ANALYSIS OF LOOP PROGRAMS USING GENERALIZATION

by

Glenn Colman

A thesis

submitted to the Victoria University of Wellington
in fulfilment of the
requirements for the degree of
Doctor of Philosophy
in Computer Science

Victoria University of Wellington
1989

ANALYSIS OF LOOP PROGRAMS USING GENERALIZATION

by
Glenn Colman

Submitted to The Department of Computer Science,
Victoria University of Wellington
in fulfilment of the requirements for the degree of
Doctor of Philosophy in Computer Science

Abstract

This thesis describes a symbolic execution system, PAN, that is able to symbolically execute loops. PAN achieves this by generalizing the effect of a few loop iterations to predict the effect of an unknown number of iterations. PAN operates on relatively unstructured loops that include 'go to' type constructs, allowing multiple exits from a loop.

PAN uses a two stage generalization approach using techniques developed in Artificial Intelligence systems. The first stage uses models of expected loop effects and requires only limited search to generalize the effect of simple loops. The second stage uses a less constrained approach that can generalize the effects of more complex loops by using extensive search.

Fundamental to PAN's generalization method is the concept of a sequence. These are identified using models and used in both stages of the generalization process.

Acknowledgements

For their contribution to this work I would like to thank:

Peter Andreae, my supervisor, for much advice and the flexibility to cope with an unusual student.

My parents for their encouragement and for applying for the Kippenburger Scholarship.

The Returned Services Association for awarding me the Kippenburger Scholarship.

Databank Systems Limited for providing flexible employment.

And finally, my family for coping with the whole project.

Contents

Figures

Chapter 1
Introduction

1.1 PROGRAM ANALYSIS

The analysis of an existing computer program to determine what it does is a problem well known to computer programmers. Even when creating a new program from correct specifications, such an analysis is often performed informally as a test of correctness. For existing programs lacking correct specifications, such an analysis often precedes program correction or modification.

This analysis can be seen as the reverse of programming: it will be given a program as input and produce as output a program specification consisting of non procedural statements in a formal logic.

The difficult and unpopular nature of this task has prompted attempts to automate it, especially by the recognition of 'cliches' or stereotyped program fragments.

This thesis examines a new approach to automated program analysis using symbolic execution.

Symbolic execution is a technique for 'executing' a program without assigning specific values to the input data. Instead, such data is given a 'symbolic' value and 'execution' will produce output as a function of the input. This technique has been particularly useful in program verification (i.e. verifying that the program conforms to its specification). However, use of symbolic execution is usually restricted to segments of code which do not allow iteration so as to avoid problems with executing loops.

A symbolic executor steps through the program in a similar fashion to a 'real' executor. The major distinction between these two methods of execution arises when the program branches at a conditional statement. Unlike a real executor, a symbolic executor cannot, in general, decide which branch to take as the condition will involve input data whose value is unknown. The usual method of handling this situation is to allow the symbolic executor to take both branches by producing two sets of output values, each associated with a constraint on the input data sufficient to make the appropriate condition true. However, in the special case where the conditional statement is a loop exit condition, this method leads to the symbolic executor performing an indefinite number of loop iterations. With each new iteration performed, different output statements and associated input constraints are generated, and the analysis task never completes.

Previous attempts to address this problem have focused on solving recurrence relations established during a single loop iteration. The original contribution of this thesis is to suggest that this problem can be approached as a generalization problem. In particular, the output statements and associated input constraints generated from a small number of iterations can be generalized to infer equivalent data for an indefinite number of iterations. This thesis demonstrates that this method overcomes some of the problems associated with solving recurrence relations, particularly the problem of conditional recurrence relations, enabling relatively unstructured loops to be successfully analysed. Additionally, this method allows an improved method of determining loop exit conditions. Thus we can make symbolic execution a more useful program analysis tool by extending the class of programs which can be analysed.

To support this viewpoint, reference will be made to a particular program analysis system, PAN, implemented using the principles described in this thesis.

A program analysis system such as PAN may usefully become a component in a larger computer system. The extended NODDY system, being developed at Victoria University, currently includes a component which automatically generates robot programs from input traces [Andreae 1985]. This system does not currently 'know' what these programs do, in the sense of having non procedural specifications of their effect. An intended use of PAN is to allow the NODDY system to automatically generate programs from traces, and then specifications from the programs.

Alternatively, PAN could find a place in an automated programmer's assistant. Such systems often store programs and their non procedural specifications, and problems arise in introducing new programs without specifications or regenerating specifications from programs manually modified. In both cases PAN could be used to generate the description from the program.

1.2 OUTLINE OF THESIS

The thesis is organized as follows:

- Chapter 1. Section 1.3 introduces some terminology, Section 1.4 presents a scenario showing informally how PAN analyses two programs from different domains. Section 1.5 contains an overview of PAN. Section 1.6 comments on the scope of the analysis task being attempted. Section 1.7 reviews related work.

- Chapter 2 specifies the languages used by programs which can be analysed by PAN, and the output specifications.

- Chapter 3 describes the symbolic execution component of PAN, and the special processing performed to aid the loop generalization and loop exit processing steps.

- Chapter 4 shows how PAN uses the output of symbolic execution to perform loop generalization, determining the effect of an indefinite number of iterations.

- Chapter 5 describes the loop exit processing, used to find the condition required to exit from the loop.

- Chapter 6 identifies cases that normal loop generalization will not handle, and describes a search intensive method used to extend the range of the generalizer.

- Chapter 7 addresses the problems of converting the state of the executor at program exit into an understandable program specification.

- Chapter 8 concludes the thesis with a discussion of the original contributions presented in this thesis and the strengths and weaknesses of PAN.

- The appendix contains further complete examples of programs that PAN has successfully analysed.

1.3 TERMINOLOGY

This section introduces sufficient terminology for the examples and overview presented later in this chapter. Symbolic execution attempts to mimic real execution. A real executor steps through a program testing conditions or making changes to its world (files, variables, blocks etc). It does not need to record either the result of a test or the effect it has had on the world. It only needs to remember the last statement executed so as to be able to continue the execution at the correct point in the program.

A symbolic executor, however, does not have a real world to run in, so must maintain a symbolic description of such a world. If a symbolic execution has reached some statement S in the program, then this description of the world will need to include both the conditions on the world required to reach S, and the cumulative effect of executing all statements up to and including S. Also, instead of having to remember a single 'last statement executed', a symbolic executor needs to control an execution which may be proceeding down several different branches concurrently. These aspects of symbolic execution are described using the following terms:

program	a set of statements, each of which has one or more statements specified as successors.
execution state	a description of the world when symbolic execution has reached some specific program statement S. We say this execution state is associated with S.
execution path = CSP trace	a sequence of execution states from the beginning of the program to some specific program statement
active execution states	the most recently created execution states in any execution path. These execution states are those which can be used to continue the symbolic execution. This is a generalization of a real executor's 'last statement executed'.
path conditions	that part of an execution state that describes the conditions that must be true in the world for execution to have traversed some specific path
effects	that part of an execution state that describes the effect the program has had on the world.

Using this terminology, the symbolic execution proceeds by selecting an execution state E from the active execution states. E will be associated with some program statement S. If S′ is a successor of S, a new execution state E′ is created from E by adding the conditions required to execute S and the effects of executing S. E′ then replaces E in the active execution states. Note that both E and E′ now both exist, E′ has only replaced E in the active execution states, not in the full set of execution states.

1.4 SCENARIO

This section presents one example from each of the two domains that PAN has been tested on, and informally steps through a PAN analysis of them. In order to introduce these examples before PAN has been described, it has been necessary to greatly simplify many details.

ROBOT DOMAIN EXAMPLE

In the robot domain, programs control a robot hand that is able to move and grasp objects if it is in contact with them. The robot is also equipped with sensors so that it can test physical properties of the object contacted (such as color).

The example program is presented in figure 1-2. In this example the robot hand 'finds' objects by repeatedly executing the statement 'move until contact up to (Φ 1)′. Every time this statement is executed the robot hand is at position pos-a. Thus the statement instructs the robot to move at angle Φ from pos-a until either it contacts an object or 1 units have been moved. If we refer to the robot's line of movement by the triple (pos-a, Φ, 1), then we can refer to the objects contacted as 1st from (pos-a, Φ, 1), 2nd from (pos-s, Φ, 1) etc.

Thus the program in figure 1-2 finds all objects from the line (pos-a, Φ, 1). The blue objects are moved to position pos-b and the red objects to pos-c. The effect of this program on the world is shown in figure 1-1. The line at pos-a did contain 8 objects. The first five have already been moved and three remain on the line. The first five have been moved to pos-b if blue and to pos-c if red.

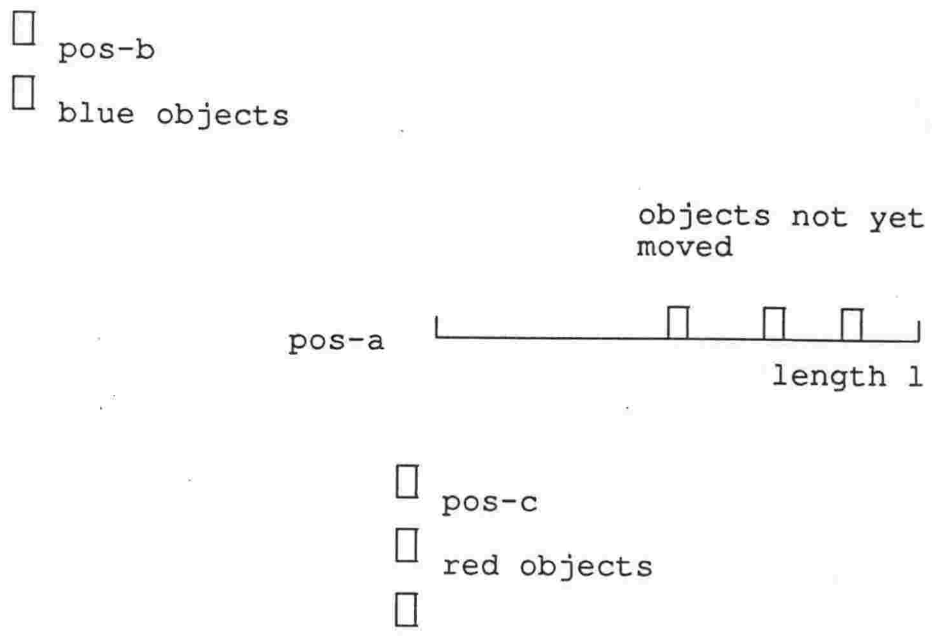


Figure 1-1 Effect of Program on World

We expect PAN to produce the following program specification:

$\forall$ object (object $\in$ (pos-a, Φ, l) $\wedge$ color(object) = blue $\rightarrow$ position(object) = pos-b)

$\forall$ object (object $\in$ (pos-a, Φ, l) $\wedge$ color(object) = red $\rightarrow$ position(object) = pos-c)

Pan's analysis is described in five phases:

symbolic execution before loop generalization
loop generalization
symbolic execution after loop generalization
exit processing
interpretation

SYMBOLIC EXECUTION BEFORE LOOP GENERALIZATION

We describe the first symbolic execution phase by specifying the effect that executing each program statement has on PAN's set of execution states. The program statement numbers refer to those in figure 1-2.

<u>Statement No</u>	<u>Statement Name</u>	<u>Effect on PAN's Execution States</u>
1	start	An execution state is initialized as having no path conditions or effects.
2	move to pos-a	A new execution state is created with effects showing that the robot hand is now at position pos-a.
3	loop entry	PAN performs bookkeeping tasks when a loop is entered, these are not described here.
4	move until contact up to (Φ, 1)	A real executor would execute this statement by moving the robot hand at angle Φ until either an object is contacted or the hand has moved 1 units. Thus this step is conditional - its effect depends upon how many objects are on the line.

Since a symbolic executor cannot determine whether any objects will be found it must handle both possibilities. Thus two new execution states are created. One of them will be made to reflect the case in which the line is empty by adding:

> number of objects on line (pos-a, Φ, 1) = 0

to the path conditions, and

> robot hand at position 1 units at angle Φ from pos-a and not in contact

Statement No	Statement Name	Effect on PAN's Execution States
		to the effects. The other execution state will be made to reflect the case in which the line is not empty by adding number of objects on line (pos-a, Φ, 1) ≥ 1 to the path conditions, and robot hand is in contact with 1st object from line (pos-a, Φ, 1) to the effects.
5	if contact	This statement is asking whether the robot hand is in contact. PAN will symbolically execute this statement in the two different execution states created in statement 4. In the first of these, the line (pos-a, Φ, 1) was empty and the robot is not in contact. Thus this execution state is inconsistent with the condition in statement 5. Consequently, no new execution state associated with statement 5 can be created. In the second execution state, the line (pos-a, Φ, 1) was not empty at statement 4, and the robot hand is in contact. Thus this execution state already contains the condition required to execute statement 5. So

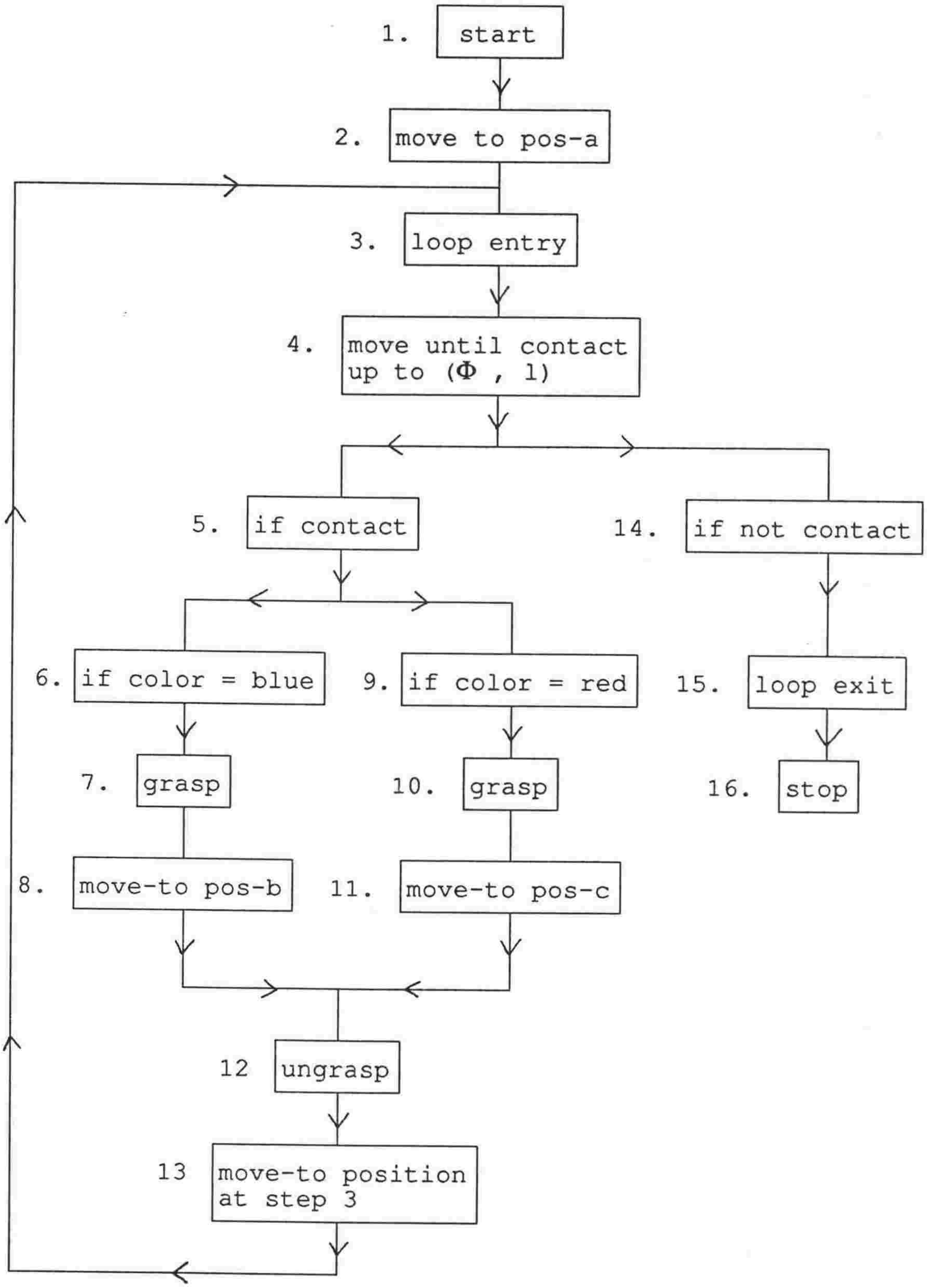


Figure 1-2 - Robot Domain Example

Statement No	Statement Name	Effect on PAN's Execution States
		a new execution state is created which only requires updating of minor bookeeping data..
6	if color = blue	The condition required to execute this statement is neither inconsistent with, nor contained in, the execution state created by executing statement 5. PAN executes this statement by creating a new execution state with color (1st object in line (pos-a, Φ, 1)) = blue added to the path conditions.
7	grasp	A new execution state is created with effects showing that the robot hand is grasping.
8	move to pos-b must specify semantics of every procedure	Since the robot hand is grasping the 1st object from line (pos-a, Φ, 1), a new execution state is created with effects showing that both the robot hand and this object are now at pos-b i.e. position(robot hand) = pos-b position(1st object from line (pos-a, Φ, 1)) = pos-b.
9, 10 11		These statements are executed as for 6, 7, 8, except that the color of the object is red and it is moved to position pos-c instead of pos-b.

Statement No	Statement Name	Effect on PAN's Execution States
12	ungrasp	The execution states established by executing statements 8 and 11 will both be used to execute statement 12. New execution states will be created with effects to indicate that the robot hand is no longer grasping.
13	move to position at step 3 Are these all maintained in separate paths?	Both execution states established by executing statement 12 will be used to execute statement 13. New execution states will be created with effects to show the robot hand is now at position pos-a.
14	if not contact	This is the converse of statement 5. Thus of the two execution states from statement 4, only the one with no object on the line (pos-a, Φ, 1) will be used to create a new execution state associated with this statement.
15	loop exit	When a loop exit statement is reached, PAN checks whether loop generalization has been performed. Since it hasn't, this statement is not executed.
3	loop entry	PAN performs bookkeeping tasks when a loop is entered, these are not described here.

Symbolic execution now proceeds through subsequent iterations of the loop until the required number of iterations has been reached. For the purpose of this example, it will be assumed that three iterations are required.

how do you determine the required number?

LOOP GENERALIZATION

PAN now has a record of the execution states associated with the loop entry statement after zero, one, two and three iterations. From these PAN produces a single generalized execution state which represents the execution state after an indefinite number of loop iterations.

The objects in each execution state are related to the number of iterations as follows:

iteration	objects in execution states
0	none
1	1st object from line (pos-a, Φ, 1)
2	1st and 2nd objects from line (pos-a, Φ, 1)
3	1st, 2nd and 3rd objects from line (pos-a, Φ, 1)

From these facts the generalization process will infer the following fact about a generalized execution state:

iteration	objects
k	1st, 2nd, 3rd,...,kth objects from line (pos-a, Φ, 1)

This sequence of objects is represented by

(sequence i = 1 to k (ith object from (pos-a, Φ, 1))).

which is given the name SEQUENCE-1.

heuristic

Some of the path conditions in the execution states describe objects. The conditions describing objects from SEQUENCE-1 are now used to create subsequences. The appropriate constraints are color(object) = red and color(object) = blue. These are used to generate two more sequences:

specification

SEQUENCE-2 = (object: object ∈ SEQUENCE-1 ∧ color(object) = blue)
SEQUENCE-3 = (object: object ∈ SEQUENCE-1 ∧ color(object) = red)

These sequences are now used to generalize the effects of the execution states. The following relationships are found:

all objects in SEQUENCE-2 have position equal to pos-b
all objects in SEQUENCE-3 have position equal to pos-c

A generalized execution state is now created, with the above relationships on SEQUENCE-2 and SEQUENCE-3 as effects. This execution state represents the effect the program would have after k iterations.

SYMBOLIC EXECUTION AFTER LOOP GENERALIZATION

Why does it again?

The symbolic execution now recommences with the generalized execution state created above. Execution proceeds as follows:

Statement No	Statement Name	Effect on PAN's Execution States
4	move until contact up to (Φ, l)	As in the previous execution of this conditional statement, two execution states are created. One of them will be made to reflect the case in which the line is now empty by adding number of objects in line (pos-a, Φ, l) = k to the path conditions (since k objects have already been obtained from this line), and robot hand at position l units at angle Φ from pos-a and not in contact to the effects.

Stat-ment No	Stat-ment Name	Effect on PAN's Execution States
		The other execution state is will be made to reflect the case in which the line is not empty by adding number of objects from line (pos-a, Φ, 1) ≥ k and robot hand is in contact with (k+1)th object from line (pos-a, Φ, 1).
5-13		Execution then proceeds through these statements beginning with the second execution state created while executing statement 4.
3	loop entry [illegible] generalization was [illegible] correct. What happens [illegible]	When a loop entry statement is reached after loop generalization has been performed, PAN verifies that the loop generalization was correct. This process is not described here.
14	if not contact	PAN will execute this statement using both execution states created while executing statement 4. However, in the second of these execution states, the line (pos-a, Φ, 1) was not empty and the robot is in contact. Thus, this execution state is inconsistent with the condition in statement 15. Consequently, no new execution state associated with statement 15 can be created. In the first of these execution states, the line (pos-a, Φ, 1) was empty, and the robot

	hand is not in contact. Thus, this execution state already contains the condition in statement 15, so a new execution state is created which only requires updating of minor bookeeping data.
15 loop exit	As a loop exit statement has now been reached after loop generalization, exit processing is invoked.

EXIT PROCESSING

The principal function of the exit process is to determine a value for the unknown iteration count k. In this example the single execution state associated with the loop exit statement contains an explicit value for k - the path condition:

k = number of objects in line (pos-a, Φ, 1)

which was added when symbolically executing statement 4. Thus a new execution state is created with this value substituted for k wherever it occurs, in the definition of SEQUENCE-1, and in the path condition for this execution state. Note that this path condition now becomes

number of objects in line (pos-a, Φ, 1) = number of objects in line (pos-a, Φ, 1)

which is always true i.e. there is no path condition required.

INTERPRETATION

The execution state created by exit processing is then used to execute the next statement. However, since this is a stop statement, the execution state is interpreted to produce the required output specification:

$\forall$ object (object $\in$ SEQUENCE-2 $\rightarrow$ position(object) = pos-b)

$\forall$ object (object $\in$ SEQUENCE-3 $\rightarrow$ position(object) = pos-c)

where

SEQUENCE-2 = (object: object $\in$ SEQUENCE-1 $\wedge$ color(object) = blue)

SEQUENCE-3 = (object: object $\in$ SEQUENCE-1 $\wedge$ color(object) = red)

SEQUENCE-1 = (sequence i = 1 to (number of objects in line (pos-a, Φ, 1)) (ith object in line (pos-a, Φ, 1))

Since an object is from line (pos-a, Φ, 1) if and only if it is in SEQUENCE-1, this is equivalent to the required specification.

DP DOMAIN EXAMPLE

In the dp domain, programs process data held in records. The data in each record is divided into fields, each of which can be referenced by name. Records are organized into files and may be accessed sequentially or by key. Existing files may be read or updated and new files created.

The dp domain example program is shown in figure 1-3. Analysis of this program shows PAN operating in a different domain, and also demonstrates PAN's response to a more difficult generalization problem. In the loop generalization phase of the first example, PAN found a simple relationship with which it generalized the effects of the execution states. However, in the current example the search for a simple relationship to explain the value of the variable v fails, and PAN must resort to a more search intensive technique.

The program in figure 1-3 finds the minimum value of the weight field of all records in file A. The reference to 'weight A' in statements 6, 7 and 8 means the value of the weight field for the current record of file A. Given a specific record x, from file A, then (weight x) means the value of the weight field in record x.

We expect PAN to produce the following output specification:

v = minimum({ weight(j): j ∈ file A}).

Program analysis is again described in five phases:

symbolic execution before loop generalization
loop generalization
symbolic execution after loop generalization
exit processing
interpretation

SYMBOLIC EXECUTION BEFORE LOOP GENERALIZATION

Symbolic execution before loop generalization will proceed in much the same way as in the previous example. We merely need to note that the first time through the loop, the condition required to execute statement 8, v < weight(A), will be inconsistent with v being high-values in the execution state. Consequently, statement 8 will not be executed. In subsequent loop iterations, the conditions at statements 6 and 8 will both be consistent with the execution state, and PAN will take both branches, adding suitable conditions to the path conditions of the execution states. Thus after three iterations there will be 8 execution states associated with the loop entry statement. The 4 created on the third iteration will contain:

1. number of iterations = 3,
   path conditions:
      number of records in file A ≥ 3,
      weight(1st record in file A) < weight(2nd record in file A)
      weight(1st record in file A) < weight(3rd record in file A)
   effects:
      v = weight(1st record in file A)

could this be explosive for a large program?

2. number of iterations = 3,

   path conditions:

   number of records in file A ≥ 3,

   weight(1st record in file A) ≥ weight(2nd record in file A)

   weight(2nd record in file A) < weight(3rd record in file A)

   effects:

   v = weight(2nd record in file A)

3. number of iterations = 3,

   path conditions:

   number of records in file A ≥ 3,

   weight(1st record in file A) ≥ weight(2nd record in file A)

   weight(2nd record in file A) ≥ weight(3rd record in file A)

   effects:

   v = weight(3rd record in file A)

4. number of iterations = 3,

   path conditions:

   number of records in file A ≥ 3,

   weight(1st record in file A) < weight(2nd record in file A)

   weight(2nd record in file A) ≥ weight(3rd record in file A)

   effects:

   v = weight(3rd record in file A)

LOOP GENERALIZATION

In the same way that objects were generalized into a sequence in the first example, PAN will generalize the 1st, 2nd, and 3rd records from the file A into a sequence

SEQUENCE-1 = (sequence i = 1 to k (ith record in file A)).

how do objects get assigned to a sequence
objects encountered in an execution path
are they typed?

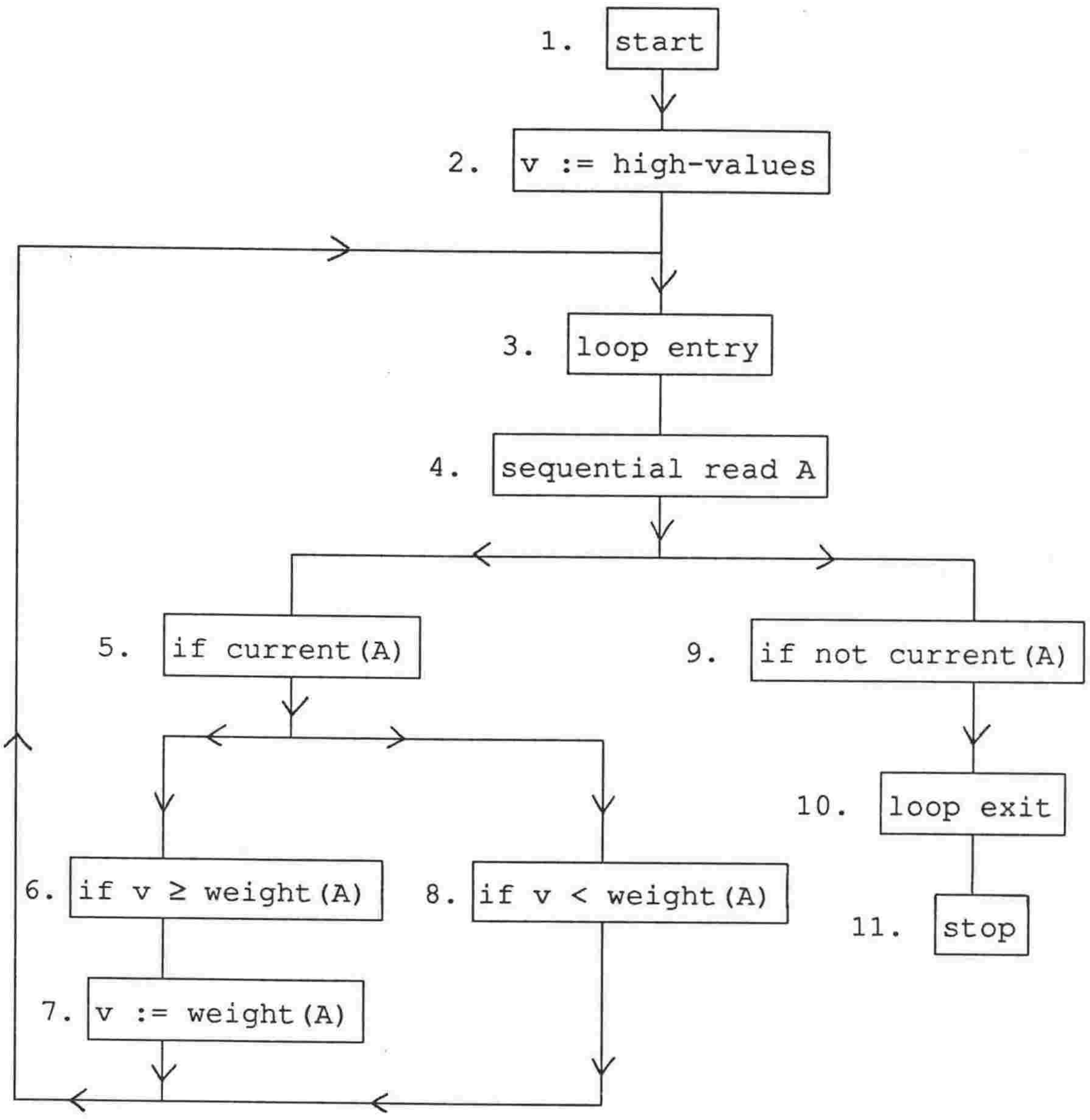


Figure 1-3 - DP Domain Example

PAN will then try to use the path conditions on records in SEQUENCE-1 to generate subsequences. In this example, however, the available conditions are more complicated and involve multiple records. PAN attempts to express these conditions in a form suitable for creating subsequences, but fails. PAN then tries to generalize the effects using the single available sequence. In this case, this means trying to find a simple relationship that expresses the value of v in terms of SEQUENCE-1. This also is unsuccessful because PAN does not include 'minimum' as one of these simple relationships.

Having failed to find a simple relationship giving a value for v, PAN now resorts to a search intensive technique. It begins by expanding the facts in each execution state which has completed at least one loop iteration. This expansion uses a set of rules fully explained in Chapter 6. For this example, however, the following rules are relevant:

heuristics or formal?
constructive induction?

a. given $x = y$ generate $x \leq y$
b. given $x \geq y$ generate $y \leq x$
c. given $x \leq y$ and $y \leq z$ generate $x \leq z$
d. given P(item) and item $\in$ S, generate $\exists y\ (y \in S \wedge P(y))$
e. given P(item) for all items in S, generate $\forall y\ (y \in S \rightarrow P(y))$

where S is a sequence and P is a predicate.

When applied to the facts in the execution state 2, for example, these rules will enable the following facts to be generated:

$v \leq$ weight(1st record in file A)
$v \leq$ weight(2nd record in file A)
$v \leq$ weight(3rd record in file A)

which in turn leads to

$\forall y$ (y $\in$ SEQUENCE-1 $\rightarrow$ v $\leq$ weight(y)).

And from

$$v = \text{weight(2nd record in file A)}$$

PAN will generate

$$\exists y\ (y \in \text{SEQUENCE-1} \wedge v = \text{weight}(y)).$$

When this expansion process is complete, PAN returns to the task of trying to find a relationship to explain the value of v. PAN assumes that any predicate involving v which has been generated using the facts from every execution state, should also be true in the generalized execution state. The conjunction of these predicates provides a relationship involving v. In this example, the only predicates involving v generated in all execution states are:

$$\forall y\ (y \in \text{SEQUENCE-1} \rightarrow v \leq \text{weight}(y))$$

and

$$\exists y\ (y \in \text{SEQUENCE-1} \wedge v = \text{weight}(y)).$$

From these PAN is able to assert that v obeys the following relationship

$$\forall y\ (y \in \text{SEQUENCE-1} \rightarrow v \leq \text{weight}(y)) \wedge \exists y\ (y \in \text{SEQUENCE-1} \wedge v = \text{weight}(y)).$$

PAN now builds a generalized execution state whose output values assert that v obeys the relationship above, or:

$$v \in \{z\text{: } \exists y\ (y \in \text{SEQUENCE-1} \wedge z = \text{weight}(y)) \wedge \forall y\ (y \in \text{SEQUENCE-1} \rightarrow z \leq \text{weight}(y))\}.$$

SYMBOLIC EXECUTION AFTER LOOP GENERALIZATION

Symbolic execution of statement 9 adds

number of records in file A = k

to the path conditions. Execution of statement 10 then initiates exit processing.

EXIT PROCESSING

Loop exit in the first example occurred when line (pos-a, Φ, 1) contained no more objects. In this example an equivalent role is played by file A having no more records. Thus, by the same reasoning, loop exit processing will provide a value for k of (number of records in file A).

INTERPRETATION

PAN will output a program specification containing

$$v \in \{z: \exists y\ (y \in \text{SEQUENCE-1} \wedge z = \text{weight}(y)) \wedge \forall y\ (y \in \text{SEQUENCE-1} \rightarrow z \leq \text{weight}(y))\}.$$

where

SEQUENCE-1 = (sequence i = 1 to (number of records in file A) (ith record in file A)).

Since a record is in SEQUENCE-1 if and only if it is in file A, this is equivalent to:

$$v \in \{z: \exists y\ (y \in (\text{file A}) \wedge z = \text{weight}(y)) \wedge \forall y\ (y \in (\text{file A}) \rightarrow z \leq \text{weight}(y))\}.$$

Why is this different to the expected specification? In the loop generalization section it was stated that PAN does not include 'minimum' as one of its possible relationships. This being the case, we can hardly expect PAN to express its output in terms of a relationship it does not know. However, PAN does the next best thing, and comes up with a relationship for v which could act as a definition of minimum. Thus PAN is not limited to expressing output specifications in terms of known relationships, but is able to use these relationships to create new, more complicated ones.

1.5 PAN OVERVIEW

PAN is a program analysis system in the sense that it accepts a program as input and produces a specification of the program as output. This specification describes the effect the program has on the world. We first discuss how a symbolic execution system, such as PAN, can produce specifications.

PAN, like any symbolic execution system, uses execution states which are descriptions of the world when execution has reached some specific program statement S. We say that an execution state describing the world when execution has reached statement S is associated with S. These descriptions contain two principal components - the path conditions which must be true in the world for execution to reach S, and the effects that the program has had on the world by executing statements up to and including S. Thus, when symbolic execution reaches the end of the program, we will have a set of one or more execution states S1,...,Sn, which are descriptions of the world after the whole program has run.

From S1,...,Sn, program specifications can be produced. Suppose each Si contains path conditions Ii and effects Ei. These execution states represent the program specification

$$(I1 \rightarrow E1) \wedge (I2 \rightarrow E2) \wedge \ldots \wedge (In \rightarrow En)$$

which is the specification output by PAN.

The above discussion shows that the problem of producing a specification can be solved if execution states associated with all program statements can be generated. We now discuss the generation of these execution states, firstly for programs without loops and then separately for loops.

1.5.1 Generation of Execution States for Non Looping Programs

Generation of execution states for programs without loops is not particularly difficult and is a feature of all symbolic execution systems. Since symbolic execution must proceed down multiple branches when a conditional statement is reached, there will be, at any point in the execution, a set of statements in different branches which are the last statement executed in that branch. The execution states associated with these statements are called active, and any one of them may be used to advance the execution.

Thus, the symbolic executor continues execution by selecting an active execution state and finding all successors of the associated statement. For each successor statement a new execution state is created which is a modified version of the selected one. The modifications consist of the following:

- if the program statement is conditional, then the condition from the statement is added to the path conditions

- any changes the program statement makes to the world is added to the effects.

This processing constitutes symbolic execution of a single statement and is accomplished by an 'executor' specific to the type of statement. An executor is a routine which understands how to interpret the parameters of the statement and the effect it has on the world. For example, an executor for the 'move-to x' statement knows that the effect of this statement is that the robot hand is now at x, and if it is grasping, then the grasped object is also at x.

Executors for conditional statements have the added responsibility of checking whether the addition of the new path conditions has made the execution state inconsistent. If such a situation is detected, then the execution state describes a world situation which could not be effected by a real executor, and is discarded.

Once execution of the successor statements is complete, the newly generated execution states replace the selected one in the active set. Repetition of this process will eventually produce execution states which are descriptions of the world after the whole program has run, and can be used to generate the required program specification.

1.5.2 Generation of Execution States for Loops

In this section we discuss the special problems faced by symbolic execution of loops. We can describe a general loop as having a loop entry, a loop body and one or more loop exits, as shown in figure 1-4.

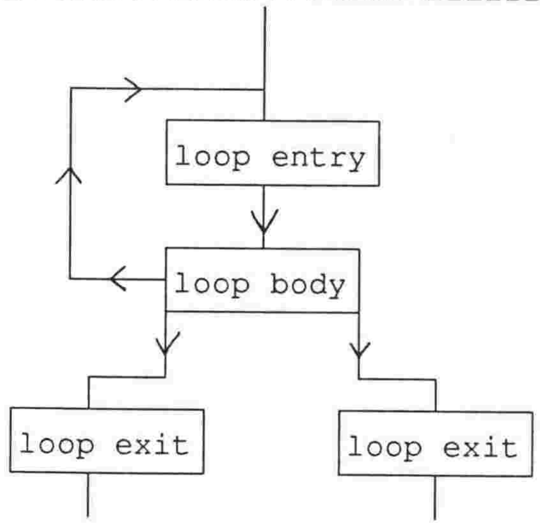


Figure 1-4 General Loop Format

what if program has an infinite loop?

In PAN programs the first statement in a loop is always a special 'loop entry' statement, and the first statements outside the loop are 'loop exit' statements. If a program containing such a loop is symbolically executed, then the process described above is sufficient to ensure that execution reaches the loop entry statement i.e. an execution state associated with loop entry will be created. PAN includes executors for loop entry and loop exit statements which only perform housekeeping tasks as these statements have no effect on the world.

If symbolic execution is continued into the loop body, then eventually condition statements will be reached which determine whether execution proceeds back to loop entry or to one of the loop exit statements. As usual, when condition statements are reached, symbolic execution will continue down all branches. Thus another execution state associated with loop entry and execution states associated with all loop exits will be created. From the execution states associated with the loop exits, symbolic execution can proceed on to the end of the program. From the new execution state associated with loop entry, another symbolic execution through the loop can be performed, and symbolic execution will never end. Execution states associated with the loop exits will be created which describe the effect of the loop after 1, 2, 3... iterations. But no complete description of the loop will be generated and , consequently, no specifications of the program can be produced.

The central contribution of this thesis is to describe a method whereby execution states can be generated that are associated with loop exit and fully describe the effect of the loop. Thus, once symbolic execution has continued from these execution states and reaches program end, full program specifications can be produced.

PAN generates execution states associated with the loop exit statements by a four stage process, as follows:

1/ symbolic execution of the loop is continued until some required number of iterations have been performed

2/ a generalization process is then performed using the execution states associated with the loop entry statement. The result of this process is to produce a single active execution state which represents the effect of an indefinite number k of loop iterations. This process is referred to as loop generalization.

3/ symbolic execution then proceeds beginning with this generalized execution state until execution states associated with each loop exit and a new execution state associated with loop entry have been created.

4/ the execution states associated with each loop exit statement are analysed to produce a value for the number of iterations, k. This process is referred to as exit processing.

5/ the new execution state associated with loop entry is used to verify the generalization.

Loop generalization is performed by the following procedure

- any items referred to in the execution states associated with loop entry are replaced by sequences wherever possible

- the conditions included in any condition statement in the loop are used to suggest subsequences which can be created from these sequences

- the effects of the execution states are expressed in terms of sequences wherever possible

- sequences and the effects expressed in terms of sequences are used to create a single generalized execution state.

Exit processing is performed by analysing the conditions occurring in any condition statements executed between loop entry and loop exit to obtain a value for the number of loop iterations, k.

1.6. SCOPE OF THE PROGRAM ANALYSIS

The scope of a program analysis system is determined by the class of programs it can accept as input and the program specifications which are expected as output.

1.6.1 Input Programs Accepted by PAN

PAN accepts programs from two distinct domains, robot manipulation and data processing (dp). Programs from both domains have the same structure and control statements (statements that control the flow of control through the program). A program is structured as a directed graph. Each node in the graph represents a program statement and the edge between adjacent nodes determines the intended order of execution. A statement may have multiple successors only if each is a conditional statement. Conditional statements are based on Dijkstra's guarded commands [1975]. Thus a conditional statement 'guards' the entry to a program branch. That branch is only executed if the guard is true.

why not use while & do etc

Directed graphs allow the representation of arbitrarily complex and unstructured loops. PAN places some restrictions on these loops, discussed in Chapter 2, but does permit loops that are more complex than structured iterative constructs such as while, do etc.

In addition to the control statements, PAN programs include domain specific statements. In the robot domain these statements control one or more robot hands. In the dp domain they allow records and files to be processed. The robot programs control a blind two dimensional robot. From these programs the size of the objects manipulated cannot be inferred. This has implications for the output specifications which is discussed below.
are

1.6.2 Output Specifications Produced by PAN

We now describe the output specifications in terms of format, correctness and completeness. Output specifications produced by PAN are expressed in the first order predicate calculus, enhanced to include concepts such as objects, sequences of objects, files, robot hands etc. For example, an output specification could consist of

$$\forall \text{object } (\text{object} \in \text{SEQUENCE-1} \rightarrow \text{position(object)} = \text{pos-b})$$
$$\wedge\ \text{SEQUENCE-1} = (\text{sequence } i = 1 \text{ to size(line(pos-a, } \Phi, 1))$$
$$(\text{ith object in line(pos-a, } \Phi, 1)))$$

meaning that the effect of the program is that all objects from the line of length 1 in direction Φ from pos-a end up at pos-b. A more formal description of this language is presented in chapter 2.

The correctness of PAN's specifications depends on the correctness of the execution states from which they are produced. The correctness of these, in turn, depends upon the correctness of the processes which create them, ordinary symbolic execution and loop generalization. Since symbolic execution creates execution states using executors which are an encoding of the effect of each type of statement, we can assume such execution states are created correctly. Loop generalization, however, is more suspect since it is inferring the effect of the loop from the few iterations actually performed.

However, PAN verifies that loop generalization is correct. If the inferred effect of the loop as described by the generalized execution state is E(k), where k is the iteration count, then the effect after k+1 iterations should be E(k+1), where E(k+1) represents the result after replacing k by k+1 in the generalized execution state. PAN performs another loop iteration beginning with the generalized execution state. This will produce new execution states associated with the loop entry statement, which describe the effect after another loop iteration. These can be compared with E(k+1). Any generalization which is not verified is discarded. This verification process allows us to assert that PAN specifications are correct.

could it fail?

PAN specifications are correct, but they are not guaranteed to be complete. We may distinguish between two sources of incompleteness: limitations of PAN which could be corrected, and fundamental limitations of the programs being analysed.

In the first category, PAN will not be able to express specifications in terms of concepts which it does not know. For example, the specifications for the dp domain program in section 1.4 should ideally, be expressed in terms of the minimum function. Since PAN does not know this function, it produced a less than ideal specification. Also in the first category, PAN's analysis will be incomplete if loop generalization fails to express the effects of the loop in terms of the sequences it has generated. Experience with PAN has shown that failures of this type usually occur when analysing programs deliberately constructed to have an obscure structure which even humans find difficult to analyse. This point is further discussed in Chapter 4.

A limitation fundamental to the programs being analysed is PAN's inability to recognise identical objects in the robot domain. For example, suppose a program first moves all objects from line 1 to line 2 and later moves all objects from line 2 to line 3. PAN's specifications for such a program would be:

```
all objects in line 1 moved to line 2
all objects in line 2 moved to line 3
```

instead of

```
all objects in line 1 moved to line 3.
```

However, in general, repeated contact with the same object is impossible to detect in a program designed to control a blind robot. Since objects may be of arbitrary shape, two separate contacts could always be with the same object.

Also in the first category is that PAN does not determine preconditions for robot statements. For example, move statements assume that there is free space between the current position and the destination. However, since the size of the robot hand and other objects in the environment is unknown, the amount of free space required to complete a move cannot be determined. For this reason, determination of the preconditions of the program was excluded from the scope of PAN.

preconditions are important.
couldn't this have been added?

1.7 RELATED SYSTEMS

In discussing work related to PAN, we need to consider PAN's goal and the method used to achieve that goal. PAN's goal is to analyse existing programs without assuming the existence of specifications. The method that PAN uses to achieve that goal is symbolic execution.

Most closely related to PAN are other symbolic execution program analysers, since they have the same goal and use the same method to achieve their goal. Less closely related are systems which perform program analysis by other methods. Only distantly related are systems which use symbolic execution to achieve other goals. We use these three categories in our discussion of related systems.

Program analysis systems which do not use symbolic execution have to directly analyse the input program, and this affects system performance. Symbolic execution systems are less sensitive to the vagaries of coding style - as long as the effects are the same, the same analysis will ensue. Systems that directly analyse the input program perform better at recognising known code fragments (cliches).

1.7.1 Other Symbolic Execution Program Analysers.

A system closely related to PAN is that of Cheatham, Holloway and Townley [1979]. This system extends previous symbolic execution analysers by attempting to handle loops with an indefinite number of iterations. The method used is to derive recurrence relations between variables modified in the loop by symbolically executing a single loop iteration.

For example, given a loop which simply increments a variable v, they define the value of v after k iterations to be v_k. They now symbolically execute the loop and show that

$$v_{k+1} = v_k + 1.$$

They then use prespecified rules for solving common recurrence relations to show

$$v_k = v_0 + k,$$

where v_0 is the value of v before the loop was executed.

Problems arise with this method when several non independent recurrence relations need to be solved, or the equations are conditional.

Cheatham et al solve these problems for limited special cases. For example, they show conditional recurrence relations can be solved if it is possible to find ranges of k within which the relations become non conditional.

