 101;                                                    by 3.5 and 7.1
          /\ x <= 111;
          /\ x > 100;
        '''
          /\ x > 110;
          x := x - 10;
```

```
            y := y - 1;
          }}}
        UNTIL y = 1;

⇔       y := 1;
        /\ x <= 100;
        REPEAT x := x + 11; y := y + 1 UNTIL x > 100;
        REPEAT
          {{{
            /\ x > 100                                          by 3.12
            /\ x <= 110                                         by 3.12
            x := x + 1;
          '''
            /\ x > 110
            x := x - 10;
            y := y - 1;
          }}}
        UNTIL y = 1;
```

This result is identical to that shown in Program 6.4, except the first constraint in the program set here is x > 100 instead of x > 99. The difference results from the use of loop invariant x > 100 in the simplification process here but not in the other. The point to be made here is that Program 6.4 is obtained much more readily through the decomposition-simplification-recomposition process.

8. Concluding Remarks

The notion of a state constraint introduced in this paper provides an effective formalism for several purposes. First of all, one can now speak of the subprogram involved in a particular test-execution. For instance, Program 6.3 is the trace subprogram that produces the result for testing Program 4.12 with x = 97. This trace subprogram can be automatically generated during the test [6], and can be simplified to the one shown below by using the relations presented in Sections 2 and 3.

```
READ (x);
/\ x = 97;
z := 91;
WRITELN ( 'z = ', z );
```

This simplified trace program shows that, in an execution of Program 4.12 with x = 97, even though it will go through a sequence of convoluted steps described in Program 6.3, in effect it does nothing but to assign 91 to the output variable z. Furthermore, Program 6.3 is a subprogram defined for x = 97 only.

Note that the same information can be obtained through symbolic execution [7]. The constraints and statements in Program 6.3 are the branch predicates and statements along the execution path. What is revealed by the reduced subprogram shown above can also be revealed by a symbolic execution along the path. There is, however, no formalism that can be used to describe precisely and concisely the details involved in a symbolic execution. For instance, the path of execution has to be given in terms of a directed graph called the execution tree. The process of symbolic execution has to be explained in terms of illustrative examples. There is no easy way to explain how the path condition can be utilized to simplify the intermediate results.

This paper provides a formalism for describing an execution path in a program, and the computation performed along it, as a subprogram. The corollaries and theorems presented herein can be interpreted as rules that allow one to transform one construct into another. The preciseness and conciseness with which these rules describe what can be done in the process of symbolic execution or program simplification is hitherto unattainable.

Next, if the program contains a loop, and if the loop expands into an infinite execution tree, then the program also decomposes into an infinite number of trace subprograms. A subprogram formed by a particular iteration of the

loop is often defined for only one element in the input domain. For instance, the subprogram shown at the beginning of this section is defined for x = 97 only. For such a program, the usefulness of symbolic execution or trace-subprogram analysis becomes severely limited. The fact that, for input x = 97, the program will assign 91 to z can be much more readily ascertained by a test execution. There is no known method to overcome this difficulty in symbolic execution.

In the present framework, we can overcome this difficulty by expanding the subprogram to include the loop. The expanded subprogram can be simplified as explained in Sections 6 and 7. To fix the idea, let us consider Program 6.4, which is a super program of Program 6.3, and is defined for all x < 100. By using the rules described in the preceding sections, one can simplify it further as shown below:

Program 6.4:

```
        READ (x);
        y := 1;
        /\ x <= 100;
        REPEAT x := x + 11; y := y + 1 UNTIL x > 100;
        REPEAT
            {{{
              /\ x > 99;
              /\ x <= 110;
              x := x + 1;
            '''
              /\ x > 110;
              x := x - 10;
              y := y - 1;
            }}}
        UNTIL y = 1;
        z := x - 10;
        WRITELN ( 'z = ', z );

⇔       READ (x);
        y := 1;
        /\ x <= 100;
        REPEAT x := x + 11; y := y + 1 UNTIL x > 100;
        /\ x > 100;                                    These are the three redundant
        /\ x <= 111;                                   constraints identified in the fourth step
        /\ y <> 1;                                     of analysis of Program 7.12.
        REPEAT
          {{{
            /\ x > 99;
            /\ x <= 110;
            x := x + 1;
          '''
            /\ x > 110;
            x := x - 10;
            y := y - 1;
          }}}
        UNTIL y = 1;
        z := x - 10;
        WRITELN ( 'z = ', z );