However, this technique will not work for the types of programs being considered in this thesis. Consider the first example in section 1.3. If recurrence relations were derived for this program they would be conditional on properties of the objects being processed, and we could not find ranges of k within which the equations would be unconditional.

Even worse, we are interested in programs in which the condition contains a variable which also occurs in the recurrence relations. The second example in section 1.3 is of this type. If such a program were used to find the minimum value of an array A, we would have the recurrence relation:

$$v_{k+1} = \begin{cases} v_k \text{ if } v_k < A[k] \\ A[k] \text{ if } v_k \geq A[k]. \end{cases}$$

While, obviously a specific rule to recognize this particular case could be introduced, the difficulty of solving such recurrence relations generally is a major reason why the PAN system uses the alternative method of generalizing from a small number of iterations.

The same techniques for solving recurrence relations were still being used by Richardson and Clarke [1985].

The CAN system of Goosens [1979, 1981] also solves recurrence relations to determine the effect of k loop iterations. This system assumes that loops will be represented as recursive calls to the procedure being analysed. The result of symbolically executing the procedure is assumed to result in interim results of the form:

```
    if path condition 1 then effect 1
    if path condition 2 then effect 2
    . .        .      .   .   .     . .
    . .        .      .   .   .     . .
    if path condition n-1 then effect n-1
    if path condition n then recursive procedure call.
```

In other words the procedure is assumed to terminate except in the case of a single path condition. Thus this system can only analyse programs presented in a suitable language. If the type of programs analysed in this thesis were represented in a language suitable for CAN, they could still not be analysed since multiple recursive procedure calls would be required. On the other hand, the CAN system is particularly concerned with extending the symbolic execution technique to LISP type languages which allow variables (and other entities) to be 'meta-described'. Thus in a LISP assignment statement such as SET, the left hand side may not simply refer to a specific variable but may itself be a conditional expression whose value cannot be determined by a symbolic executor. This extension is outside the scope of both PAN and the Cheatham et al system.

Both the CAN and Cheatham et al systems analyse loop exit conditions by including them in the set of recurrence relations to be solved. This contrasts with PAN, which first finds a generalized execution state and then separately determines the condition required to exit from the loop. This approach allows PAN to cope with programs having multiple loop exits, which do not appear to have been allowed for in any previous system.

A less closely related system is that of Cohen [1983], which symbolically executes specifications instead of programs. In this case only 'simple loops' are allowed, ones in which 'the same thing is done to each of a set of objects'. In this way the main issues being addressed in this thesis are avoided. Similar comments apply to the symbolic execution of specifications in Kemmerer [1985].

1.7.2 Other Program Analysers

A significant project at MIT in recent years has been the Programmer's Apprentice [Rich, Shrobe and Waters, 1979]. One aspect of this project concerns the analysis of existing programs. Analysis begins by obtaining a program description in terms of plan building methods (PBMs) [Waters, 1979]. Thus a loop of the form

```
do i = 1 to 10
if A(i) < 0
     A(i) = 0
od
```

would be divided up into a basic loop - do i = 1 to 10, a 'filter' A(i) < 0, and an 'augmentation' A(i) = 0. Also, the 'temporal sequences' A(i) i = 1 to 10 and $\{A(i): 1 \le i \le k \land A(i) < 0\}$ are recognized. These temporal sequences are equivalent to the sequences produced in a PAN analysis. However, in this early work the analysis output was only concerned with being able to describe a program in terms of its parts (i.e. PBMs). This left unresolved the question of whether PBMs could be used to generate program specifications. Indeed, as Waters [1979] states 'it can be arbitrarily difficult to determine what a basic loop does given the behaviour of its parts'.

In order to produce program specifications two further stages have since been developed. First, the PBM output, called Plan Calculus, is converted into a 'flow graph', which does not contain loops [Brotsky, 1984]. The looping connections have been 'cut' and replaced with annotations stating their relationships.

The second stage parses the flow graph and produces a program description, by matching the parse output against plans from a prespecified library [Zelinka 1986]. This system has a similar goal, but a very different approach, from the PAN system. The reliance on a plan library has the advantage that a new program can be recognized as the same or similar to an earlier one. The disadvantage is that only limited analysis can be performed for a program whose 'parse tree' fails to match an existing plan. In this case a parse is started at every non terminal node of the flow graph in order to at least produce a partial program description. This produces some good results, but as stated by Zelinka [1986], the examples used only work because the successfully analysed fragments 'are disjoint...connected by data flow with no unrecognizable sections in between'.

1.7.3 Other Uses of Symbolic Execution

An extensive number of symbolic execution systems have been developed. However, since these do not generally deal with program analysis of loop programs with unknown specifications, they are only distantly related to PAN. Consequently, only a small sample of these systems is discussed here.

For a general introduction into the use of symbolic execution see, for example, Clarke and Richarson [1981]

The DESIGNER system of Steir and Kant [1985] uses symbolic execution to test program fragments against expected inputs/outputs and time constraints. This system does not currently handle an indefinite number of loop iterations. However, in considering possible future work they state

'People, on the other hand, usually recognize a correct loop when a small number of test cases work, and we wish to capture this ability to recognize familiar patterns from the structure of the algorithm and from symbolic or test case execution in our system.'

Although PAN is primarily concerned with deriving program specifications, rather than testing programs against expected results, it goes some way to providing such an ability for symbolic execution.

The tutoring system of Laubsch and Eisenstadt [1981] uses symbolic execution to analyse student programs and compare their effect with an expected solution. Loops are handled by deriving recurrence relations as described in 1.6.1, but in this case the relations are looked up in a database containing patterns of expected relations.

The system of Dannenburg and Ernst [1982] is representative of systems using symbolic execution to prove that a loop obeys prestated loop invariants. Such systems have only a distant relationship to PAN.

Chapter 2

Representation

As a program analyser, PAN accepts programs as input and produces specifications as output. This chapter describes how the input programs and output specifications are represented.

2.1 PROGRAM LANGUAGE

PAN was primarily developed to explore the effectiveness of generalization as a program analysis method. Thus, to be convincing, PAN must analyse programs which include those features which make program analysis difficult - loops and variables. Also, to demonstrate the domain independence of the generalization process, PAN must analyse programs from multiple domains.

A secondary goal of the PAN system is to act as a component of the Noddy system being developed at Victoria University. As such, PAN is expected to analyse Noddy programs.

To meet both these goals, PAN has been designed to analyse programs in a language which is an extension of that generated by the Noddy system. The Noddy system programs are limited to the robot domain, and do not allow variables. The extended program language used by PAN allows variables and also applies to the dp domain. The language is for side-effect programs - programs that manipulate or modify external objects in some world outside the program, rather than manipulating values internal to the program. In this sense, it is the complete opposite of a functional language. A consequence is that it has no data structures and programs are not intended to make much use of internal variables.

what affect does this have on complexity?

The remainder of this section describes the structure of PAN's analysis language, discusses the expressiveness of this language and then describes the syntax and semantics in detail.

2.1.1 Structure of PAN's Program Language

Why not a structured language?

Like the Noddy system, PAN's program representation is based on flow diagrams. A flow diagram consists of boxes connected by arrows representing the flow of control. Each flow diagram box contains either a *condition*, an *action* or a *flow control marker*. The box, together with its contents, is referred to as a *statement*. Two flow control marker statements are start and stop. Each program contains a single start statement and one or more stop statements. Each statement, except the stop statement, has one or more arrows proceeding from it. Each statement except the start statement has one or more arrows pointing to it. The *successors* of a statement S are the statements pointed to by the arrows out of S and the *predecessors* of S are the statements that have arrows pointing at S.

For example, figure 2-1 shows a simple program containing seven statements. Statements 1, 2, 5 and 7 are flow control markers. Statements 3 and 4 are condition statements and 6 is an action statement.

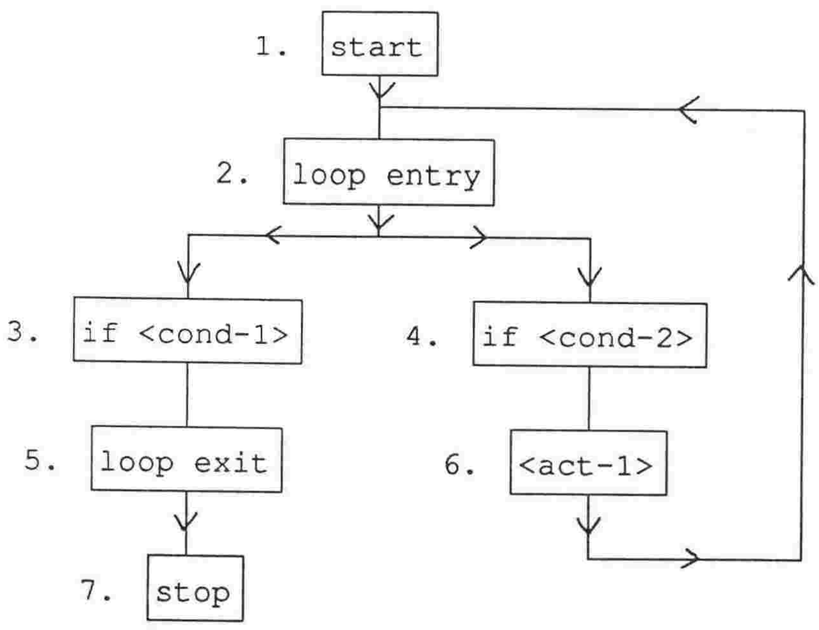


Figure 2-1 A Simple PAN Program

A program is executed by beginning at the start statement and following the arrows to the successors until a stop statement is reached. Given two statements S and S′, we say there is a *path* from S to S′ if we can find statements S1,...,Sn such that S1 is a successor of S, Si+1 is a successor of Si, and S′ is a successor of Sn. For every statement S there must be a path from the start statement to S, and a path from S to one of the stop statements. The flow control markers would be ignored by a 'real' executor - they are purely for the benefit of a symbolic executor and are discussed later in this chapter. An action statement is executed by performing the specified action. Condition statements are of the form if <condition>. They are executed by evaluating <condition>. If <condition> is true then control continues on to the successors to the condition statement. Conditional branching can therefore be represented by multiple arrows out of one statement with each successor statement being a condition statement, as shown in figure 2-2.

Do we assume programs are correct

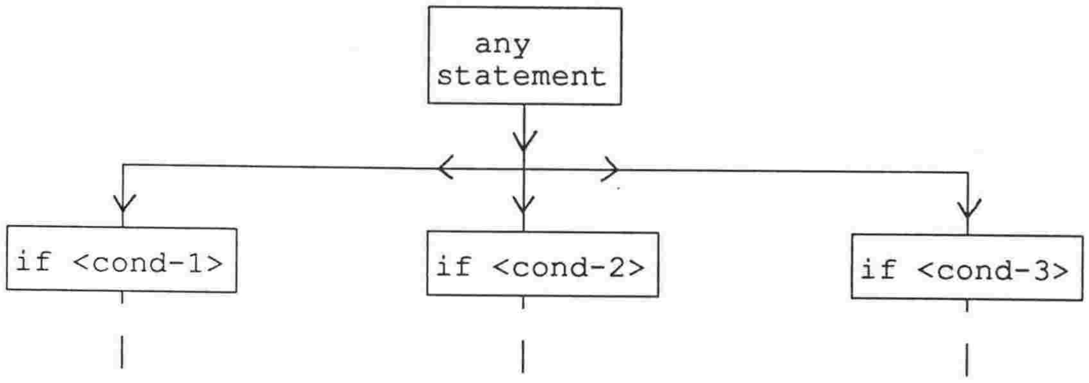


Figure 2-2 Conditional Branching

A statement with multiple arrows going out is said to be *at a fork* in the program (statement 2 in figure 2-1). A statement with multiple arrows going in is said to be *at a merge* in the program (also statement 2 in figure 2-1 - though a statement which is at a fork does not have to be at a merge). A *branch* in a program is a linear sequence of statements beginning at the start statement, a statement at a merge, or a statement immediately following a statement at a fork, and continuing to a stop statement, a statement at a fork or a statement immediately preceding a statement at a merge.

We insist that all successors of a statement at a fork are condition statements. These act as guards or gates into a branch. The construct for conditional branching described above is similar to guarded commands, introduced by Dijkstra[1975]. Dijkstra intended that the construct be non deterministic - an executor can continue down any one of the branches with a true guard. Programs generated by the Noddy system also implement conditional branching using this construct, but the condition statements are ordered and execution will continue down the first branch with a true guard. For a symbolic execution system, such as PAN, these distinctions are not significant, since generally the guard conditions cannot be proved either true or false, and execution will proceed down all branches.

Iteration is represented by a loop in the flow diagram. In a loop, there must be at least one statement at a fork and one of the statements following the fork must be external to the loop if the program is to be able to halt successfully.

2.1.2 Restricting Loop Structure

The flow diagrams as presented in the previous section can represent a wide range of iterative constructs, including unrestricted use of 'go tos'. This is deliberate, since a major goal of PAN is to investigate the problems of analysing loops with limited structure - in particular, loops with multiple unstructured exit paths.

However, PAN is not able to analyse completely unstructured loops. PAN requires the following (informal) restrictions to be placed on loops:

- each loop can only be entered via a single statement

- this 'entry statement' is executed once for each iteration of the loop.

The benefits that can be derived from these restrictions are developed in chapters 3 and 4. We now express these requirements more formally as restrictions on the flow diagrams to be analysed by PAN. An earlier attempt to represent PAN's input language as a formal grammar was abandoned because of the difficulty of representing these restrictions.

could have done it

We first need to introduce our loop terminology. We want to be able to specify loops by identifying the first and last statements in a loop, referred to as loop entry and loop exit. If we ignore nested loops, the loop entry statement of a loop is the statement S such that there is a path from S back to S, but this is not true of any of S's predecessors. The loop exits of a loop are those statements from which there is no path back to the loop entry, while from all predecessor statements there are paths back to the loop entry. The formal definitions of loop entry and loop exit which follow are complicated by having to cope with nested loops. This is done by using inductive definitions. We say that a statement in a flow diagram is *at a loop entry* if it obeys the following inductive definition

- a statement S is at loop entry at step 0 if there is a path from S to S and there is a predecessor, S' of S such that there is no path from S' to S'

- a statement S is at loop entry at step n+1 if there is a path from S to S which does not pass through any statements at loop entry at steps 0 to n, and S has a predecessor, S' such that there is no path S' to S' except via statements at loop entry at steps 0 to n.

- a statement S is at loop entry if there is some n such that S is at loop entry at step n.

made awful by choice of flowchart

grammatical description using recursive description of statements would have made this a lot easier

For example, in figure 2-3, S2 will be identified as at loop entry at step 0, as there is a path from S2 to S2 and S2 has a predecessor, S1, such that there is no path from S1 to S1. S3 and S4 will not be identified as at loop entry at step 0 because although there are paths from S3 to S3 and from S4 to S4, all the predecessors of S3 and S4 also have paths back to themselves.

However, at step 1 S4 will be identified as at loop entry since there is a path from S4 to S4 which does not go through S2 and this is not true for S4's predecessor S3.

We say that a statement S' in a flow diagram is *at loop exit for S* where S is a statement at loop entry if it obeys the following inductive definition

---

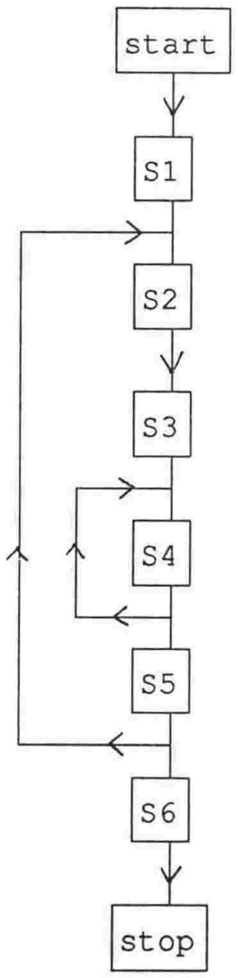


Figure 2-3 Identifying Statements at Loop Entry and Exit

- S′ is at loop exit for S at step 0 if there is no path from S′ to S and from every predecessor of S′ there is a path back to S

- S′ is at loop exit for S at step n+1 if there is no path from S′ to S except through another loop entry statement that has a loop exit identified at steps 0 to n, and from every predecessor of S′ there is a path back to S that does not go through another loop entry statement that has a loop exit identified at steps 0 to n.

- S′ is at loop exit for S if there is some n such that S′ is at loop exit for S at step n.

Thus referring again to figure 2-3, statement S6 is at loop exit for S2 at step 0, and statement S5 is at loop exit for S4 at step 1.

Given a statement S at loop entry we define the *loop beginning at S* as all statements S′ which obey the following:

- there is a path from S to S′ without going through a statement at loop exit for S.

Given these definitions, we can now precisely specify the loop restrictions that PAN places on flow diagrams:

Loop Restriction 1:
the only statement in a loop beginning at S which has a predecessor outside the loop is S.

Loop Restriction 2:
if a loop L beginning at statement S′ contains a statement S, such that there is a path from S to S within L that does not include S′, then S is contained in a loop beginning at some statement S′′ in L.

In fact, restriction 2 is a consequence of our the definition of a loop. To prove this we first need the following lemma.

Lemma: any path from statement S back to statement S goes through a statement S′ at loop entry, such that there is no loop exit for S′ on the path.

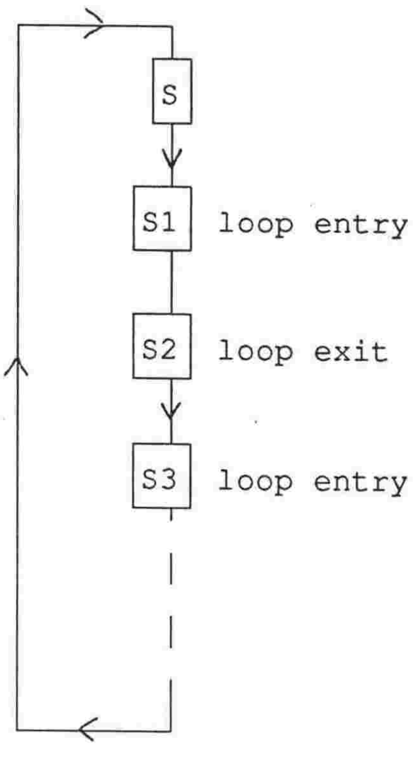
Proof:

We first prove that the path contains some statement at loop entry. We prove this by contradiction. Suppose that the path from S back to S does not contain any statement at loop entry. Then by the definition of at loop entry, all predecessors of S must have the same property, or otherwise S would have been defined as at loop entry. Continuing this reasoning, all predecessors of the predecessors must also have this property etc. Since by the definition of the start statement, there is a path from start to S, then we eventually have to conclude that the start statement has this property, which is not possible since the start statement is not permitted any predecessors.

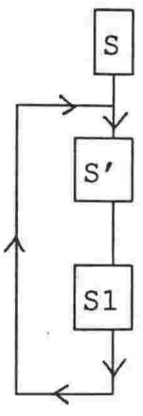
We now know that the path from S to S must include at least one statement at loop entry, say S1. We prove the remainder of the lemma by showing that if the proposition is not true there must be an infinite number of statements on the path S to S. If S1 does not have an exit on this path then the lemma is proved. Suppose, conversely, S1 does have an exit on the path, say S2. For S2 to have been defined as at loop exit, then any path from S2 to S1 must be via a loop entry statement which had previously had loop exits found, say S3.

If S3 does not have an exit on the path, then again the lemma is proved, but if it does, then we can again show that another loop entry exists on the path which had exits found before S3. This reasoning can be continued indefinitely, as shown in figure 2-4(a).

Since there cannot be an infinite number of statements on the path S to S, one of them must have no exit on the path as required.



(a)



(b)

Figure 2-4 (a) Proof of Lemma (b) Proof of Loop Restriction 2

Proof of Loop Restriction 2:

Suppose that the loop L starting at statement S does have a statement S′ with a path from S′ to S′ which does not include S, and S′ is not in any loop L′ in L. Then by the lemma, there is a statement S1 on the path S′ to S′ which does not have an exit on the path (see figure 2-4(b)). By the definition of the loop starting at S1, S′ is in this loop, as required.

The expressiveness of this restricted language is discussed in section 2.1.5.

2.1.3 Restricting Conditional Branching

The flow diagrams presented in section 2.1.1 can represent a wide range of conditional constructs, including unrestricted use of 'go to's. The method that PAN uses to analyse loops requires that some restrictions are placed on the structure of conditional branching. PAN requires the following (informal) restrictions to be placed on conditional branching:

like guarded commands [illegible] [illegible] program

- in all cases at least one of the condition statements following a statement at a fork must have a true condition

- conditional branching must be cleanly nested - two conditional branches cannot overlap. [illegible] mutually exclusive ?

The benefits of these restrictions are developed in Chapters 3 and 4. To give a graphical representation of the second, more complicated restriction, consider the program fragments in figure 2-5. We allow the fragment in 2-5(a). Thus we do not insist that all paths starting at S1 merge back at the same point. But we do not allow the fragment in 2-5(b) because the conditional branching beginning at S2 overlaps that beginning at S1. In order to motivate the formal statement of conditional branching restrictions, note that one way of characterising what is 'wrong' with 2-4(b) is 'there are paths beginning at S2 and S3 which merge at S5 but there is a path beginning at S2 which does not pass through S5'. In other words, we want to restrict conditional branching so that all paths from S2 must pass through S5. Now if the whole of the conditional branch is in a loop, all paths from S2 will pass through S5 by performing another loop iteration and then executing the path S1, S3, S5. This still does not make the conditional branch in 2-5(b) acceptable, so we need to state the restriction on conditional branching so that all paths from S2 pass through S5 without performing another loop iteration.

We now want to express the above requirements for conditional branching as restrictions on the flow diagrams to be analysed by PAN. We first need to introduce our conditional branching terminology. We have previously defined a statement being at a merge and at a fork. We now relate these two concepts. Let the successors to the statement S at a fork be the condition statements S1,...,Sn. We say that a statement S′ which is at a merge is *at a merge for the fork at S* if:

- there are paths from at least two of the Si's to S′ without going through loop entry or loop exit of any loop S is in

- no predecessors of S′ have this property for the same Si's.

Given this definition, we can now specify precisely the conditional branching restrictions that PAN places on flow diagrams as

---

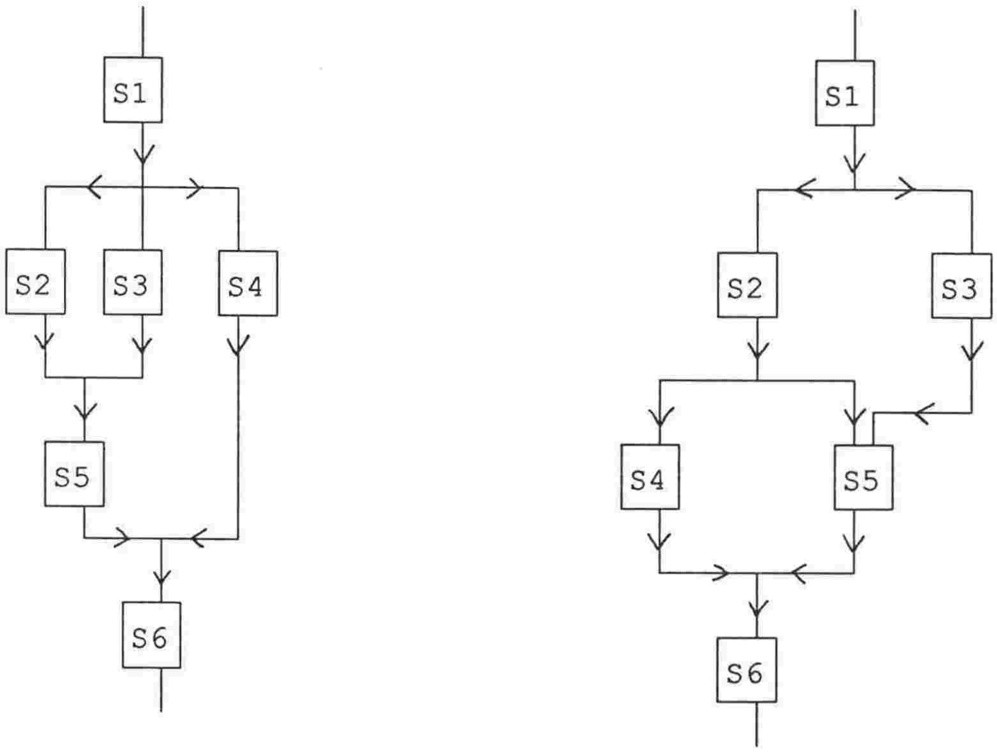


Figure 2-5 (a) Cleanly Nested and (b) Not Cleanly Nested Conditional Branches

Conditional Restriction 1:

If a statement S at a fork has successors S1,...,Sn, with conditions C1,...,Cn, then C1 ∨ C2 ∨ ... ∨ Cn = T

Conditional Restriction 2:

if S′ is at a merge for the statement S at a fork, with successors S1,...,Sn, then if there is a path from an Si to S′, then every path starting at Si can be extended to include S′ without going through statements at loop entry or loop exit for any loop that S is in.

Conditional Restriction 3:

a statement cannot be at a merge for two forks.

We now show some of effects that these restrictions have on the structure of conditional branches. Firstly we present a theorem which shows that the conditional branching restrictions ensure that a conditional branch starting at a fork SF can only be entered via SF.

Theorem 2-1: Given a statement SM at merge for the statement SF at a fork, there is no path from the start statement to SM except via SF.

Proof: Suppose the contrary is true. Then there are paths P1 and P2 from the start statement to SM, P1 via SF and P2 not via SF. Let the first place where these paths deviate be the statement SF2, which must be at a fork. Thus SF2 has condition statements SC1 and SC2 as successors, so that SC1 is on P1 and SC2 on P2. Now there are paths from SC2 and SC1 to SM. But SM cannot be at a merge for SF1, by the conditional branch restrictions since it is at merge for SF. Therefore there must be some predecessor of SM such that there are paths from SC1 and SC2 to this predecessor. By looking at predecessors, predecessors of predecessors etc of this statement we must eventually find a statement S, which is at merge for SF2. Since by the conditional branch restrictions all paths from SC1 and SC2 go through S, P1 and P2 do. We now have the situation shown in figure 2-6, though it is not known where S is on the path SF2-SC1-SF-SM.

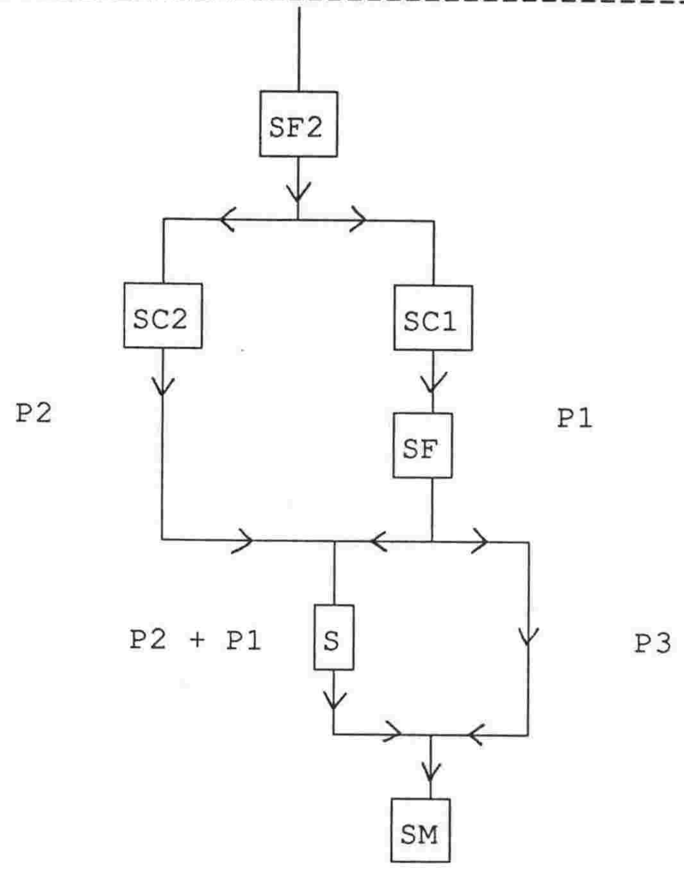


Figure 2-6 A Conditional Branch Entered at Non Fork.

We complete the proof by showing that S must be equal to SM. First suppose that S is before or equal to SF. This violates conditional branch restrictions for conditional branch SF2

because then there is a path from SC2 to S, but also a path, P2 from SC2 which does not go through S. (If P2 went through S, then since S is before SF, P2 would also go through SF, contrary to the definition of P2). Alternatively, if S is after SF, but not equal to SM, then choose some path, P3, from SF to SM which does not go through S. This is possible since otherwise SM would not be at merge for SF since any successors of SF which have paths to SM would also have paths to the predecessor of SM on the path from S to SM. But now the path SF2-SC1-SF-P3 does not go through S, which violates conditional branch restrictions for conditional branch beginning at SF2, since there is a path from SC1 to S, which requires all paths from SC1 go through S. Thus S must be equal to SM, which violates the restriction that a statement can only be at merge for one statement at a fork.

For the next result we need to define the point where all branches of a conditional branch merge. We say that a merge statement SM is the *last merge* for a statement SF at a fork if SM is at a merge for SF and there is a path from every condition successor of SF to SM without going through any loop entry or exit for any loop SF is in.

The last merge can be thought of as the end of the conditional branch. We now prove our main result on the structure of conditional branches. The following theorem shows that when we have nested conditional branches, any inner conditional branches end before merges are encountered for an outer conditional branch.

Theorem 2-2: If statement SM is at a merge for the statement SF at a fork, and a path from SM to SF passes through a statement SF2 at a fork, then this path also contains a last merge for SF2.

Proof: Every path through SF2 must go through SM by the conditional branch restrictions. Also by these restrictions, SM cannot be at a merge for SF2. Thus there must be a predecessor of SM which also has the property that all paths from SF2 pass through it. Look at predecessors, predecessors of predecessors etc of SM until one if found which has all paths from SF2 going through it but its predecessor doesn't. This is a last merge for SF2 as required.

These theorems are used in Chapters 3 and 4. The expressiveness of this restricted language is discussed in section 2.1.5.

2.1.4 Flow Control Markers

As stated in section 2.1.1, PAN's program statements are either conditions, actions, or flow control markers. We now describe what the flow control markers are, why they are required, and the extent to which they reduce the generality of PAN's input language.

The flow control markers used by PAN are:

- loop entry
- loop exit
- merge
- start
- stop.

In section 2.1.1 and 2.1.2 we defined what is meant by a statement being at loop entry, at loop exit and at merge. These concepts have been deliberately defined to have names corresponding to flow control markers. In fact, we want to be able to refer indiscriminately to a statement being at loop entry (loop exit, merge) or being a loop entry (loop exit, merge) statement. Thus we require:

all statements at loop entry are loop entry statements. All loop entry statements are at loop entry.

all statements at a merge but not at loop entry are merge statements. All merge statements are at a merge.

all statements at loop exit are succeeded by one loop exit statement for each loop being exited. All loop exit statements succeed a statement at loop exit.

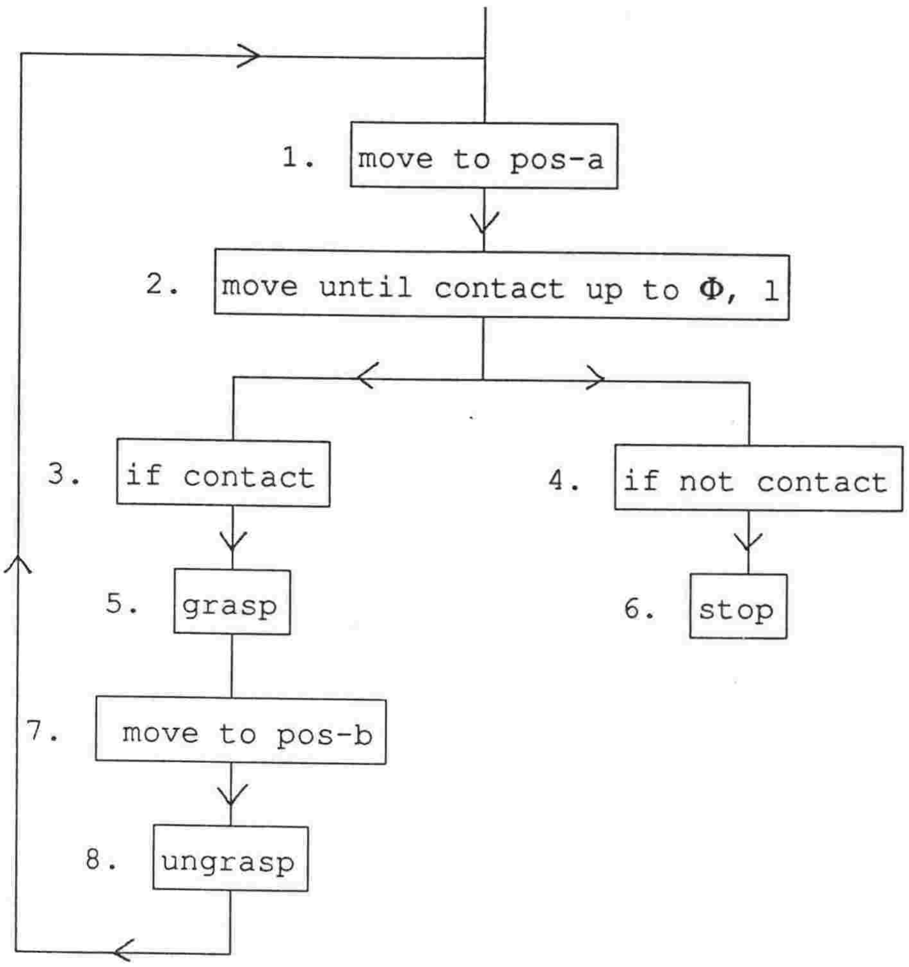


Figure 2-7 (a) Program Fragment Without Flow Control Markers

---

structured language

The flow control marker statements loop entry, loop exit and merge would be ignored by a real executor, so why are they required in PAN's analysis language? The reason is that it is useful for a symbolic executor to know when key positions in the program have been reached. The actions taken by PAN when loop entry, loop exit and merge statements are reached is described in Chapter 3.

We now address the question of whether the requirement to include flow control markers reduces the generality of PAN's input language. In fact, we claim that there is no loss of generality, as the flow control markers could be added automatically. For example, consider the program shown in figure 2-7(a). The definition of at loop entry and at loop exit specified in section 2.1.2 could be turned into a procedure which would recognise that statement 1 is at loop entry and statement 6 is at loop exit. New statements could then be inserted to produce the program in figure 2-7(b).

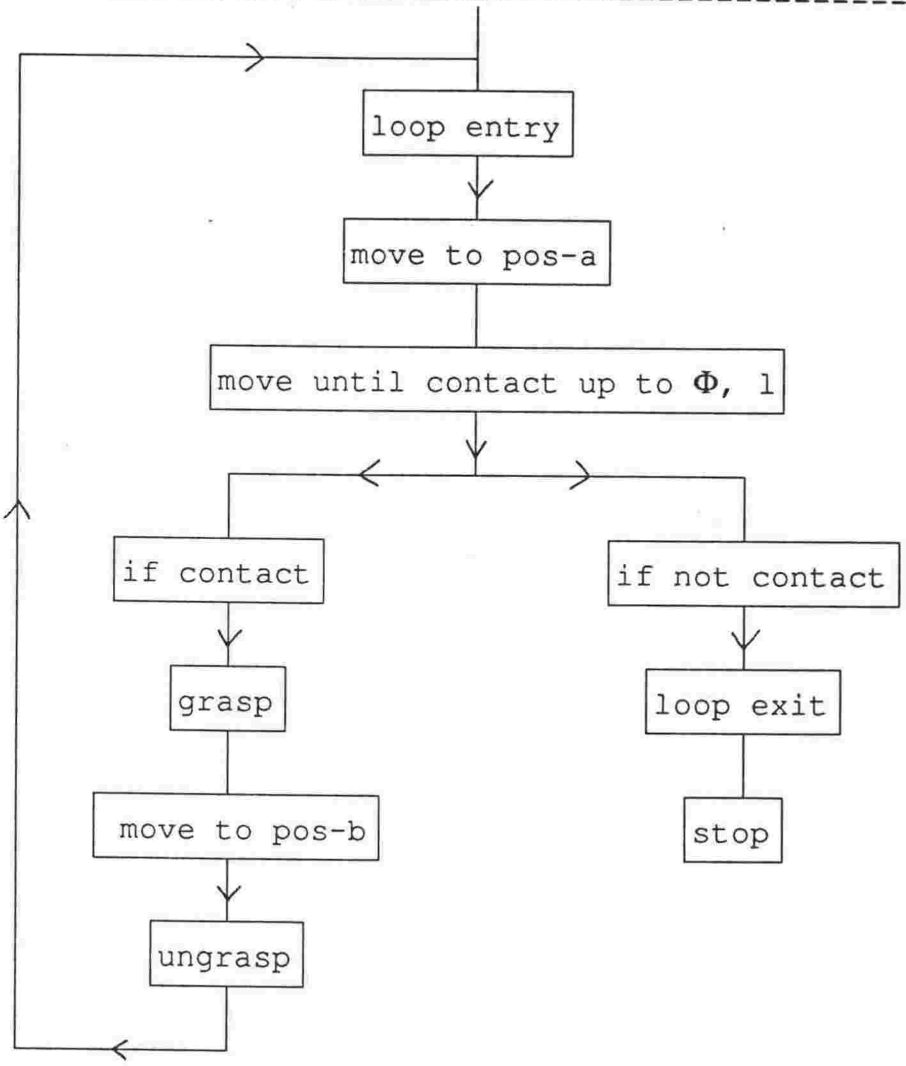


Figure 2-7 (b) Program Fragment With Flow Control Markers Added

---

If the program has nested loops, then multiple loop exit statements can be inserted. Merge statements are even easier to generate automatically. Although such a program preprocessor has not been built for PAN, it would be straightforward to produce one.

2.1.5 Expressiveness of PAN's Analysis Language

Having introduced the structure of PAN's input analysis language, we need to address the question of how expressive this language is. A program analyser which is successful because it only allows a very restricted input language is not interesting. Thus we want to show that conditional and iterative constructs from more conventional languages can be represented in PAN's restricted flow diagrams.

For conditional constructs we consider the common 'if...then...else' construct and the LISP 'cond' construct.

'If <condition-1> then <action-1> else <action-2>' can be represented in PAN's analysis language as shown in figure 2-8,

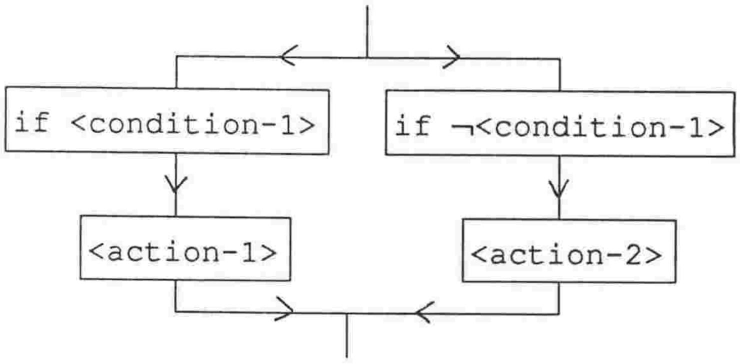


Figure 2-8 Representation of IF THEN ELSE

```
whereas '(cond (condition-1 action1)
               (condition-2 action2)
               (condition-3 action3)'
```

can be represented as shown in figure 2-9.

These representations trivially satisfy the restrictions in section 2.1.3.

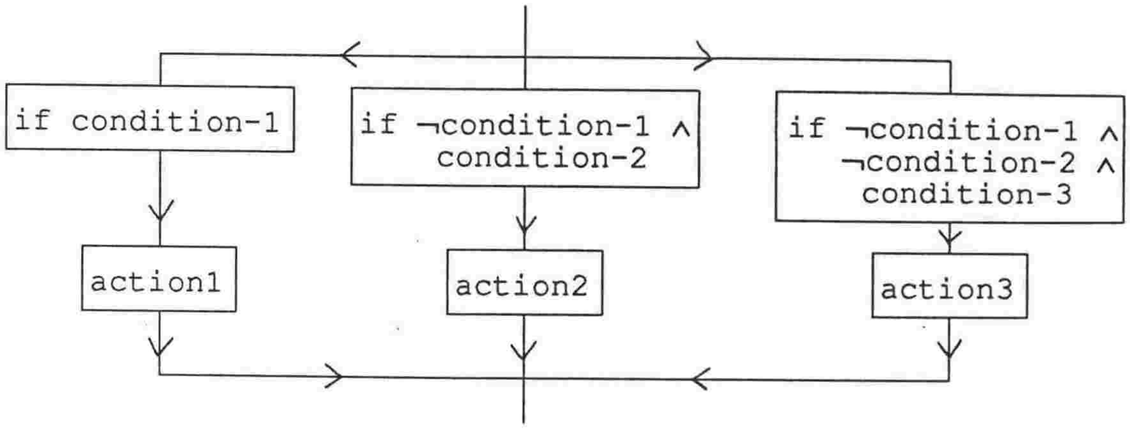

if condition-1
if ¬condition-1 ∧ condition-2
if ¬condition-1 ∧ ¬condition-2 ∧ condition-3
action1
action2
action3

Figure 2-9 Representation of COND

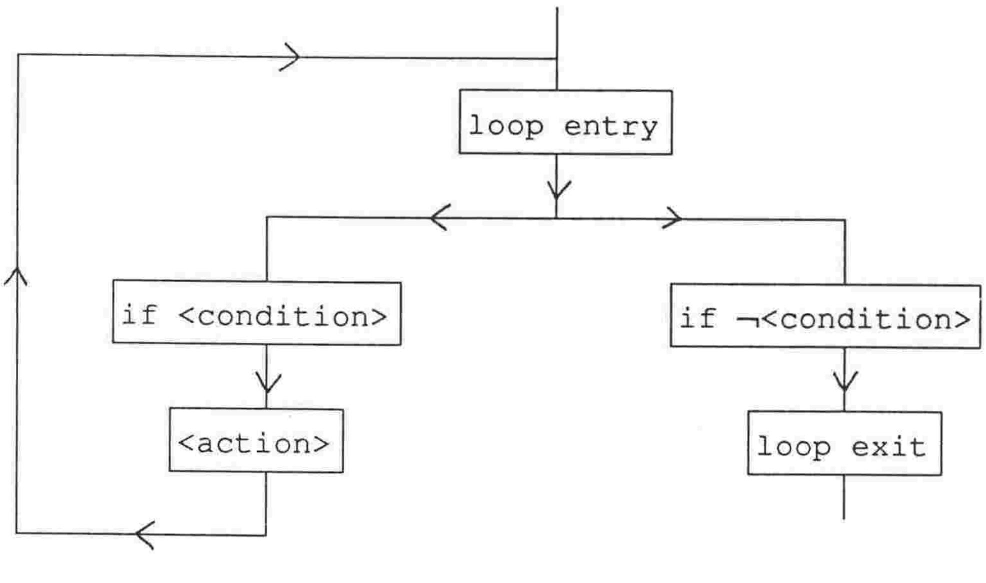


Figure 2-10 Representation of WHEN

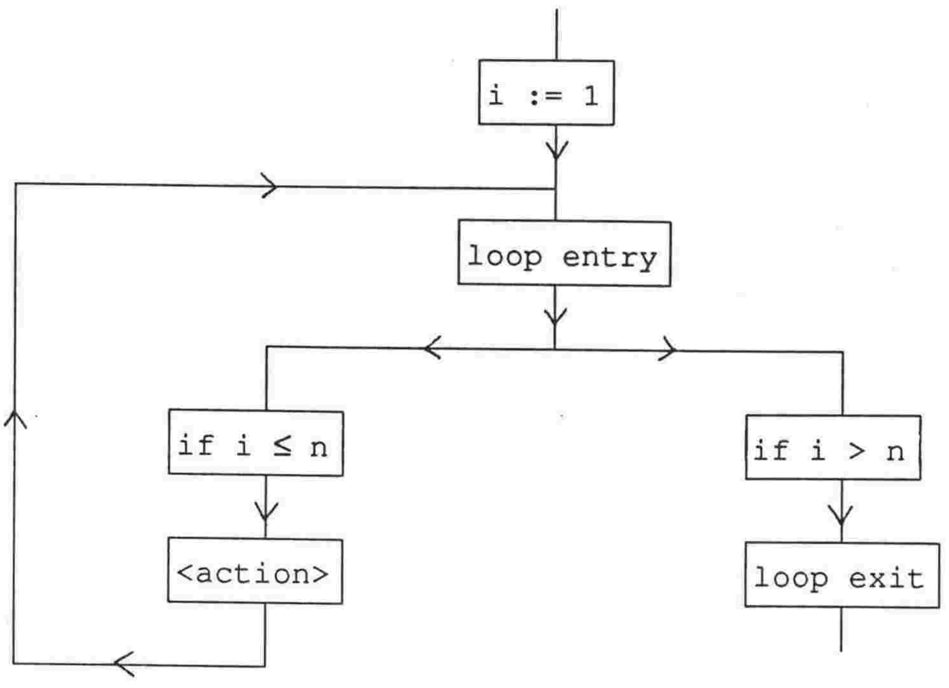


Figure 2-11 Representation of FOR

For iterative constructs, we choose 'when' and 'for' constructs. 'When <condition> <action>' can be represented as in figure 2-10, and 'for i = 1 to n <action>' can be represented as shown in figure 2-11.

However, the PAN input language also allows less structured iteration, including loops with multiple exits from different parts of the loop as shown in figure 2-12.

Thus we have shown that PAN's success at program analysis cannot be attributed to a restricted program domain.

2.1.6 Expressions

The remainder of section 2.1 describes the allowable PAN statements. Each statement is described by its syntax and semantics. Many of these statements contain boolean or arithmetic expressions, which are described first.

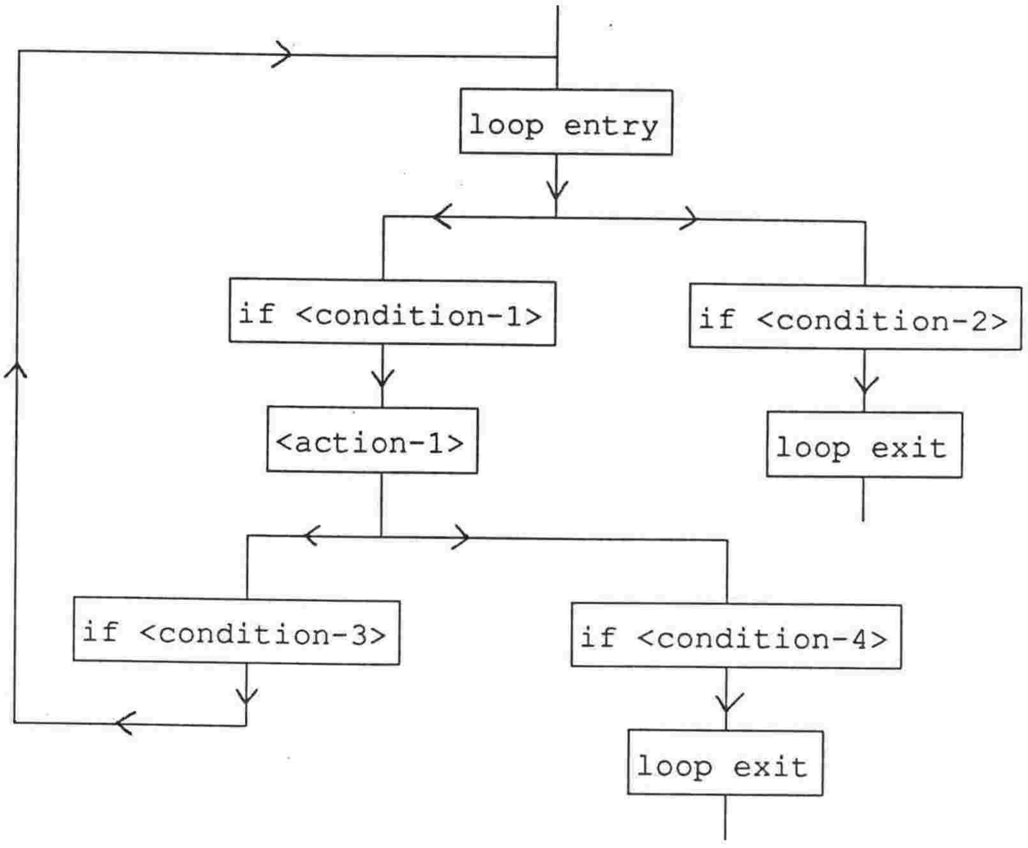


Figure 2-12 Representation of Less Structured Iteration

Syntax of Arithmetic Expressions

```
<arithmetic expression> ::= <arithmetic operand>
                            <arithmetic operator>
                            {<arithmetic operand>} |
                            <arithmetic operand>
<arithmetic operator> ::= + | - | * | /
<arithmetic operand> ::= <arithmetic expression> |
                         <property>(<identifier>) | <variable> |
                         <number>
<identifier> ::= <file identifier> | <hand identifier>
```

Syntax of Boolean Expressions

```
<boolean expression> ::= <boolean operand> <boolean operator>
                         <boolean operand> | not <boolean operand>
                         | <boolean operand>
<boolean operator> ::= and | or
<boolean operand> ::= <boolean expression> | <boolean term> | T | F |
                      <variable>
<boolean term> ::= <logical operand> <logical operator>
                   <logical operand>
<logical operator> ::= = | > | < | ≥ | ≤
<logical operand> ::= <property>(<identifier>) | <variable>
                      | <arithmetic expression>
```

Semantics of Expressions

The meaning of these terms generally follows the usual computer science interpretation. The operand <property(<file identifier>) is only used in the dp domain and refers to the field called <property> in the current record of file <file identifier>. The operand <property>(<hand identifier>) is only used in the robot domain and refers to the physical attribute called <property> of the object contacted by hand <hand identifier>.

The operand <variable> should be interpreted as the usual computer science concept - an area of memory used for temporary storage of value which can be retrieved by the variable name. All variables are global in scope and extent and no syntactic distinction is made between variables having arithmetic, boolean or other values. However, to legitimately occur as an arithmetic operand a variable must have a numeric value, whereas to occur as a boolean operand it must have a boolean value.

2.1.7 Condition Statements

PAN's input language contains only a single condition statement.

If

Syntax: if <boolean expression>

Semantics: The <boolean expression> is evaluated. If it has a value of T, execution can continue with the successors of this statement, otherwise it can't.

2.1.8 Flow Control Markers

As discussed in section 2.1.4, the flow control markers do not have any meaning for a real executor. However, for completeness, they are all described here.

Start

Syntax: start

Semantics: Signals the beginning of the program. Obviously both real and symbolic executors need to know where the program starts. However, this can be done by finding the single statement with no predecessors, or by insisting that the statement is first in the flow graph. The explicit start statement was mainly included for compatibility with Noddy programs.

Stop

Syntax: stop

Semantics: Signals end of the program. Equivalent comments to those in start also apply here.

Merge

Syntax: merge

Semantics:	Signals a position in the program where two (or more) branches merge. Would be ignored by a real executor.

Loop Entry

Syntax:	loop entry
Semantics:	Signals a position in the program where a loop begins. The restrictions on loops introduced in section 2.1.2 insures that each loop has a single position having this property. Would be ignored by a real executor.

Loop Exit

Syntax:	loop exit
Semantics:	Signals a position in the program where a loop ends. There may be several such positions for each loop. Would be ignored by a real executor.

2.1.9 Action Statements

We now describe the action statements, which make up the majority of PAN's analysis language. Most of these statements are specific to a particular program domain. Thus we describe these statements in three categories; domain independent, robot domain and dp domain.

2.1.9.1 Domain Independent Statement

The only domain independent statement currently included in PAN's input language is the assignment statement. In the robot domain this statement allows the robot to use any special abilities it may have to modify properties of a contacted object. For example, if a painting robot is able to modify an object's color, this will be done using an assignment statement specifying color as the property modified.

This use of the assignment statement allows all object properties other than position to be changed (position is changed using move statements because these affect the robot as well as the contacted object). Thus the assignment statement allows PAN programs to have the same effect as languages which include more specific commands, such as 'paint' without unnecessarily encumbering the PAN input program language.

The assignment statement in the dp domain allows fields in records to be given new values.

In both domains assignment statements can also be used to change the value of variables.

Assignment

```
Syntax:  <lhs assignment> := <rhs assignment>

    where

         <lhs assignment> ::= <variable> |
                              <property>(<file identifier>) |
                              <property>(<hand identifier>)
         <rhs assignment> ::= <arithmetic expression> |
                              <boolean expression>
```

Semantics: if the <lhs assignment> is <property>(<file-identifier>), then the assignment statement is updating the property (field) in current record of file specified by file-identifier; if the <lhs assignment> is <property>(<hand-identifier>), then the statement is modifying the object contacted by hand-identifier by changing the specified property; otherwise the statement is changing the value of the specified variable.

The <rhs assignment> is evaluated and assigned to the field, property or variable as determined above.

2.1.9.2 Robot Domain Statements

Programs written for the robot domain are intended to manipulate a blind 2-dimensional robot with one or more 'hands'. Under program instruction the robot is able to move its hands and to grasp and ungrasp objects. Objects can be grasped by a robot hand only if that hand is currently in contact with the object. Once an object has been grasped by a robot hand, any movement by that hand also moves the object.

In order that a robot hand can come into contact with an object without damaging it, special 'move until contact' instructions are provided, which presumably cause the robot hand to move more slowly. For each hand the boolean expression contact<hand-identifier> will have the value T if hand-identifier is in contact, and F otherwise. Thus 'contact' is a special 'property' detected by the robot hand.
The robot is also able to modify attributes of the contacted object, other than just its position. These modifications are handled by the general 'assignment' statement discussed above in section 2.1.9.1.

Program statements specific to this domain are as follows.

Move-to

Syntax: move-to <hand-identifier> <x coordinate> <y coordinate>

where

<x coordinate> ::= <arithmetic expression>
<y coordinate> ::= <arithmetic expression>

Semantics: the robot hand specified by hand-identifier moves to position specified by (x coordinate, y coordinate). If the robot hand is grasping an object, it also moves.

This instruction assumes there are no obstructions in the path of the robot hand.

Move-by

Syntax: move-by <hand-identifier> <x coordinate> <y coordinate>

Semantics: same as move-to, except that new position is specified as an increment from the current position, instead of as an absolute value.

Move-until-contact

Syntax: move-until-contact <hand-identifier> <angle>

where

<angle> ::= <arithmetic expression>

Semantics: the robot hand specified by hand-identifier, moves at the specified angle in a special 'cautious' mode and stops as soon as it detects contact. Movement continues indefinitely until contact is achieved.

Move-until-contact-up-to

```
Syntax:  move-until-contact-up-to <hand-identifier> <angle>
                                  <distance>
```

where

<distance> ::= <arithmetic expression>

Semantics: same as move-until-contact except that if the robot hand is still not in contact when <distance> has been moved, the robot hand stops.

Grasp

Syntax: grasp <hand-identifier>

Semantics: only allowed if the specified robot hand is in contact and not grasping. The contacted object is grasped.

Ungrasp

Syntax: ungrasp <hand-identifier>

Semantics: reverse of grasp

2.1.9.3 Data Processing Domain Statements

Programs written for this domain process files and records. Files can be read from or written to, either sequentially or by key. Whenever read statements are used the boolean expression current(<file-identifier>) will be set to T for a successful read from file file-identifier and F otherwise. Thus having a 'current' record is a special property of each file. No such facility is used for write statements i.e. they are assumed to always be successful. A key used to specify a record in a keyed read can be any predicate on fields within the record.

No special provision is made to handle duplicates - during a keyed read any record satisfying the predicate is retrieved; during a keyed write a new record is always written even if it is identical to an existing one, in which case a duplicate will be created.

Once a successful read statement has been executed, the current file record is available for processing.

Within a single program, sequential files can only be either read from or written to, but not both. There is no such limitation on keyed files. No explicit open statements are used. The first file access (whether read or write) performs this function.

Program statements specific to the dp domain are as follows.

<u>Sequential-read</u>

Syntax: sequential-read <file-identifier>

Semantics: either the next record is read from the file specified by file-identifier, and becomes the current record from that file, or the file is exhausted and no record is read (in which case there is now no current record).

<u>Sequential-write</u>

Syntax: sequential-write <file-identifier>

Semantics: the current record from the file specified by file-identifier is written to the file.

<u>Keyed-read</u>

Syntax: keyed-read <file-identifier> <boolean expression>

Semantics: either a record obeying the boolean expression is read for the file specified by file-identifier, or no such record exists and the read fails (in which case there is now no current record).

<u>Keyed-write</u>

Syntax: keyed-write <file-identifier>

Semantics: the current record from the file specified by file identifier is written to the file.

2.2 SPECIFICATION REPRESENTATION

The goal of the PAN system is to produce non iterative program specifications. To make these specifications both concise and unambiguous, we represent them in a logical language. This section describes this language, somewhat informally, but in sufficient detail that a formal description of the language could be constructed if required. We begin with a typed first order predicate calculus, extended to include arithmetic and set theory. Thus we assume the language includes the concepts of:

- types (integers, real numbers, strings, objects, sources and sequences)
- logical and arithmetic operators
- constants
- cardinality
- numbers
- variables
- predicates
- functions
- terms
- well formed formulae (wff)
- truth and falsity.

Further extensions to this language are required to represent concepts from our two domains, such as files and lines, and also to represent the relationships between objects, sources and sequences. Some simple functions on sequences are also needed. Thus, we extend the language to include:

- for each of the types; objects, sources and sequences, the special constants, OBJECT-n, SOURCE-n and SEQUENCE-n, for all integers n, where OBJECT-n are of type object, SOURCE-n are of type source and SEQUENCE-n are of type sequence.

- the representation of files and lines, using the functions

```
file: strings -> sources
line: numbers X numbers X numbers X numbers -> sources.
```

The intended interpretation of file(file-name) is the source that is a file with name file-name. The intended interpretation of line(i, j, k, l) is the source that is a line of length l, at angle k, from the point (i, j) in 2 dimensional space.

- a language construct for representing the objects retrieved from a source, using the functions

```
sequential-object-in-source: integers X sources ->
                             objects
keyed-object-in-source:unary predicates X sources ->
                             objects.
```

and the membership predicate ∊.

Thus if n is an integer and S is a source, then the intended interpretation of sequential-object-in-source(n, S), is the nth object retrieved from S. If P is a predicate with the free variable 'item', then the intended interpretation of keyed-object-in-source(P(item), S) is an object retrieved from source S which satisfies P. If x is an object then x ∊ S is interpreted to mean x was retrieved from S.

- a language construct for constructing primitive sequences from objects, sources, or other sequences and the predicate ∊ for representing membership of primitive sequences. Thus if D is a function on integers whose range is a set of items of type objects, sources, or sequences, and n is an integer, then

```
(sequence i = 1 to n D(i))
```

is a term of type sequence. Thus, this construct is an abbreviation of the n-tuple (D(1), D(2),...,D(n)), where each D(i) is an object, source or sequence. If S is a primitive sequence, then $x \in S$ is a well formed formula with the intended interpretation that $x = D(i)$ for some i, $1 \leq i \leq n$.

- a language construct for creating subsequences from sequences and the predicate $\in$ for representing membership of subsequences. Thus if S is a sequence, and P is a unary predicate, then

  $$(\text{item}: \text{item} \in S \wedge P(\text{item}))$$

  is of type sequence. The intended interpretation is the objects included in S which also satisfy P. If S′ is such a subsequence, then $x \in S'$ is a well formed formula with the intended interpretation that $x \in S \wedge P(x)$.

- representation of size of sources and sequences, meaning the number of elements they contain. Since we have defined membership of sources and sequences and our basic language is assumed to include set theory, we can define the size as the cardinality of the set of members in each construct by the axiom

  $$\text{size}(x) = \text{cardinality}(\{\text{item}: \text{item} \in x\})$$

  where x is of type source or sequence.

- a function, item-in-sequence, for referring to an item by its position in a sequence. Thus if n is an integer and S is a sequence, then item-in-sequence(n, S), is a term of the same type as the elements of S. The intended interpretation is that item-in-sequence(n, S), is the nth item in S.

- a function, position-in-sequence, for referring to the position an item occupies in a sequence. Thus if $x \in S$ where S is a sequence, then position-in-sequence(X, S) is an integer. The intended interpretation is that position-in-sequence(x, S) = n, if x is the nth element of S.

- a function, map, for specifying corresponding members of sequences of equal length. Thus if S and S′ are sequences such that size(S) = size(S′), then map(item, S, S′) is defined as

  item-in-sequence(position-in-sequence(item, S), S′)

  The interpretation of map is that it maps the nth element of S to the nth element of S′.

In this extended language, a specification is represented by a wff in the form

$$(P1 \rightarrow Q1) \wedge (P2 \rightarrow Q2) \wedge \ldots \wedge (Pn \rightarrow Qn),$$

meaning that if the input to the program satisfies Pi, then the program has effect Qi, for i = 1 to n.

2.2.1 Some Examples of Using the Specification Language.

We now provide some examples of specifications represented in this language.

Suppose we have a program which moves the first object from the line (pos-a, Φ, 1) to pos-b if it is blue and to pos-c otherwise. In our formal specification language, this would be expressed as:

```
(SOURCE-1 = line(pos-a, Φ, 1) ∧
 OBJECT-1 = sequential-object-in-source(1, SOURCE-1) ∧
 color(OBJECT-1) = blue
    → position(OBJECT-1) = pos-b)

∧

(SOURCE-1 = line(pos-a, Φ, 1) ∧
 OBJECT-1 = sequential-object-in-source(1, SOURCE-1) ∧
¬ color(OBJECT-1) = blue
    → position(OBJECT-1) = pos-c)
```

For another example, consider again the robot domain example from section 1.3. This program moves all blue objects from line (pos-a, Φ, 1) to pos-b and red objects to pos-c.

This would be expressed as:

```
SOURCE-1 = line(pos-a, Φ, 1) ∧
 SEQUENCE-1 = (sequence i = 1 to SIZE(SOURCE-1)
                 sequential-object-in-source(i, SOURCE-1)) ∧
 SEQUENCE-2 = (item: item ∈ SEQUENCE-1 ∧ color(item) = blue) ∧
 SEQUENCE-3 = (item: item ∈ SEQUENCE-1 ∧ color(item) = red)
    → (∀item (item ∈ SEQUENCE-2 → position(item) = pos-b) ∧
       ∀item (item ∈ SEQUENCE-3 → position(item) = pos-c))
```

To demonstrate the usefulness of the map function, consider the program in figure 2-13. In this program the specification will include the sequences :

```
    SEQUENCE-1 = (sequence i = 1 to n sequential-object-in-source(i,
                  SOURCE-1)
    SEQUENCE-2 = (sequence i = 1 to n sequential-object-in-source(i,
                  SOURCE-2)
```

where

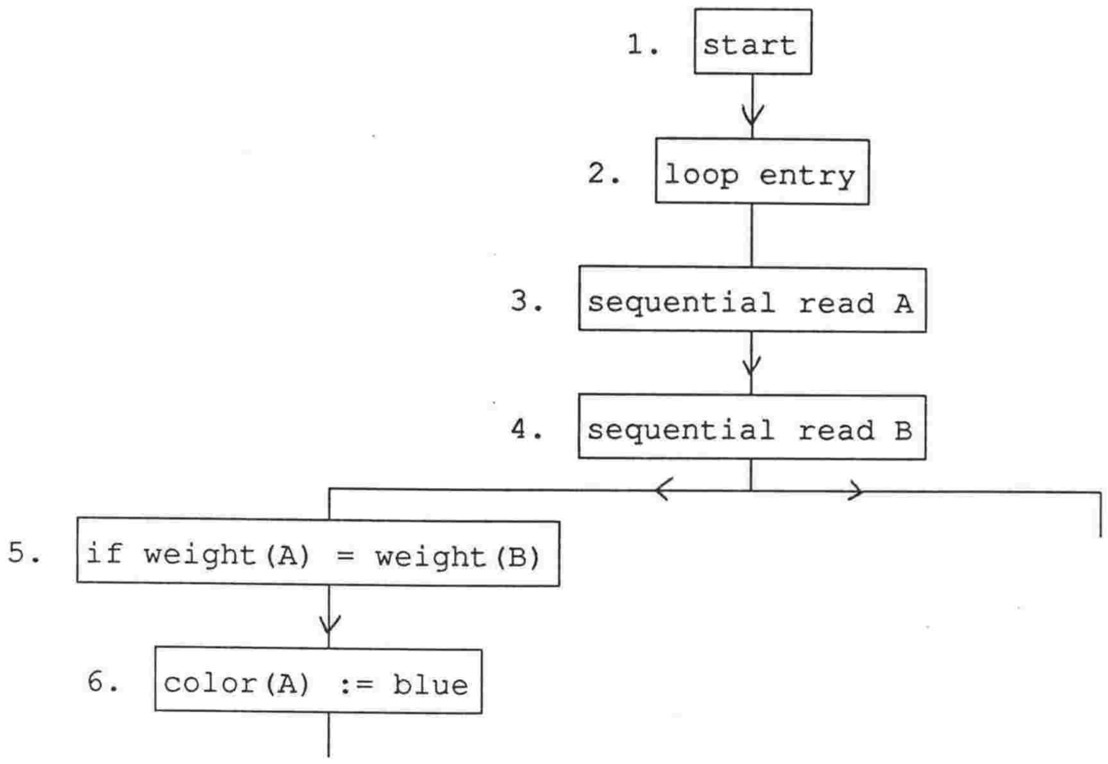


Figure 2-13 - Subsequences using two sources.

SOURCE-1 = file(A)

SOURCE-2 = file(B).

To describe the action taken at statement 6, we need a subsequence of all objects in SEQUENCE-1 which obey the condition at statement 5. This subsequence consists of all objects from SEQUENCE-1 whose weights are equal to the weight of corresponding objects from SEQUENCE-2. These can be expressed using the map function as

```
SEQUENCE-3 = (item: item ∈ SEQUENCE-1 ∧ weight(item) =
              weight(map(item, SEQUENCE-1, SEQUENCE-2))).
```

We can now express the specification of the program fragment as:

```
SOURCE-1 = file(A) ∧
 SOURCE-2 = file(B) ∧
```

```
SEQUENCE-1 = (sequence i = 1 to n sequential-object-in-source(i,
               SOURCE-1)) ∧
SEQUENCE-2 = (sequence i = 1 to n sequential-object-in-source(i,
               SOURCE-2)) ∧
SEQUENCE-3 = (item: item ∈ SEQUENCE-1 ∧ weight(item) =
               weight(map(item, SEQUENCE-1, SEQUENCE-2)))
   → ∀item(item ∈ SEQUENCE-3 → color(item) = blue)
```

In some programs, objects have properties modified in a way that is dependent on the object's position in a sequence. For example, a program may move all objects from line (pos-a, Φ, 1), so that the first object is moved to (1,1), the second to (2,2) etc. The specification for this could be expressed using the specification language as:

```
SOURCE-1 = line(pos-a, Φ, 1) ∧
 SEQUENCE-1 = (sequence i = 1 to size(SOURCE-1)
               sequential-object-in-source(i, SOURCE-1))
   → ∀item (item ∈ SEQUENCE-1 → position(item) =
           (position-in-sequence(item, SEQUENCE-1),
            position-in-sequence(item, SEQUENCE-1)))
```

Specifications of more complex programs can be constructed from these language components. Further examples are presented in the appendix.

Chapter 3

Execution States, Program Analysis and Symbolic Execution

3.1 INTRODUCTION

Given some initial state of the world, real execution of a program will produce some particular effect. Observing such a single real execution does not allow us to describe the general relationship between initial world states and effects. Symbolic execution, on the other hand, attempts to simulate program execution for a general 'symbolic' input. The result of such a simulation should allow program effects to be expressed as a function of the initial state.

Both real and symbolic execution of an entire program are achieved by repeated execution of individual program statements. Real execution of a single program statement may involve updating the executor's internal state and/or modifying the domain in which the program operates. A symbolic executor mimics these effects by maintaining symbolic descriptions of both the executor's internal state and the effect the program has had on its domain of execution at each point in the program. These symbolic descriptions are referred to as execution states. Thus the effect of symbolically executing a program statement S is to produce the execution state at the point following the statement. This execution state describes the effect of executing all statements up to, and including, S.

Conditional statements are those which control entry to different program branches. The simplest conditional statements are those which control entry to two program branches. One branch is entered if the condition is true and the other if it is false. Upon reaching such statements, a real executor will evaluate the conditions to determine which branch to follow. A symbolic executor, on the other hand, in general, will not be able to determine whether the condition is true or false. Instead, the symbolic executor will take both branches. A separate execution state will be created for each branch and the fact that the condition is true will be added to one execution state and the fact that it is false added to the other. Thus the execution states associated with any execution state S will contain facts derived from all conditions on the path traversed to reach S. We distinguish between these facts and the remainder of the execution state, by referring to the former as path conditions and the latter as the effects.

Each execution state can be associated with a single program statement - the statement executed by the symbolic executor in producing that state. A statement ,however, can be associated with many execution states. This arises because the same program statement may be reached by different paths through the program.

Since different paths through the program may have different effects the execution of a program statement requires that a separate execution state be produced for each such path. Thus a program statement, S, may be associated with all the different execution states produced by executing S.

The execution states associated with a single statement cannot be simply 'merged' into a single execution state. To see why, suppose that the statement was in a loop. Then the execution states associated with this statement will reflect the effects of one, two, three... loop iterations. A single 'merged' execution state would have to describe the effect after an indefinite number of iterations, which is exactly the problem which has made symbolic execution of loops hard, and is the major issue addressed by this thesis.

For a program without loops, symbolic execution is an adequate program analysis technique. Each execution state associated with a stop statement implicitly contains a partial program specification of the form path conditions $\rightarrow$ effects. The conjunction of these partial specifications are the full program specification. However, for programs with loops, symbolic execution alone is not adequate, as the complete set of execution states associated with stop statements can never be produced. Thus the PAN program analysis system combines symbolic execution with separate techniques for loop analysis. Symbolic execution and loop analysis techniques are organised as separate tasks, both of which use execution states. The remainder of this chapter describes the contents of the execution states, shows how PAN schedules the various tasks it has available and describes in detail the symbolic execution task.

3.2 EXECUTION STATES

Because of the problem of conditional statements discussed above, any symbolic executor needs to maintain multiple execution states. Most symbolic execution systems discard an execution state once it has been used to forward the symbolic execution. In the PAN system, however, loop generalization requires the execution states after one, two, three or more loop iterations to be available to the generalization process. To ensure that sufficient information is available, PAN retains all execution states. This requires PAN to record some additional bookkeeping information with each execution state, to be able to distinguish those execution states being retained for generalization from those still actively involved in symbolic execution. Again to assist loop generalization, PAN also records some history in the execution states. Of course, PAN also records the 'standard' data of path conditions and effects. These categories of information are used in the following detailed description of PAN's execution states.

3.2.1 Effects

A symbolic executor needs to describe the effect the program has had on the world in which the program executes. This data is generally standard information which all symbolic executors need to maintain. The only unusual features are the use of objects to refer to either physical objects or file records, and the information recorded on sequences. The information is recorded in the following categories.

Variable Data

PAN programs are allowed to refer to simple variables, and the program may have had the effect of changing the value of some of these variables. So, we record the value of each variable so far encountered in the program. Variables are identified by the name used in the program. This information is used whenever a variable occurs in program expressions which need to be evaluated (e.g. conditional expressions, or parameters of action statements).

Thus, we may record v = 6 or w = weight(OBJECT-1).

Object Data

For each object encountered during program execution, we record the value of any changed properties. The word 'objects' is used generically to mean the fundamental element in the world the program runs in. In the robot domain, these are physical objects, whereas in the dp domain they are records from a file. Properties refer to physical properties of physical objects or the value of fields in a record. Each property is identified by the name used in the program. This information is used whenever an object property that has been modified occurs in a program expression which needs to be evaluated.

However, since updated properties record the principal effect the program has had on its environment, this data is primarily used in program interpretation - the process of producing program specifications from the execution states. This data is generalized into sequences during loop generalization. Two examples of updated properties are:

```
    color(OBJECT-1) = red
```

and

```
    position(OBJECT-2) = (1,2)
```

Source Data

For each source identified during program execution, we record:

- the current object
- the number of retrievals attempted (sequential input sources only) or number of writes (sequential output sources only)
- an exhausted indicator (sequential sources only)

The word source is used generically to mean the method of specifying objects in the program. In the robot domain objects are specified by the lines on which they are contacted, whereas in the dp domain records are specified by the files from which they are retrieved.

The current object specifies the most recently accessed object from the source, if there is one. In the dp domain it is used whenever a program statement refers to a source as a way of specifying the current object from that source. For example, the program statement

```
x := weight(A)
```

means that the variable x is assigned the value of the weight of the current object from source A.

The number of objects retrieved is used to determine the definition of the next object to be retrieved. For example, if the number of objects retrieved = 2, then the next object retrieved will be defined as the 3rd object from that source.

The number of retrievals attempted acts like a special type of variable, not referenced in the program but required by the symbolic executor to keep track of its position in a sequential file. (A real executor would also need such a variable if the language allowed such statements as 'read nth record'.) PAN's use of this variable is further discussed in section 3.3.5 in describing symbolic execution of statements which sequentially retrieve objects from sources.

The exhausted indicator, if set, shows that the previous attempt to obtain an object from the source was unsuccessful (because the source had no more objects). If this indicator is set, subsequent attempts to access the source will always be unsuccessful.

Object Sequence Data

For each object sequence, we record the same information as for objects i.e. the value of updated properties.

Sequence information is only created in the loop generalization process. Sequence information is used in program interpretation and, for programs containing nested loops, for further generalization.

Robot Status Data

In the robot domain only, PAN records, for each robot hand:

- its position
- object contacted (if any)
- an indicator to show whether hand is grasping or not.

This information is needed to determine which object (if any) has its position changed when the robot hand moves, and what its new position is.

3.2.2 Path Conditions

The path condition in an execution state contains a boolean expression that the initial state of the world must satisfy in order for the program to have the effect described in the remainder of the execution state.

Any given execution state is associated with a program statement and records the effects of symbolically executing some path through the program up to and including that statement. This path can only be traversed if the initial state of the world is such that all conditions along the path are true. The path condition is therefore the conjunction of these conditions.

To formalise this definition, we first need to introduce our terminology for the instantiation of a predicate. Conditions may contain references to files, hands and variables. A condition is evaluated in any execution state by instantiating files by the current object, hands by the contacted object and variables by their current value. We use the convention that the instantiation of predicate $C(x_1,...,x_n)$, with free variables, $x_1,...,x_n$, by values $v_1,...,v_n$ is represented by $C(x_1/v_1,...,x_n/v_n)$. If we want to show the instantiation with the appropriate values in execution state E, then we write $C(x_1,...,x_n)/E$ or simply C/E if the free variables in the predicate do not need to be individually identified.

Now suppose PAN execution has proceeded from the start statement down some path P, to reach statement S. Suppose further that PAN created execution states $E_1,...,E_n$, when executing condition statements containing conditions $C_1,...,C_n$ on this path, and execution state E when executing S. The *path condition* for execution state E is then defined as

$$C_1/E_1 \wedge ... \wedge C_n/E_n,$$

simplified if necessary.

For example, path condition could consist of:

$$(\text{color(OBJECT-1)} = \text{red}) \wedge (\text{size(SOURCE-1)} < 10).$$

3.2.3 History

PAN also records some historical information, for use in the loop generalization process. Chapter 4 gives a detailed description of the use of this information. At this point we merely state that this information enables PAN to reconstruct from the execution states the structure of conditional branching within the loop and the conditions required to enter each branch. To specify this information, we first need to introduce the concept of a statement condition. Given any statement within a loop, the statement condition, for any iteration of the loop, is the condition which needs to be true for execution to reach that statement from loop entry, using any possible program path.

Before formalising this definition, we introduce an alternate form for representing instantiation. This is required because PAN needs to be able to identify the instantiation on statement conditions. Given a predicate $C(x_1,\ldots,x_n)$ with free variables $x_1,\ldots,x_n$, we use $C(x_1[n_1],\ldots,x_n[v_n])$ to represent an instantiation of $x_1,\ldots,x_n$ by $v_1,\ldots,v_n$ which is explicitly recorded, rather than simply substituted into C. In other words, both x_i and v_i are recorded. If C is instantiated with appropriate values from execution state E and individual values do not need to be identified, we write C[E].

To formalise the definition of the statement condition, suppose a statement S in a loop can be reached down paths $P_1,\ldots,P_n$ from loop entry without passing again through loop entry or a loop exit for this loop entry. Beginning with a single execution state E associated with loop entry, suppose that execution of P_i produces execution states $E\text{-}i1,\ldots,E\text{-}im_i$, when executing conditions $C\text{-}i1,\ldots,C\text{-}im_i$ on P_i. We define the *statement condition* C to reach S starting in execution state E as

$$C = (C\text{-}11[E\text{-}11] \wedge \ldots \wedge C\text{-}1m_1[E\text{-}1m_1]) \vee \ldots \vee$$
$$(C\text{-}n1[E\text{-}n1] \wedge \ldots \wedge C\text{-}nm_n[E\text{-}nm_n]).$$

Since the instantiation is explicitly recorded, we can always derive the *uninstantiated form* of a statement condition by replacing terms of the form $x_i[v_i]$ by x_i.

Note that when we define the statement condition for a statement in a loop, the loop may contain inner loops. Such inner loops do not appear explicitly in the definition of the statement condition. For the purpose of statement conditions we treat each inner loops if it were of the form shown in figure 3-1.

Of course, condition-1,...,condition-n are not explicitly defined in the program, but we assume that these conditions can be determined as part of loop processing. Given such a representation of an inner loop, condition-1,...,condition-n, will, being conditions, be included in statement conditions in the outer loop.

This method of handling inner loops means that the statement condition for a statement S in a loop cannot be determined until all inner loops on the path from the loop entry to S have been analysed. PAN scheduling, described later in this chapter, ensures that loops are analysed in the correct order so that statement conditions can always be determined. How the conditions in figure 3-1 can be determined is addressed in Chapter 5.

The use of the statement condition requires that it satisfies the following:

- if a loop contains two statements S and S′ that intuitively require the same conditions in order to reach them from loop entry, then the statement condition to reach S starting from execution state E will be equivalent to the statement condition to reach S′ starting from the same execution state E

- given two execution states E and E′ associated with loop entry, the uninstantiated form of the statement condition to reach some statement S in a loop starting from E will be equivalent to the statement condition to reach S starting from E′.

These conditions are related in that they are affected by the extent to which the statement condition is simplified. The second requirement is trivially true if the statement condition is not

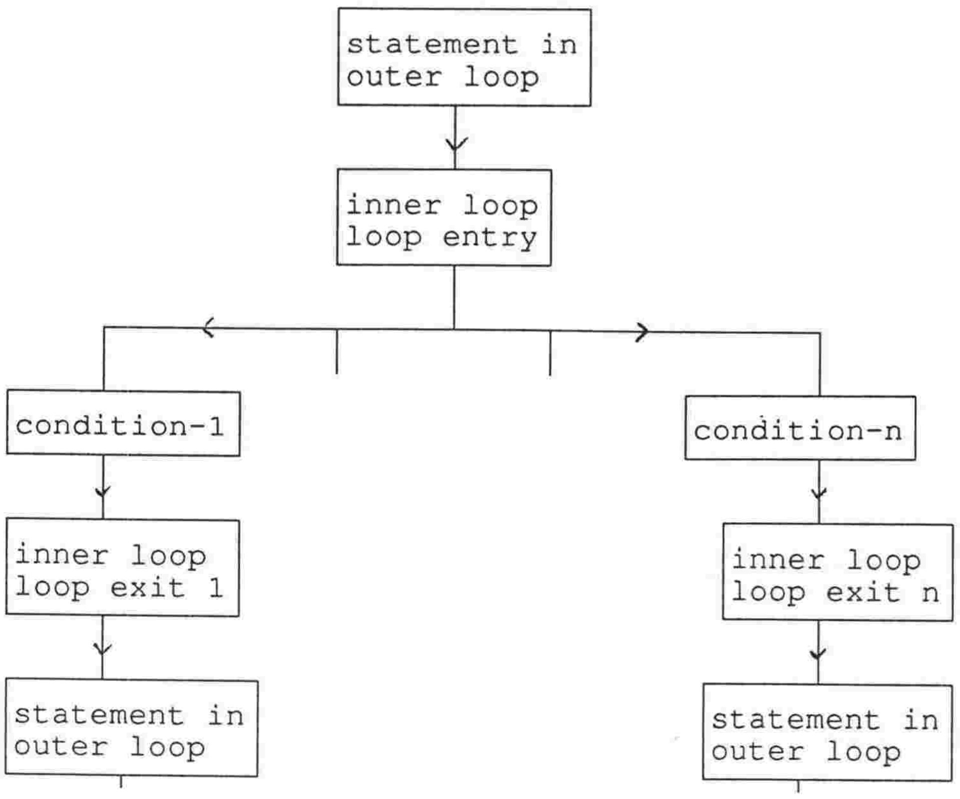


Figure 3-1 Inner Loops for Statement Condition

---

simplified at all. However, the first requires that some simplification is performed. To see why, consider the program fragment in figure 3-2. If OBJECT-1 is read from file A and the value of x is 5, the statement condition to reach S is

```
color(A[OBJECT-1]) = red
```

while that to reach S′ is

$$(\text{color}(A[\text{OBJECT-1}]) = \text{red} \wedge x[5] < n) \vee$$
$$(\text{color}(A[\text{OBJECT-1}]) = \text{red} \wedge x[5] \geq n).$$

Simplification of the statement condition for S′ produces

```
color(A[OBJECT-1]) = red
```

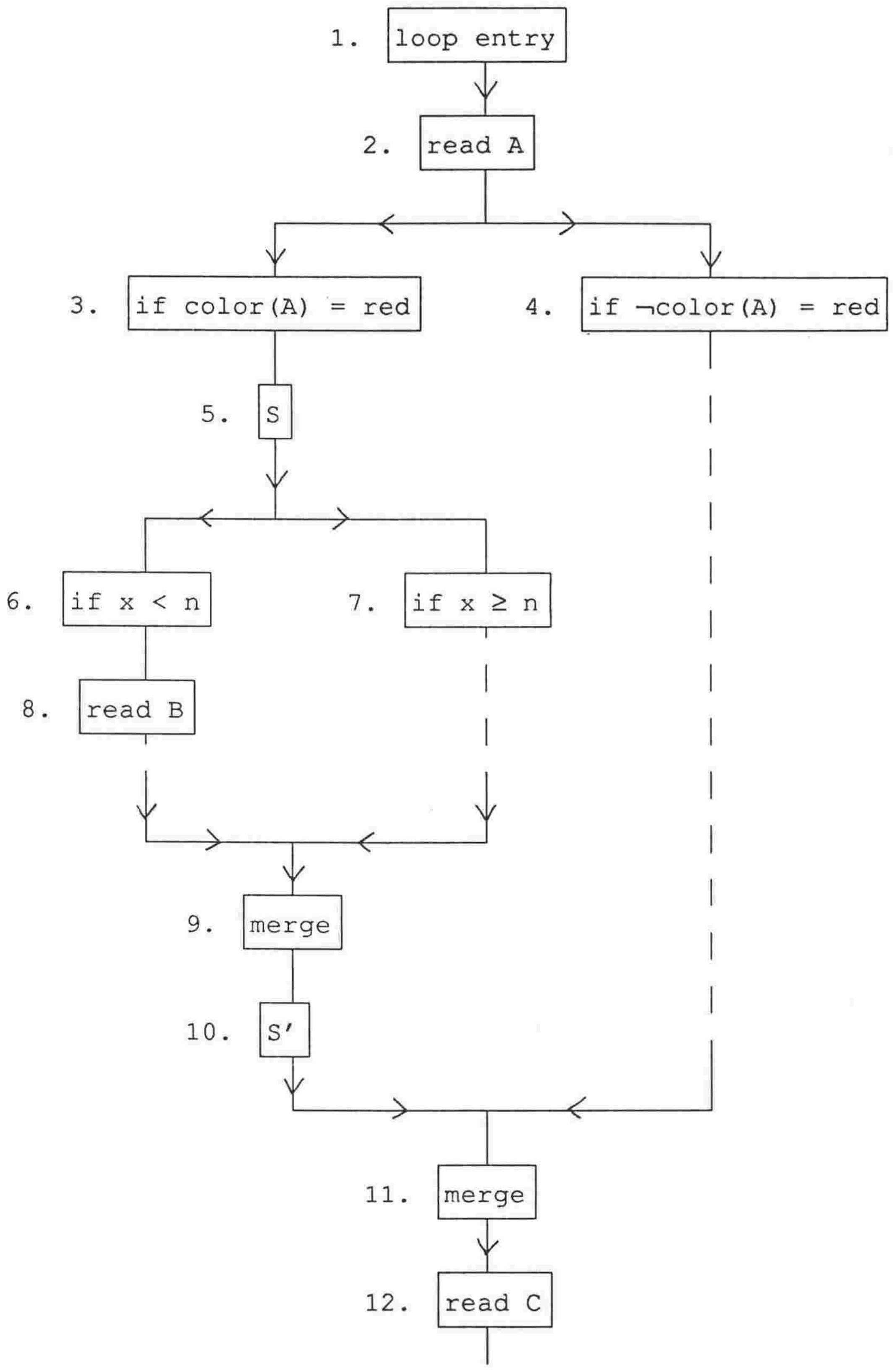


Figure 3-2 - Statements with Equal Statement Conditions

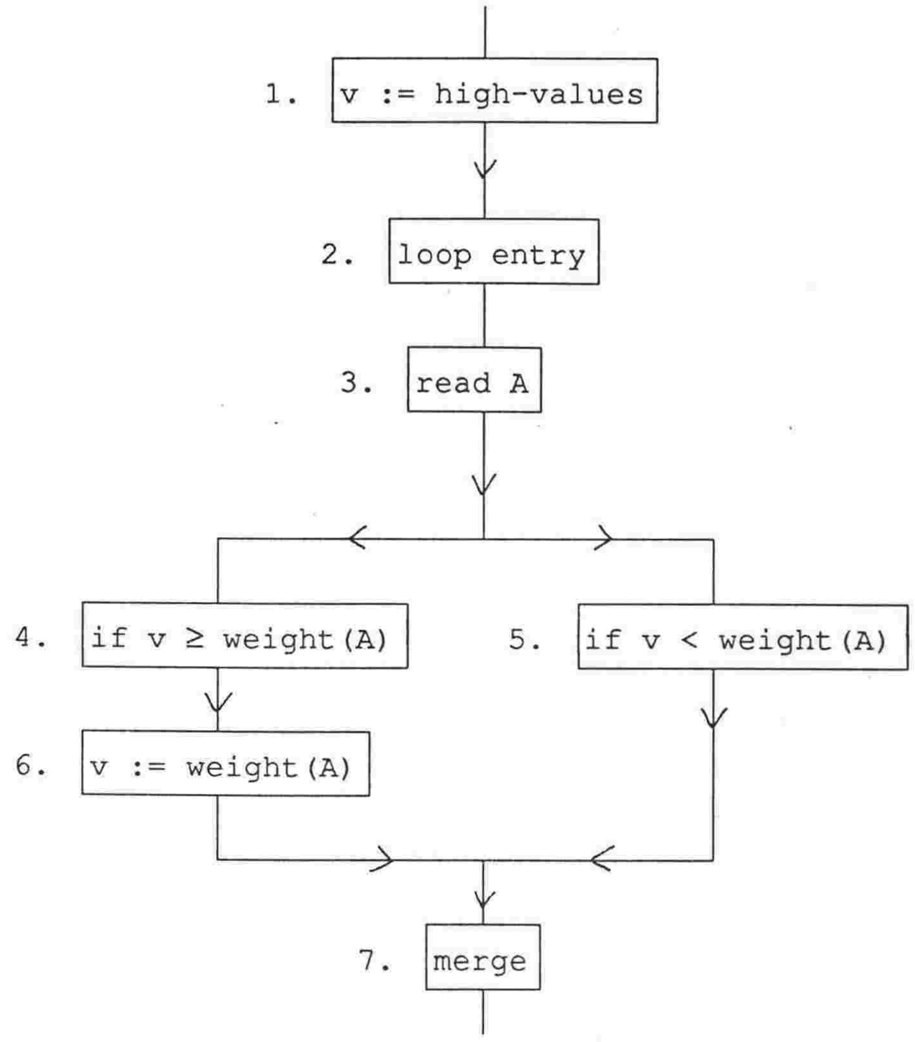


Figure 3-3 - Statement Conditions on Different Iterations

---

which is equal to the statement condition for S, meeting the first requirement. In this simplification A[OBJECT-1] can be treated as a term distinct from OBJECT-1. If, however, terms of the form x[v] are treated as being equivalent to v, then the second requirement above will not be satisfied. To see why, consider the program fragment shown in figure 3-3.

Suppose that during symbolic execution of this program, we have two execution states E and E′ associated with loop entry, E being the state after no loop iterations and E′ being the state after one iteration. The statements conditions to reach statement 6 from E and E′ will be

weight(A[OBJECT-1]) ≤ x[high-values]

and

weight(A[OBJECT-1]) ≤ x[weight(OBJECT-1)]

assuming the first two objects read from file A are OBJECT-1 and OBJECT-2, since on the first iteration the path through statement 6 will be taken and x will be given the value weight(OBJECT-1). But, assuming the simplification rule, n ≤ high-values, for any number n, the first of these expressions may be simplified to T, if x[high-values] is treated as equivalent to high-values. To avoid this problem, PAN simplifies the statement condition treating x[v] and v as distinct terms.

The statement condition has some similarities to the path condition. However, they differ in that

- the statement condition is only defined for statements in a loop, and only describes the conditions between loop entry and the statement

- the statement condition records the condition to reach a statement down any path, not a particular path

- the instantiation of the statement condition can be identified, and this limits the simplification which can be performed

We are now in a position to describe the history recorded by PAN. Given an execution state, E, associated with a statement S in a loop, created after execution has proceeded from execution state E′ associated with loop entry, then the statement condition to reach S from E′ is recorded in E. This historical information is recorded for each loop the statement is in.

This history allows PAN to reconstruct the program structure during loop generalization. However, this information could not be obtained simply by examining the program. The predicates from the condition statements in the loop could be determined this way, but the instantiation could not.

3.2.4 Bookkeeping Data

To enable the symbolic executor to keep track of its numerous execution states, we record on each state the associated statement, parent execution state, status and loop data.

The status is used to determine what type of processing is required for the execution state. Possible values are:

-	active	normal symbolic execution can be continued from this execution state
-	dead	no further processing is required for this execution state
-	waiting loop generalization	execution state has reached loop entry after some number of loop iterations
-	waiting exit processing	execution state has reached loop exit statement after loop generalization
-	waiting merge	execution state has reached a merge statement
-	waiting interpretation	execution state has reached program end

The symbolic execution process described below always creates a new execution state from an existing one. Thus execution states have a parent/child relationship, which is recorded in the parent execution state. This relationship allows us to describe one execution state as *descended* from another.

If the last statement executed is within one or more loops, then PAN requires information about these loops for two reasons. Firstly, in order to ensure that only a specified number of iterations are performed before loop generalization, PAN needs to record the number of iterations performed. Secondly, because symbolic execution only allows loop exit after loop generalization, an indicator is required of whether a loop has yet been generalized.

3.2.5 Names and Definitions

Much of the information recorded in the execution states involves objects, sources or sequences. For clarity, these are referred to using identifiers of the form OBJECT-n, SOURCE-n and SEQUENCE-n. These identifiers need to be related to the definition of each item. This information is the same for all execution states and we refer to it as *global data*.