⇔       READ (x);
        y := 1;
        /\ x <= 100;
        REPEAT x := x + 11; y := y + 1 UNTIL x > 100;
        /\ x > 100;
```

```
/\ x <= 111;
/\ y <> 1;
REPEAT
  /\ x > 100;                                                    by 7.7
  /\ x <= 111;                                                   by 7.7
  /\ y <> 1;                                                     by 7.5
  {{{
    /\ x > 99;
    /\ x <= 110;
    x := x + 1;
  '''
    /\ x > 110;
    x := x - 10;
    y := y - 1;
  }}}
UNTIL y = 1;
z := x - 10;
WRITELN ( 'z = ', z );
```

⇔

```
READ (x);
y := 1;
/\ x <= 100;
REPEAT x := x + 11; y := y + 1 UNTIL x > 100;
REPEAT
  {{{
    /\ x > 100;                                                  by 4.14
    /\ x <= 111;
    /\ y <> 1;
    /\ x > 99;
    /\ x <= 110;
    x := x + 1;
  '''
    /\ x > 100;                                                  by 4.14
    /\ x <= 111;
    /\ y <> 1;
    /\ x > 110;
    x := x - 10;
    y := y - 1;
  }}}
UNTIL y = 1;
z := x - 10;
WRITELN ( 'z = ', z );
```

⇔

```
READ (x);
y := 1;
/\ x <= 100;
REPEAT x := x + 11; y := y + 1 UNTIL x > 100;
REPEAT
  {{{
    /\ x > 100;                                  by logical simplification
    /\ x <= 110;
    /\ y <> 1;
    x := x + 1;
  '''
    /\ x = 111;                                  by logical simplification
    /\ y <> 1;
    x := x - 10;
    y := y - 1;
```

```
        }}}
      UNTIL y = 1;
      z := x - 10;
      WRITELN ( 'z = ', z );

⇔     READ (x);
      y := 1;
      /\ x <= 100;
      REPEAT x := x + 11; y := y + 1 UNTIL x > 100;
      REPEAT
        {{{
          x := x + 1;
          /\ x > 101;                                                    by 3.12
          /\ x <= 111;
          /\ y <> 1;
        '''
          x := x - 10;
          y := y - 1;
          /\ x = 101;                                                    by 3.12
          /\ y <> 0;
        }}}
      UNTIL y = 1;
      z := x - 10;
      WRITELN ( 'z = ', z );

⇔     READ (x);
      y := 1;
      /\ x <= 100;
      REPEAT x := x + 11; y := y + 1 UNTIL x > 100;
      REPEAT
        {{{
          x := x + 1;
          /\ x > 101;
          /\ x <= 111;
          /\ y <> 1;
        '''
          x := x - 10;
          y := y - 1;
          /\ x = 101;
          /\ y <> 0;
        }}}
        /\ (x > 101   x <= 111   y <> 1   x = 101   y <> 0);             by 4.15
      UNTIL y = 1;
      z := x - 10;
      WRITELN ( 'z = ', z );

⇔     READ (x);
      y := 1;
      /\ x <= 100;
      REPEAT x := x + 11; y := y + 1 UNTIL x > 100;
      REPEAT
        {{{
          x := x + 1;
          /\ x > 101;
          /\ x <= 111;
          /\ y <> 1;
        '''
          x := x - 10;
```

```
    y := y - 1;
    /\ x = 101;
    /\ y <> 0;
  }}}
  /\ (x > 101   x <= 111   y <> 1   x = 101   y <> 0);
UNTIL y = 1;
/\ (x > 101   x <= 111   y <> 1   x = 101   y <> 0);          by 7.10
/\ y = 1;                                                     by 7.4
z := x - 10;
WRITELN ( 'z = ', z );