3.3 PROGRAM ANALYSIS

3.3.1 Introduction

Having described execution states, we are in a position to describe in more detail how PAN uses symbolic execution to perform program analysis. The intention is to allow symbolic execution to continue until execution has reached program end.

Once this point has been reached, then the execution states associated with stop statements are interpreted to produce the program specification which is the output from the program analysis. If these execution states are E1,...,En containing path conditions P1,...,Pn and effects Q1,...,Qn, then the required program specification can be derived by simplifying the expression

$$(P1 \rightarrow Q1) \wedge (P2 \rightarrow Q2) \wedge \ldots \wedge (Pn \rightarrow Qn).$$

The apparent simplicity of this scheme is complicated by loops in the input program. Regardless of language and representation a loop must always contain a condition which controls exit from the loop. The inability of a symbolic executor to determine the truth of this condition will lead to loop execution continuing indefinitely.

PAN addresses this problem by analysing loops using a three stage approach: first, the execution states produced during a few loop iterations are generalized to a single execution state representing the effect of an indefinite number of iterations; second, the generalizations produced are verified to be invariants by performing another loop iteration; third, symbolic execution using the generalized execution state continues until all possible loop exits have been reached, at which point loop exit processing determines the number of loop iterations performed.

Normal symbolic execution then continues from the loop exits using the execution states produced by the loop exit analysis.

Both loop generalization and loop exit processing differ from normal symbolic execution in that they use multiple execution states, and cannot be performed until all these states are available. This is also true of a merge process which is required for updating the statement condition stored in the execution state history.

Thus symbolic execution is only one component of the PAN system. In fact, PAN consists of several distinct components called processes. At any one time, only one of these processes is active, controlled by a scheduling process as shown in figure 3-4.

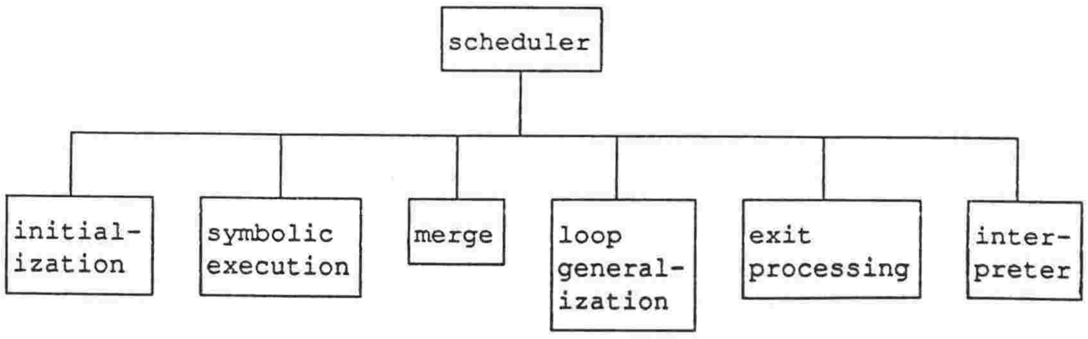


Figure 3-4 - Structure of PAN

The function of each process is summarized as follows:

Process	Function
scheduler	Decide which process to run next.
initialization	Begins program analysis.
symbolic execution	Symbolically executes one or more statements.
merge	Updates statement condition at a merge statement inside a loop.
loop generalization	Generalizes all execution states at loop entry.
exit processing	Determines number of loop iterations performed.
interpreter	Outputs program specification from execution states at program end.

3.3.2 Process Description

This section describes the individual PAN processes in more detail.

Initialization Process

Input: program to be analysed
Output: a single execution state

This process is invoked once, when PAN is executed. PAN requests an input program and then finds the single 'start' statement within the program. PAN then creates an execution state, associates it with this start statement and sets its status set to active.

Symbolic Execution Process

Input: a single active execution state representing execution up to and including some statement S
Output: new execution states representing execution of all statements up to and including all successor statements of S

PAN finds the statement associated with the input execution state. It then scans the input program to find all successors of this statement. PAN changes the status of the input execution state to dead unless it is associated with a loop entry statement of an ungeneralized loop, in which case it is changed to 'waiting loop generalization'. Then, for each successor statement PAN:

- creates a new execution state which is initially a copy of the input execution state

- updates the execution state according to the instructions in the statement. PAN has an update subprocess for each type of statement and simply calls the update subprocess associated with the name of the command in the statement.

This process is simple because the actual knowledge of how to execute a particular statement is in the subprocesses. There are three groups of statements - condition statements, flow control markers and action statements. PAN symbolically executes condition statements by updating the path and statement conditions. It executes flow control markers by updating history and bookkeeping information. It executes action statements by updating the effects. The subprocesses for these three groups are described in sections 3.3.3, 3.3.4 and 3.3.5 respectively.

Merge Process

Input:	a set of execution states associated with a single merge statement, that are descended from a single execution state associated with the loop entry
Output:	same set of execution states updated

Section 3.2.3 defined the statement condition recorded in any execution state created during execution of a statement in a loop. Using this definition the statement condition could be derived by finding the appropriate conditions from the program and instantiating them using previously created execution states. A more efficient way

of finding the statement condition is to create it from the one recorded on the parent execution state. By the definition of the statement condition it is easy to show that given execution states E and E′ associated with statements S and S′, where S′ is a predecessor of S and S is not a merge statement, E′ is the parent of E

- if S is not an if statement, then the statement condition in E equals the statement condition in E′

- if S is an if statement, with condition C, then the statement condition for E is the conjunction the statement condition in E′ and C[E].

Thus the statement condition in E can easily be determined from that in E′, except possibly in the case that S is a merge statement. In this case we want to show that the statement condition can be derived by forming the disjunction of the statement conditions from all execution states associated with predecessors of S.

Theorem 3-1: If a statement SM is at a merge, then the statement condition tp reach SM is the disjunction of the statement condition of every predecessor of SM.

Proof: Suppose there are paths $P_1,...,P_n$ from loop entry to SM, and the predecessors of S are $S_1,...,S_m$. Then $P_1,...,P_n$ can be divided into disjoint sets $SP_1,...,SP_m$, so that SP_i contains paths which reach SM via S_i. Since by definition the statement condition to reach SM is the disjunction of the conditions on all of $P_1,...,P_n$, while the statement condition for each S_i is the disjunction of all the conditions on SP_i, the result follows.

Corollary: If a statement SM is at a merge, and the predecessors of SM have associated execution states $E_1,...,E_n$, with statement conditions $C_1,...,C_n$, then the statement condition in any execution state E associated with SM is $= C_1 \vee C_2 \vee ... \vee C_n$.

Proof: Since the statement condition recorded in an execution state is simply the statement condition of the associated statement the result follows from the above theorem.

Thus, when a merge statement is reached, the statement condition to be recorded in any execution state associated with the merge statement can be derived from the statement conditions in all execution states associated with the predecessors of the merge statement. This function is performed by the merge process. It cannot be invoked until all the execution states associated with predecessors have been created. This is ensured by the scheduling process discussed in section 3.3.6. Note that the merge process works in conjunction with the merge subprocess which simply changes the status of execution states reaching a merge statement to 'waiting merge'. The merge process is then invoked to create the new statement condition.

Thus the merge process retrieves the statement conditions from each input execution state and forms the disjunction of these expressions. This expression then becomes the statement condition on the output execution states which are given a status of active.

We now prove some additional results on the form of the statement condition at a merge statement. These results are used in the loop generalization described in Chapter 4.

For the first result we need a definition of the depth of conditional branching. We say that a conditional branch starting at fork SF1 is *inside* a conditional branch starting at fork SF2 if SF1 is on a path from SF2 to a statement at merge for SF2. A conditional branch starting at fork SF has *depth n* if there are n conditional branches starting at forks SF1,...,SFn, such that each SFi+1 is inside SFi, and SF1 = SF.

For the first theorem, recall that a merge statement SM is the last merge for the statement SF at a fork if SM is at a merge for SF and there is a path from every successor of SF to SM.

Theorem 3-2: If SM is a last merge for fork SF, then the statement condition at SF is equivalent to the statement condition at SM.

Proof: Proof is by induction on the depth of conditional branching of the conditional branch beginning at SF.

If the conditional branch starting at SF has depth 0, then there are no inner branches. Suppose there are paths $P_1,\ldots,P_n$ from loop entry to SM, with conditions $C\text{-}11,\ldots C\text{-}1m_1,\ldots,C\text{-}n1,\ldots,C\text{-}nm_n$. Then, by definition, the statement condition at SM is

$$(C\text{-}11 \wedge \ldots \wedge C\text{-}1m_1) \vee \ldots \vee (C\text{-}n1 \wedge \ldots \wedge C\text{-}nm_n).$$

Suppose the conditions occurring in paths from SF to SM are $C_1,\ldots,C_p$, some subset of $C\text{-}11,\ldots C\text{-}1m_1,\ldots,C\text{-}n1,\ldots,C\text{-}nm_n$. Since, by the induction hypothesis, there are no statements at a fork between SF and SM, $C_1,\ldots,C_p$ must be the conditions in the successor statements to SF. Thus, each path between SF and SM will include a single condition from $C_1,\ldots,C_p$. Now consider any path P_i from loop entry to SM. If P_i contains a condition from $C_1,\ldots,C_p$, then there is another path, P_j, identical to P_i except going through a different condition from $C_1,\ldots,C_p$. So, for any term, $(C\text{-}i1 \wedge \ldots \wedge C\text{-}im_i)$, in the statement condition at SM which includes a condition from $C_1,\ldots,C_p$, there will be a set of terms identical to this one except including different members of $C_1,\ldots,C_p$. But since $C_1 \vee \ldots \vee C_p = T$, these terms can be simplified by removing $C_1\ldots,C_p$.

Thus the statement condition at SM will not include any conditions occurring between SF and SM. Also, by theorem 2-1, any path from loop entry to SM goes through SF, so the statement condition at SM does not include any conditions not in the statement condition at SF, and therefore they are identical, as required.

We now suppose that the result is true for conditional branches of depth n-1, and we want to show it is true for conditional branches of depth n. We follow the statement condition as it varies down a path P from SF to SM. Suppose the statement condition at SF is C. If the successor to SF on P has condition C_i, then the statement condition at this statement will be $C \wedge C_i$. For any succeeding statement on P, we observe that if it is neither a condition or a merge, then the statement condition will be unchanged. If a statement at a fork is encountered on P, then by theorem 2-2, there will be another statement further down P which is a last merge for this statement at a fork. The conditional branch at this fork can have depth at most n-1, so by the induction hypothesis, the statement condition at the last merge will be the same as at the fork. Thus, apart from statements at merge for SF, the statement condition at the end of P will still be $C \wedge C_i$. But since, by theorem 3-1, the effect of merge statements on the statement condition is to form the disjunction of statement conditions at preceding branches, this means that the statement condition when SM is reached will be a disjunction of terms of the form $(C \wedge C_i)$. Now since SM is a last merge for SF, there are paths from each successor of SF to SM, and so each Ci will occur in at least one such term. Thus the statement condition at SM is

$$\begin{aligned} &(C \wedge C_1) \vee \ldots \vee (C \wedge C_p) \\ &= C \wedge (C_1 \vee \ldots \vee C_p) \\ &= C \wedge T \\ &= C \end{aligned}$$

as required.

Theorem 3-3: Suppose SM is at a merge for SF at a fork, and SF has successors S1,...,Sp which are on paths from SF to SM If S1,...,Sp contain conditions $C_1,...,C_p$, then the statement condition at SM is equivalent to the statement condition at SF $\wedge$ $(C_1 \vee ... \vee C_p)$.

Proof: Suppose the conditional branch starting at SF has conditional branching depth of n. The argument in the theorem above did not use the fact that SM was a last merge for SF except to assert $C_1 \vee ... \vee C_p = T$. Therefore, using the same reasoning we can show that the statement condition at SM = statement condition at SF $\wedge$ $(C_1 \vee ... \vee C_p)$ as required.

<u>Loop Generalization Process</u>

Input: a set of execution states associated with a loop entry statement that are all descended from a single execution state associated with the same loop entry statement (This execution state will be the one which has done zero iterations. All other execution states in the set will have completed a different number of iterations, varying from zero to the number of iterations required for generalization).

Output: a single generalized execution state, with the number of loop iterations unknown.

The restriction of the input to execution states descended from a single execution state associated with the loop entry requires some explanation. Suppose that a branch in the program prior to the loop has the effect that execution reaches loop entry down two different paths. This will result in two execution states being associated with the loop entry statement which have different path conditions or effects that are not caused by the loop. The loop generalization processes are intended to generalize the effects of the loop, so that the different execution states created during loop execution are generalized into a single execution state. However, we cannot expect these processes to generalize differences which arose outside the loop. Thus these processes operate on sets of execution states which only contain differences introduced during loop execution. This is achieved by restricting the set of execution states to ones descended from a single execution state associated with loop entry.

Chapters 4 and 7 describe this process.

<u>Loop Exit Process</u>

Input:	a set of execution states associated with all loop exit statements of a loop and all descended from a single execution state associated with the loop entry statement.
Output:	a set of updated execution states, with duplicate execution states removed and the value of the number of loop iterations known.

Chapter 5 describes this process.

<u>Interpreter Process</u>

Input:	all execution states having reached program end (i.e. those that with status 'waiting interpretation').
Output:	program specifications

Chapter 6 describes this process.

3.3.3 Symbolic Execution of Condition Statements

The only condition statement in the program language is the if statement.

If Statement

The input statement is in the form if <boolean expression>. The first step in processing this statement is to instantiate the boolean expression as required for updating the statement condition, by:

- replacing each variable by variable-identifier[value]

- replacing any expression of the form property(<file-identifier>) with property(<file-identifier>[(OBJECT-n]) where OBJECT-n is the current object from file <file-identifier> and any expression of the form property(<hand-identifier>) with property(<hand-identifier>[OBJECT-n]) where OBJECT-n is the object contacted by hand <hand-identifier>.

This instantiated expression is then used to update the statement condition if the execution state is in a loop whose associated execution states have not been generalized. If the statement condition on the input execution state is P, and the instantiated expression is Q, then the new statement condition is $P \wedge Q$. The remainder of this subprocess does not require that instantiations can be identified, so each term of the form variable[value] in the instantiated expression is simply replaced by 'value'. Also any expression of the form property(OBJECT-n) is replaced with the updated value of that property from the object data held on the execution state (if any)

At this point a 'real' executor would be able to determine whether the instantiated expression was true or false. In general, a symbolic executor will not be able to do this. However, in some cases, all or part of an expression may be provably true or false.

For example, we could have an if statement of

```
    if color(file-A) = green ∧ length(file-A) > 5
```

Now, if the execution state has OBJECT-1 as the current object in file-A, and the path conditions of the execution state contain

```
    length(OBJECT-1) = 20,
```

then we can prove that the second conjunct of the above expression is true. This still doesn't determine the truth of the whole expression, but leaves an 'unresolved expression' of

```
    color(OBJECT-1) = green.
```

The advantages and disadvantages of trying to determine the truth of of an if predicate are discussed below in section 3.3.7. The current version of PAN allows the user to specify whether a theorem prover should be invoked when an if statement is processed.

The theorem prover used by PAN is discussed in section 3.3.7.1, and includes the ability to return the 'unresolved expression' if the expression cannot be proved either true or false.

Assuming then that this feature is enabled, PAN passes the theorem prover the instantiated expression and a list of known facts obtained from the execution state.

If the theorem prover returns F, PAN simply sets the status of the new execution state to dead.

If the theorem prover returns T, PAN simply exits from the if subprocess.

If the theorem prover returns an unresolved expression, P, then this is used to update the path conditions. If the input execution state has a path condition of Q, then it is updated to $P \wedge Q$.

3.3.4 Symbolic Execution of Flow Control Markers

Stop Statement

The stop subprocess simply sets the status of the execution state to 'waiting interpretation'.

Loop Entry Statement

If the execution state currently has no data on the loop being entered, then the new loop is added to the bookkeeping data with iteration count of zero, and generalization indicator set to 'ungeneralized'.

If this loop has already been generalized, then the effects of a further loop iteration are used to verify the generalization. This is described in Chapter 5. If the loop has not been generalized, then the iteration count is incremented. If the iteration count has now reached the number of iterations required before loop generalization, then the execution state is changed to 'waiting generalization'.

Loop Exit Subprocess

If the data for the innermost loop on the execution state shows generalization has not been performed, then loop exit is premature, and PAN changes the status of the execution state to 'dead'. Otherwise PAN changes the status to 'waiting exit processing' so that this execution state will act as input to the loop exit process.

Merge Statement

The only processing required when a merge statement is reached, is to update the statement condition by the merge process described in section 3.3.2. This information is only maintained for execution states in loops, therefore if the input execution state is in a loop, then its status is changed to 'waiting merge', otherwise the subprocess does nothing.

3.3.5 Symbolic Execution of Action Statements

Action statements provide the program language with the ability to modify the world in which the program executes. Thus, these statements vary depending upon the domain. Introduction of a new domain or extensions to the language facilities provided in the dp and robot domains would require additional action statements and corresponding subprocesses to symbolically execute them. No other changes to the PAN system should be required.

Move-to Statement

The move-to statement changes properties of the hand specified in the statement and the position of any object the hand is grasping. The position of the hand is set to the position in the statement. If the hand was grasping, the grasped object's position is also set to this position. Otherwise the robot status is set to no contact.

Move-by Statement

Processing is the same as for move-to, except that the new position is the current position of the specified hand plus the increment specified in the input statement.

Move-until-contact Statement

The move-until-contact statement defines a line (source) by (position, angle), where position is the current position of the robot hand referenced in the input statement, and the angle is as specified in the input statement.

This source may not have been previously encountered, in which case it will need to be added to the global data, and execution state source data initialized.

Since this statement always succeeds in finding a new object, PAN increments the number of objects retrieved from the source before determining the definition of this new object as

    sequential-object-in-source(number of objects retrieved,
                                                  source identifier).

This object is added to the global data if necessary. Finally, we set the position of the robot hand specified in the input statement to be at the same position as the new object i.e. to be at

position(object-identifier), and both the current object for source and object contacted by hand to be this new object.

Move-until-contact-up-to Statement

The move-until-contact-up-to statement defines a line (source) by (starting-position, angle, length), where starting-position is the current position of the robot hand specified in the statement, while angle and length are as specified directly in the input statement.

As for move-until-contact, this source may need to be added to the global data and the execution state source data initialized. Unlike move-until-contact, the processing for this statement has to deal with the complexity of a conditional outcome - a new object may or may not be found. The outcome of this condition depends on the number of retrievals attempted from the source compared to the number of objects originally in the source, referred to as size(SOURCE-n), where SOURCE-n is the identifier of the source. Another object will be found if size(SOURCE-n) ≥ number of retrievals attempted, and no object will be found if size(SOURCE-n) < number of retrievals attempted. As discussed in section 3.2.1, the 'number of retrievals attempted' is a special type of variable, not explicitly referenced in the input program, but implicitly required by the use of a sequential source. For a given source, SOURCE-n, we can give this variable the name SOURCE-n-number-of-retrievals-attempted. Using this variable, we can express the conditions for the source having or not having another object as:

```
    size(SOURCE-n) ≥ SOURCE-n-number-of-retrievals-attempted
```

and

```
    size(SOURCE-n) < SOURCE-n-number-of-retrievals-attempted.
```

These conditions, will be recorded in the statement condition as

```
    size(SOURCE-n) ≥ SOURCE-n-number-of-retrievals-attempted
                        [value of
                          SOURCE-n-number-of-retrievals-attempted]
```

and

```
    size(SOURCE-n) < SOURCE-n-number-of-retrievals-attempted
                      [value of
                          SOURCE-n-number-of-retrievals-attempted].
```

Thus, to process the move-until-contact-up-to statement, we first check to see if the line is already exhausted. If so, we set the hand position to the end of the line (i.e. current position plus length at specified angle). In addition, if the execution state is in a loop which has not been generalized, we add

```
    size(SOURCE-n) < SOURCE-n-number-of-retrievals-attempted
                        [value of
                          SOURCE-n-number-of-retrievals-attempted]
```

to the statement condition.

If the source has not already been marked exhausted, we need to show two possible results i.e. the source is now exhausted or it is not. To do this we create a new execution state as a copy of the input execution state.

Then one execution state is updated to reflect the condition where the line is exhausted by:

- setting the position of the robot hand to current position plus length at specified angle

- adding

```
  size(SOURCE-n) < value of
                      SOURCE-n-number-of-retrievals-attempted
```

  to the path conditions.

- adding

```
 size(SOURCE-n) < SOURCE-n-number-of-retrievals-attempted
                   [value of
                    SOURCE-n-number-of-retrievals-attempted]
```

  to the statement condition if the execution state is in a loop whose associated execution states have not been generalized.

The other execution state is updated to reflect the condition where the line is not exhausted by:

- creating a new object as in move-until-contact

- setting position of robot hand to be the initial position of this object, specified as position(object-identifier), and object contacted to be object-identifier.

- setting last object taken from the source to be object-identifier

- adding

```
size(SOURCE-n) ≥ value of
                      SOURCE-n-number-of-retrievals-attempted
```

to the path conditions.

- adding

```
size(SOURCE-n) ≥ SOURCE-n-number-of-retrievals-attempted
                  [value of
                   SOURCE-n-number-of-retrievals-attempted]
```

  to the statement condition if the execution state is in a loop whose associated execution states have not been generalized.

<u>Grasp Statement</u>

The robot status slot is updated to show the hand specified in the statement is grasping.

<u>Ungrasp Statement</u>

The robot status data is updated to show the hand specified in the statement is not grasping.

<u>Sequential Read Statement</u>

The processing required for this statement is the same as for move-until-contact-up-to apart from the following differences:

- object source is identified by file name rather than by (starting-position, length, angle)
- robot status data is not used.

<u>Sequential Write Statement</u>

The current object for the specified source (file) is found from the source data in the execution state. It is an error if there is no current object (since in this case the data to be written to the file has not been specified).

The number of objects written is incremented.

The object definition of the current object is created as sequential-object-in-source(number of objects written, source-identifier) and the source data is updated to show that this source has no current object.

Keyed Read Statement

The keyed read statement retrieves an object from the specified source (file) by key. The key is specified by a predicate which the object must satisfy. The source may not have been previously encountered, in which case it will need to be added to the global data and execution state source data initialized.

This is a conditional statement in that an object satisfying the predicate P as specified in the statement may not exist. Thus we create a new execution state as a copy of the input one.

Then one execution state is updated to reflect the condition where no object that satisfies the predicate exists by:

- recording that the source has no current object
- adding

  $$\neg \exists\, o\ (P(o) \wedge o \in \text{source-identifier})$$

  to the path conditions. This is also added to the statement condition if the execution state is in a loop whose associated execution states have not been generalized.

The other execution state is updated to reflect the condition where an object satisfying the predicate does exist by:

- creating a new object defined as

  keyed-object-in-source(P, source-identifier)

  with an identifier of OBJECT-n.

- set current object in source to OBJECT-n

- adding

  $\exists$ o (P(o) $\wedge$ o $\in$ source-identifier)

  to path conditions. This is also added to the statement condition if the execution state is in a loop whose associated execution states have not been generalized.

<u>Keyed Write Statement</u>

The current object for the specified source is found from the sources data in the execution state. It is an error if there is no current object (since in this case the data to be written to the file has not been specified).

The source data is updated to show no current object.

<u>Assignment Statement (:=)</u>

The right hand side of the assignment statement is first evaluated by:

- substituting for each variable its value

- substituting property(OBJECT-n) for any expression of the form property<file-identifier> where OBJECT-n is the current object from file file-identifier and property(OBJECT-n) for any expression of the form property(hand-identifier) where OBJECT-n is the object contacted by hand hand-identifier

- substituting the value of property(OBJECT-n) for any expression of the form property(OBJECT-n) where the value property(OBJECT-n) is obtained from the object data held on the execution state (if any)

This expression is then simplified and used to update the variable value or object property as specified in the left hand side of the assignment statement.

In the case of a dp program writing an output file, this may be the first reference to the modified object or even the first reference to the file the object is in. In the first case the object is created; in the second case both the object and the source are created and the object is made the current object for this source. However, the creation of an object in this way does not establish the object's definition (in fact the object may never be written to the file) and so it left undefined until the sequential write is processed as described above.

3.3.6 PAN Scheduling

Given the number of processes which PAN has available, how is it to decide which to run?

The simplest of these processes is the initialization process. It is run once, when PAN begins analysis of a new program. Conversely, the interpretation process analyses the results of the symbolic execution, and is only run when all other processes are complete.

The three processes merge, loop generalization and loop exit require a set of execution states to be available before they can run successfully. However, instead of trying to run one of these processes and having to check whether all required execution states are available, PAN uses a simpler approach. It assigns priorities to the processes in such a way that a process will never be initiated unless all required execution states are available.

To determine these priorities, we first observe that PAN's loop analysis method requires that it first symbolically executes loops, then generalizes them and finally performs exit processing. This means that within a given loop, the scheduler must assign priorities so that

priority (symbolic executor) > priority (loop generalizer)

and

priority (loop generalizer) > priority (exit process).

To place the merge process in this priority list, we note that the merge process cannot take place until all execution states to be merged have been generated i.e. we require

priority (symbolic execution) > priority (merge process).

Also, since the merge process can occur inside a loop (but not vice versa), in order to complete the symbolic execution of the loop, we require

priority (merge process) > priority (loop generalization).

This is not yet adequate to define the priority of the merge process as a single loop may contain several merge statements each having associated execution states which are waiting merge processing. This problem may be addressed by the observation that we need to ensure that the merge process will not be initiated until an execution state has arrived from all merging branches. This can only be guaranteed if we process the merge statements in 'execution order' i.e. give higher priority to merging execution states associated with statement m over those associated with statement n, where there is a path from m to n within the loop.

To clarify this point, consider the merge statements in figure 3-5. Suppose statement 1 has associated execution states from branches A and B, and statement 2 has associated execution states from branch C. We must perform the merge of execution states associated with statement 1 before these execution states can reach statement 2, allowing the merge at statement 2 to proceed.

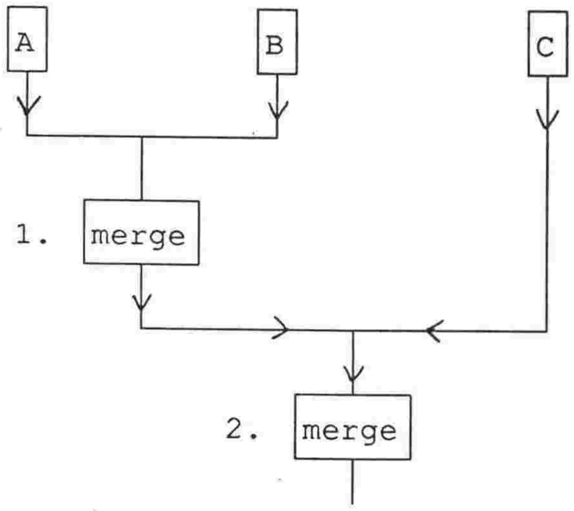


Figure 3-5 - Priority of Merging Execution States

Finally we consider the priority of programs with nested loops. In this case we may think of an inner loop as a single compound statement within the outer loop. This 'statement' will need to be executed before loop generalization of the outer loop can proceed. Since the inner loop will contain its own loop entry, loop exit and, possibly merge statements, we require that the scheduler will give higher priority to processing execution states associated with statements in the inner loop over processing execution states from the outer loop.

The above requirements are met by the following scheduling algorithm:

Scheduling Algorithm

```
IF there are no execution states

    perform initialization

ELSE
         IF all execution states are either dead or 'waiting
         interpretation'

              perform interpretation
              exit PAN

         ELSE IF any active execution states exist

              pick one and pass to symbolic executor

         ELSE

              collect all execution states which have the maximum
              number of loops in loop data (i.e. are associated with
              statements from the innermost loop).

              IF any of these execution states have status 'waiting
              merge'

                   pass to merge scheduling

              ELSE IF any of these execution states have status
              'waiting generalization'

                   pass to generalizer scheduling

              ELSE

                   pass all of these execution states with status
                   'waiting exit processing' to exit scheduling.
```

Merge Scheduling

From all input execution states, retain those which are earliest in execution order. Statement n is earlier than statement m if there is a path from n to m inside the loop (i.e. without passing through a loop exit statement for this loop - see example in figure 3-5 above).

From these execution states, pick any set which are associated with a single merge statement, and are descended from a single execution state associated with the loop entry, and pass to the merge process.

Generalizer Scheduling

Pick any set of execution states associated with a single loop entry statement, that are descended from a single execution state associated with the loop entry statement, and pass to the loop generalization process.

Loop Exit Scheduling

Pick any set of execution states associated with all loop exit statements from a single loop, which are descended from a single execution state associated with the loop entry statement, and pass to the loop exit process.

3.3.7 The Case for Theorem Proving

As has been discussed above, a symbolic executor is generally not able to prove conditional statements either true or false. Nevertheless, PAN's 'if' statement processing discussed above does attempt to do so by calling a theorem prover. Since theorem proving considerably slows the program analysis, is it worth doing at all? This is a question which does not seem to have been addressed in the literature on symbolic execution.

There can even be some advantages in not trying to do it. Consider the program shown in figure 3-6. Assuming that PAN performs less than

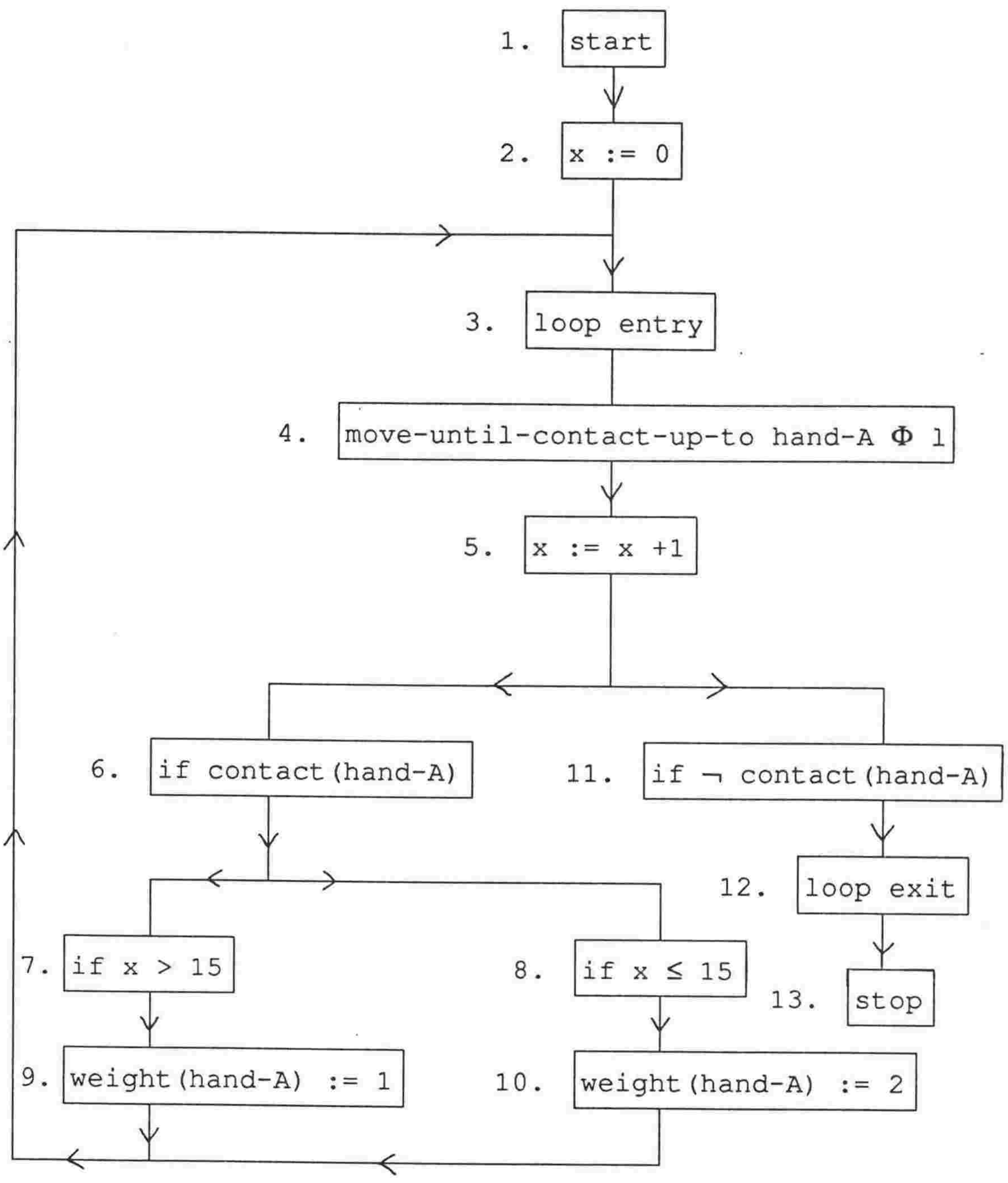


Figure 3-6 - Program Analysed Better Without Theorem Proving

---

15 loop iterations before loop generalization, then by using theorem proving PAN will always be able to prove that the condition in statement 7 is false and statement 9 will never be executed. This will result in incomplete loop analysis.

If, however, no attempt is made to prove statements 7 and 8 true, then PAN will traverse each path on each iteration. At loop generalization, the following sequences will be generated:

SEQUENCE-1 = (sequence i = 1 to k sequential-item-in-source(i, SOURCE-1))

SEQUENCE-2 = (item: item $\in$ SEQUENCE-1 $\wedge$ x > 15)

SEQUENCE-3 = (item: item $\in$ SEQUENCE-1 $\wedge$ x $\leq$ 15).

Since the definition of SEQUENCE-2 and SEQUENCE-3 contain a variable, x, PAN will try to express the value of x in such a way that it can be removed from the definitions of these sequences. In this case x is always equal to the length of SEQUENCE-1, which will allow SEQUENCE-2 and SEQUENCE-3 to be reexpressed as:

SEQUENCE-2 = (item: item $\in$ SEQUENCE-1 $\wedge$
position-in-sequence (item, SEQUENCE-1) > 15)

SEQUENCE-3 = (item: item $\in$ SEQUENCE-1 $\wedge$
position-in-sequence (item, SEQUENCE-1) $\leq$ 15)

The process used here to remove variables from sequence definitions is fully explained in Chapter 4. Once these sequences have been generated a complete analysis of the program will follow.
Although this method works well in this case, it does so at the cost of allowing execution states to hold inconsistent information. In particular, the first frame which passes statement 7 will have

x > 15 added to the path conditions

and

x = 1 in the variables data.

Even worse, in other programs, this method can lead to incorrect analysis. Consider the program fragment in figure 3-7.

In this case, at the end of symbolic execution, the execution states to be interpreted will include an execution state which describes the effect of passing through statements 2, 4 and 7. This execution state, and will contain a path condition of

weight(OBJECT-1) = heavy $\wedge$ color(OBJECT-1) = red

and corresponding effects. However, since the path 2, 4, 7 could not be traversed by any real executor, the resulting analysis is incorrect.

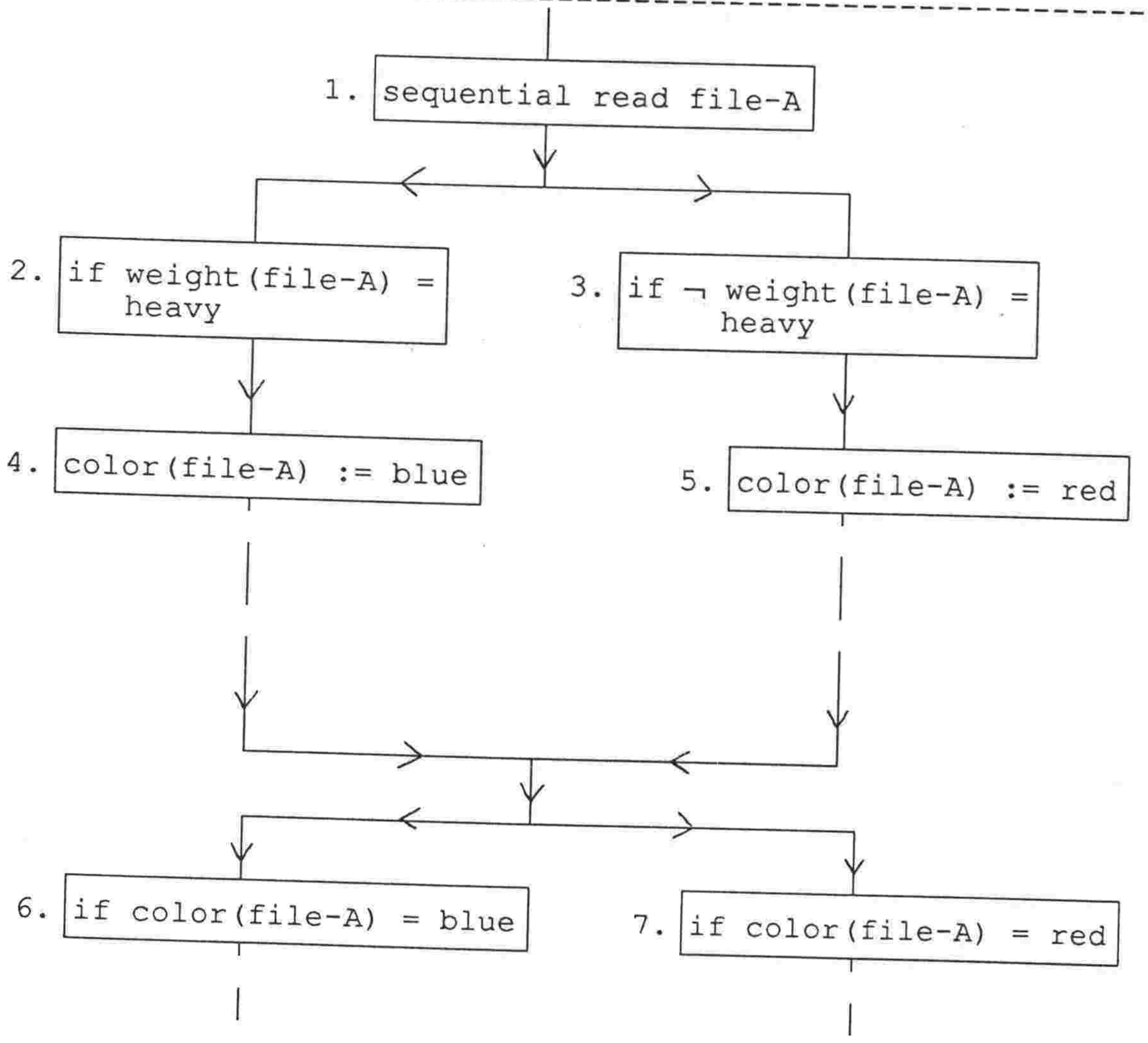

1. sequential read file-A
2. if weight(file-A) = heavy
3. if ¬ weight(file-A) = heavy
4. color(file-A) := blue
5. color(file-A) := red
6. if color(file-A) = blue
7. if color(file-A) = red

Figure 3-7 - Program Analysed Incorrectly Without Theorem Proving

To avoid producing such results, PAN always attempts to prove conditional statements either true or false using the theorem prover described below.

The correct method to perform a symbolic execution analysis on such a program is to let the generalized execution state produced from the first loop generalization initiate another set of loop iterations, producing more execution states for generalization. These would then be used to produce a second generalized execution state and this process would continue until loop generalization produces no new information.

PAN does not currently perform more than one loop generalization, but should be extended to do so.

3.3.7.1 Theorem Proving

Theorem proving could be supplied by a system external to PAN. Obviously, PAN would also need to supply an external theorem prover with the currently known facts. These are available from the path conditions, variables, objects and object sequences data held on the execution state.

Since a convenient theorem prover was not available when PAN was constructed, a simple theorem prover was developed. This theorem prover is not adequate as a general theorem prover but is satisfactory for the programs so far analysed by PAN. It should not been seen as a part of the contribution of this thesis and is only described here briefly for completeness.

This theorem prover is passed a candidate predicate and a list of known facts.

The theorem prover extends this list of facts by using the following rules:

from	(P ∧ Q)			generate	P, Q
"	(< P Q)			" "	(≤ P Q)
"	(= P Q)			" "	(≤ P Q)
"	(= P Q)			" "	(= Q P)
"	(≥ P Q)			" "	(≤ Q P)
"	(> P Q)			" "	(< Q P)
"	(≤ P Q)			" "	(≥ Q P)
"	(< P Q)			" "	(> Q P)
"	(< P Q)	and	(< Q R)	generate	(< P R)
"	(≤ P Q)	"	(≤ Q R)	" "	(≤ P R)
"	(≤ P Q)	"	(≥ P Q)	" "	(= P Q)

The components of the candidate predicate are now examined recursively. For each component P, if P is a known fact, replace P by T in the candidate predicate. If ¬P is a fact, replace P by F in the candidate predicate.

Once this process is complete, the resulting predicate is simplified and returned.

Chapter 4

Loop Generalization

4.1 INTRODUCTION

The most distinctive feature of PAN as a program analysis system is its method of analysing loops. PAN analyses loops by generalizing the effect of a few iterations, to produce the effect after an indefinite number of iterations. The idea of using generalization for this purpose was largely derived from the observation that human programmers can often describe a loop after an informal symbolic execution of two or three iterations.

The task of such a generalization process is to produce a single execution state that generalizes the effect of several execution states. More specifically, PAN must determine the objects and sources that will exist in this generalized execution state, describe their updated properties (modified values of fields in the dp domain or physical properties in the robot domain) and determine the value of any variables.

The task of determining the objects and sources in the generalized execution state is a specific case of the more general task:

> given a set of observations $\{O_1,...,O_n\}$, each of which describe a set of items, determine the items which would be in an observation O_k associated with integer k.

Note that the items in Ok may not simply be a function of k, but may depend on other facts in Ok, particularly properties of other items. For example, if we had

O_1:	items = {1 red ball, 1 block}
O_2:	items = {1 red ball, 1 green ball, 1 block}
O_3:	items = {3 red balls, 3 blocks}
O_4:	items = {1 red ball, 3 green balls, 1 block}

then a possible value for O_k is

O_k: k balls, n blocks (where n = number of red balls)

So, in this case, the number of blocks depends on the number of red balls.

A system addressing a generalization problem similar to this is the SPARC/E system described in Dietterich and Michalski[1985]. This system has been designed to play the game of Eleusis, which involves finding patterns in sequences of playing cards. Given a sequence of k cards $<card_1,...,card_k>$, SPARC/E finds the set Q_{k+1} of admissible next cards. This set may depend on any of the properties of cards $<card_1,...,card_k>$.

PAN's task is in some ways harder in that an exact description of the objects in the generalized execution state is required, not just a set of admissible ones. On the other hand, sequences in Eleusis are often quite complicated, whereas investigations with PAN have shown that sequences required for program analysis are usually much simpler.

Thus the specific techniques used by Dietterich and Michalski were not found suitable for use in PAN. However, their approach in trying to fit the input data to various parametized models has been successfully adopted in PAN. Deitterich and Michalski define a model as follows:

> "A model is a structure that specifies the syntactic form of a class of descriptions. A model consists of model parameters and a set of constraints that the model places on the forms of the descriptions. The process of specifying the values for the parameters of a model is called parametizing the model. The process of filling in the form of the parametized model is called instantiating the model."

never worked well in SPARC!

Separate models are used for determining the objects in the generalized execution state, describing their updated properties and finding the value of variables. These are further discussed below. None of PAN's models use parameters.

Once PAN has determined the objects and sources that should be included in a generalized execution state it must determine the updated properties of the objects. This task is a special case of the more general task:

> given a set of observations $\{O_1,...,O_n\}$, each of which describes the properties of a fixed set of items $\{I_1,...,I_m\}$, determine the properties of these items in observation O_k.

If we added the restriction that the properties can be divided into descriptors (those which can be observed) and symptoms (those which we are trying to predict) and the symptoms of an item must only depend on other descriptors of that same item, this generalization could be performed using generalization methods developed for learning systems, such as those described in Michalski[1983].

To use these methods, the observations $\{O_1,...,O_n\}$ must only contain expressions of the form

$$P_i(\text{item}) \rightarrow Q_i(\text{item})$$

where P_i is some predicate made up from descriptors and Q_i is a symptom.

These would be generalized to some expression

$$P(\text{item}) \rightarrow Q(\text{item})$$

true in all observations.

We now show that PAN's generalization task cannot be expressed in this way, so that the methods described in Michalski[1983] cannot be used. Since each observation used by PAN is, in fact, an execution state, we can, for each observation, obtain an expression

$$P \rightarrow Q$$

by setting P to be the path conditions and Q the effects of the execution state. To obtain the form above, the expression $P \rightarrow Q$ would need to be restated as:

$$(P1 \rightarrow Q1) \wedge (P2 \rightarrow Q2) \wedge \ldots \wedge (Pn \rightarrow Qn)$$

where each $Pi \rightarrow Qi$ involves only a single item, Pi being constructed from descriptors and Qi from symptoms. However, the execution state expression, $P \rightarrow Q$ generated by PAN cannot always be represented as subexpressions involving only one item. For example, if PAN is analysing a loop which includes the condition

```
if weight(A) = weight(B)
```

then the execution states to be generalized may include expressions of the form

$$(\text{weight(OBJECT-1)} = \text{weight(OBJECT-2)}) \rightarrow Q$$

for some Q, which cannot be reexpressed to involve a single object.

Because of this PAN instead uses the model approach mentioned above to generalize execution states.

Producing a generalized execution state also requires PAN to generalize the value of variables. The special nature of variables adds peculiarities to the generalization task which seem to occur only in program related problems (e.g. program analysis or program verification), and do not seem to have been previously approached using generalization. PAN uses models to express variable values in terms of generalized objects in the generalized execution state, as described below.

variables?   plotting?

In addition to the use of these models, another generalization technique used in Dietterich and Michalski[1985] and adapted for PAN, is that of adding derived properties. Simply stated, this technique consists of adding derived attributes to those explicitly provided, before generalizing. Generalization may sometimes be successful using the derived attributes where it would be unsuccessful if restricted to the explicitly provided input attributes. For example, in the game Eleusis, a particular card may be described as 'jack of hearts', but generalization may need to use the fact that the card is 'red' or 'a picture card'.

constructive induction (defined also with Michalski)

In PAN this technique has been considerably extended. Instead of just looking for derived attributes of a single object, PAN can use all facts available in an execution state to derive additional facts for use in generalization. Since, in PAN, this technique has become very computationally expensive, it is only used when necessary. Thus PAN first tries to generalize the execution states without using derived facts, and only includes them if this process fails. The generation of derived facts is described in Chapter 7, and this chapter merely mentions how the decision to use this method is made.

4.2 OVERVIEW OF PAN GENERALIZATION

The task addressed by PAN generalization is to produce a single execution state that describes the effect after an indefinite number of iterations, given the execution states describing the effect after a few iterations.

PAN divides this task into three major components:

- describe the objects that would exist in a generalized execution state using the sequence terminology described in Chapter 2.

- describe the effect that the loop has on updated object properties by describing them in terms of sequences.

- find the value of each variable in the generalized execution state, expressed in terms of sequences.

These tasks show that sequences are a key component in PAN's loop analysis method. A PAN sequence is conceptually similar to the temporal sequences developed as part of the Programmers Apprentice Project [Waters 1979]. Given a statement in a loop which is repeatedly executed, we can form a sequence of execution states associated with that statement, that were created during the first, second, third,.. etc iteration of that loop. From this sequence of execution states, sequences of items created in each execution state can be formed. (This is a simplification since PAN will actually have many execution states which have completed any given number of iterations.) Such a sequence is temporal in the sense that items are created over time i.e. one in each iteration. Treating this sequence as a non temporal aggregate item allows the loop effects to be described in a non iterative manner, eventually leading to non procedural program specifications being produced.

For example, consider the program fragment in figure 4-1. The program has been coded so that each record read from file A is separately updated so that its color field has the value red. The loop is used as the mechanism for processing each record. If, however, the sequence of records created at statement 2, say SEQUENCE-1, is treated as an aggregate object, the program can be represented as having the effect shown in figure 4-2. The programs shown in figures 4-1 and 4-2 have the same effect, but figure 4-2 does not include a loop and so leads easily to the production of specifications.

PAN generalization does not try to explicitly produce programs expressed in terms of sequences as shown in figure 4-2. Instead it uses sequences to describe the effects of the loop in a non iterative manner.

As mentioned above, PAN sequences are conceptually similar to the temporal sequences used in the Programmer's Apprentice. However there are differences. The Programmer's Apprentice constructs sequences

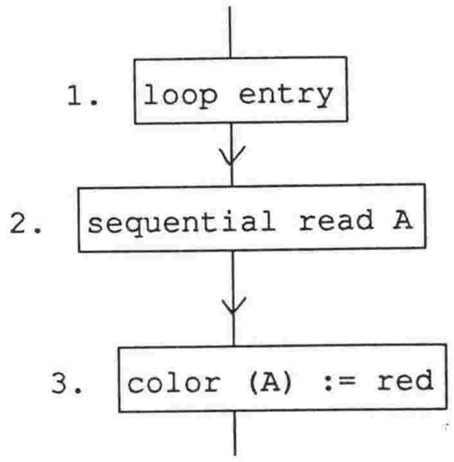


Figure 4-1 Sequence of Records Being Processed Individually

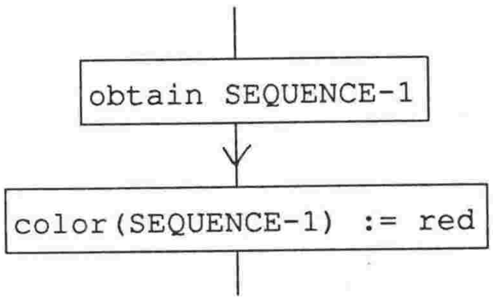


Figure 4-2 Sequence of Records Processed as an Aggregate Object

---

from examination of program code, whereas PAN constructs sequences from examination of execution states. This ensures that PAN can construct sequences regardless of peculiarities in coding. A more significant difference is in the use of sequences. Since the Programmer's Apprentice is not a symbolic execution system, it does not have a description of the effects of the program. Since PAN does have such a description, it is able to use sequences to produce a non procedural description of programs containing iteration.

4.3 SEQUENCE GENERATION

4.3.1 Introduction

Having given an overview of the role that sequences can play in producing non iterative specifications for loops, we now introduce the method by which sequences are generated. PAN generates two type of sequences, primitive sequences and subsequences. Primitive sequences are expressed as (sequence i = 1 to n D(i)), where D(i) is the definition of the ith item in the sequence. Thus to generate a primitive sequence, PAN needs to

- identify the objects in it
- determine the generalized description of the objects in terms of the sequence variable
- determine an expression for the length of the sequence.

Subsequences are expressed as (item: item ∈ S ∧ P(item)), where S is a sequence and P is a predicate. Thus to generate a subsequence PAN needs to find a predicate which can be applied to the items in the sequence S. Subsequences are equivalent to the Programmer's Apprentice temporal sequences generated by a filter.

Generating primitive sequences is an inductive process since a sequence describes items which will be generated in iteration k, from the items which have actually been produced in the first few iterations. This process is hard, principally because each execution state being generalized will have a different combination of objects to be included in any given sequence, and because we allow items to be defined in terms of other items. Consequently, if item-4, item-5 and item-6 are defined in terms of item-1, item-2 and item-3 which are included in sequence S1, a sequence S2, which can be generated from item-4, item-5 and item-6 may not become obvious until the definition of these items is reexpressed in terms of S1. This implies that sequences need to be generated in the correct order.

Subsequences are generated by applying predicates to the primitive sequences or previously generated subsequences. These predicates are based on the conditions which occurred in the conditional statements within the loop. Generation of these predicates to apply to a sequence S is, however, far from trivial because:

- the condition may refer to several items, some included in S and others not

- the condition may have included a variable with a value set in some previous iteration of the loop. Thus variables can disrupt the orderly temporal nature of the iteration by retaining values from an earlier time.

To make these ideas more concrete, we now look at some example loops and the sequences required to analyse them. Considering how PAN can generate these sequences will suggest a sequence generation procedure which we then try to apply to a general loop structure.

<u>Example Sequence Generation</u>

As a first example, consider again the program fragment first presented as figure 2-13, and reproduced here as figure 4-3. The interpretation in section 2.2 used the following sequences

```
SEQUENCE-1 = (sequence i = 1 to n sequential-object-in-source(i,
                                          SOURCE-1))
SEQUENCE-2 = (sequence i = 1 to n sequential-object-in-source(i,
                                          SOURCE-2))
SEQUENCE-3 = (item: item ∈ SEQUENCE-1 ∧ weight(item) =
                    weight(map(item, SEQUENCE-1, SEQUENCE-2)))
```

where SOURCE-1 and SOURCE-2 are defined as file(A) and file(B).

How are these sequences generated? The execution states to be generalized will all include objects defined as:

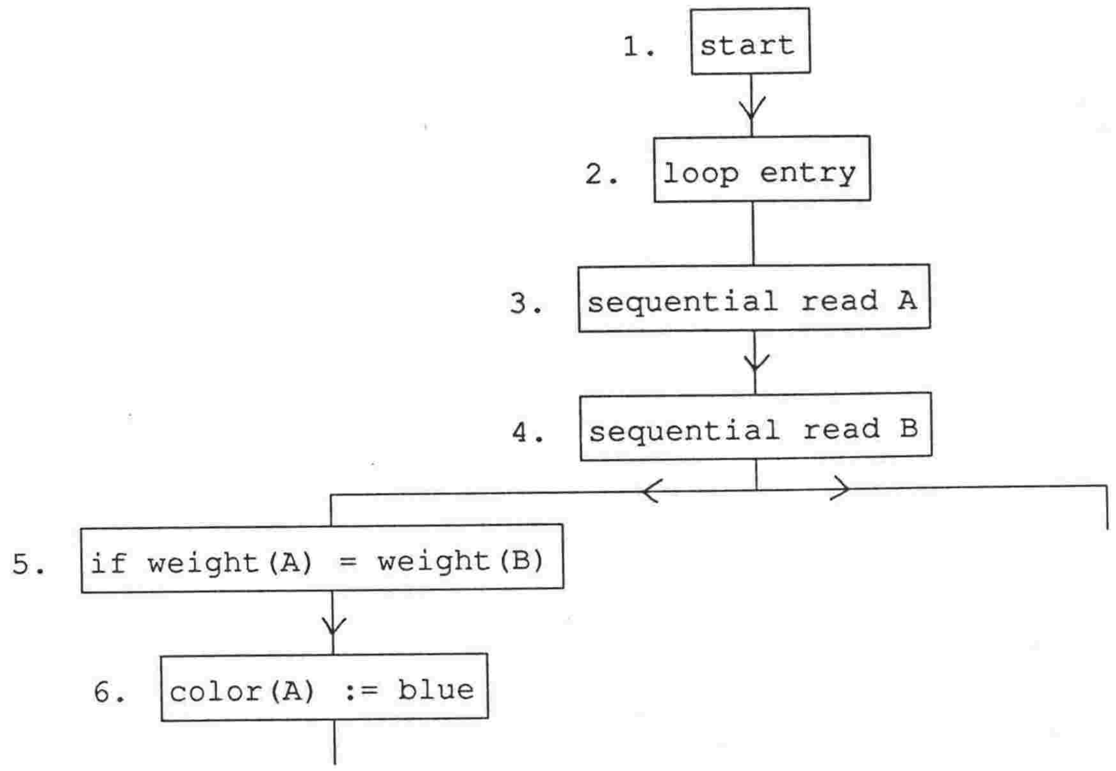


Figure 4-3 - Subsequences using two sources.

---

sequential-object-in-source(i, SOURCE-1)

and

sequential-object-in-source(i, SOURCE-2)

for i from 1 to the number of loop iterations performed. Thus SEQUENCE-1 and SEQUENCE-2 can be generated by

- recognizing that there are sets of objects whose definitions vary only in a single integer
- putting the objects into order of creation, giving this integer the values 1, 2, 3,... i.e. the ith value is i.

The length of these sequences will, at this point, simply be the number of iterations k (a value for k will be found during exit processing as explained in chapter 5).

Generating SEQUENCE-3 requires the recognition that the condition in statement 5 is applied to the objects in SEQUENCE-1 and SEQUENCE-2. During program execution, the condition in statement 5 will have been instantiated with objects which are now included in SEQUENCE-1 and SEQUENCE-2; this may suggest that the rule for using conditions to produce subsequences is

> if the condition has been instantiated with objects which are included in SEQUENCE-n, then use the condition to form subsequences of SEQUENCE-n.

However, this is not adequate as can be shown by considering the program in figure 4-4.

Interpretation of this program requires that a value for v, as updated by statement 6, be expressed in terms of a sequence. Since v is the sum of weights of objects passing through statement 6, which are those objects from SEQUENCE-1 that satisfied the condition in statement 5, we now require the subsequence

```
SEQUENCE-3 = (item: item ∈ SEQUENCE-1 ∧
                  weight(map(item, SEQUENCE-1, SEQUENCE-2)) < 10)
```

So, in this case we need to form a subsequence of SEQUENCE-1 using the condition in statement 5, even though that condition is never instantiated with objects in SEQUENCE-1.

Thus, in general, we need to use the condition in an if statement to form subsequences of every sequence which has objects which can be referenced in the statements succeeding the if statement. This means generating subsequences from all sequences consisting of objects that were the current object from a file or the object in contact with a robot hand when the if statement was executed. Using of the map function in creating these subsequences requires the sequences to

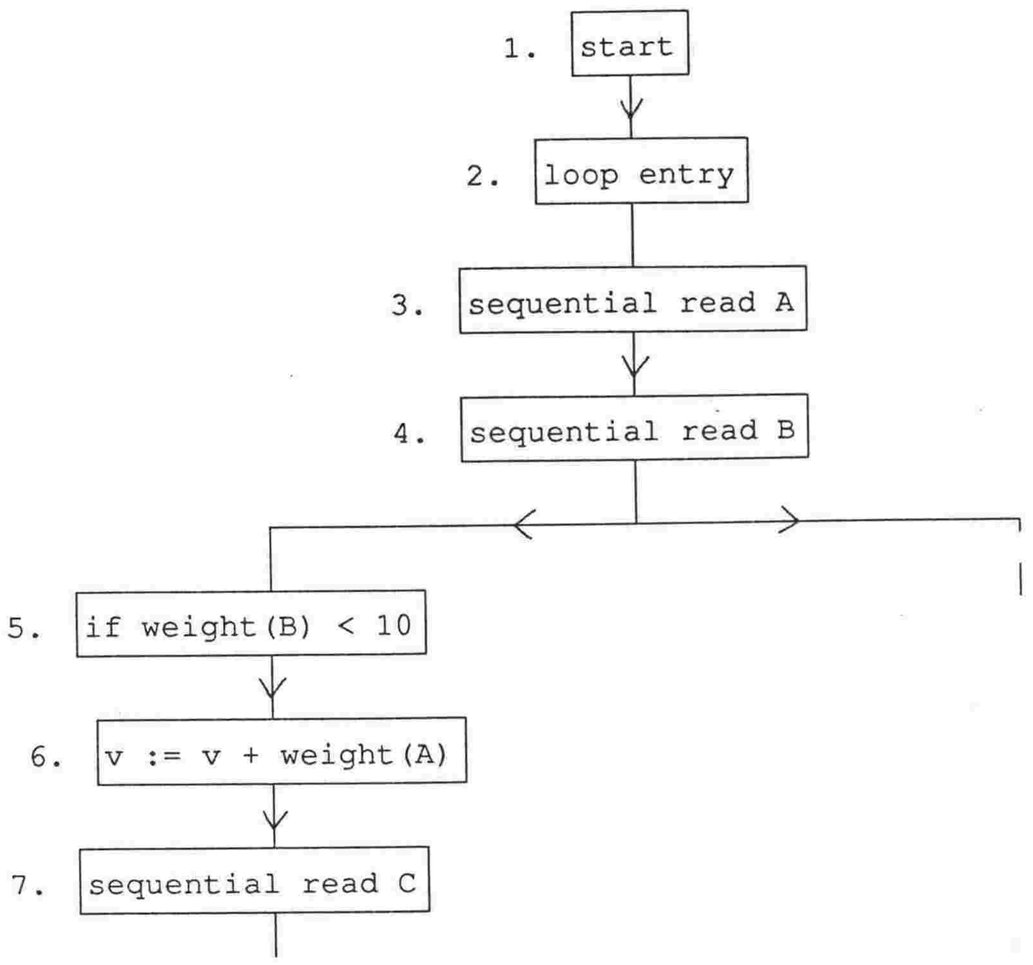


Figure 4-4 - Example of Loop Structure

---

have the same length. We will see, later in the chapter, that PAN's sequence creation procedure ensures that sequences that consist of objects current when an if statement was executed will be of equal length.

Having looked at the subsequences that need to be created using the condition in an if statement, we now consider the length of sequences consisting of objects created when executing statements in the branch beginning at that if statement. Such a sequence which will be generated by objects created by statement 7 in figure 4-4:

```
SEQUENCE-4 = (sequence i = 1 to n sequential-object-in-source(i,
                                          SOURCE-3)).
```

The length of this sequence, n, in any execution state will equal the number of times statement 7 was executed. But this is equal to the number of times statement 5 was true, which is the length of SEQUENCE-3. Obviously, if we want to express n in terms of SEQUENCE-3, we need to have generated SEQUENCE-3 before SEQUENCE-4.

This suggests that generation of sequences for analysing the program in figure 4-4 needs to take place in the following order:

- generate sequences for objects created in the unconditional code of statements 3 and 4

- form subsequences of these sequences using the conditions in all the if statements succeeding statement 4

- form sequences for objects created in statements following these if statements such as statements 6 and 7.

<u>Developing a Sequence Generation Algorithm</u>

We now extend the reasoning above by considering loops with an arbitrarily complicated structure of conditional branching. We first examine the effect of condition statements in sequence generation and then the effect of merge statements.

Consider the condition statements in the loop shown in figure 4-5. Each "branch i" is a branch as defined in Chapter 2. Suppose that on each iteration objects were created when executing statements in branch 0. To generate sequences from these objects, we can look for groups of objects whose definitions vary only in numbers whose values can be expressed as a function of the object's position in the group, arranged in order of creation. Let the sequences generated be referred to as S1,...,Sp.

We initially consider the simpler case in which condition 1 is an expression involving objects created in branch 0, and does not include variables. (Condition 1 'involves' objects if it refers to a file's current object or a robot hand's contacted object.)

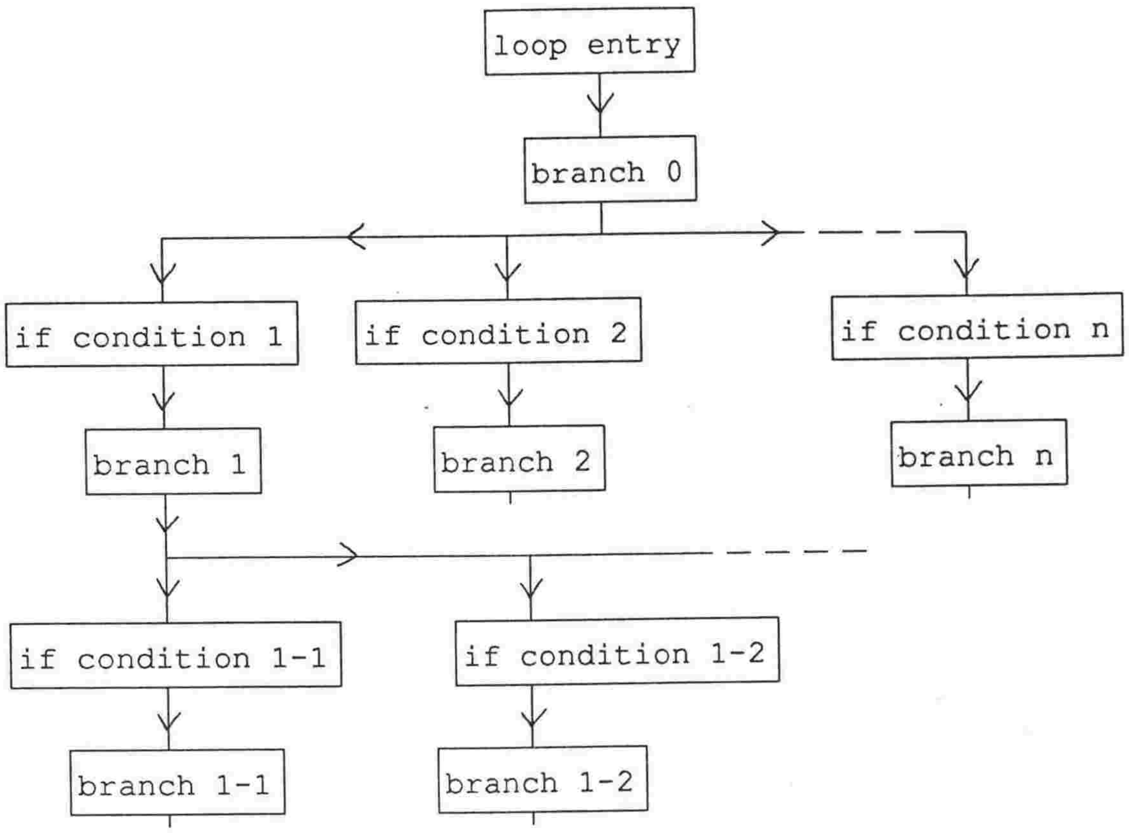


Figure 4-5 - Program Structure within a Loop

To be able to describe the actions taken by branch 1, which may include modifying properties of those objects from sequences S1,...,Sp that are current when branch 1 is executed, we need to be able to express the subsequences of S1,...,Sp which satisfy condition 1.

Symbolic execution of condition 1 in some iteration of the loop will result in the current objects being instantiated for expressions of the form file(<file-identifier>) or hand(<hand-identifier>). Condition 1 will then be in the form

P(current objects at condition 1)

for some P.

If we make the simplifying assumption that every sequence generated from objects created in branch i contains exactly one object from each execution of branch i, and every object created in every iteration has been included in a sequence, then the current objects on the ith iteration will consist of the ith object from each sequence, so condition 1 can be stated as

P(ith S1 item,..., ith Sp item).

The subsequence of Sj items satisfying condition 1, say S'j, will include those Sj items which satisfy P, i.e.

S'j = (item: item $\in$ Sj $\wedge$ P(ith S1 item,..., ith Sj-1 item, item, ith Sj+1 item,..., ith Sp item)).

But since the above assumption means that all sequences S1,...,Sp will be the same length, this correspondence can be expressed using the map function defined in Chapter 2 as

S'j = (item: item $\in$ Sj $\wedge$ P(map(item, Sj, S1),...,map(item, Sj Sj-1), item, map(item, Sj, Sj+1),..., map(item, Sj, Sp))).

If S1,...,Sp were of different lengths, we would need to introduce relationships more complex than 'map'. This has not been found necessary for any of the programs which have been analysed. Restricting the relationship between sequences to map is a current limitation of the PAN system. Note that, in any execution state, all subsequences S'1,...,S'p created by condition 1 are also the same length, since each contains those items from S1,...,Sp which make P(current objects at condition 1) true (this is proved more formally below).

Referring again to figure 4-5, objects may also have been created in branch 1. As before, these objects are put into groups with definitions varying only in numeric parameters, and from each group a sequence is created. Our simplifying assumption ensures that the length of these sequences will be the number of times branch 1 is executed, which is equal to the length of any of the S'j sequences.

Thus the following sequences contain objects which were current when condition 1-1 was executed:

- the subsequences of the branch 0 sequences, that contain items satisfying condition 1 i.e. S'1,...,S'p

- sequences containing items created in branch 1.

We now form subsequences of all these sequences in the same way as above, using condition 1-1 in place of condition 1.

This process can be repeated until all branches beginning with conditions have been processed.

We now consider the effect of merge statements on sequence generation. Consider the merge shown in figure 4-6, in which paths from condition-1,...,condition-n merge at single statement.

To describe the actions taken in branch-i we need to be able to express the subsequences which consist of objects current when the merge statement is executed. If sequences S1,...,Sp consist of objects which were current when the statement at a fork was executed, then the items in any Si which were also current when the merge was executed will be those which satisfy the conditions required for execution to reach the merge statement. By the definition of the statement condition, this will be the difference between the statement condition to reach the merge from that to reach the statement at a fork. But by theorem 3-3 this is

$$\text{condition-1} \vee \text{condition-2} \vee \ldots \vee \text{condition-n}.$$

The instantiation of any condition-i when the condition was executed will consist of items current when the fork statement was executed. Thus if we set

$$P' = \text{condition-1} \vee \text{condition-2} \vee \ldots \vee \text{condition-n}$$

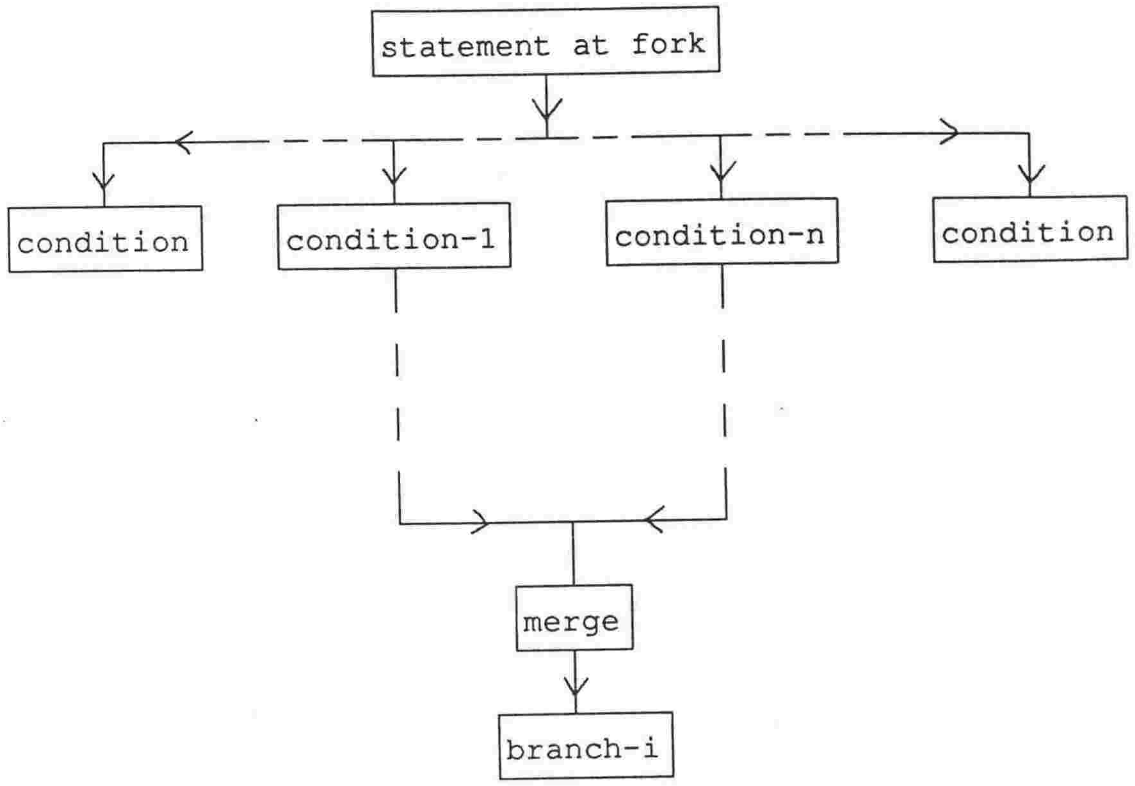


Figure 4-6 General Merge Format

then P′ fulfils the same requirements as P above, i.e. the subsequence of any Si current when the merge statement is executed can be expressed as

S′j = (item: item ∈ Sj ∧ P′(map(item, Sj, S1),...,map(item, Sj Sj-1), item, map(item, Sj, Sj+1),..., map(item, Sj, Sp))).

It is the restrictions on conditional branching introduced in section 2.1.3 that ensures that subsequences current when a merge statement is executed can be expressed in this form. Without these restrictions, the statement condition to reach the merge statement may contain objects created after the fork, which could not be used to form subsequences in the form of S′j above.

Sequences generated from items created in the branch beginning at the merge will, as above, have the same length as these subsequences.

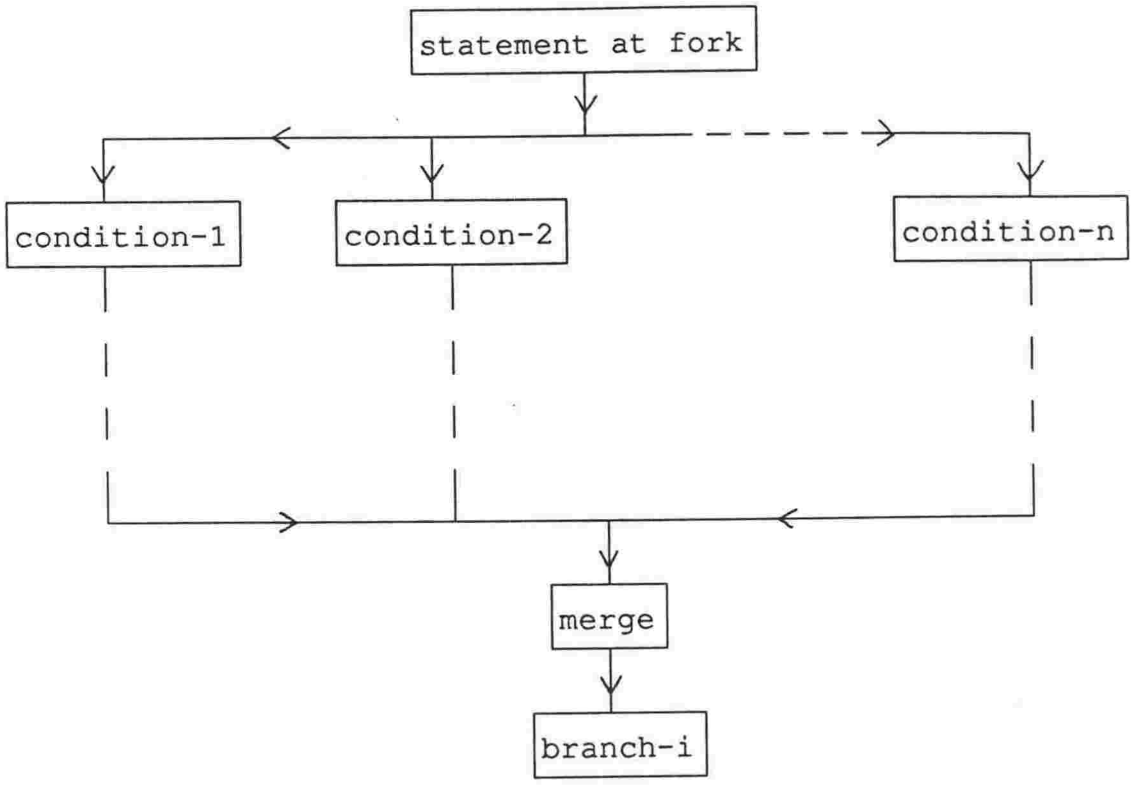


Figure 4-7 Last Merge Format

As a final observation on branches beginning with a merge, we consider branches which follow the last merge for a fork. In this case we have the situation as shown in figure 4-7.

In this case, if we consider a sequence which consists of items current at the fork, then by the reasoning above the subsequence of items current at the merge will consist of those items which satisfy the disjunct of the conditions. But in this case the disjunct of the conditions is simply T, so sequences created in the branch beginning with the merge will be the same as those created in the branch ending with the fork. Thus these branches can be treated as a single branch for the purpose of generating sequences. This applies to any branches which are reached from loop entry under the same conditions.

Conditions Containing Variables

We now consider the more complex case of forming subsequences from conditions containing variables.

Suppose now we have condition-1 following a fork, involving variables v1,...,vn, in addition to current objects. Condition-1 will now be in the form:

P(current objects at condition-1,v1,...,vn).

If sequences S1,...,Sp have been generated in processing the branch ending with the fork, then by our previous assumption, the current object in the ith iteration will be the ith member of S1,...,Sp. Thus condition-1 can be expressed as

P(ith S1 item,..., ith Sp item,v1,...,vn).

As a first stage in generating subsequences of S1,...,Sp which satisfy P we simply allow P to continue to refer to variables by their identifiers. So, as above, for each j from 1 to p, we would form:

S'j = (item: item $\in$ Sj $\wedge$ P(map(item, Sj, S1),..., map(item, Sj Sj-1), item, map(item, Sj, Sj+1),..., map(item, Sj, Sp), v1,...,vn)).

Note that this can only be done if the occurrence of variables in a condition can be explicitly recognized. Such a sequence definition cannot be used in the generalized execution state as it contains unknown values for v1,...,vn. Thus a second stage is invoked which tries to express the values of vi in terms of previously defined sequences and their current objects. This process is described in section 4.4 below.

Summary of Progress

The sequence generation process outlined above depends on determining:

1/ the items which are candidates for generating primitive sequences - those created in the same branch or a branch reached under the same conditions

2/ the conditions to use for generating subsequences. These conditions either originate from if statements (for specifying those objects that satisfy the conditions required to enter a branch beginning with an if statement) or, from the disjunction of such conditions (for specifying those objects that satisfy the conditions required to enter a branch beginning with a merge statement.) In both cases the occurrence of variables needs to be explicitly identified.

Both these requirements can be met by the statement condition, which has been expressly designed for this purpose.

Requirement 1 is met by the fact that all items created in branches reached under the same conditions will have been created in execution states having identical statement conditions if instantiation is ignored.

The first part of requirement 2 is met by the fact that if sequences have been created from items in some branch, branch-i whose statements have statement condition P, then any condition Q succeeding this branch will have caused a statement condition of $P \wedge Q$ to be recorded in the execution state created when the condition is executed.

The second part of requirement 2 is met by the fact that if sequences have been created from items in some branch, branch-i, whose statements have statement condition P, then if the branch is succeeded by conditions C1,...,Cn in the first statements of paths which later merge, then P ∧ (C1 ∨ ... ∨ Cn) will be recorded in the execution state created when the merge is processed.

To summarize, the statement conditions preserve the structure of the conditional branching within the loop, and it is this structure which determines the sequences and subsequences required for loop generalization.

Items Defined in Terms of Other Items

The sequence generation model creates sequences using the definitions of objects. The method of representing the definition of objects which are defined in terms of other items requires careful handling. Consider the program fragment in figure 4-8.

Two execution states, after four iterations could contain the following

Execution state 1:

```
    OBJECT-1 = sequential-object-in-source(1, SOURCE-1)
    OBJECT-2 = sequential-object-in-source(2, SOURCE-1)
    OBJECT-3 = sequential-object-in-source(3, SOURCE-1)
    OBJECT-4 = sequential-object-in-source(4, SOURCE-1)
    OBJECT-5 = keyed-object-in-source(weight(item) =
                                      weight(OBJECT-1), SOURCE-2)
    OBJECT-6 = keyed-object-in-source(weight(itemct) =
                                      weight(OBJECT-3), SOURCE-2)
    color(OBJECT-1) = red, ¬color(object-2) = red
    color(OBJECT-3) = red, ¬color(object-4) = red
```

Execution state 2:

```
    OBJECT-1 = sequential-object-in-source(1, SOURCE-1)
    OBJECT-2 = sequential-object-in-source(2, SOURCE-1)
    OBJECT-3 = sequential-object-in-source(3, SOURCE-1)
```

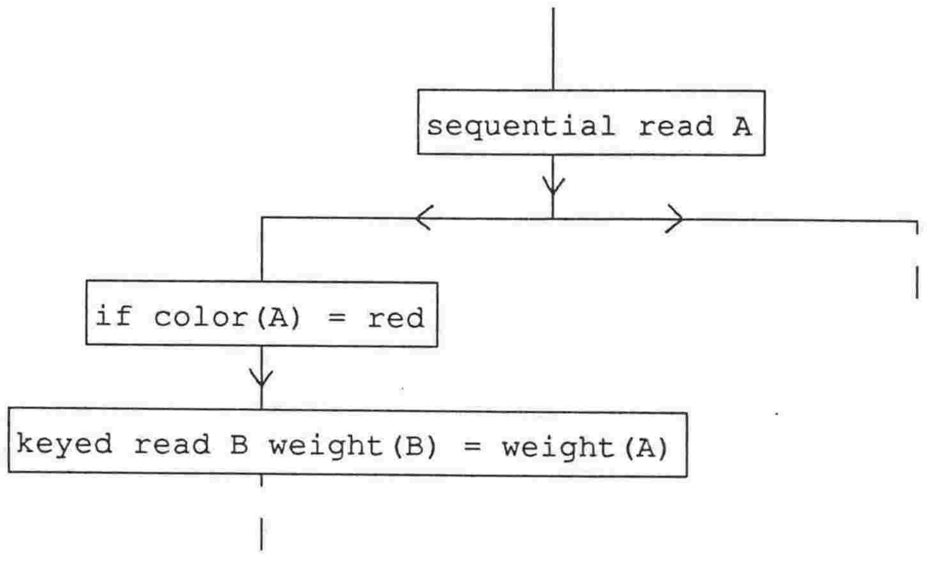


Figure 4-8 Definition of Objects for Sequence Generation

```
    OBJECT-4 = sequential-object-in-source(4, SOURCE-1)
    OBJECT-7 = keyed-object-in-source(weight(item) =
                                         weight(OBJECT-2), SOURCE-2)
    OBJECT-8 = keyed-object-in-source(weight(item) =
                                         weight(OBJECT-4), SOURCE-2)

    ¬color(OBJECT-1) = red, color(object-2) = red
    ¬color(OBJECT-3) = red, color(object-4) = red
```

In each execution state, the definitions of OBJECT-5, OBJECT-6, OBJECT-7 and OBJECT-8 have terms containing other objects. If we substitute the definitions of these objects into the definitions of OBJECT-5, OBJECT-6, OBJECT-7 and OBJECT-8, we obtain

Execution state 1:

```
    OBJECT-5 = keyed-object-in-source(weight(item) =
     weight(sequential-object-in-source(1, SOURCE-1), SOURCE-2)
    OBJECT-6 = keyed-object-in-source(weight(item) =
     weight(sequential-object-in-source(3, SOURCE-1), SOURCE-2)
```

Execution state 2:

```
    OBJECT-7 = keyed-object-in-source(weight(item) =
     weight(sequential-object-in-source(2, SOURCE-1), SOURCE-2)
    OBJECT-8 = keyed-object-in-source(weight(item) =
```

Given these definitions, there is no simple function f, which will allow these definitions to be expressed as

```
    OBJECT-5 = keyed-object-in-source(weight(item) =
     weight(sequential-object-in-source(f(1), SOURCE-1), SOURCE-2)
    OBJECT-6 = keyed-object-in-source(weight(item) =
     weight(sequential-object-in-source(f(2), SOURCE-1), SOURCE-2)
    OBJECT-7 = keyed-object-in-source(weight(item) =
     weight(sequential-object-in-source(f(1), SOURCE-1), SOURCE-2)
    OBJECT-8 = keyed-object-in-source(weight(item) =
     weight(sequential-object-in-source(f(2), SOURCE-1), SOURCE-2)
```

Thus no sequence can be generated from these definitions. However, OBJECT-1, OBJECT-2, OBJECT-3 and OBJECT-4 will already have generated the sequence

SEQUENCE-1 = (sequence i = 1 to k sequential-object-in-source(i, SOURCE-1)).

Then the condition on color(A) = red, will have been used to generate the subsequence

SEQUENCE-2 = (item: item $\in$ SEQUENCE-1 $\wedge$ color(item) = red).

In each execution state, we can therefore represent the objects used in definitions of OBJECT-5, OBJECT-6, OBJECT-7 and OBJECT-8 as

Execution state 1:

```
    OBJECT-1 = item-in-sequence(1, SOURCE-1)
    OBJECT-3 = item-in-sequence(2, SOURCE-1)
```

Execution state 2:

```
    OBJECT-2 = item-in-sequence(1, SOURCE-1)
    OBJECT-4 = item-in-sequence(2, SOURCE-1)
```

Substituting these facts into the definition of OBJECT-5, OBJECT-6, OBJECT-7 and OBJECT-8, we obtain

Execution state 1:

```
    OBJECT-5 = keyed-object-in-source(weight(item) =
                 weight(item-in-sequence(1, SEQUENCE-2), SOURCE-2))
    OBJECT-6 = keyed-object-in-source(weight(item) =
                 weight(item-in-sequence(2, SEQUENCE-2), SOURCE-2))
```

Execution state 2:

```
    OBJECT-7 = keyed-object-in-source(weight(item) =
                 weight(item-in-sequence(1, SEQUENCE-2), SOURCE-2))
    OBJECT-8 = keyed-object-in-source(weight(item) =
                 weight(item-in-sequence(2, SEQUENCE-2), SOURCE-2))
```

It is now straightforward to generate the sequence

```
    SEQUENCE-3 = (sequence i = 1 to size(SEQUENCE-2)
                   keyed-object-in-source(weight(item) =
                   weight(item-in-sequence(i, SEQUENCE-2), SOURCE-2)))
```

in both execution states.

For this reason, whenever an object, say OBJECT-1, has a definition which refers to another item (object, source or sequence) that is a member of a sequence, we represent that OBJECT-1 using item-in-sequence(i, SEQUENCE-n), where SEQUENCE-n is the smallest sequence that the item is in.

We have now introduced the sequences which PAN needs to generate and the role played by the statement condition in sequence generation.

The models for sequence generation described in the next section are designed to generate these sequences.

4.3.2 Sequence Generation Models

We are now ready to specify in detail the sequences generated by PAN. This specification is in the form of two models. Each model describes a sequence which will be produced if the specified conditions are satisfied by the execution states being generalized. After the models have been presented, an algorithm is presented, showing how execution state conditions can be tested against the models in a efficient manner.

Two models are used, one to generate primitive sequences and the other to generate subsequences.

Model 1 Primitive Sequences

Suppose we are given execution states $E_1, \ldots, E_g$, to be generalized. A primitive sequence will be generated whenever we can find a set of items I_j, $1 \leq j \leq g$, in each execution state, which satisfy the following:

- the items were created by statements with the same uninstantiated statement condition P for the loop being generalized.

- the definition of any item in any I_j is of the form $D(n_1, \ldots, n_m)$, where $n_1, \ldots, n_m$ are integers

- functions f1,...,fm, can be found, such that the ith member of I_j is

  D(f1(i),...,fm(i))

- the size of I_j equals either the iteration count in each E_j if P is T, or the size of a previously generated sequence associated with P.

If these conditions are satisfied the primitive sequence

```
    (sequence i = 1 to h D(f1(i),...,fm(i)))
```

will be generated, and *associated* with P, where either h = k (the iteration count) or h = size(S) for some previously generated sequence S.

For the next model, we note that an execution state E associated with loop entry describes the effect the program has had after execution has traversed some path H through zero or more iterations. During each of these iterations, statement conditions will have been recorded on the execution states created while traversing H.

The statement conditions recorded in these execution states provide a history of the path traversed to reach the execution state E. We say that an execution state has a statement condition R in *the history of the ith iteration* if R is recorded in an execution state that is created while executing path H in the ith iteration of the loop. If the iteration is not significant we simply say that an execution state has R *in the history.*

Given a set of statement conditions in a history containing R and P, we say that R is *minimally stronger* than P, if $R \rightarrow P$, R is not equal to P and there is no R′, R not equal to R′, such that $(R \rightarrow R') \wedge (R' \rightarrow P)$. Hard to do in logic?

Model 2 Subsequences

Suppose we are given execution states $E_1,\ldots,E_g$, to be generalized. If sequences S1,...,Sp, have previously been generated, and are associated with a statement condition, P, from the histories of $E_1,\ldots,E_g$, and these histories contain a minimally stronger condition $R = P \wedge Q$, then generate S1′,...,Sp′, *associated* with R, where Si′ is defined as

$$Si' = (item: item \in Si \wedge Q')$$

where Q′ is obtained from Q by replacing any instantiation [<source-identifier>/<object-identifier>] either by item if <object-identifier> is a member of Si, or by map(item, Si, Sj) if <object-identifier> is a member of Sj, j not equal to i.

As discussed in 'Developing a Sequence Generation Algorithm' above, it is the conditional branch restrictions of section 2.2.3 and their consequences (especially Theorem 3.3) that guarantee that each <object-identifier> is in one of S1,...,Sp, as long as sequence generation has been successful up to and including sequences associated with P.

4.3.3 Sequence Generation Algorithm

These models are used to generate sequences as follows:

1. Find any items (objects, sources or sequences) created by a statement in the loop being generalized with a statement condition of T. Retrieve the definitions of these items and replace any other item I in the definition by item-in-sequence(i, SEQUENCE-n), if I is the ith item in SEQUENCE-n and SEQUENCE-n is the smallest sequence I is in. Use sequence model 1 to generate sequences for these items. Let these sequences be S1,...,Sp.

2. Now take the history of statement conditions from every execution state, drop duplicates, and put the remainder into a partial order of strength of the condition, i.e. $P \le Q$ if and only if $Q \rightarrow P$.

   Let the weakest conditions be {C1,...,Cn}. Use each Ci, $1 \le i \le n$, to form subsequences of Sj, $1 \le j \le p$, using sequence generation model 2.

3. Repeat steps 1 and 2 for every condition, C, in order, so that in step 1, use items created by statements with statement condition C instead of T, and in step 2, using conditions minimally stronger than C instead of the weakest conditions.

4. Continue until all conditions have been processed.

4.3.4 Properties of Sequences

We now demonstrate two important properties of sequences created using these models.

Theorem 4-1: Let S1,...,Sp, be a set of sequences created using the sequence generation models and associated with the same statement condition P. S1,...,Sp, satisfy the following:

- they are all the same size in any execution state
- for every execution state, each member of any Si was created on a different iteration
- for every execution state the jth members of each Si was created on the same iteration, the jth to include P in the history.

Proof: Proof is by induction on the statement condition that S1,...,Sp are associated with. For sequences associated with the statement condition T, model 1 requires that these sequences are always equal in length to the iteration count. For this to be true for execution states which have completed 1, 2, 3,... iterations, these sequences must have one member created on each iteration. Since the statement condition T is in every iteration of every execution state (by definition of the statement condition) these sequences obey the proposition.

Now suppose that the proposition is true for sequences associated with statement condition P and R = P $\wedge$ Q is a minimally stronger statement condition in the history of some execution state. We want to show the proposition is true for all sequences associated with R. We first demonstrate this for sequences created using model 2 and then using model 1.

Let the sequences associated with P be S1,...,Sp, and subsequences S1′,...,Sp′ be the sequences generated from them using model 2. Thus Si′ = (item: item ∈ Si ∧ Q′(item)), where Q′ is as defined in model 2. Let the iterations in which R is in the history of some execution state be I1,...,Im. We first show that for any Si, 1 ≤ i ≤ p, the members of Si that are included in Si′, are those created during one of I1,...,Im.

Intuitively we are trying to show that if items in Si were created by statements with statement condition P, then the items in Si′ = (item: item ∈ Si ∧ Q(item)) consist of those items in Si created by statements with statement condition P ∧ Q.

First suppose that R is in the history of some iteration. Then by the definition of the history, R = P ∧ Q was recorded in that iteration. Now as shown in the proof of theorem 3-2, only one of two program structures give rise to a change in the statement conditions. Either Q is the condition in a statement succeeding one at a fork, or Q is the disjunct of several such conditions at the beginning of paths which meet at a merge. In the first case, the fact that P ∧ Q was recorded means that the statement containing the condition was executed and so the condition must have been satisfied by the current members of S1,...,Sp. In the second case the fact that the P ∧ Q was recorded means that the merge statement was executed and consequently one of the paths to that merge was executed. Thus the condition at the beginning of that path must have been satisfied by the current members of S1,...,Sp and since Q is the disjunct of such conditions, it also must have been satisfied. Thus in either case Q is satisfied by the items in S1,...,Sp for this iteration. But since Q is satisfied if and only if Q′ is satisfied, then these items are in S1′,...,Sp′.