⇦  READ (x);
y := 1;
/\ x <= 100;
REPEAT x := x + 11; y := y + 1 UNTIL x > 100;
REPEAT
  {{{
    x := x + 1;
    /\ x > 101;
    /\ x <= 111;
    /\ y <> 1;
  '''
    x := x - 10;
    y := y - 1;
    /\ x = 101;
    /\ y <> 0;
  }}}
  /\ (x > 101   x <= 111   y <> 1   x = 101   y <> 0);
UNTIL y = 1;
/\ y = 1   x = 101;                                  by logical simplification
z := x - 10;
WRITELN ( 'z = ', z );

⇦  READ (x);
y := 1;
/\ x <= 100;
REPEAT x := x + 11; y := y + 1 UNTIL x > 100;
REPEAT
  {{{
    x := x + 1;
    /\ x > 101;
    /\ x <= 111;
    /\ y <> 1;
  '''
    x := x - 10;
    y := y - 1;
    /\ x = 101;
    /\ y <> 0;
  }}}
  /\ (x > 101   x <= 111   y <> 1   x = 101   y <> 0);
UNTIL y = 1;
z := x - 10;
/\ z = x - 10;                                               by 7.2
/\ y = 1   x = 101;                                          by 3.12
WRITELN ( 'z = ', z );

⇦  READ (x);
y := 1;
```

```
/\ x <= 100;
REPEAT x := x + 11; y := y + 1 UNTIL x > 100;
REPEAT
  {{{
    x := x + 1;
    /\ x > 101;
    /\ x <= 111;
    /\ y <> 1;
  '''
    x := x - 10;
    y := y - 1;
    /\ x = 101;
    /\ y <> 0;
  }}}
  /\ (x > 101   x <= 111   y <> 1   x = 101   y <> 0);
UNTIL y = 1;
z := x - 10;
/\ z = 91;                                          by logical simplification
WRITELN ( 'z = ', z );
```

Since z is the only output variable, and its value is constrained to 91, it immediately follows that the program can be reduced to the one show below:

```
⇔       READ (x);
        /\ x <= 100;
        z := 91;
        WRITELN ( 'z = ', z );
```

In a traditional test, one can only say that the program prints out `z = 91` as the test result. With recomposition and analysis of trace subprograms illustrated above, one can additionally conclude that the program will produce exactly the same result for any `x` less than or equal to 100. This conclusion is hitherto unreachable through testing.

A basic step involved in proving program-correctness by using the inductive assertion method is to show that an assertion inserted in the program is always true. For construction of proof, assertions can be regarded as constraints. Showing a constraint to be always true (i.e., redundant) can be done as discussed in Sec. 7. Thus proving program-correctness can also be done in the present framework as well. To be more specific, to show that a program S is (partially) correct with respect to input predicate I and output predicate R is to show that constraint R in program `/\I;S;/\R` is redundant.

Most corollaries and theorems in this paper can be interpreted as program verification rules; some of which are restatements of known verification rules in the new formalism. For instance, Theorems 7.6a and 7.7a are a variation of the invariant relation theorem [3]. Thus, the work presented in this paper not only provides an effective method for program analysis but also a unified framework for developing various validation and verification techniques as well.

Acknowledgment

This material is based in part upon work supported by the Texas Advanced Research Program under Grant No. 1014-ARP.

References

[1] R. B. Anderson, *Proving Programs Correct,* Wiley, New York, 1979.

[2] E. Charniak and D. McDermott, *Introduction to Artificial Intelligence,* Addison Wesley, Reading, Massachusetts, 1985, p. 147.

[3] E. W. Dijkstra, A Discipline of Programming, Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

[4] N. Gehani, *ADA: An Advanced Introduction,* Prentice-Hall, Englewood Cliffs, New Jersey, 1983.

[5] J. C. Huang, "A New Verification Rule and its Applications," IEEE Trans. Software Engineering, vol. SE-6, no. 5, Sept. 1980, pp. 480-484.

[6] J. C. Huang, "Instrumenting Programs for Symbolic-Trace Generation," COMPUTER, vol. 13, no. 12, Dec. 1980.

[7] J. C. King, "A New Approach to Program Testing," Proc. of 1975 International Conference on Reliable Software, Los Angeles, Calif., 1975.

[8] W. Leler, *Constraint Programming Languages,* Addison-Wesley, Reading, Massachusetts, 1988.

[9] L. G. Stucki, "Automatic Generation of Self-Metric Software," Proc. IEEE Symposium on Computer Software Reliability, 1973, pp. 94-100.