Conversely, suppose that for some iteration, the items in S1,...,Sp are also in S1',...,Sp'. By the induction hypothesis P must be in the history for this iteration. For the items to be in S1',...,Sp', then they must satisfy Q. But since Q is satisfied, then if Q is in a condition succeeding a statement at a fork, that statement will be executed. This will cause the statement condition to be updated to $P \wedge Q$. If Q is the disjunct of conditions succeeding a fork, then one of these will be executed and when the merge statement is reached again, $P \wedge Q$ will be recorded.

We now know that, for each i, the members of Si' are those members of Si created on the ith iteration that had R in the iteration history. But since, by assumption, each member of Si is created on a different iteration, so must each member of Si' be. Also, since each Si is the same size, and the nth element of Si is in Si' if and only if R is in the history of the nth iteration, if and only if the nth element of Sj is in Sj', Si' and Sj' are the same size.

For sequences associated with R, created using model 1, their lengths, by the rules of model 1, must be equal to the length of any Si'. This shows that they are the same length. Also, for this to be true for execution states having completed 1, 2, 3,... iterations, the members of these model 1 sequences must have been created in the same iteration that Si' members were created. Since Si' members obey the proposition, so must sequences created using model 1.

The next theorem shows that there is a relation between the items in sequences and items used in instantiation if the program satisfies a constraint on instantiation. An item which is the current record from a file or the object currently contacted by a robot hand, will be used to instantiate any program statements which refer to that item by specifying the file or hand. We expect that a well behaved program will not try to refer to an item in this way unless it is certain

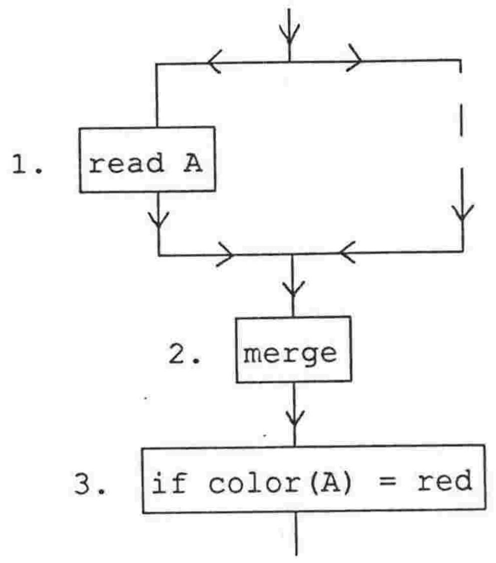


Figure 4-9 Possibly Invalid Instantiation

that such an item exists. For example, consider the program fragment shown in figure 4-9.

The reference to A in statement 3 is not valid unless A is read in all paths which reach statement 3. In the simple case in which a single statement is used to create an item, we can express this requirement as follows:

> if an item is created at statement S and used to instantiate statement S′ then all paths to S′ must pass through S.

We say a program uses *simple instantiation* if it obeys this condition. For statements in a loop this means that if the statement condition to reach S is C, and that to reach S′ is C′, then C′ → C. For programs which satisfy this constraint we can prove the following theorem.

Theorem 4-2: Suppose, in the symbolic execution of a program statement S, an item is used to instantiate file(<file-identifier>) or hand(<hand-identifier>). If the program uses simple instantiation, then if this item is included in any sequences, it will be in a sequence SP associated with the statement condition P required to reach S.

Furthermore, the ith time the statement S is executed, the instantiated item will be the ith member of SP.

Proof First suppose that the item was created by a statement with the same statement condition as S. In this case if the item is in any sequences it will be in a sequence SP generated using model 1, and S will be associated with P as required.

Now suppose that the item is created in some earlier statement with statement condition Q. Then, since the program uses simple instantiation, $P \rightarrow Q$. Since the item is in a sequence it must be in a sequence SQ generated using model 1 and associated with Q. Now since execution of the program reaches statement S in the iteration in which the item is referenced, P will be in the history for this iteration. When subsequences were created, the statement conditions will have been put into a partial order, with Q before P. Suppose the part of this ordering containing P and Q is, $Q, R_1, \ldots, R_n, P$ such that

$$P \rightarrow R_n \rightarrow \ldots \rightarrow R_1 \rightarrow Q,$$

and R_1 is minimally stronger than Q, R_{i+1} is minimally stronger than R_i, and P is minimally stronger than R_n. Then model 2 will have created subsequences

$$\begin{aligned}
S1 &= (\text{item: item} \in SQ \wedge R_1) \\
&\ldots \\
&\ldots \\
Sn &= (\text{item: item} \in Sn-1 \wedge R_n) \\
SP &= (\text{item: item} \in Sn \wedge P)
\end{aligned}$$

But because $P \rightarrow R_n \rightarrow \ldots \rightarrow R_1 \rightarrow Q$, items will be in SP if and only if they are in SQ and satisfy P. Since we know that the item is in SQ, and P is in the history for this iteration, then by theorem 4-1, the item will also be in SP.

To prove the extension we note that the ith time that the statement S is executed is the ith iteration that contains P in the history. But, by theorem 4-1, this is the iteration in which the ith member of SP was created as required.

As shown by these theorems, sequences associated with the same statement conditions have items which are current at the same time. For this reason these sequences are called *concurrent*.

4.3.5 Evaluation of the Sequence Generation Method

Having presented a method of generating sequences it is appropriate to ask under what conditions is this method successful. By successful, we mean that all items created during execution of some branch which is repeatedly executed, have been included in sequences.

We claim that the method will be successful if and only if the following proposition holds:

Proposition 4-1:

The definitions of items generated by statements with the same statement condition can be expressed in a way that varies only on the number of times these statements have been executed.

This is expressed more formally in the following theorem.

Theorem 4-3: If one execution of a set of statements with the same statement condition generates n items, then these items will be in sequences if and only if the definitions of the items generated on the jth execution of the statements can be expressed as

D1(f1(j)),...,Dn(fn(j))

for some functions f1,...,fn.

Proof: If the definitions of the n items can be expressed as

```
D1(f1(j)),...,Dn(fn(j))
```

then since these items will have the same statement condition at the time they were created, the sequence generation process will produce the sequences:

```
(sequence i = 1 to m D1(f1(i)))
   .  .  .  .  .  .  .
   .  .  .  .  .  .  .
(sequence i = 1 to m Dn(fn(i))).
```

Conversely, if the items are included in sequences, then they must have first been put into primitive sequences as above using model 1. Then the jth iteration produces objects defined as D1(j),...,Dn(j), as required.

Since the success of the method depends on the truth of the above proposition, we want to know whether this proposition is likely to hold for programs expressible in the language described in chapter 2.

For objects generated from sequential sources, the answer depends upon whether a single statement is generating objects from the same source. If it is, then the proposition above will be true, since objects will have definitions sequential-object-in-source(1, SOURCE-1), sequential-object-in-source(2, SOURCE-1) etc. If some other statement is also generating objects from the same source, then, the above proposition will not be satisfied. To see that this is so, first suppose that the other statement has the same statement condition, then the definition of items produced on the jth execution of these statements will be

```
sequential-object-in-source(2j-1, SOURCE-1)
sequential-object-in-source(2j, SOURCE-1)
```

which does not satisfy the theorem. Alternatively, if the other statement generating objects from the same source has a different statement condition then the objects created from either set of statements with the same statement conditions will not include all the objects from the source and again the proposition will not be satisfied.

For keyed sources the answer depends upon whether the predicate P used in the keyed read statement contains variables. If it doesn't, then on each execution of the keyed read statement, P can only vary in the objects current when the keyed read statement was executed. Now as shown by theorem 4-2, if P refers to other objects which have been put into a sequence, then the ith time the keyed read is executed, these objects will be the ith members of some sequence. Thus, if the objects created by the keyed read are object-1,...,object-n, then when the other objects referred to in the definition of object-i are replaced by the item-in-sequence(m, S) form as required by the sequence generation algorithm, m will be i. In this form the definitions of object-1,...,object-n will easily be made into a sequence since they will be the same apart from the ith definition having the integer i where the jth has integer j. An example of this procedure has been previously given at the end of section 4.3.1.

Thus P, and therefore the objects created by the keyed read that uses P, do obey the proposition if no variables occur in P. If P does contain a variable, then the proposition will only be obeyed if the variable's value on executing the keyed read statement only depends on the number of times the statement has been executed.

We can conclude that we can expect many 'normal' programs to obey the proposition and hence that the sequence generation as described will be successful for these programs. This does not mean that it is hard to create programs which do not obey the proposition. Either we can include sequential reads from the same source in different branches, or keyed reads using variables in the predicate whose values cannot be expressed as a function of the number of times the branch has been executed (e.g. a variable could count the number of times some other branch has been executed). However, since such programs are difficult for a human to analyse we need not be too disappointed that PAN cannot cope with them.

We can also evaluate the sequences generation method by comparing it with some obvious alternatives. As a first alternative we consider whether instead of grouping objects with the same statement condition, as done in model 1, we should simply group objects created by the same statement. Since all PAN sequences generated using model 1 do actually consist of items created from a single statement, this could be done without changing the sequences created.

However, we still need to apply model 2 to these sequences. As discussed in section 4.3.1, model 2 needs to apply a single predicate to all sequences associated with the same statement condition. Thus if we simplified model 1 by grouping items by their creation statement, we would need to add a new step to group these sequences by statement condition before being able to apply model 2.

As a second alternative to the sequence generation method, we consider whether we should apply model 1 to all items created by the same source. This appears attractive when one considers the program in figure 4-10, for which, as we have seen by the above discussion PAN is not able to generate sequences of objects from file (source) B. If, instead of the PAN method, we try to generate sequences from objects created from file B, regardless of which statement created them, then we can generate the sequence

```
SEQUENCE-1 = (sequence i = 1 to n
                   sequential-object-in-source(i, B)).
```

However, there are several problems with this apparently attractive approach. As a first problem, we again have a difficulty with applying model 2. There will be two subsequences of sequence 1, those items generated from the statement in the branch beginning with condition 1, and those items generated from the statement in the

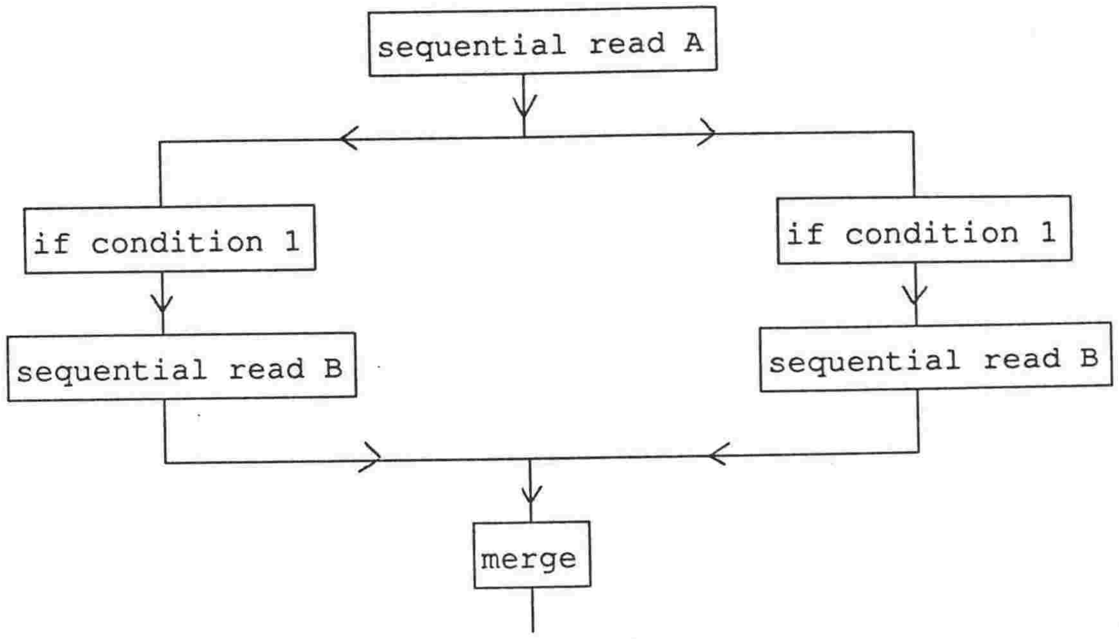


Figure 4-10 Same Source Used in Two Branches

---

branch beginning with condition 2. These subsequences will be generated by model 2 only if it is recognized that SEQUENCE-1 is in fact concurrent with

```
SEQUENCE-2 = (sequence i = 1 to n
                 sequential-object-in-source(i, A))
```

and that condition 1 and condition 2 are to be applied to both of these sequences.

Thus application of model 2 requires that SEQUENCE-1 and SEQUENCE-2 are to be grouped for generation of subsequences, which is a non trivial task. We return to this subject shortly. A second problem with this alternative is that PAN does not always generate sequences from a single source. When analysing a robot program which finds the first object in line 1, the first object in line 2, then the first object in line 3, PAN will generate the sequence

```
(sequence i = 1 to n (first object in line i))
```

which could not be done if sequence generation is done by grouping items from the same source. A related problem occurs with two keyed reads in different branches using the same source. In this case, we saw earlier in this section that PAN's method of grouping the objects from each read separately will generate sequences (unless, possibly variables are used in the key). This alternative method will not work because the definitions of the objects from both sources will not have a simple relationship between them. A final problem with this method is that it cannot be easily extended to programs of the form shown in figure 4-11. Trying to find a sequence from the objects created from file B now runs into problems with finding the length of the sequence. We now return to the fact that PAN cannot find sequences for the program in figure 4-10. Rather than adopting this alternate method, with all its disadvantages, a better solution would be to extend PAN's model 1 so that it first tries to find sequences for items created with the same statement condition, as currently, and, if this fails, to then try to find sequences of items which were created with statement conditions S1,...,Sn which satisfy the following:

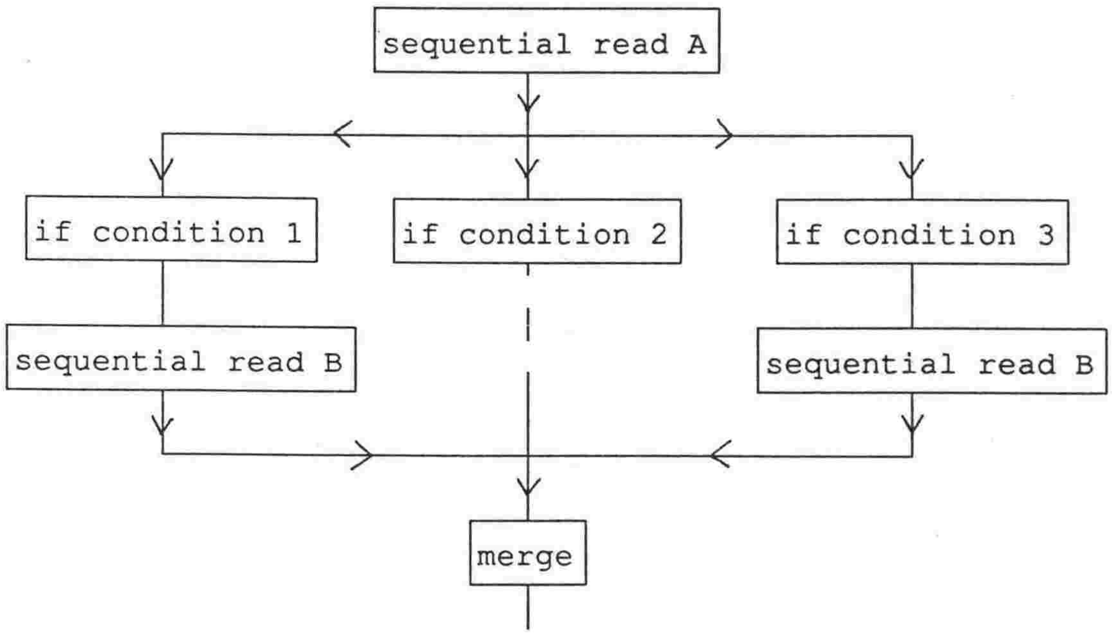


Figure 4-11 - Same Source Used in Two Out of Three Branches

S1,...,Sn are of the form $P \wedge Q1, ..., P \wedge Qn$, where

$Q1 \vee ... \vee Qn = T$.

If any such sequences can be found they should be the same length as sequences associated with P and used with them to form subsequences. PAN is not currently able to do this but should be extended to do so.

4.4 RESOLUTION OF THE VALUE OF VARIABLES USED IN SEQUENCE DEFINITIONS

4.4.1 Introduction

We now address the problem that some of the subsequences generated by the sequence generation models will be defined in terms of a variable of unknown value. Since these sequences have not yet been used, apart from possibly appearing in the definition of other sequences, we are free to discard them if necessary. Resolution means that either the subsequence is successfully redefined without variables, or it is not used further in the loop generalization process.

Before presenting models and an algorithm for resolving these values, we first discuss the program situations in which variables occur in sequences.

Without the use of variables a program can reference any current object from a source. Generally, variables increase the expressiveness of program by allowing the program to either reference non current objects, or reference the size of some set of objects.

There are two trivial cases. Firstly, we may find that the variable in question is in fact not being updated in the loop being generalized i.e. has the same value in every execution state input to the generalization process. In this case we simply replace the variable by its value in all sequence definitions. Secondly, the variable may always reference only current objects. This is illustrated by the two programs in figure 4-12. The use of a variable in this case does not increase the expressiveness of the program. If

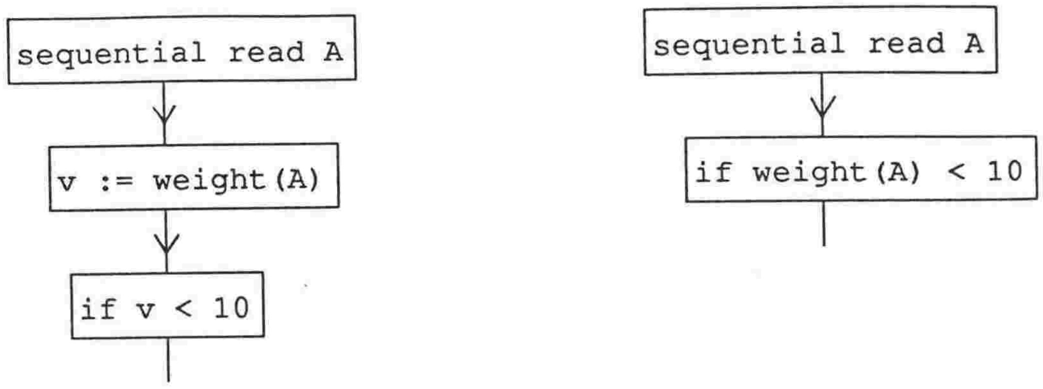


Figure 4-12 - Unnecessary use of a Variable

---

the left hand program fragment in figure 4-12 is part of a loop, then we can expect the following sequences to be generated:

```
SEQUENCE-1 = (sequence i = 1 to
                 sequential-object-in-source(i, SOURCE-1)
SEQUENCE-2 = (item: item ∈ SEQUENCE-1 ∧ v < 10)
```

Let the objects in SEQUENCE-1 be OBJECT-1, OBJECT-2, OBJECT-3. Then the value of v as used in the condition and recorded in the statement conditions will have values weight(OBJECT-1), weight(OBJECT-2), weight(OBJECT-3). The significant characteristics of this use of a variable as part of a condition generating a subsequence of some sequence S are that:

- if a statement condition containing v is in the history of iteration i, its instantiation will involve items created in iteration i

and

- all the items involved in the instantiation of v are members of S. (This can be generalized so that the items are in a sequences concurrent with S).

Identification of this case allows it to be handled by a separate model.

We now turn our attention to more interesting cases. A subsequence S2, whose definition contains variables must have been created by applying some condition P involving variable v to a sequence S1 i.e.

$$S2 = (item: item \in S1 \wedge P(item, v)).$$

We want to find an expression Q(item), so that

$$item \in S1 \rightarrow Q(item) \equiv P(item, v)$$

allowing S2 to be expressed as

$$S2 = (item: item \in S1 \wedge Q(item)).$$

Two cases which occur frequently are:

- v is counting the size of the sequence S1
- v is a function on some property of all objects in S1.

For the first case, consider the program fragment in figure 4-13. If v is initialized to zero, and this fragment occurs in a loop, the sequence generation method will produce

---

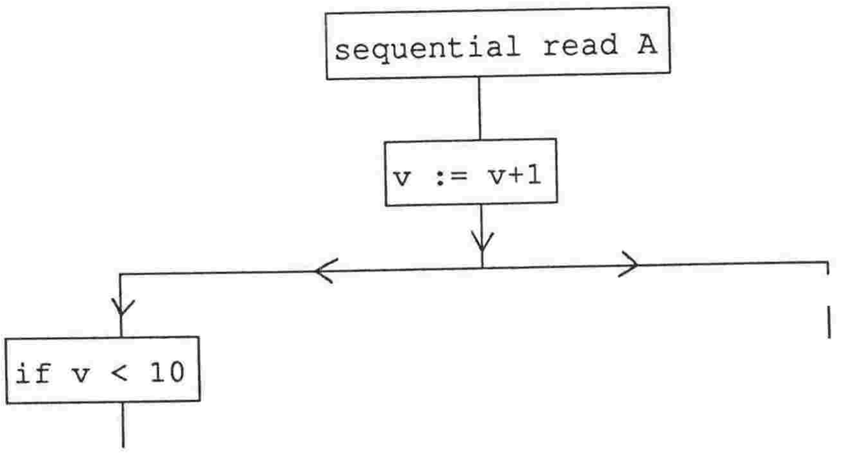


Figure 4-13 - Variable Counting Size of Sequence

S1 = (sequence i = 1 to n sequential-object-in-source(i, SOURCE-1))

S2 = (item: item $\in$ S1 $\wedge$ v < 10}

For any item in S1, at the time the condition on v is tested, v will be the size of S1, which is the position of the item in S1. Thus we can express S2 as

S2 = (item: item $\in$ S1 $\wedge$ position-in-sequence(item, S1) < 10)

For the second case consider the program fragment in figure 4-14. Again, we assume v is initialized to zero, and this fragment occurs in a loop. The same sequences

S1 = (sequence i = 1 to n sequential-object-in-source(i, SOURCE-1))

S2 = (item: item $\in$ S1 $\wedge$ v < 10}

are generated. But in this case, for any item in S1, at the time the condition on v is tested, v will be the sum of the weight of all items in S1 at that point, from the first up to the position of that item in S1 i.e. we will have

$$v = \sum_{i=1}^{size(S1)} weight(item\text{-}in\text{-}sequence(i, S1))$$

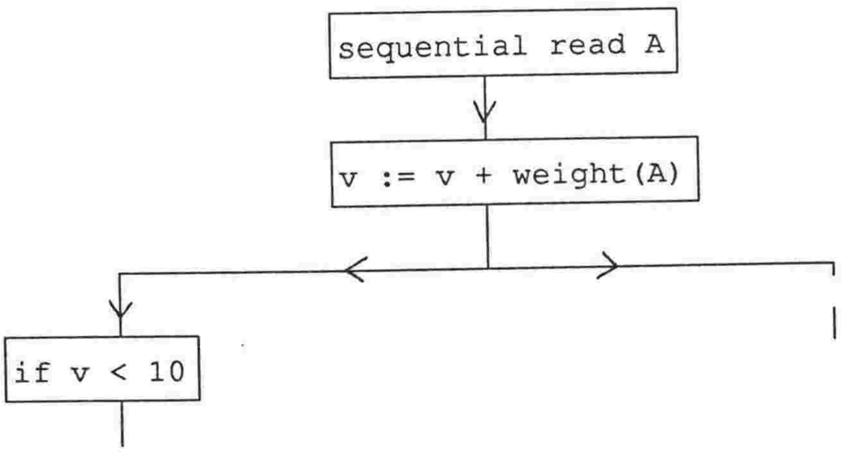


Figure 4-14 - Variable Summing Weight of Sequence

and S2 can be expressed as

$$S2 = (\text{item: item} \in S1 \wedge \sum_{i=1}^{\text{position-in-sequence(item, S1)}} \text{weight(item-in-sequence(i, S1))} < 10)$$

Both of these examples can be extended to allow a superset of S1 to be substituted for S1.

4.4.2 Resolution of Variables in Sequence Definition Model

We now present models which formalise the cases in which PAN is able to resolve a variable occurring in the definition of a sequence. A variable v occurring in the definition of the sequence S2

$$S2 = (\text{item: item} \in S1 \wedge P(\text{item}, v))$$

will be called *resolvable* if it obeys one of the following models.

Model 1

v has the same value in all input execution states, in which case it is replaced by its value.

Model 2

The instantiation of v in every occurrence of P(item, v) in the history of every execution state is an expression involving only items which were the last in S1 or concurrent sequences at the time that P(item, v) was recorded.

If this is the case, suppose the instantiation of v in some occurrence of P(item, v) in the history of an execution state, refers to items: item-1,...,item-n. Form Q from P(item, v) by

- replacing each item-i by map(item, S′, S1) if itemi $\in$ S′, concurrent with S1 and S′ not equal to S1

- replacing item-i by item if item $\in$ S1.

S2 is then redefined as

S2 = (item: item $\in$ S1 $\wedge$ Q(item)).

Model 3

There is some sequence S, that is a superset of S2, and the instantiation of v in every occurrence of P(item, v) in the history of every execution state is the size of S at the time that P(item, v) was recorded. In this case replace v by position-in-sequence(item, S).

Model 4

There is some sequence S, that is a superset of S2, and the instantiation of v in every occurrence of P(item, v) in the history of every execution state is the sum or product of some property of all items in S at the time that P(item, v) was recorded i.e.

$$v = \sum_{i=1}^{\text{size(S)}} p(\text{item-in-sequence}(i, S))$$

or

$$v = \prod_{i=1}^{\text{size(S)}} p(\text{item-in-sequence}(i, S))$$

for some property p.

In this case we replace v by

$$\sum_{i=1}^{\text{position-in-sequence(item, S)}} p(\text{item-in-sequence}(i, S))$$

or

$$\prod_{i=1}^{\text{position-in-sequence(item, S)}} p(\text{item-in-sequence}(i, S))$$

4.4.3 Resolution of Variable in Sequence Process

In using the above models, sequences must be processed in the correct order to ensure we do not resolve a variable by replacing it by an expression involving a sequence which may later be removed (because this sequence contains a different unresolvable variable).

Thus PAN ensures that it processes any sequence Si before processing any sequence Sj that is defined in terms of Si. A sequence S is processed by checking whether its definition involves variables. If it does, then an attempt is made to resolve each variable using the above models. If this is unsuccessful, the sequence, and all sequences defined in terms of S are removed.

4.5 VARIABLE VALUES AT LOOP ENTRY

4.5.1 Introduction

We have now dealt with the special case of generalizing the value of variables used in sequence definitions. These values are not necessarily the same as the value of the variable at loop entry.

The generalized execution state, which is the output of the loop generalization process, reflects the situation at loop entry after an indefinite number of iterations. Thus the value of variables at loop entry needs to be generalized from the input execution states, using the values in the variables data, regardless of whether the variables occur in sequence definitions.

For variables in sequence definitions, we specified models for the resolvable values of the variable. If a variable could not be resolved the sequence was simply removed from the loop generalization process. Failure to find a value of the variable at loop entry, however, is more serious and results in a generalized execution state with an unknown value for the variable. Thus we use a two stage approach. We first look for a value for the variable using a model that is an extension of the one for variables in sequences. If this fails then the 'brute force search', described separately in chapter 6, is invoked. This two stage approach has the advantage of being able to quickly analyse those variables with simple values and only uses the more powerful but time consuming method if it is really required.

The following analysis refers specifically to variables, but applies equally well to any value generated by all iterations of the loop. For example, it applies to an updated property of a single object present in all input execution states, as in the program in figure 4-15. In this case, file A is a report showing the total weight of all records in file B. Program analysis of the value of total(A) at loop entry is the same as if it were a variable. An object used in this way will have no definition when the loop is generalized, because it has not been read or written and, as discussed in chapter 3, object definitions are always assigned by read or write statements. Thus PAN tries to apply the following variable analysis not only to variables, but to any updated property of any object without a definition.

To begin the discussion of what we would expect as a model of variable values, we first note that, as in the sequence definition variable analysis, we may have the simple cases - either the variable is not being altered by the loop, or is always an expression involving only objects which were current when the variable was updated. In the first case no further analysis is required. In the second case the value of v in any execution state will involve those objects current when v was last updated. If v is updated in branch i, then by theorems 4-1 and 4-2, if branch i is executed n times in total, these objects will be the nth members of concurrent sequences of length n. Thus they can be expressed as item-in-sequence(size(S), S) for some sequence S.

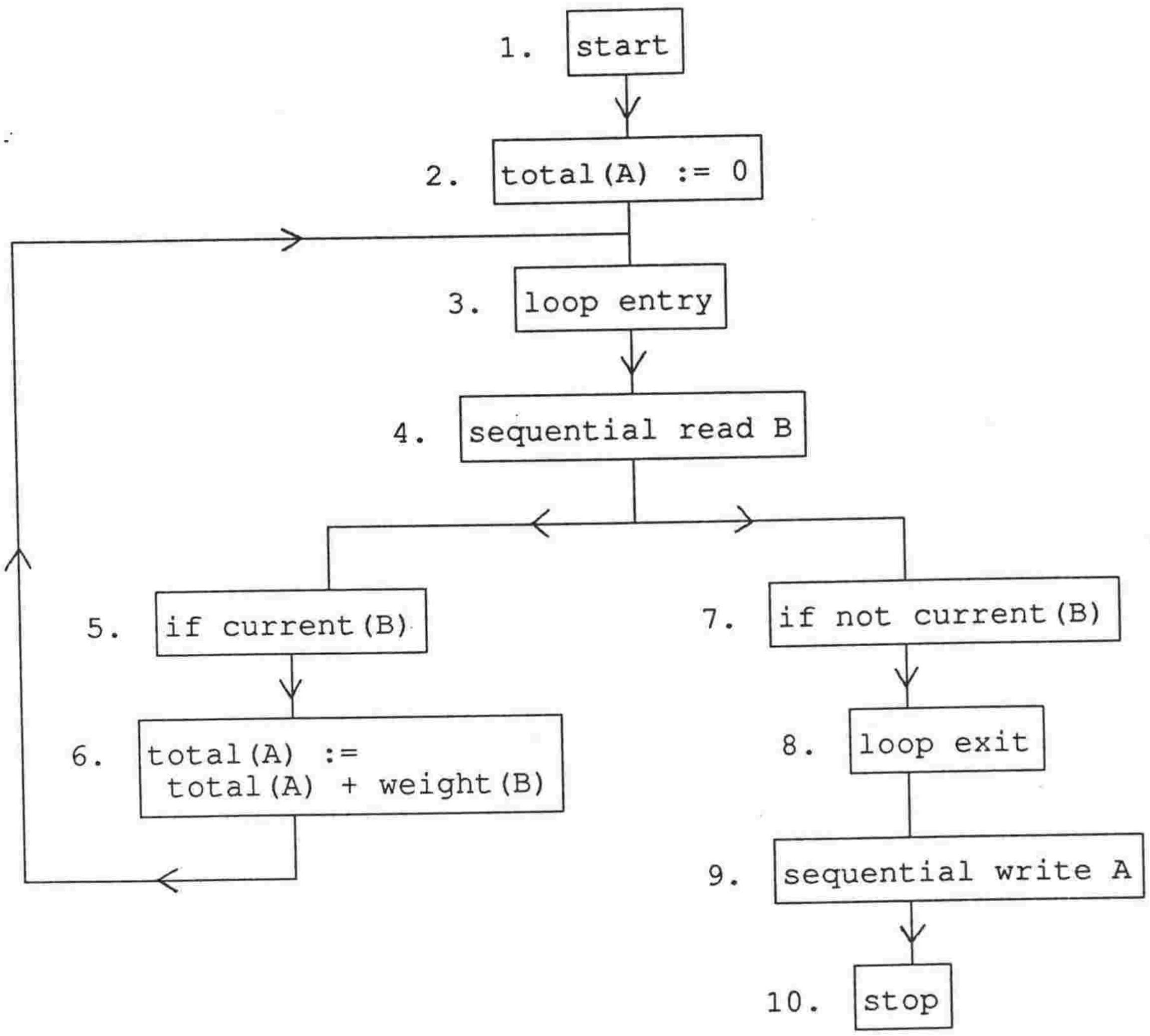


Figure 4-15 - Report Total.

For non trivial cases, we extend the model for variables occurring in sequence definitions. We still need to find a value for the variable expressed in terms available in the generalized execution state, but we are no longer constrained to find a value which can be substituted into the definition of a subsequence.

The case of a variable counting the size of a sequence is extended to a variable counting the sum or difference of the sizes of several sequences. The case of the variable being a sum or product of some property of all objects in a sequence is extended to the variable being a function of the sum or product of one or more properties of all objects from several sequences.

For an example of the second case, a variable v may have the value

```
    weight(OBJECT-1) * weight(OBJECT-2)
     - weight(OBJECT-3) * weight(OBJECT-4)
     + weight(OBJECT-5) * weight(OBJECT-6)
     - weight(OBJECT-7) * weight(OBJECT-8)
     + weight(OBJECT-9) * weight(OBJECT-10)
```

where

```
    S1 = {OBJECT-1 OBJECT-5 OBJECT-9}
    S2 = {OBJECT-2 OBJECT-6 OBJECT-10}
    S3 = {OBJECT-3 OBJECT-7}
    S4 = {OBJECT-4 OBJECT-8}.
```

with {S1 S2} and {S3 S4} concurrent pairs of sequences. v can then be expressed as:

```
  (weight(item-in-sequence(1, S1)) * weight(item-in-sequence(1, S2))
 +   "          "          2  "       "          "          2  "
 +   "          "          3  "       "          "          3  ")))

    -

  (weight(item-in-sequence(1, S3)) * weight(item-in-sequence(1, S4))
     "          "          2  "       "          "          2  ")))
```

or

$$\sum_{i-1}^{size(S1)} \quad weight(item\text{-}in\text{-}sequence(i,\ S1)) * weight(item\text{-}in\text{-}sequence(i,\ S2))$$

-

$$\sum_{i=1}^{size(S3)} weight(item\text{-}in\text{-}sequence(i,\ S3))$$

$$*\ weight(item\text{-}in\text{-}sequence(i,\ S4)).$$

This type of rearrangement is possible because the original value for v can be expressed in the form:

v = (term1 + term2 + term3) - (term4 + term5)

where

- term1, term2 and term3 only contain items from concurrent sequences {S1 S2}
- term1, term2 and term3 are the same except term1 references the first item from a sequence, where term2 references the second and term3 the third
- size({term1 term2 term3}) = size(S1) = size(S2)
- similar conditions are true for term4 and term5.

We are now ready to present in detail the models and algorithm for the values of variables at loop entry.

4.5.2 Models for Variable Values at Loop Entry

The value of a variable v at loop entry will be called *resolvable* if it satisfies one of the following models.

Model 1

v has the same value in all input execution states in which case this value is used in the generalized execution state.

Model 2

There is a set of concurrent sequences S1,...,Sp such that the value of v recorded in the variables data in all input execution states is the same expression involving the last items of these sequences. In this case, the value used in the generalized execution state is the expression formed by replacing each item which is the last member of sequence S by item-in-sequence(size(S), S)

Model 3

There is a set of sequences S1,...,Sp such that the value of v is ± size(S1) ± size(S2) ± ... ± size(Sp) in all input execution states. In this case the same expression is used as the value of v in the generalized execution state.

Model 4

In all input execution states, the value of v is some expression $P(e_1,...,e_n)$, where each e_i is a subexpression obeying the following:

- each e_i is of the form $\sum_{i=1}^{size(S)}$ term(i) or $\prod_{i=1}^{size(S)}$ term(i) for some sequence S, where each term(i) is identical except term(i) may refer to the ith member of some sequence S′ concurrent with S, where term(j) refers to the jth member of S′

In this case the value of v used in the generalized execution state is the expression $P'(e_1,...,e_n)$, where P′ is formed from P by replacing each item that occurs in term(i) and is a member of S′, concurrent with S, by item-in-sequence(i, S′).

4.5.3 Algorithm for Finding Variable Values at Loop Entry

The algorithm proceeds by trying to match the value of all variables at loop entry to each model. Thus for each variable, proceed as follows.

1. Try to match against the first model by simply checking the value of the variable in each execution state.

2. Try to match against the second model by checking whether the value of v, apart from reference to items in sequences, is the same in all execution states. If it is, then choose any execution state and find a set of concurrent sequences whose last members are the items in the value of v in that execution state. If such a set can be found, see if they have the same property for every execution state. If so, the value of v is formed by replacing each item by an expression of the form item-in-sequence(size(S), S). Otherwise, return to the first execution state and find another set of concurrent sequences to try. If no more sets can be found, then the variable is not resolvable using this model.

3. Choose any execution state and try to find a match of the value of v to the size of a sequence in this execution state. If this can be found, see if v is the size of the same sequence in all other execution states. If not, try other single sequences. If this is also not successful, try the sum of two sequences, and then the difference of two sequences. Continue in this way until all combinations of æ size(S1) æ ... æ size(Sp) have been tried. If still not successful, then value of v is not resolvable using this model.

4. Testing whether a value of v satisfies the fourth model may require some manipulation of v's value. Much of this will be already done by the Expression Simplification process described in Chapter 7, since this will group + and * terms. Thus the following algorithm will suffice:

   - pick any execution state and examine the value of v. If v contains any subexpressions of the form

     term1 + term2 + ... + termn,

     then use associative and reflexive properties of addition to reexpress these as

$(\text{term1-1} + \ldots + \text{term1-n}_1) + \ldots + (\text{termh-1} + \ldots + \text{termh-n}_h)$

so that each form $(\text{termi-1} + \ldots + \text{termi-n}_i)$ can be associated with a set of concurrent sequences $\text{Si1},\ldots,\text{Sip}_i$ so that

- termi-1,...,termi-n_i are identical except that termi-j contains the jth item from one of the concurrent sequences where termi-k contains the kth item from the same sequence.

- n_i = size of any of the current sequences

if this can be done, then replace each item from any sequence S of the concurrent sequences, occurring in termi-j, by item-in-sequence(j, S), which allows the subexpression to be written as

$$\sum_{i-1}^{\text{size(S11)}} \text{term1(i)} + \ldots + \sum_{i-1}^{\text{size(Sh1)}} \text{termh(i)}.$$

Do similar processing for subexpressions involving *, using $\prod$ format. At this point we will have a set of alternate values of v, say $v_1,\ldots,v_p$. If any of these is derived in all execution states, and contains no items which are in sequences, then it satisfies the model and is used as the value of the variable in the generalized execution state.

4.6 UPDATED OBJECT PROPERTIES

4.6.1 Introduction

We now address the last major task of the generalization process: analysing the updated properties of any objects created in the loop being generalized. As for finding the values of variables at loop entry, we first try to explain these properties in terms of a model. Only if this fails is the brute force search of chapter 6 invoked.

We first consider the likely values for updated properties. Suppose we are considering the updated properties of some sequence S, updated in branch j of the program. On any given loop iteration, the assignment statement performing the property update could have referenced either the value of variables or the current items available in branch j. By theorem 4-2, the ith time this update occurs it will be updating the ith member of S and the only other current objects will also be the ith members of sequences concurrent with S. So, apart from the value of variables, the updated property of the ith item in S can only reference the ith items from sequences S1,...,Sp, concurrent with S.

Variables can greatly increase the complexity of updated properties. However, a variable being updated during the loop is generally computing some value which only has meaning when all iterations are included. Thus the interim value of such a variable i.e. the value on the ith execution of branch j, is unlikely to be used in updating the property of the ith member of S.

One exception to this, however, is when the updated property of the ith item in S depends upon i. This would occur in a dp program which numbers each record (e.g. in a report line) or in a robot program which moves the first object in S to (1 , 1), the second to (2 , 1), the third to (3 , 1) etc.

Thus the updated property model described below only allows for variables used to update properties in this simple way.

To find an algorithm to use such a model, we must bear in mind that we may find that only some objects in a sequence have updated properties. For example, consider the program fragment of figure 4-16. Sequence generation will produce:

$S1 = (\text{sequence } i = 1 \text{ to } k \text{ sequential-item-in-source}(i, \text{SOURCE-1}))$

$S2 = (\text{item: item} \in S1 \wedge \text{color(item)} = \text{blue})$

$S3 = (\text{item: item} \in S1 \wedge \neg\text{color(item)} = \text{blue})$

$S4 = (\text{item: item} \in S2 \wedge \text{length(item)} > 10)$

$S5 = (\text{item: item} \in S2 \wedge \text{length(item)} \leq 10)$

In this case only some of the items in S1 will have an updated weight property i.e. those also in S2. Obviously we cannot expect to obtain a general statement about the updated property of all items in a sequence if only some of them have that property updated. Thus we should only try to fit the model to those sequences in which all items have the property updated.

We also want to ensure that we use the most general possible sequence to state the property updates. For example, referring again to the above sequences, we would not want to state that items in S4 and S5 have updated weight property with value of 1. While this is true, it is more economical to state this of sequence S2 instead. We can ensure that our analysis fits these requirements by analysing sequences, subsequences, subsubsequences etc, until an explanation of the updated property is found.

4.6.2 Updated Property Model

We say that the value of updated property p is *resolvable* for sequence S of length n, if the values of p, for all members of S, $p_1, \ldots, p_n$, vary only in two ways

- there is some sequence S′ concurrent with S such that p_i refers to the ith member of S′

- p_i contains the numbers $n_{i1}, \ldots, n_{im}$

and functions $f_i, \ldots, f_p$ can be found, so that

$f_i(j) = n_{ji}$, for $1 \leq j \leq n$, $1 \leq i \leq m$.

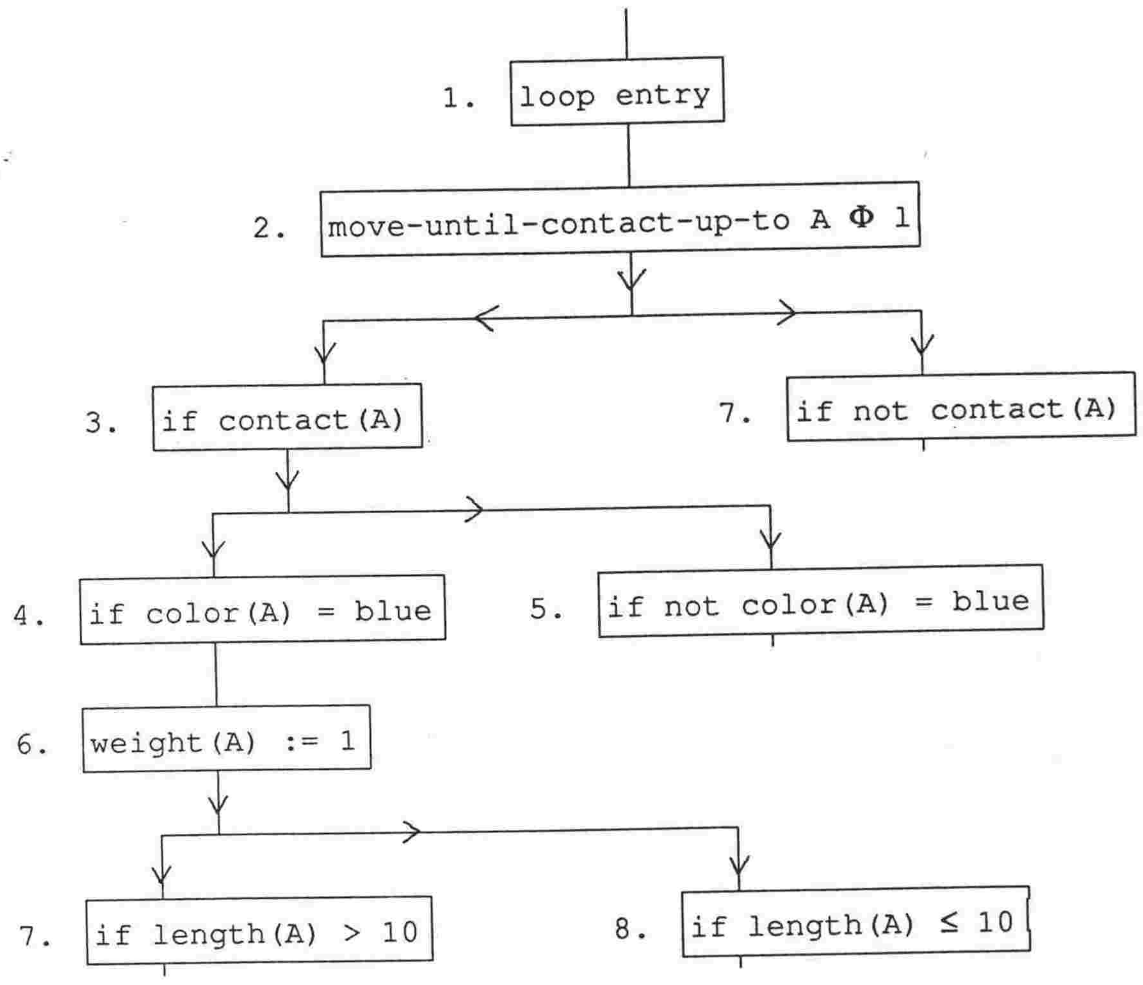


Figure 4-16 - Partial Property Update Example

In this case the sequence has the updated property described by

$$p(\text{item-in-sequence}(j, S)) = p', \ 1 \le j \le n$$

where p' is formed from p_i by

- replacing n_{ij} by $f_j(i)$ for $1 \le j \le n$
- replacing the ith element of any sequence Sm concurrent with S by item-in-sequence(i, Sm).

4.6.3 Updated Property Algorithm

1. Place sequences into a partial order, so that S1 < S2 if S2 is a subsequence of S1.

2. Process sequences so that if S1 < S2, S1 is processed before S2. If a sequence S contains any element with an updated property, see if every element of S has this property updated. If so, then try to fit the updated property to the above model. If successful do not analyse the property for any subsequence of S. If not every element of S has property updated or the attempt to fit the model is unsuccessful, then continue processing if S has any subsequences. Otherwise analysis has failed for this property and will be referred to the brute force search described in Chapter 6.

4.7 GENERALIZATION OUTPUT

The analysis above of variable values and updated object properties may need to invoke the brute force search. If so it will be done before the processing described in the remainder of this chapter. However, rather than divert from the discussion of the usual analysis flow, this has been left for description in Chapter 6. For now we proceed onwards with the production of a generalized execution state.

Some minor housekeeping is discussed first - we may be able to obtain a generalized value of the position of robot hands in the robot domain, and a value for the number of retrievals attempted from each sequential source.

4.7.1 Position of Robot Hands, Object Contacted, Grasping.

The position of each robot hand is expressed as coordinates in 2D space i.e. in the form (x , y). For a given robot hand, only one such position will exist for each hand. Thus there is nothing to prevent us from treating x and y as variables, and using the full variable analysis process described above, including, if necessary, the brute force search of Chapter 6.

However, this was not realized when PAN was implemented and PAN instead tries to explain each robot hand position as a function of the iteration count. That is, for a given hand h, we try to find functions f1 and f2 such that

$$x_h = f1(k) \text{ and } y_h = f2(k)$$

where k is the iteration count. If such functions f1 and f2 cannot be found, then these values are set to 'unknown'. This usually doesn't present problems in practice, as programs which have complicated expressions for robot hands at loop generalization usually return the hands to some fixed point at loop exit.

For generalizing the object contacted by a robot hand, we have much less scope. Our generalized value must be expressed only in terms available in the generalized frame. Also, the object contacted by a robot hand on the last loop iteration must have been created in the last execution of some branch, branch i. Thus the object will be the last object of one of those sequences generated by objects created in branch i.

Our generalization is then very simple:

- if, in every execution state, hand h is not in contact, then it will not be in contact in the generalized execution state.

- otherwise, if there is a sequence S, such that the object contacted by hand h is always the last object in this sequence, then in the generalized execution state the object contacted is set to

  item-in-sequence((size S), S)

- otherwise, object contacted is set to 'unknown'.

Finally, to derive the generalized value of grasping, we simply set this to T or F, if it has that value in every execution state, otherwise to 'unknown'.

4.7.2 Number of Retrievals Attempted, Current Object and Exhausted Indicator of Sources.

As for the robot position described above, the number of retrievals attempted is really an implicit variable, and should have been analysed in the same way as other variables. However, when PAN was constructed it was recognized that the number of retrievals attempted will equal the number of times the program code retrieving objects is executed. If this is contained in some branch, branch i, this will be the same as the number of times this branch is executed, which, in turn is the size of any sequences generated from objects created in this branch.

Thus, we expect that this implicit variable will end loop execution with a value equal to the size of some sequence. We look for such a sequence S, and if one can be found, set the value of the implicit variable to size(S). If no such S can be found, the number of retrievals attempted is set to 'unknown'. Current object is analysed as for robot hand in contact, as described above. Exhausted indicator is analysed as for grasping as described above.

4.7.3 Production of Generalized Execution State

The final task of loop generalization is to actually produce a single generalized execution state. Since producing this execution state has been the aim of the whole chapter, we now only need to describe how the results of the previous sections are used for this purpose.

We begin with the single execution state E that is associated with loop entry and has an iteration count of zero. We then retrieve additional information from the other execution states associated with loop entry and the results of the loop generalization process. We add to E all objects and sources identified during loop execution which have not been put into sequences. We then add all the sequences generated by the method described in sections 4.3.2 and 4.4.2. The updated properties of these sequences determined in section 4.6.2 are also recorded. Any variables identified during loop generalization are recorded and any variable values resolved in section 4.5.2 are also recorded. The robot hand and sequential source attributes described in section 4.7.1 are recorded. Finally housekeeping data is updated to ensure that symbolic execution can continue from this generalized execution state. This involves setting the iteration count to the indeterminate value k, setting the status of the execution state to active and the loop status to exiting.

Chapter 5
Loop Exit

Once a loop has been generalized there are two outstanding loop processing tasks. One is to find a value for the loop iteration count on exit and the other is to verify that the generalization is correct. These processes correspond to the two paths that execution can take starting with the generalized execution state. Either a loop exit statement is reached, in which case a value for the iteration count can be determined, or loop entry is reached, in which case the generalization can be verified. We first describe the processing when loop exit is reached, referred to as exit processing and then describe verification. These descriptions show that verification and loop exit processing require similar components.

5.1 EXIT PROCESSING

One characteristic feature of PAN's loop analysis process, is that analysis of loop exit conditions has been made a separate process distinct from loop generalization. The primary function of this process is to determine a value for the loop iteration count at loop exit. Having a distinct process to analyse loop exit conditions appears to be unique to PAN. The 'usual' method that symbolic executors use to analyse loops, using recurrence relations, treats the value of the iteration count on exit as just one more relation to be solved simultaneously with any other relations describing the effect of the loop. See, for example, Cheatham et al [1979].

Having a separate process to analyse exit conditions enables PAN to analyse loops with more complex exit conditions than those handled by other analysis systems. In particular, PAN can analyse programs having loops with multiple exits, whose position in the loop is not determined by the iteration construct used. As described in Chapter 4, PAN's loop analysis method first generalizes the execution states at loop entry, and then continues symbolic execution. Once all possible execution states have been created at loop exit statements, these execution states are used in analysis of exit conditions. However, this is not the only way that generalization could be used to analyse loops. Consider the loop structure in figure 5-1. One possible way of analysing such a program would be to allow the symbolic executor to perform several iterations as far as the loop exit statements. Generalization would then be performed on the execution states that are associated with loop exit 1 and loop exit 2. This generalization would include determining the exit conditions and would therefore appear simpler than PAN's method of using a separate process for analysing exit conditions. However, such a method would suffer from requiring two generalizations, and since loop generalization is the most time consuming task performed by PAN, would almost double the time required to analyse a program. Also, correct analysis of loop exit conditions means that the generalization processes would not be fully independent.

Exit from the loop in figure 5-1 occurs when either cond-1 or cond-5 are true, and this information will only be available by examining the execution states at both loop exit 1 and loop exit 2.

To avoid these problems, PAN loop analysis only performs generalization of execution states at loop entry. The result of this generalization is a single execution state, as described in Chapter 4. The normal symbolic executor, described in Chapter 3, will then continue until execution has reached all possible loop exits statements. At this point the loop exit processing described in this section is invoked. It analyses all execution states to determine the value of the iteration count, which was left undetermined by the generalization at loop entry

One complication with this process is that the sequences created when generalizing execution states at loop entry may need to be extended to properly describe the situation at loop exit. This complication is not serious enough to outweigh the advantages of PAN's method of loop analysis.

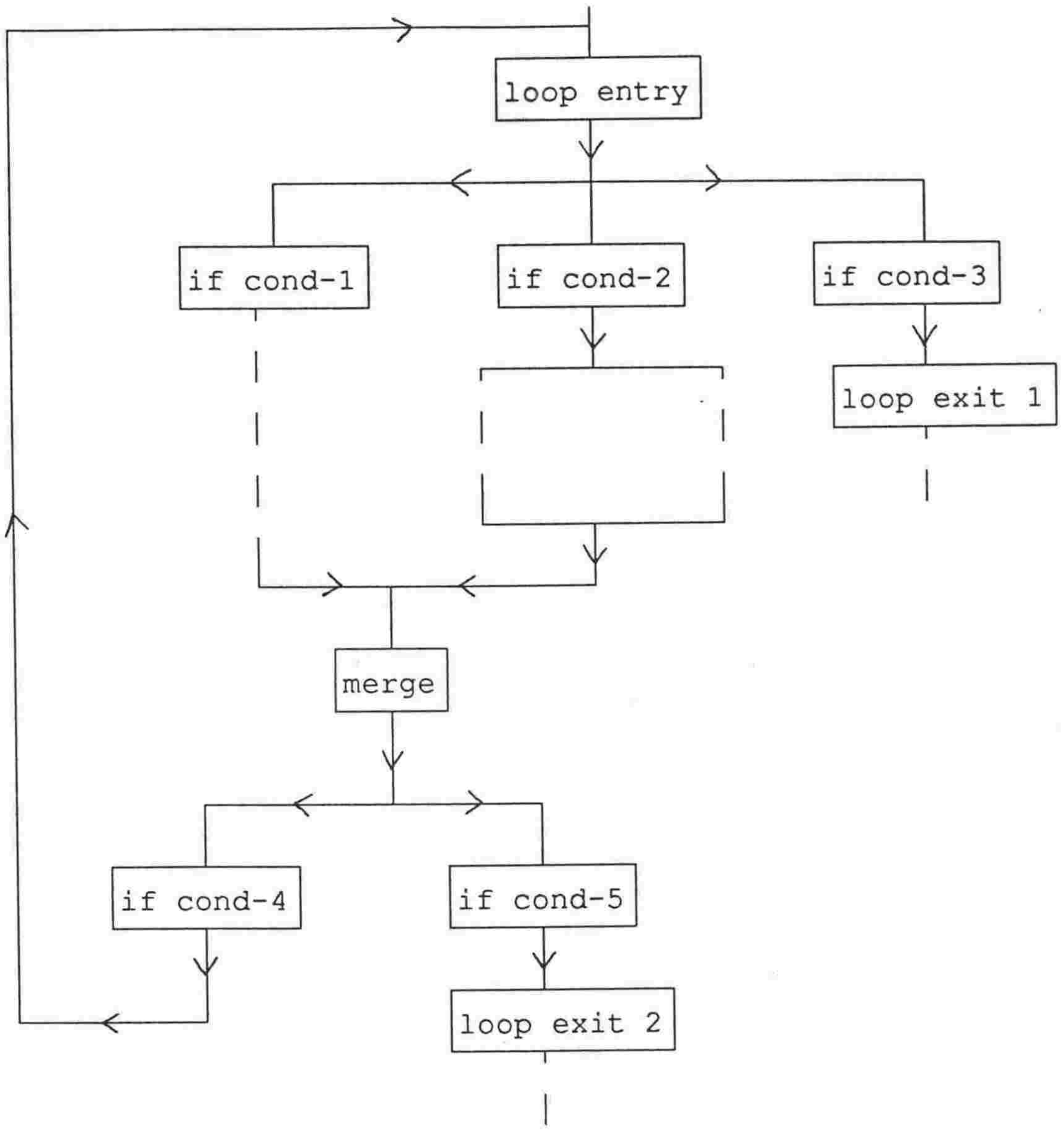


Figure 5-1 - Multiple Loop Exits

Therefore, there are two stages to PAN loop exit analysis: first extending the sequences to include any items created between loop entry and the loop exit; second, determining a value for the iteration count. Note that extending sequences involves analysing each execution state separately, whereas determining a value for the iteration count involves analysing all execution states associated with loop exit statements.

5.1.1 Extending Sequences

The information in the execution states associated with loop exit will be carried forward, eventually to the specifications output. Thus, we want to ensure that they contain the best representation of sequences and items.

The execution states input to the loop exit process will have sequences created during loop generalization and, possibly, additional sources and objects created between loop entry and loop exit. For nested loops, the exit path for an outer loop may have contained an inner loop. Thus, even new sequences may have been created. Although not essential for program analysis, it is nevertheless desirable to check whether these new items can be fitted into existing sequences. To see the impact that this process has on the program analysis, consider the program in figure 5-2. The generalized execution state will have sequences

```
S1 = (sequence i = 1 to k
        sequential-object-in-source(i, SOURCE-1))
S2 = (item: item ∈ S1 ∧ color(item) = red)
S3 = (item: item ∈ S1 ∧ ¬ color(item) = red)
```

having updated properties of

$$\forall \text{ item (item} \in \text{S2} \rightarrow \text{position(item) = pos-b)}$$

and

$$\forall \text{ item (item} \in \text{S3} \rightarrow \text{position(item) = pos-c)}.$$

Two execution states will be associated with loop exit. One, E1, will be the result after execution has gone through statement 6 and will have a new red object, with an updated position property of pos-b. The other, E2, will be the result after execution has gone through statement 7, and will have a new not red object, with an updated

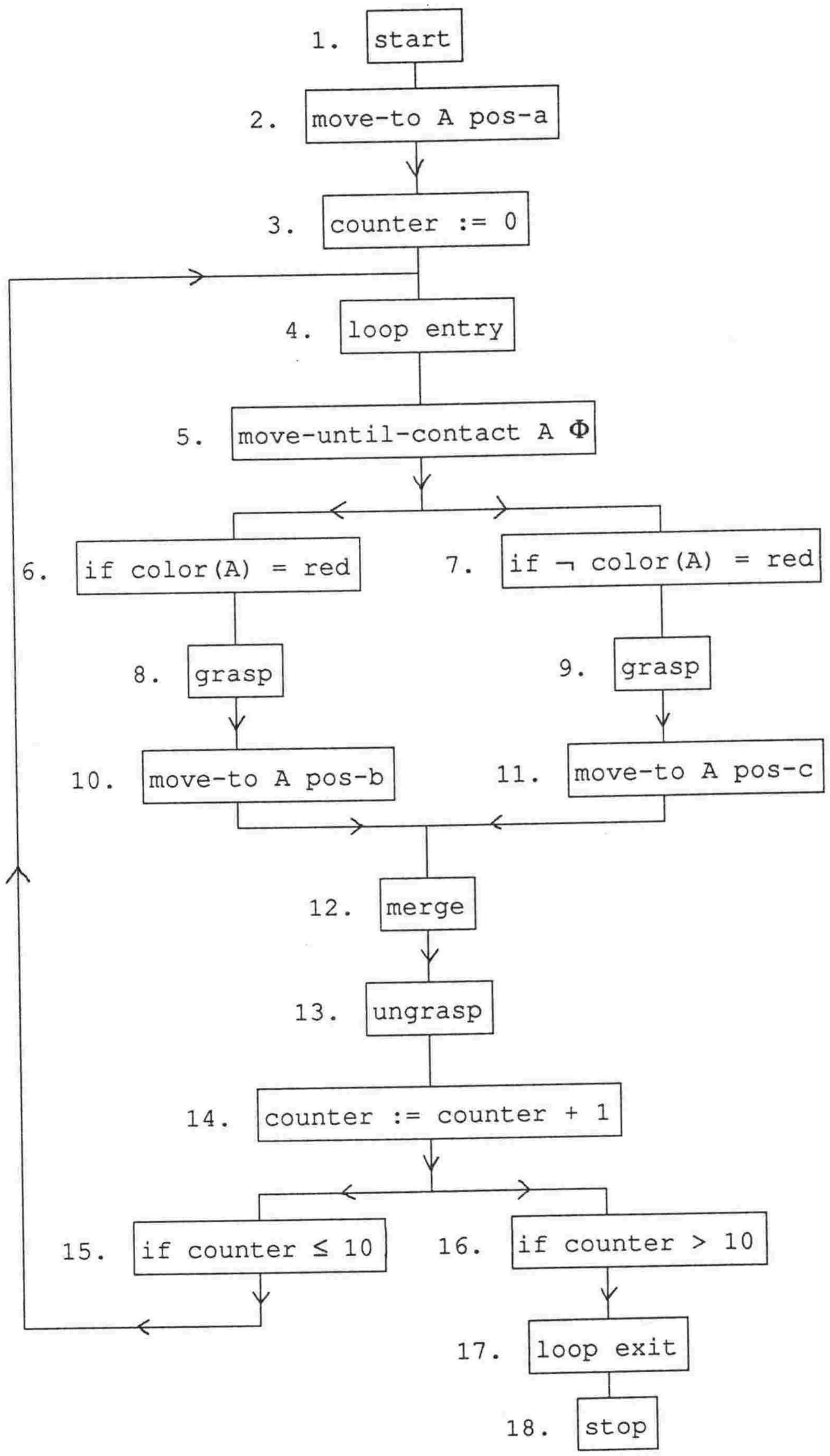


FIGURE 5-2 - New items on Loop Exit

position property of pos-c. Both execution states must be the result of execution having gone through statement 16, at which time the value of counter will have been k+1. Thus, these execution states will have path conditions of (color(OBJECT-1) = red $\wedge$ k+1 > 10) and ($\neg$color(OBJECT-1) = red $\wedge$ k+1 > 10), respectively, where

OBJECT-1 = sequential-object-in-source(k+1, SOURCE-1).

If sequence extension is not performed, then assuming k has been resolved to the value 10 by the process described below, then since 10+1 > 10 can be simplified to T, the output specifications produced will be

```
if color(OBJECT-1) = red

then

    ∀ item (item ∈ S2 → position(item) = pos-b)
    ∀ item (item ∈ S3 → position(item) = pos-c)
    position(OBJECT-1) = pos-b

where

    OBJECT-1 = sequential-object-in-source(11, SOURCE-1)
    S1 = (sequence i = 1 to 10
              sequential-object-in-source(i, SOURCE-1))
    S2 = (item: item ∈ S1 ∧ color(item) = red)
    S3 = (item: item ∈ S1 ∧ ¬ color(item) = red)
```

and

```
if ¬ color(OBJECT-1) = red

then

    ∀ item (item ∈ S2 → position(item) = pos-b)
    ∀ item (item ∈ S3 → position(item) = pos-c)
    position(OBJECT-1) = pos-c
```

where

```
OBJECT-1 = sequential-object-in-source(11, SOURCE-1)
S1 = (sequence i = 1 to 10
        sequential-object-in-source(i, SOURCE-1))
S2 = (item: item ∈ S1 ∧ color(item) = red)
S3 = (item: item ∈ S1 ∧ ¬ color(item) = red)
```

This is not wrong, but is clumsy compared to the alternative specification

```
∀ item (item ∈ S2 → position(item) = pos-b)
∀ item (item ∈ S3 → position(item) = pos-c)
```

where

```
S1 = (sequence i = 1 to 11
        sequential-object-in-source(i, SOURCE-1))
S2 = (item: item ∈ S1 ∧ color(item) = red)
S3 = (item: item ∈ S1 ∧ ¬ color(item) = red)
```

To achieve this second form of specification, PAN needs to extend the sequences to include OBJECT-1, after which the two execution states will have different path conditions but identical effects. Merging of these execution states is discussed at the end of section 5.1.2, where this example is revisited. Note that extending the sequences to include OBJECT-1 requires recognizing that the updated properties of OBJECT-1 are consistent with the updated properties of S2 and S3. We now consider the general requirements of extending sequences.

Initially ignoring the problems associated with updated properties, a sequence of the form

```
S = (sequence i = 1 to n D(i))
```

may be extended to

S = (sequence i = 1 to n+1 D(i))

if the execution state contains an item-j, whose definition is D(n+1).

If a sequence S is extended to include a new item, item-j, the item may also be included in subsequences of S. Given a subsequence

S′ = (item: item ∈ S ∧ P(item))

then item-j will be in S′ if P(item-j) is satisfied.

Now considering that both items and sequences have updated properties, it is necessary to ensure that the updated properties of the items are consistent with those of the sequences it is added to. Recall from section 4.6 that a sequence including items with updated properties may be generalized by recording updated properties on subsequences. Thus to determine whether an item added to a sequence has consistent updated properties, we need to consider the subsequences the item is in, and then compare the updated properties of the sequence and all of these subsequences with those of the item.

We now consider two complications in this apparently simple process. The first complication concerns references to other sequences. The definition of the next item in a sequence, the definition of the new item to be included, the values of the updated properties, and the definitions of subsequences, may all contain references to other sequences in such subexpressions as:

- size(S)
- item-in-sequence(i, S)
- position-in-sequence(item, S)
- map(item, S, S′).

The use of such expressions is complicated by the fact that sequences are being modified by the sequence extension process, and the values of such expressions may change when a sequence is extended. This complication can be addressed by extending sequences in an order such that if the definition of sequence S1 refers to sequence S2, then S2 is extended before S1, and all sequences are extended before properties are tested for consistency. Non circularity is guaranteed by the fact that if any sequence S′ is used in the definition or updated properties of some other sequence S, then S′ must be a superset of a sequence concurrent with S. This can be verified by considering all processes described in sections 4.3, 4.4 and 4.6.

The second complication involves concurrent sequences. The use of the map function to define subsequences requires that concurrent sequences are the same length. It is therefore invalid to extend a sequence unless all concurrent sequences are also extended.

We are now ready to present an algorithm to extend sequences. This algorithm makes use of the association of statement conditions with sequences, and the distinction between sequences generated using sequence generation model 1 and those generated using model 2, as described in section 4.3.2.

Sequence Extension Algorithm

1. Let the weakest statement conditions associated with sequences be $C_1,...,C_m$. Repeat steps 2 through 6 for l = 1 to m.

2. For every sequence S = (sequence i = 1 to n D(i)) in the execution state that is associated with statement condition C_l, try to find an item I, so that the definition of I = D(n+1). If this cannot be done for every such sequence then exit without extending any sequences.

3. For every sequence S extended in 1, find all subsequence, subsubsequences, ... of S. Each of these, will be of the form S2 = (item: item ∈ S1 ∧ P(item)), where S1 is either extended in 1, or is itself a subsequence of such a sequence. See whether the item, I, added to S, is in S2 by testing whether I ∈ S1 and using the theorem prover to establish whether P(I) is true.

4. For every subsequence S2, found in step 3 to include a new item, try to extend every primitive sequence concurrent with S2. If this cannot be done, then exit without extending any sequences.

5. Now for every primitive sequence, S, which has been extended with item I, we check consistency of the updated properties. Let subsequences, subsubsequences, etc of S which include I be {S1,...,Sn}. Suppose item I has properties $p_1,...,p_q$ updated with values $v_1,...,v_q$. For all i = 1 to q, check that there is a j, $1 \le j \le n$, so that Sj has updated property p_i with value v_i. Conversely, given any property p, with value v of any Sj, check that p is one of the p_i, $1 \le i \le q$, and v is equal to v_i. If this consistency test fails, then exit without extending any sequences.

6. If a primitive sequence S is extended to include the new item, I, then remove I from the execution state, replacing any reference to I by item-in-sequence(size(S), I).

5.1.2 Resolving the Iteration Count

We now consider the loop analysis tasks which involve multiple execution states. The most obvious of these is to determine the value of the iteration count. Even for programs having multiple exits, this process is relatively straightforward because of the quality of information which has been previously determined. Each execution state associated with a loop exit explicitly records the path condition which must be satisfied for execution to reach loop exit from the start statement. If we remove from the path condition the component required to reach loop entry from the start statement we are left with the condition which must be satisfied to traverse some specific path from loop entry to loop exit. We refer to this condition as the *exit condition* for this execution state. Suppose that execution states E1,...,En are associated with all loop exit statements and have exit conditions of P1,...,Pn. Since execution will exit from the loop as soon as any of these conditions is satisfied, the condition which needs to be satisfied for exit is

$P1 \vee \ldots \vee Pn$. The iteration count on exiting the loop will be the minimum integer value which makes $P1 \vee \ldots \vee Pn$ true. So, if we can express $P1 \vee \ldots \vee Pn$ as a function of the iteration count k, i.e.

$$P(k) = P1 \vee \ldots \vee Pn$$

then the value of the iteration count on exit will be

$$k_{exit} = (\text{minimum } \{j: P(j) \wedge (\text{integer } j)\}).$$

We now consider some examples of what is required to express the disjunction of the exit conditions, $P1 \vee \ldots \vee Pn$, as a function of the iteration count.

In some cases the connection between the exit conditions and the iteration count k is explicit. For example, loop generalization of the program fragment in figure 5-3 will determine that counter = k at loop entry. Thus the execution state associated with the loop exit statement will have exit condition of $k+1 > 10$. Since on exit, k is always the minimum integer to make the exit condition true, we easily establish that $k_{exit} = 10$.

In other cases, the connection is more indirect. Consider the program fragment in figure 5-4. In this case we will have two execution states associated with the two loop exit statements. One will have an exit condition of color(OBJECT-n) = red and the other color(OBJECT-n) = blue. So far these exit conditions do not seem to involve k. However, when we substitute in the definition of the objects, we obtain:

color(sequential-object-in-source(k+1, SOURCE-1)) = red

and

color(sequential-object-in-source(k+1, SOURCE-1)) = blue.

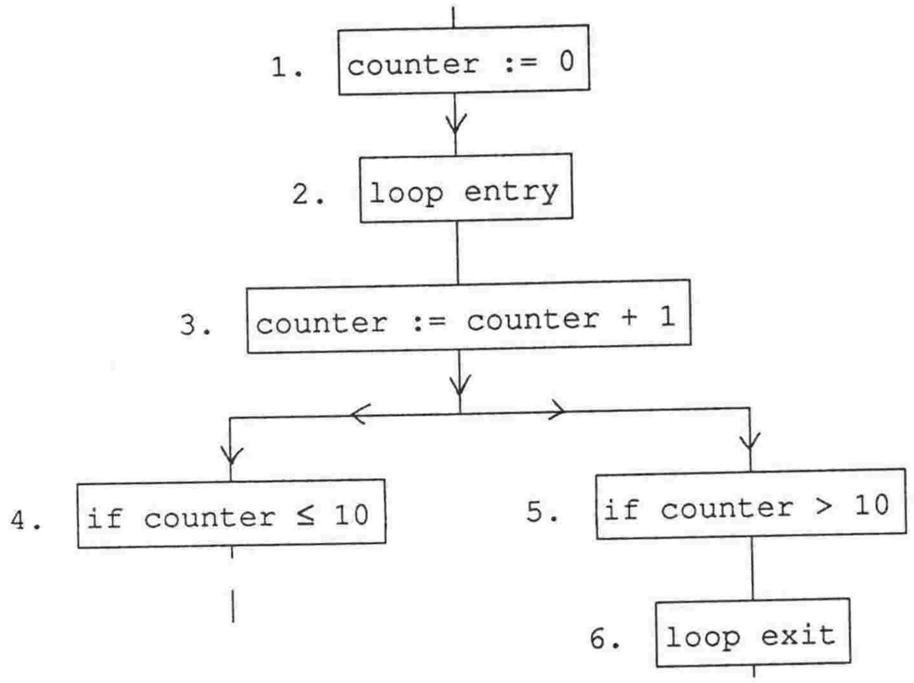


FIGURE 5-3 - Simple Exit Constraints

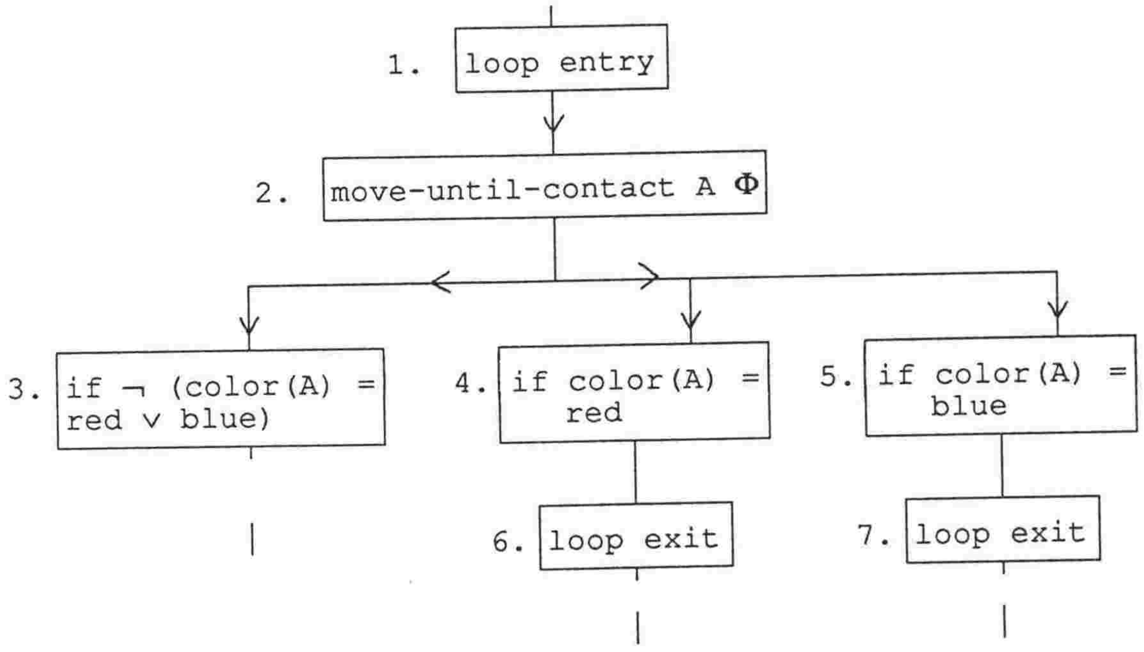


FIGURE 5-4 - More Complex Exit Constraints

Since exit will occur as soon as one or the other of these predicates becomes true we can derive:

```
k = minimum({j: color(sequential-object-in-source(j+1,
                          SOURCE-1)) = red
                ∨ color(sequential-object-in-source(j+1,
                          SOURCE-1)) = blue})
```

Thus, in general, we see that we need to substitute into the exit conditions the definition of any items whose definitions involve k.

Once the iteration count has been resolved, what other loop exit analysis is there involving multiple execution states associated with loop exit statements? In Chapter 3, when the statement condition was defined, we stated that each inner loop was to be treated as shown in figure 5-5. If execution reaches loop exit i, then condition i needs to be added to the statement condition. We can now determine the value of condition i. If loop exit i has execution states E1,...,En associated with it, having exit conditions of P1,...,Pn, then the condition required to reach loop exit i down any path is P1 ∨ ... ∨ Pn. Thus, if the exiting loop, L, is nested in an outer loop, with statement condition Q required to reached the loop entry of L, then the statement condition to reach loop exit i is Q ∧ (P1 ∨ ... ∨ Pn).

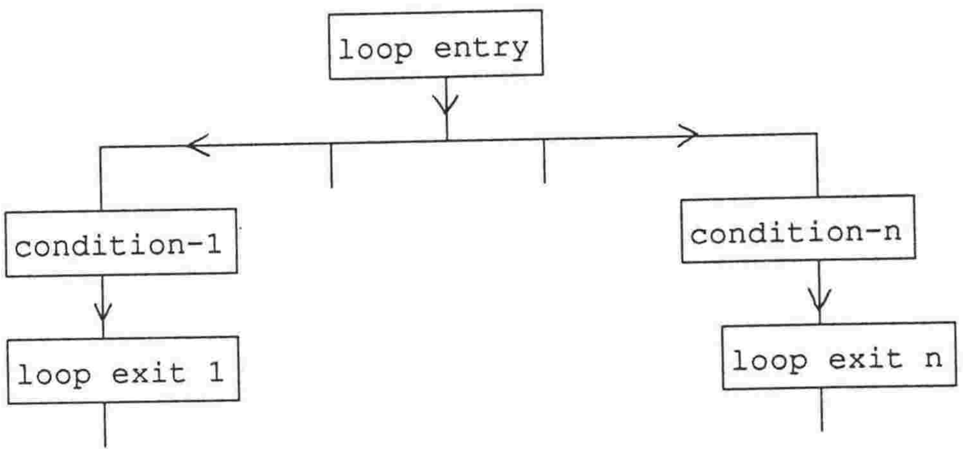


Figure 5-5 Inner Loops for Statement Condition

The final task of loop exit analysis is to ensure that no unnecessary execution states are allowed to exit from the loop. Consider again the program in figure 5-2. Since we now have a method for extending sequences and determining the value of k, let us see if we are now in a position to produce the more concise specification identified in section 5.1.1. The sequence extension process will extend sequences S1, S2 and S3 to include OBJECT-1, and any reference to OBJECT-1 will be replaced by item-in-sequence(k+1, S1), since the extended size of S1 is k+1.

The iteration count at exit will then be resolved to 11, using the process described earlier in this section. However, there will still be two execution states associated with loop exit. Their path conditions will have been modified to be

    color(sequential-object-in-source(11, SOURCE-1)) = red

and

    ¬ color(sequential-object-in-source(11,SOURCE)) = red.

by the removal of OBJECT-1 from the execution states. The effects in each execution state are now the same. The fact that we still have two execution states will lead to an eventual specification being produced of:

    if color(sequential-object-in-source(11, SOURCE-1)) = red

    then

        ∀ item (item ∈ S2 → position(item) = pos-b)
        ∀ item (item ∈ S3 → position(item) = pos-c)

    where

```
        S1 = (sequence i = 1 to 11
                sequential-object-in-source(i, SOURCE-1))
        S2 = (item: item ∈ S1 ∧ color(item) = red)
        S3 = (item: item ∈ S1 ∧ ¬ color(item) = red)
```

and

```
    if ¬ color(sequential-object-in-source(11,SOURCE-1)) = red.

    then

        ∀ item (item ∈ S2 → position(item) = pos-b)
        ∀ item (item ∈ S3 → position(item) = pos-c)

    where

        S1 = (sequence i = 1 to 11
                sequential-object-in-source(i, SOURCE-1))
        S2 = (item: item ∈ S1 ∧ color(item) = red)
        S3 = (item: item ∈ S1 ∧ ¬ color(item) = red)
```

This specification is still not satisfactory. The problem is caused by the fact that we have two execution states which only vary in path conditions. Since execution states can be interpreted as

```
    if path condition then effect
```

then whenever we have associated with a statement S, execution states E1,...,En identical except for path conditions P1,...,Pn, then we can replace E1,...,En with a single execution state with a path condition of P1 ∨ ... ∨ Pn. Since this situation is likely to occur because sequences have been extended, loop exit analysis checks whether there are any set of execution states associated with any loop exit statements which are identical except for path conditions, and if so, merges them. Performing such a simplification at loop exit will enable PAN to analyse the program in 5-2 correctly.

We are now ready to describe the algorithm for the loop exit processing on multiple execution states.

Exit Algorithm

1. Suppose the exit conditions in exiting execution states contain P1,...,Pn. Find P = P1′ ∨ P2′ ... ∨ Pn′, the combined exit condition, where Pi′ is formed from Pi by replacing any item referred to in Pi by its definition, if that definition contains the iteration count k.

2. The value of k on exit is then given by (minimum {j: P(j) ∧ (integer j)}), where P(j) is derived from P by replacing all occurrences of k by j. This value then replaces all uses of k in all exiting execution states.

3. Now for each exit statement and the execution states associated with it:

   Group the execution states associated with this statement, so that in each group the execution states are identical except for path conditions. For each group:

      Let the path conditions for these execution states in the group be Q1,...,Qn with exit conditions P1,...,Pn. Define Q = Q1 ∨ ... ∨ Qn and P = P1 ∨ ... ∨ Pn. Now pick one of the execution states in the group, replace the path conditions with Q and add P to the statement condition of the next outermost loop (if there is one). Change the status of the execution state to active and update the loop data by removing the innermost loop. Change the status of all other execution states in the group to dead.

5.2 GENERALIZATION VERIFICATION

This section describes how the correctness of PAN's loop generalization can be verified. The generalized execution state produced at the end of Chapter 4 is intended to represent the effect of k iterations. This execution state contains k as a variable and may be referred to as E(k). Since E(k) was generalized from execution states which included all those having completed one iteration, we know that E(1) does represent the effect after one iteration. We want to show that if E(k) represents the effect after k iterations then E(k+1) represents the effect after k+1 iterations. We can then assert by induction that the generalization is correct.

To find the effect after k+1 iterations we begin symbolic execution from E(k) and proceed until we have execution states E1,...,En associated with loop entry, having performed k+1 iterations. For each i, we want to show that the effect represented by Ei is the same as the effect represented by E(k+1). The only complication with this scheme is the now familiar one of extending sequences. Consider again the program in figure 5-2, this time in the context of generalization verification. Once execution resumes from E(k), two execution states E1 and E2 will be associated with loop entry having completed k+1 iterations. Their effects can be summarised as:

E1:

```
        ∀ item (item ∈ S2 → position(item) = pos-b)
        ∀ item (item ∈ S3 → position(item) = pos-c)
        position(OBJECT-1) = pos-b

    where

        OBJECT-1 = sequential-object-in-source(k+1, SOURCE-1)
        S1 = (sequence i = 1 to k
                 sequential-object-in-source(i, SOURCE-1))
        S2 = (item: item ∈ S1 ∧ color(item) = red)
        S3 = (item: item ∈ S1 ∧ ¬ color(item) = red)
```

E2:

$\forall$ item (item $\in$ S2 $\rightarrow$ position(item) = pos-b)
$\forall$ item (item $\in$ S3 $\rightarrow$ position(item) = pos-c)
position(OBJECT-1) = pos-c

where

OBJECT-1 = sequential-object-in-source(k+1, SOURCE-1)
S1 = (sequence i = 1 to k
sequential-object-in-source(i, SOURCE-1))
S2 = (item: item $\in$ S1 $\wedge$ color(item) = red)
S3 = (item: item $\in$ S1 $\wedge \neg$ color(item) = red)

Whereas the effect of E(k+1) will be

$\forall$ item (item $\in$ S2 $\rightarrow$ position(item) = pos-b)
$\forall$ item (item $\in$ S3 $\rightarrow$ position(item) = pos-c)

where

S1 = (sequence i = 1 to k+1
sequential-object-in-source(i, SOURCE-1))
S2 = (item: item $\in$ S1 $\wedge$ color(item) = red)
S3 = (item: item $\in$ S1 $\wedge \neg$ color(item) = red).

So far the effects of E1 and E2 are not the same as those of E(k+1). However, once sequences have been extended as described for loop exit the effects of E1 and E2 will be the same as that of E(k+1).

Thus the generalization verification requires the same sequence extension process as that used by loop exit processing. Also, the mechanical procedure for checking that the effects of Ei are the same as the effects of E(k+1) is the same as that required for merging those execution states associated with loop exit that are identical apart from path conditions. Thus generalization verification requires similar processing to loop exit processing.

The algorithm for generalization verification can be stated as follows.

Algorithm for Generalization Verification

1. Beginning with the generalized execution state E(k), use symbolic execution to generate execution states E1,...,En, associated with loop entry, representing the effects after k+1 iterations.

2. Extend sequences in E1,...,En using the algorithm in section 5.1.1.

3. Verify that the effect of each Ei = the effect of E(k+1).

Chapter 6
Brute Force Search

6.1 INTRODUCTION

The brute force search is the second of PAN's two generalization techniques, the first being the models described in Chapter 4. The brute force search, being far more time consuming, is only invoked if the models fail to find a generalization.

The idea behind the brute force search is very simple - a generalization task which fails using raw input data may succeed after additional facts have been derived from the input data. This technique has been successfully used by Dietterich and Michalski[1985] in the SPARC/E system which can play the game Eluesis. For example, given a card described as 'jack of hearts', this system will add the derived fact that the card is red, before generalization is attempted. However, in the SPARC/E system, additional data on any object is derived without considering data available about other objects. Given properties $p_1,...,p_n$ of some object, new properties $p_{n+1},...,p_m$ of the same object are derived. While useful for playing Eluesis, this restricted technique would not provide sufficient power to significantly extend the programs that PAN can analyse. Thus we extend the technique so that not only does PAN use all relevant facts to derive new ones, but this derivation process actually produces the generalization. In PAN's case this means that this derivation process produces the generalized facts required for the generalized execution state. The task being attempted can be described in terms not specific to program analysis as follows:

Suppose we are provided with a set of observations O1,...,On. Each Oi contains observations in the form of well formed formulas in some logical language. We want to derive a single observation

O, which is a generalization of O1,...,On in the sense that a formula P is in O if and only if P is derivable from the formulas in Oi for i = 1 to n, by the following method:.

- for each Oi, continue to apply any derivation rules that are available, to the formulas in Oi until a stable set Oi′ is obtained

- define O as O1′ ∩ O2′ ∩ ... ∩ On′.

Why is such a method not commonly used for generalization tasks? The problem lies in producing the Oi′ sets. It may not be possible to produce them in any finite amount of time. For example, suppose Oi contains the formulas:

```
weight(OBJECT-1) = 1
weight(OBJECT-2) = 2
weight(OBJECT-3) = 3
```

and derivation rules include arithmetic substitution. We can generate

```
weight(OBJECT-1) = weight(OBJECT-2) - 1
weight(OBJECT-1) = 2 * weight(OBJECT-2) - 3
weight(OBJECT-1) = 3 * weight(OBJECT-2) - 5
  .       .       .       .       .       .
  .       .       .       .       .       .
```

etc, so that Oi′ is not a finite set. Of course, we could simply apply the derivation rules to each Oi for some specified time to derive Oi′′, and use O1′′ ∩ O2′′ ∩ ... ∩ On′′ as an approximation of O. This however makes the order in which the derivation rules are used the critical factor. In the above example, we may never generate

```
weight(OBJECT-3) = weight(OBJECT-1) + weight(OBJECT-2).
```

which may be the desired generalization. Thus, limiting the derivation of new formulas by time does not address the real problems with this method: that most applications of derivation rules will not produce formulas useful for the generalization task. Thus the challenge is to limit the derivation rules to those which will produce only a limited set of new formulas containing those most likely to be a valid generalization.

To address this problem, we recall from Chapter 4 that in the partly generalized execution states available to PAN, we have, in addition to formulas, the powerful concept of a sequence. Since sequences describe those sets of items which have been distinguished by the program being analysed, any useful generalizations are likely to involve sequences and we can restrict many derivation rules by expressing them in terms of sequences instead of unrestricted sets. Experiments with PAN have shown that with a carefully chosen set of derivation rules, the brute force search can provide solutions to many different program analysis generalization problems in reasonable time.

Programs that PAN can analyse using this technique, but which cannot be analysed using the models of Chapter 4 include:

1. programs which determine the maximum or minimum of a sequence. An example of PAN analysing such a program was given in Chapter 1.

2. programs in which variables have values which are a function of both the size of a sequence as well as the values of properties of items in a sequence. An example of this type of program in given in figure 6-1, in which v has the value

$$\sum_{i-1}^{size(S)} \text{position-in-sequence(item-in-sequence(i, S))} *$$

$$\text{weight(item-in-sequence(i, S)}$$

```
where S = (sequence i = 1 to n
           sequential-object-in-source(i,A))
```

3. programs in which variables are used to refer to the properties of non current items. An example of this type of program is given in figure 6-2, in which v is used in this way, enabling a single record of file B to be created from two records of file A.

4. programs in which the updating of some property of items in a sequence S is controlled by a variable counting the size of some sequence not equal to S. An example of this type of program is given in the appendix.

Obviously the brute force search cannot extend the analysis system beyond the capabilities of the specification language described in Chapter 2. The strength of the method is in being able to describe effects using the language components available, such as size, position-in-sequence etc, in a flexible way. This contrasts with the models of Chapter 4, which are only able to describe effects which are structured in the way required by the model.

The remainder of this chapter first describes in detail PAN's use of this technique, the initial set of formula and the derivation rules used. It then describes the use of the derived formulas to complete the generalization task. The chapter finally describes how the process has been made efficient enough to be practical.

6.2 PAN'S USE OF BRUTE FORCE SEARCH

As discussed in Chapter 4, the brute force search is only invoked if other methods have failed. In particular, it is used when loop generalization has failed to explain:

- the value of a variable at loop entry

or

- the value of an updated property for items included in a sequence.

This process is called by loop generalization and its input is:

- all execution states input to loop generalization which have completed the full number of iterations

- the identifiers of any variables not successfully analysed

- a list of (object , property) pairs, the objects being members of some sequence, and having an unsuccessfully analysed property.

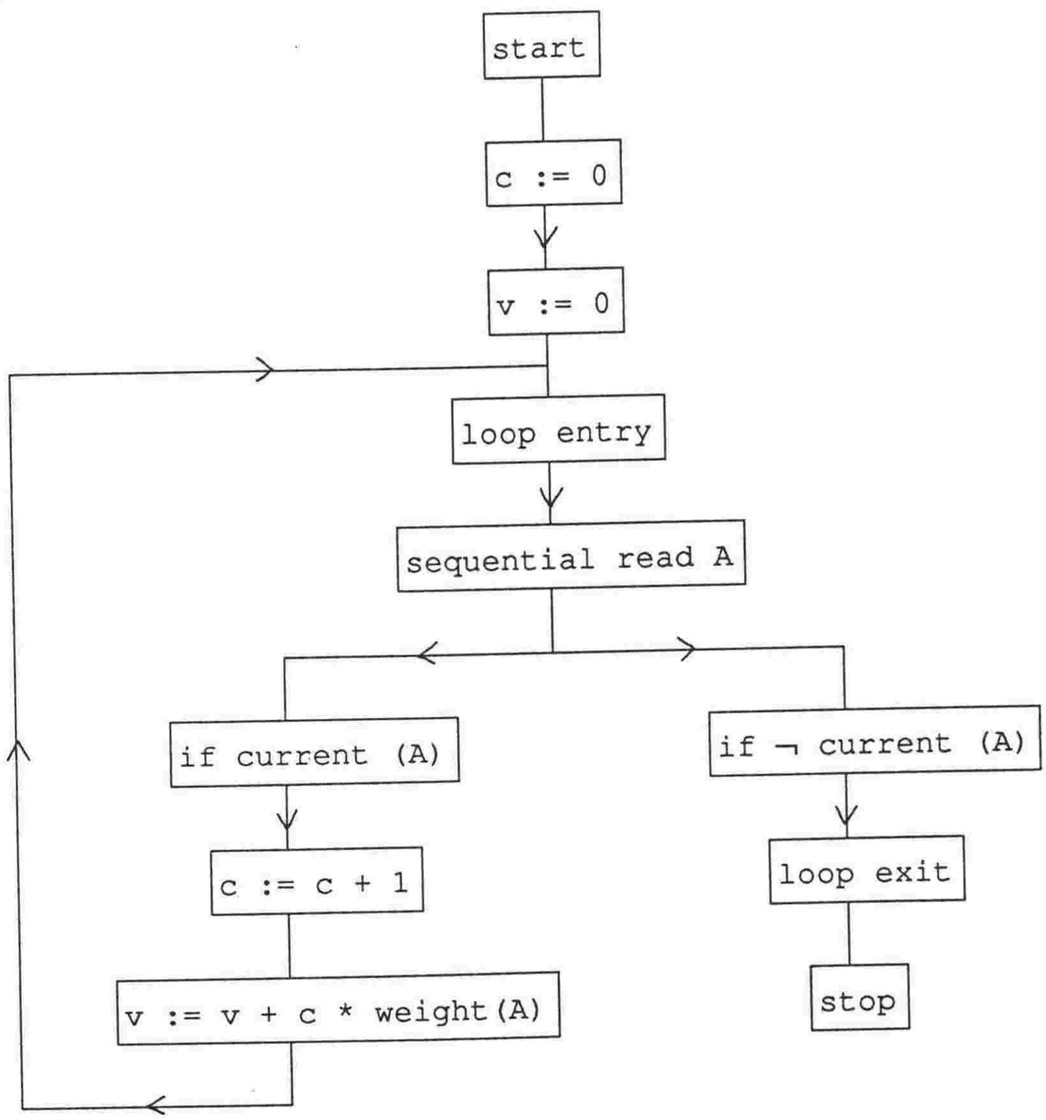


Figure 6-1 Complex Variable Value

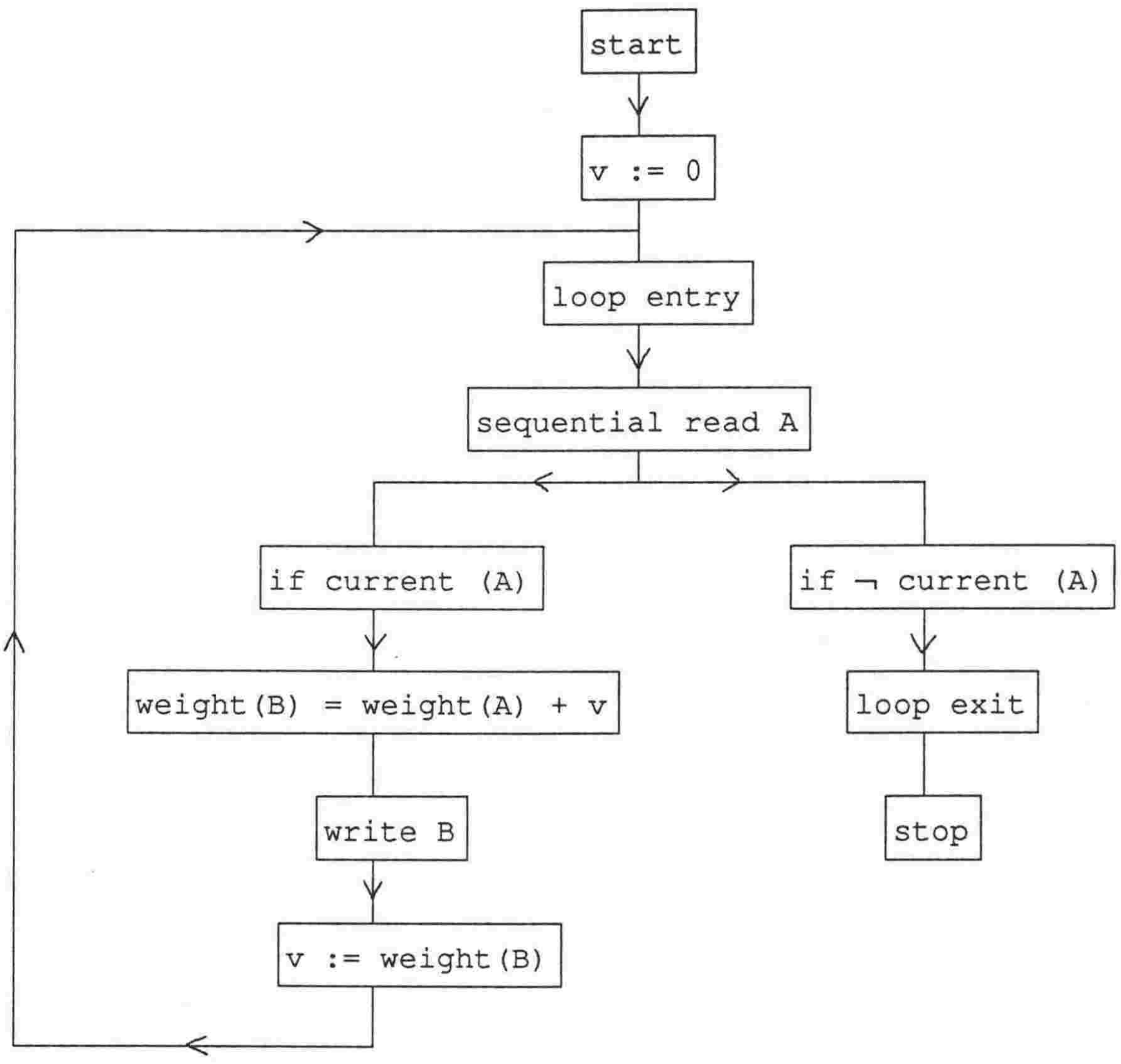


Figure 6-2 Variable Referencing Previous Item

The output from this process consists of new values for variables and new properties of sequences. Each variable will either have a generalized value, or the value 'unknown'. Similarly, each updated object property will either be explained by an updated property of a sequence the object is in, or else will have the value 'unknown'.

The brute force search operates on each execution state in turn. For each execution state an initial set of formulas is obtained and expanded. The results from all expansions are then analysed.

6.3 INITIAL SET OF FORMULAS

The brute force search for a single execution state operates by expanding an initial set of formulas obtained from the execution state, global data, and the partial results of loop generalization.

The set of formulas obtained from the execution state consists of:

- the value of all variables
- the value of all updated properties of objects
- the path condition.

The set of formulas obtained from global data consists of:

- the definition of each object.

the set of formulas obtained from the partial results of loop generalization consists of:

- the definition of each sequence
- the size of each sequence
- the members of each sequence.

6.4 BRUTE FORCE EXPANSION

Once the initial set of formulas for an execution state has been found, the brute force search proceeds to generate additional formulas using the rules shown in table 6-1. These rules are ad hoc, they were selected by considering how a language component such as size or position-in-sequence could be introduced into an expression. The rest of this section describes these rules in more detail.

Table 6-1 Brute Force Expansion Rule

In this table S means a sequence, O is an object and n, m are integers.

Category	Name	Action
Simple Substitution	Sequence Size	replace n by size(S)
	Item in Sequence	replace O by item-in-sequence(i, S)
	Position in Sequence	replace n by position-in-sequence(O, S)
Logical Deduction	all rules used in the theorem prover	
	Existential Quantification	replace P(O) by $\exists$ x (x $\in$ S $\wedge$ P(x))
	Universal Quantification	replace P(O1),...,P(On) by $\forall$ x (x $\in$ S $\rightarrow$ P(x))
Sequence Substitution	All Items in Sequence	replace term-1 + ... + term-n by $\sum_{i=1}^{size(S)}$ term(i)
		replace term-1 * ... * term-n by $\prod_{i=1}^{size(S)}$ term(i)
	Property of Sequence	replace property(item-in-sequence(i,S)) = Q(m_{i1},...,m_{ik}) for i = 1 to size(S) by $\forall$ i (1 $\le$ i $\le$ size(S) $\rightarrow$ property(item-in-sequence(i,S) = Q(f_1(i),...,f_k(i)))

Sequence Size

If there is a formula P, that contains an integer n, equal to size(S) for some sequence S, then generate P′, which is derived from P by replacing n by size(S).

An example of the use of this rule is to determine that a variable is counting the size of a sequence. Note that this allows the brute force search to generate a formula equivalent to the most important of those generated by model 3 for variable values discussed in section 4.5. However, because this rule operates on any integer of the right value within a (possibly) larger expression, it is more general.

Item in Sequence

If there is a formula of the form P(OBJECT-n), where OBJECT-n is the jth element of some sequence S, then generate

```
    P(item-in-sequence(j, S)).
```

An example this rule is to generate

```
    weight(item-in-sequence(1, S1)) = weight(item-in-sequence(1,
                                                         S2))
                                       +
                                      weight(item-in-sequence(1,
                                                         S3))

    weight(item-in-sequence(2, S1)) = weight(item-in-sequence(2,
                                                         S2))
                                       +
                                      weight(item-in-sequence(2,
                                                         S3))
    weight(item-in-sequence(3, S1)) = weight(item-in-sequence(3,
                                                      S2))
                                      +
                                      weight(item-in-sequence(3,
                                                      S3))
```

from

```
    weight(OBJECT-1) = weight(OBJECT-4) + weight(OBJECT-7)
    weight(OBJECT-2) = weight(OBJECT-5) + weight(OBJECT-8)
    weight(OBJECT-3) = weight(OBJECT-6) + weight(OBJECT-9)
```

where S1, S2 and S3 are sequences such that S1 = {OBJECT-1 OBJECT-2 OBJECT-3}, S2 = {OBJECT-4 OBJECT-5 OBJECT-6}, S3 = {OBJECT-7 OBJECT-8 OBJECT-9}.

Position in Sequence

If there is a formula of the form P(m), where m is an integer which equals position-in-sequence(OBJECT-n, S) for some sequence S and OBJECT-n, and n does not appear in P as part of item-in-sequence(n, S'), then generate a new formula

```
    P(position-in-sequence(OBJECT-n, S)).
```

An example of this rule is to generate

```
    weight(OBJECT-1) = position-in-sequence(OBJECT-1, S')
    weight(OBJECT-2) = position-in-sequence(OBJECT-2, S')
    weight(OBJECT-3) = position-in-sequence(OBJECT-3, S')
```

from

```
    weight(OBJECT-1) = 1
    weight(OBJECT-2) = 2
    weight(OBJECT-3) = 5
```

where OBJECT-1 is 1st in S′, OBJECT-2 in 2nd in S′ and OBJECT-3 is 5th in S′.

Some comment is required on the limitation that n must not appear in item-in-sequence(n, S′). If we do allow n to appear in this subexpression, then this rule interacts in an unfortunate way with the Item in Sequence rule. Starting with

```
    P(m)
```

these rules would allow us to generate

```
    P(position-in-sequence(OBJECT-n, S)) by Position in Sequence

    P(position-in-sequence(item-in-sequence(i, S), S))
         by Item in Sequence

    P(position-in-sequence(item-in-sequence
              (position-in-sequence(OBJECT-n, S), S), S))
         by Position in Sequence
    etc
```

Such a series of formulas is very unlikely to be useful, and is prevented by the above restriction.

Simple Logical Rules

All the rules previously discussed in the theorem prover, section 3.2.7.1, are used. These allow simple logical deductions on our set of formulas.

Existential Quantification

If there is a formula P(OBJECT-n) and OBJECT-n $\in$ S, for some sequence S, then generate

$$\exists o\ (o \in S \wedge P(o)).$$

This rule was used in the generation of minimum weight of a sequence in the dp domain example in section 1.4.

Universal Quantification

If there are formulas P(OBJECT-i) for i = 1 to n, and S is a sequence such that S = {OBJECT-1 ... OBJECT-n}, then generate

$$\forall o\ (o \in S \rightarrow P(o)).$$

This rule was also used in the generation of the minimum weight of a sequence.

All Items in a Sequence

If there is a formula P that can be expressed so that it contains a subexpression term-1 + ... + term-n, such that term-1,...,term-n are the same except term-i refers to the ith element from concurrent sequences {S1 ... Sp}, where term-j refers to the jth element, and the size of each Si is n. We can then generate P', which is derived from P by replacing

    term-1 + ... + term-n

by

$$\sum_{i=1}^{\text{size(S1)}} \text{term}(i)$$

where term is derived from term-1 by replacing any reference to an element of {S1 ... Sp} by item-in-sequence(i, Sj) for appropriate j.

The only manipulation on P that is done to try to achieve the required form are the associative and commutative laws for addition.

The rule also applies to subexpression term-1 * ... * term-n, by replacing + by * and Σ by Π.

An example of the use of this rule is for a symbolic variable v whose value is given by the formula

$$v = \text{weight(OBJECT-1)} + \text{weight(OBJECT-2)} + \text{weight(OBJECT-3)}$$

and

$$S = \{\text{OBJECT-1 OBJECT-2 OBJECT-3}\}.$$

We will then generate a new formula

$$v = \sum_{i-1}^{\text{size}(S)} \text{weight(item-in-sequence}(i, S))$$

Note that this rule allows the brute force search to generate formulas equivalent to those generated by model 4 for variable values as discussed in section 4.5. However, because this rule operates on any subexpression of the right form it is more general than model 4.

Property of Sequence

If there is a set of formulas

$$\text{property(item-in-sequence}(i, S)) = Q(m_{i1} \dots m_{ik})$$

for i = 1 to n, whose right hand sides differ only in the value of k numbers and where S is a sequence of size n, then generate a new formula

$$\forall i\ (1 \le i \le \text{size}(S) \rightarrow \text{property(item-in-sequence}(i, S)) = Q(f_1(i) \dots f_k(i)))$$

where $f_j(i) = m_{ij}$ for $i = 1$ to n, $j = 1$ to k.

This rule allows us to record the updated properties of all objects in a sequence in a single formula.

An example of the use of this rule is to generate

$$\forall i \ (1 \le i \le \text{size}(S1) \rightarrow \text{weight}(\text{item-in-sequence}(i, S1)) = \text{weight}(\text{item-in-sequence}(i, S2)) + \text{weight}(\text{item-in-sequence}(i, S3)))$$

from the formulas established in the example for Item in Sequence.

6.5 ANALYSING THE RESULTS

Use of the rules described in section 6.4 gives us an expanded set of formulas for each execution state. How are these used to analyse variables and updated object properties?

6.5.1 Analysing Variable Values

The preferred solution to finding a variable value is that for every execution state the expanded set of formulas contains a member v = Q, where Q is expressed in terms of known values which will appear in the generalized execution state. Note that values which have themselves been updated by the program do not count as known values, since this can result in describing one unknown in terms of another.

To achieve such a solution, we require the following:

- v = Q occurs in the expanded formulas for every execution state

- Q contains no variables or items which are members of sequences.

- Q contains no updated properties of objects or variables.

If, however, we cannot find such an expression for v, then we still may be able to find a solution for v by considering those formulas which contain v. If these formulas are Q1, ...,Qn, then the value of v is constrained by satisfying $Q1 \wedge Q2 \wedge \ldots \wedge Qn$. Of course, more than one value may obey this constraint, but we can state

$$v \in \{z: Q'(z)\}$$

where Q′ is $Q1 \wedge Q2 \wedge \ldots \wedge Qn$ with v replaced by z. This procedure was used in the dp domain example in section 1.4, where v was found to be the minimum weight of the items in a sequence.

To use this form we need to find at least one formula Q that contains v in the expanded formulas for all execution states, and is otherwise expressed in terms of known values which will appear in the generalized execution state.

Thus we need to find the set of formulas, Q, obeying the following:

- Q occurs in the expanded formulas for every execution state
- Q contains v, but contains no other variables or items which are members of sequences.
- Q contains no updated properties of objects or sequences.

We then express v using the expression as above.

If we cannot find any such formulas, we set the value of v to 'unknown'.

6.5.2 Analysing Updated Object Properties

Since, in the generalized execution state, updated properties of items in sequences are recorded against the sequence, the only relevant formulas are those which express the updated property of a whole sequence. These formulas are of the form

$$\forall i\ (1 \le i \le \text{size}(S) \rightarrow \text{property}(\text{item-in-sequence}(i, S) = Q(i))$$

Such formulas express the updated property for sequence S.

As for variable values, we require that such a formula be generated from the initial formulas of every execution states, and that Q is expressed in terms of known values which will appear in the generalized execution state. However, in this case, as discussed in section 4.6, we want to express updated properties in the most general way possible. Thus, given a sequence S1 with subsequences S2 and S3, the expanded formulas may contain:

$$\forall i\ (1 \le i \le \text{size}(S1) \rightarrow \text{property}(\text{item-in-sequence}(i, S1)) = Q1(i))$$

$$\forall i\ (1 \le i \le \text{size}(S2) \rightarrow \text{property}(\text{item-in-sequence}(i, S2)) = Q2(i))$$

$$\forall i\ (1 \le i \le \text{size}(S3) \rightarrow \text{property}(\text{item-in-sequence}(i, S3)) = Q3(i))$$

for the same property. In this case only the formula for S1 should be used, as it is more general than those for S2 and S3 (i.e. it 'explains' the updated properties of more objects).

Thus we require a formula P obeying the following:

- P is of the form

  $$\forall i\ (1 \le i \le \text{size}(S) \rightarrow \text{property}(\text{item-in-sequence}(i, S)) = Q(i))$$

  and occurs in the expanded formulas for every execution state

- Q contains no variables or items which are members of sequences

- Q contains no updated properties of objects or sequences

- there is no formula P′ of the form

$$\forall i\ (1 \le i \le \text{size}(S') \rightarrow \text{property}(\text{item-in-sequence}(i, S')) = Q'(i))$$

  obeying the above conditions, where S′ is a superset of S.

If such a formula P can be found, it is recorded as the updated property of the sequence S.

The updated property of any item which is not explained by the updated properties of any sequence the item is in is set to 'unknown'.

6.6 MAKING THE BRUTE FORCE SEARCH EFFICIENT

6.6.1 Search Speed

In the introduction to this chapter we discussed why the brute force search technique is not generally used - the generation of new formulas will never terminate. Having overcome this problem in PAN's use of the technique by limiting the expansion rules, we are faced with the problem that the speed of generation of new formulas may decrease as the number of formulas increases. In this section we show that a nice feature of PAN's derivation rules allows new formulas to be generated at nearly constant speed.

Given a set of rules R1,...,Rn that operate on single formulas and a set Q of formulas {P1 ... Pm} to expand, a simple brute force search algorithm could operate by repeating the following:

- for each i from 1 to n, try to apply Ri to each Pj for j = 1 to m. If the rule fires and results in a formula P not in Q, then add P to the end of Q and increment m.

This algorithm suffers from the disadvantage that it repeatedly tests whether rule Ri can operate on formula Pj. This problem becomes worse as the size of the set Q increases. Thus the time required to produce one new formula increases with the size of Q and the algorithm becomes less and less efficient.

Now let us consider rules that operate on multiple formulas. We may say that rule Ri can produce a new formula from formulas P1,...,Pp ∈ Q, if P1,...,Pp satisfy some condition C i.e. C(P1,...,Pp) is true.

An algorithm to expand Q using such rules could operate by repeating the following:

- for each Ri, search Q for a formula P1 which could make C(P1,.....) true. Search again through Q for P2, P3 etc until either the search fails or P1,...,Pp are found so that C(P1,...,Pp) is true. If rule Ri now produces a new formula P not in Q, add it to the end of Q and increment m.

Such an algorithm potentially requires p passes over the set Q before it can fire a single rule. Such a method may be required for rules having a condition, C, such that we cannot determine whether Pi can satisfy C until P1,...,Pi-1 are already known. However, an examination of the brute force search rules shows that this is not required here. The multiformula rules are such that once any formula Pi is identified, the set of all the other formulas is determined, and so can be found in a single pass over Q.

Thus for PAN's brute force search expansion we can suggest the alternate multiformula algorithm as:

- for each Ri, search Q for a formula Pj which could make C(...Pj...) true. Then in a single pass of Q, try to find P1,...Pj-1,Pj+1,...,Pp to complete condition C. If rule Ri now produces a new formula P and P is not in Q, add it to the end and increment m.

The main remaining problem with this algorithm is now similar to the problem with the single formula rule algorithm. Most of the time will eventually be spent testing whether C(...Pj...) could be true.

We now look at a way of addressing these problems, first for single formula rules.

Given a rule Ri and a set Q of formulas, it is easy to see that if Ri fails to fire on formula Pj ∈ Q, or Ri can fire on Pj but produces a P already in Q, then Ri will never be able to generate a new formula from Pj. Thus we would like to try to apply Ri only to those members of Q which have not already failed to produce a new formula. This can be done by maintaining a new set Qi, in addition to Q, where Qi contains only those elements of Q which have not yet failed for rule Ri. Initially, we set Qi = Q. Thus a more efficient single formula algorithm would be:

- for each i, try to apply Ri to each Pj ∈ Qi. If the rule fires and produces a new formula P not in Q, then add P to Q and to every Qi. If the rule fails to produce such a new formula, then remove Pj from Qi.

This modified algorithm results in some additional memory costs. However, the difference in processing time should be dramatic. As time passes and Q increases in size, there will be some additional processing required to see if a new formula is in Q. This seems unavoidable, but is overshadowed by the gain in efficiency achieved by only allowing rule Ri to fail on Pj once (either because Ri can't fire on Pj, or because the resulting formula P is already in Q).

Can this approach also be used for rules that operate on multiple formulas? The equivalent algorithm would be:

- for each Ri, try every Pj ∈ Qi to see whether Pj could make C(...Pj...) true. If so, then search Q for P1,...,Pj-1,Pj+1,...,Pp to complete condition C. If these can be found, and application of the rule results in a new formula P not in Q, then add P to Q and to every Qi. If any of the above steps fail, then remove Pj from Qi.

This algorithm achieves the same efficiency gain as the algorithm for single formula rules. However, it is not trivially obvious that it does not limit the formulas which can be generated. A formula Pj which is removed from Qi because P1,...,Pj-1,Pj+1,...,Pp cannot be found, cannot be totally discarded because if P1,...,Pj-1,Pj+1,...,Pp are later generated, then we need formula Pj to complete condition C.

However, as each P1,...,Pj-1,Pj+1,...,Pp is generated it will initially be added to Qi and so another attempt to fire Ri will be made. When the last of these is added to Qi, then Ri will indeed fire. This is because the brute force search rules which require multiple formulas are independent of which formula is used to begin the search for the remainder.

An example may make this clearer. Consider the simple multiformula rule included as one of the Simple Logical Rules from section 6.4

Ri: from (< P Q) and (< Q R), generate (< P R).

Suppose that (< x y) is initially added to Q and Qi, but there is at that time no formula of the form (< y z) in Q for any z. Since (< x y) is in Qi an attempt to fire Ri will be made, but will fail. (< x y) will then be deleted from Qi. At a later stage, (< y z) is added to Qi and Q. Another attempt will then be made to fire Ri and will succeed since given (< y z) we will find the formula (< x y) in Q, which will then allow Ri to generate (< x z), which will be added to Q and Qi.

Thus, as long as multiformula brute force search rules are written in such a way that given any formula Pj which could make C(...Pj...) true, a search of Q is done for all other P1,...,Pj-1,Pj+1,...,Pp, then the above efficient algorithm is complete, in the sense that it will generate the same formulas as the initial simple algorithm.

We see from the above that the brute force expansion has been made efficient by utilising properties of the expansion rules. The brute force expansion rules were not selected to satisfy these properties.

In fact, these properties was only noticed when the brute force search had been implemented without making use of them and found to be too slow. It is an open question as to whether there are useful expansion rules that do not satisfy these properties.

The efficient form of the algorithms has been tested in PAN, and results in an almost constant rate of production of new formulas. In practice the Qi sets are usually only a very small subset of Q.

6.6.2 Memory Requirements

The brute force search will generally produce a large number of formulas from any given execution state. Section 6.5 showed that most of these formulas will not be used in analysis i.e. their only function was to allow other, more useful formulas to be generated. The only formulas used in analysis are those which:

- include an unanalysed variable and do not contain any items from sequences

or

- are of the form

  $$\forall i\ (1 \le i \le \text{size}(S) \rightarrow \text{property}(\text{item-in-sequence}(i, S) = Q(i))$$

  and do contain any items from sequences or variables or updated properties of objects or sequences.

Thus, once the brute force search is complete for some execution state, we can drop all formulas which do not satisfy either of the above conditions. If the number of execution states to analyse is large, this can result in significant memory savings.

Chapter 7
Interpretation and Simplification

7.1 INTRODUCTION

When symbolic execution has finished, the program has effectively been analysed. However, the results of this analysis are contained in the execution states that are associated with the stop statements in the program. It is the task of the interpretation process to interpret these results into a human readable form. Interpretation is in fact very simple since all the real work has been done by previous processes. The only complication in this process is that, for human readability it is necessary that the expressions produced by the interpretation process are in a suitably simplified form. For example, it may be correct to state

```
if

    color(OBJECT-1) = red ∨ ¬ color(OBJECT-1) = red

then

    v = weight(OBJECT-1) - weight(OBJECT-1)
```

but this is hardly a satisfactory way of saying that v always has the value zero at program end. This chapter first describes the information from the execution states which needs to be presented and then describes how this information is simplified for readability.

7.2 EXECUTION STATE INTERPRETATION

Each execution state to reach program end represents a formula

$$P \rightarrow Q$$

where P is the path condition of the execution state and Q is the effects. The effects consist of the value of variables and the updated properties of objects and object sequences. Both the path condition and the effects may refer to objects, sources or sequences. The definition of these items needs to be included in order to make the specifications complete.

The form of the interpretation is as follows:

```
    if

          path condition

    then

          variable-id-1 = variable-value-1
            .  .    . .    .  .  .   .
            .  .    . .    .  .  .   .
          variable-id-n = variable-value-n

          object-property-1(object-1) = object-property-value-1
            .  .  .  .    .  .  .  .    .  .  .  .   .   .  .
            .  .  .  .    .  .  .  .    .  .  .  .   .   .  .
          object-property-m(object-m) = object-property-value-m

          ∀i (1 ≤ i ≤ size(sequence-1) →
           sequence-property-1(item-in-sequence(i, sequence-1)) =
           sequence-property-value-1(i))
            .  .  .  .    .  .  .  .    .  .  .  .   .   .  .
            .  .  .  .    .  .  .  .    .  .  .  .   .   .  .
          ∀i (1 ≤ i ≤ size(sequence-p) →
           sequence-property-p(item-in-sequence(i, sequence-p)) =
           sequence-property-value-p(i))

    where
```

```
item-id-1 is defined as def-1
.   .  .  .   .   .    .  .  .
.   .  .  .   .   .    .  .  .
item-id-q is defined as def-q
```

The variable-id's and variable-value's are obtained from the variables data. For programs in the robot domain, the robot status can also be included as special variables if required.

The updated properties are obtained from the object and object sequences data.

The item-id's consist of all items referred to elsewhere in the execution state interpretation. Their definitions are retrieved from global data.

The above interpretation is produced for each execution state associated with a stop statement. The disjunction of all such interpretations constitutes the program specification produced by PAN.

Note that in the above interpretation the path condition will be expressed in terms of the state of the world before the program starts. To see that this is so, recall that path conditions are created when condition statements in the program are executed. Any such condition statement tests the value of a variable or object property. The value that a variable or object property can take is determined by the action statements. These allow variables and object properties to be given values that are a function of variables, object properties and constants (literals or numbers occurring in the program). Thus an inductive argumant can be used to show that any variable or object property has a value which is a function of the initial values of object properties and variables. Since these values are used to create path conditions, path conditions will be expressed in terms of the initial values of variables and object properties.

7.3 EXPRESSION SIMPLIFICATION

The description of expression simplification has been included in this chapter as it is only in the interpretation process that unsimplified expressions become visible. In fact, PAN simplifies expressions whenever they are created or updated, using the process described in this section.

The role which the expression simplification process plays in the PAN system is somewhat similar to the theorem prover: expression simplification involves issues independent of those being explored in PAN and this process could have been supplied by an external system. As such a system was not readily available, a simple simplification system has been developed, sufficient to cope with PAN expressions so far encountered.

The input to this process is an expression, and the output is a simplified form of the expression. The simplification process works by applying a set of simplification rules, of the form

input form $\rightarrow$ output form

specifying that any expression matching the input form should be rewritten as the output form. The input expression is recursively dissected until the simplest subexpressions are reached. These are then simplified by applying all simplification rules whose left hand sides match the expression. The original expression is then reassembled, with simplification being performed as each subexpression is rebuilt.

For example, suppose the input expression is

((a - 0) + (2 + 1)) - (b * 0))

then dissection would result in the simplest subexpressions of:

(a - 0), (2 + 1), (b * 0)

which would be simplified to

    a, 3, 0.

Reassembly would then yield

    ((a + 3) - 0)

which would be simplified to (a + 3).

A problem arises with this scheme when a rule creates a new inner subexpression. In this case this inner subexpression may itself need to be simplified, and PAN solves this by recursively invoking expression simplification for this subexpression.

For example, one of the simplification rules used by PAN is

    (x - (w - y) - z) -> (x + y) - (w + z).

If this is applied to

    (a - (3 - b) - 2)

we get

    (a + b) - (3 + 2).

However, (3 + 2) itself needs to be simplified to 5, so (3 + 2) will be passed to expression simplification from within this rule. Use of this method ensures that a simplification rule never returns an expression with an unsimplified subexpression.

7.4 SIMPLIFICATION RULES

PAN's simplification process uses the following rules.

7.4.1 Logical Rules

If x y z are any expressions,

```
(¬ F)           -> T
(¬ T)           -> F
(x ∧ T ∧ y)      -> (x ∧ y)
(x ∧ F ∧ y)      -> F
(x ∧ y ∧ (¬ x))    -> F
((x ∨ y) ∧ (x ∨ z))    -> (x ∨ (y ∧ z))
(x ∧ y ∧ z)      -> (x  ∧  z)  if  x  →  y  can  be  proved  by  theorem
prover
(x ∨ y ∨ (¬ x))    -> T
((x ∧ y) ∨ (x ∧ z))    -> (x ∧ (y ∨ z))
(z ∨ (x ∧ y) ∨ ((¬ x) ∧ y))  -> (z ∨ y)
```

7.4.2 Arithmetic Rules

If w x y z are any expressions and n m are numbers,

```
(n + x + m)   -> (x + value(n + m))
(n * x * m)   -> (x * value(n * m))
(n - m)      -> value(n - m)
(n / m)      -> value (n / m)
(x + (w + y) + z) -> (x + w + y + z)
(x * (w * y) * z) -> (x * w * y * z)
(x + (w - y) + z) -> ((x + w + z) - y)
(x - (w - y) - z) -> ((x + y) - (w + z)) if x is not of the form
(a - b)
((x - y) - z)   -> (x - (y + z))
(x / (w / y) / z) -> ((x * y) / (w * z)) if x is not of the form
(a / b)
((x / y) / z)   -> (x / (y * z))
(x + 0)      -> x
(x - 0)      -> x
(x * 1)      -> x
(x * 0)      -> 0
```

Note that where the expression remains meaningful, terms can be omitted - e.g. the rule for (n + x + m) is intended to include (n + m). Also, the order of the operands is not significant if the operator is commutative - e.g. the rule for (n + x + m) is intended to include (n + m + x).

7.4.3 Comparison Rules

If w y y z are any expressions and n m numbers,

(n > m)	->	value(n > m)
(n < m)	->	value(n < m))
(n = m)	->	value(n = m)
(n ≥ m)	->	value(n ≥ m)
(n ≤ m)	->	value(n ≤ m)
(high-values < x)	->	F
(high-values ≥ x)	->	T
(x > (x + n))	->	F
(x < (x + n))	->	T

7.4.4 Minimum Rules

If x y z are any expressions, n is an integer and P a formula,

minimum({j: (j = x)})	-> x
minimum({j: ((n + j) > x) ∧ integer(j)})	-> (x - (n - 1))
P(minimum({j: P(j)})	-> T

7.5 COMMENTS ON SIMPLIFICATION RULES

The simplicity of PAN's simplification process necessitates simplification rules with stronger preconditions than would be required in a more sophisticated system. The fact that any eligible rule will be used to modify the expression means that the rules must immediately produce a simpler form. This can be compared with a system which allowed backtracking in which the rules could be tried to see if they eventually resulted in a simpler expression.

For example, the effect of the logical rule

$$(z \vee (x \wedge y) \vee ((\neg x) \wedge y)) \text{ -> } (z \vee y)$$

could then be achieved by including the simpler rules

$$(z \vee x \vee y) \text{ -> } (z \vee (x \vee y))$$
$$(x \wedge y) \rightarrow (y \wedge x).$$

Then given

$$(a \vee (b \wedge c) \vee ((\neg b) \wedge c))$$

we could generate

$$(a \vee ((c \wedge b) \vee (c \wedge (\neg b)))).$$

Then, applying the existing rule

$$((x \wedge y) \vee (x \wedge z)) \text{ -> } (x \wedge (y \vee z))$$

to the subexpression

$$((c \wedge b) \vee (c \wedge (\neg b)))$$

would produce

$$(c \wedge (b \vee (\neg b))).$$

The other existing rules would then generate c giving $(a \vee c)$ for the full expression which is equivalent to applying PAN's rule.

The first step in this process is to apply

$$(z \vee x \vee y) \text{ -> } (z \vee (x \vee y))$$

which results in a more complex expression. Since at this point it is not obvious that any benefit will result from applying this rule, we do not allow it in the simple, non backtracking system used by PAN.

Inclusion of the minimum rule

```
    P(minimum({j: P(j)}) -> T
```

while obviously true may require some justification. Consider the example of a program which includes a loop which reads objects from a sequential source, SOURCE-1, and exits when a red object is found. The path condition will be

```
    P(k) = color(sequential-object-in-source(k+1, SOURCE-1)) = red.
```

The exit process will use this to generate a value for k_{exit} of

$$
\begin{aligned}
k_{exit} &= \text{minimum}(\{j: \text{color}(\text{sequential-object-in-source}(j+1, \\
&\qquad \text{SOURCE-1})) = \text{red}\}) \\
&= \text{minimum}(\{j: P(j)\})
\end{aligned}
$$

This value for k_{exit} will then be substituted wherever k occurs in the execution state, including the path condition P(k), to produce P(minimum({j: P(j)}) i.e. T. Since the path condition is used in the interpretation process, such a simplification is required.

Chapter 8

Conclusion

This thesis has described a program analysis system, PAN, based on symbolic execution and generalization. The primary contribution of this thesis, however, is not to describe a particular program analysis system, but to demonstrate that the combination of symbolic execution and generalization can produce a powerful system for analysing loop programs.

Such a system requires the generalization to produce an execution state representing the effect of some unknown number of iterations from the execution states produced during the first few iterations. The secondary contribution of this thesis is to propose one method of performing such a generalization. This method generalizes by using sequences generated from those items in the execution states with similar definitions. It should be noted that performing the generalization by other means (either without using sequences or using a different sequence generation method) would not detract from the central concept of using generalization with symbolic execution.

The remainder of this chapter makes this discussion more concrete by relating these concepts to PAN. It describes the original contributions of the PAN system, the strengths and weaknesses of such a system and identifies aspects of the system which could usefully be extended.

8.1 WHAT PAN DOES

PAN accepts a program to be analysed as input, and produces a specification as output. No additional information, such as intended specifications or program cliches, is assumed. Thus PAN does not address the tasks of checking that a program satisfies an intended specification or describing a program in terms of known cliches.

8.2 ORIGINAL CONTRIBUTIONS OF THE PAN SYSTEM.

8.2.1 Identification of a New Program Class

PAN programs are described using the standard terminology of directed graphs. Directed graphs can represent programs with unrestricted iterative and conditional constructs. PAN accepts for analysis a subset of these programs - those satisfying the restrictions identified in Chapter 2. This subset represents a middle ground between completely unconstrained programs and those only allowing iterative and conditional constructs such as while, if ... then etc. Defining and establishing the properties of this class of programs has occupied a significant part of this thesis.

PAN's success in analysing this class of programs may indicate that it would be worthwhile trying other program analysis techniques on the same set of programs.

8.2.2 Loop Generalization

Traditionally, the major weakness of symbolic execution as a program analysis technique has been its inability to handle loops. The principal contribution of the PAN system is to demonstrate that this problem can be addressed by using a technique based on generalization. PAN generalizes the effect of a few loop iterations to determine the effect of an arbitrary number of iterations.

8.2.3 Two Level Generalization

Performing loop analysis using generalization requires the identification of generalization rules that predict the effect of an indefinite number of loop iterations from the few iterations actually performed. PAN's first task in the generalization process is to generate sequences. PAN does this using structured parameterized rules referred to as models, following the terminology of Dietterich and Michalski[1985]. Models are also used as a first method of expressing the effect of the loop in terms of sequences.

By examining numerous programs from the two domains under consideration, the following conclusions were reached on the effectiveness of models for loop analysis:

- most of the effects produced by loops can be predicted from a small number of iterations using a few relatively simple generalization models

- a few loops have unusual loop effects. A generalization model to handle each such loop would produce a cumbersome system which would still be unlikely to cope with an unusual loop not previously encountered.

PAN's design reflects these findings. PAN has two generalization strategies - the first using a small number of loop generalization models, and the second using a more general but computationally expensive technique. Simple programs can be analysed quickly using the loop generalization models.

Any unusual loop which cannot be handled by these rules is analysed by a method referred to as the brute force search. This technique uses a combination of simple rules and extensive search to greatly extend the program effects which can be generalized. Compared to using models, the brute force search has the significant advantage of allowing PAN to analyse loop effects using arbitrary combinations of known functions. We saw the example in section 1.4 in which PAN analyses a program whose loop effect is finding the minimum of a sequence of values. The minimum function is, in fact, unknown to the loop generalizer, but the effect was successfully described in terms of more primitive functions.

8.2.4 Separate Analysis of Exit Conditions

PAN also introduces a new method of analysing loop exit conditions. This is not included in the loop generalization process. Once the effect of an arbitrary number of loop iterations has been determined, PAN continues symbolic execution until the loop exit(s) are reached.

At that point the loop exit conditions are analysed to determine the number of loop iterations actually performed.

This method enables PAN to clearly identify the conditions required to reach any given loop exit. Thus PAN is relatively insensitive to the complexity of loop exit conditions or the number of loop exits. This contrasts with more traditional methods such as in Cheatham, Holloway and Townley[1979], in which loop exit conditions are determined simultaneously with determining the effect of the loop. The problem of whether these methods can cope with multiple loop exits does not seem to have been addressed.

8.2.5 Execution State Representation

Development of PAN required an execution state representation suitable for both symbolic execution, loop generalization and exit processing. PAN's execution states, expressed in terms of objects, sources, sequences and variables have shown that a domain independent representation is possible. Thus the essential domain independent features of sources, objects and sequences have been identified and only these are recorded in the execution states. Also, the new concept of statement condition has been introduced and its value to the loop generalization process demonstrated.

8.2.6 Generalization Verification

PAN demonstrates that a program analyser based on symbolic execution can easily verify loop generalization by executing another iteration of the loop and comparing the results (in the execution states) with those predicted by the loop generalization. Of course, the hard part is to actually perform the loop generalization. Thus, although PAN's loop generalization is inductive and therefore not guaranteed correct, the generalization can be checked to ensure correctness, though not completeness.

8.2.7 Theorem Proving

PAN also addresses a less significant problem with symbolic execution - failure to recognize conditional statements as provably true or false can lead to incorrect analysis. It has been argued in this thesis that it is unacceptable to ignore this fact. PAN addresses the problem by invoking a theorem prover as each conditional statement is symbolically executed.

8.3 EVALUATING PAN'S PERFORMANCE

PAN was constructed to test the effectiveness of generalization as a program analysis technique rather than as a practical program analysis tool. Nevertheless, the significance of the PAN system may be clarified by contrasting the strengths and weaknesses of PAN as a program analysis system.

8.3.1 Strengths of the PAN System.

Range of Programs Analysed

PAN has demonstrated the ability to analyse programs which, the author believes, could not be analysed by any other existing program analysis system without modification or augmenting its libraries (a cliche driven system can always analyse a program if that program is added to the cliche library!). PAN can analyse programs that include the following features:

- loops may contain variable and object property updates which are both conditional and interdependent
- unstructured branching using 'go to's
- loops may contain inner loops
- loops may have multiple exits from any position in the loop.

This thesis has not attempted to determine precisely the set of programs which PAN can analyse. The program restrictions in Chapter 3 and the theorems and discussions in Chapter 4 do allow a subset of PAN analysable programs to be specified as those programs that

- are specified in terms of directed graphs and satisfy the restrictions in Chapter 2

- have effects satisfying the models in Chapter 4.

However this is not satisfactory for two reasons. Firstly, we do not know how to describe those programs with effects satisfying the models in Chapter 4 in terms of restrictions on directed graphs. Secondly it ignores the programs analysable by the brute force search method. A fuller investigation of this subject is outside the scope of this thesis.

Code Independence

Symbolic execution mimics real execution by 'executing' each program statement to determine its effect on the execution state. Once execution of a statement is completed, further analysis deals exclusively with these effects (as recorded in execution states) rather than the program statements themselves. Thus alternative ways of coding which produce the same effect will lead to identical program analysis. For a trivial example, reversing the order of two program statements may have no effect on program execution and if this is the case, no change will be caused in a PAN analysis. This contrasts with analysis systems which directly examine the program code. In these systems alternate coding may cause considerable difficulty in program analysis.

Multiple Domains

PAN was originally developed for the robot domain. The generality of this method of program analysis has been demonstrated by the ease with which PAN was extended to include the dp domain. New domains can be added by simply adding the ability to symbolically execute any new statements required. This is possible because PAN's execution states are domain independent. Once PAN has executed a domain dependant statement by creating a new execution state, all further processes that use that execution state are domain independent.

8.3.2 Weaknesses of the PAN System

Sequence Restrictions

PAN uses sequences as a fundamental part of its generalization process. These sequences must contain distinct items. Thus PAN is limited to programs in which each loop iteration processes a new item. This excludes programs containing loops which only manipulate the value of a fixed set of variables. Thus PAN is not able to use the values of the variables in each iteration as a sequence. PAN is able to analyse loops in which each iteration processes a distinct element of a variable array.

Failure to Use Previous Analysis

We have said that PAN does not address the task of identifying cliches. In fact, PAN does not even recognise the same code in a single program analysis. This leads to code being reanalysed. For example, when analysing programs with nested loops, any inner loops will be executed and generalized several times. To rectify this problem PAN would have to represent the effect of an inner loop as if it were a single statement. No serious consideration has been given to extending PAN in this way.

Difficulty in Extending Generalization Rules

If a PAN analysis fails, then either new models may need to be added or the brute force search rules extended. PAN has been designed to make this an easy task, but only for a programmer i.e. no "user" interface exists for adding models or rules.

The significance of this weakness depends on the view that one takes of the PAN system. Viewed as a test bed for investigating loop generalization as a program analysis technique, this is not an important weakness. However, if PAN were to be seriously considered as part of an automated programming system, then this weakness is one which would require correction.

8.4 EXTENSIONS TO PAN

8.4.1 Adding a Learning Component

We have discussed in section 9.4.2 above the need to easily be able to add new models and brute force search rules. A more challenging extension would be to allow relationships identified by the brute force search to be automatically added to the generalization models. This would enable PAN to learn models by itself and greatly increase the speed of analysing programs which satisfy the new model.

8.4.2 Production of Preconditions

PAN does not currently analyse the preconditions necessary for a program to operate. To extend PAN to generate preconditions would firstly require adding the preconditions for each type of statement to PAN's existing knowledge of the statement. As each statement is executed, preconditions would be stored as a new data type on PAN's execution states. At loop generalization time, these preconditions would have to be generalized so as to be expressed in terms available on the generalized execution state. The usefulness of this extension is questionable in the domains which have been investigated. For example, as discussed in section 1.6.2, the preconditions for robot movement cannot be determined from the program statements, since the size of the robot is unknown.

8.4.3 Additional Loop Generalizations

As a result of using theorem proving to prove conditional statements true or false, PAN's loop generalization may be incomplete. For example, a loop which contains the test 'if counter > 100' will always be provably false in the first few iterations, if the counter is initialized to zero and incremented on each iteration. In this case some of the loop has not been executed, and will not be included in loop generalization. This problem can be 'solved' by not performing theorem proving, but only at the cost of allowing the possibility of inconsistent execution states. A better solution would be provided by letting the generalized execution state produced by the first loop generalization initiate another set of loop iterations, producing more execution states for generalization. These would then be used to produce a second generalized execution state, and this process would continue until loop generalization produced no new information (i.e. until a fixed point is found). Such an extension would not require any new concepts but would require changes to both the loop generalizer and the scheduling components of PAN.

8.4.4 More Flexible Sequence Generation

In Chapter 4 we discussed the problems caused by only applying sequence generation models to items created by statements having the same statement condition. In particular, we discussed the fact that PAN cannot currently analyse the program shown in figure 8-1 because it cannot create sequences of items from file B. In Chapter 4 a solution to this problem was discussed that would allow PAN to form sequences from items created by statements having statements conditions of the form $P \wedge Q1, \ldots P \wedge Qn$, where $Q1 \vee \ldots Qn = T$. This would enable PAN to analyse programs like the one in figure 8-1 and would be a relatively trivial extension to PAN.

8.4.5 Reduction of the Conditional Branch Restrictions

The restrictions on conditional branching introduced in section 2.7.3 ensure that statement conditions used to create subsequences on some set of concurrent sequences $S_1,\ldots,S_n$ are only expressed in terms of items in $S_1,\ldots,S_n$. As a counter example, consider the program in figure 8.2. In this case the statement condition to reach the merge statement is

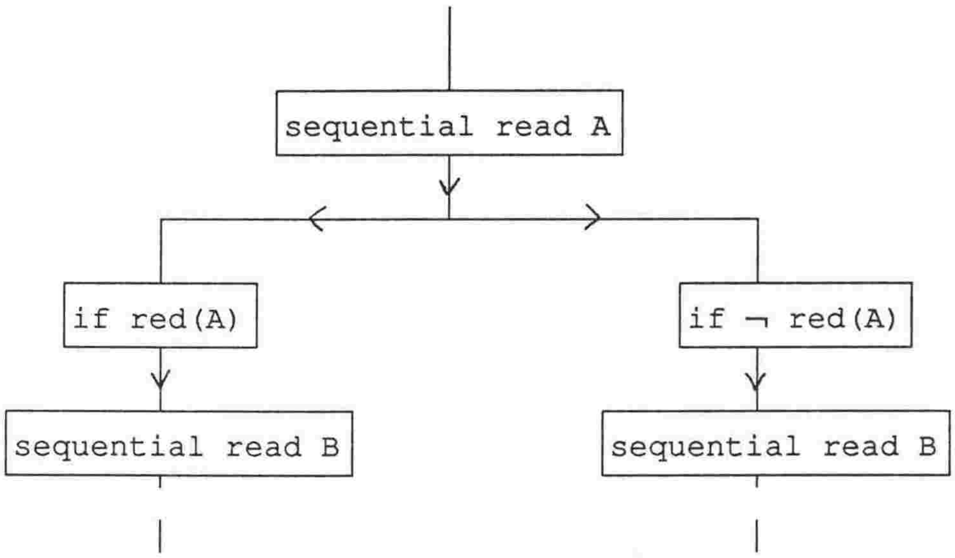


Figure 8-1 Program not Analysable by PAN

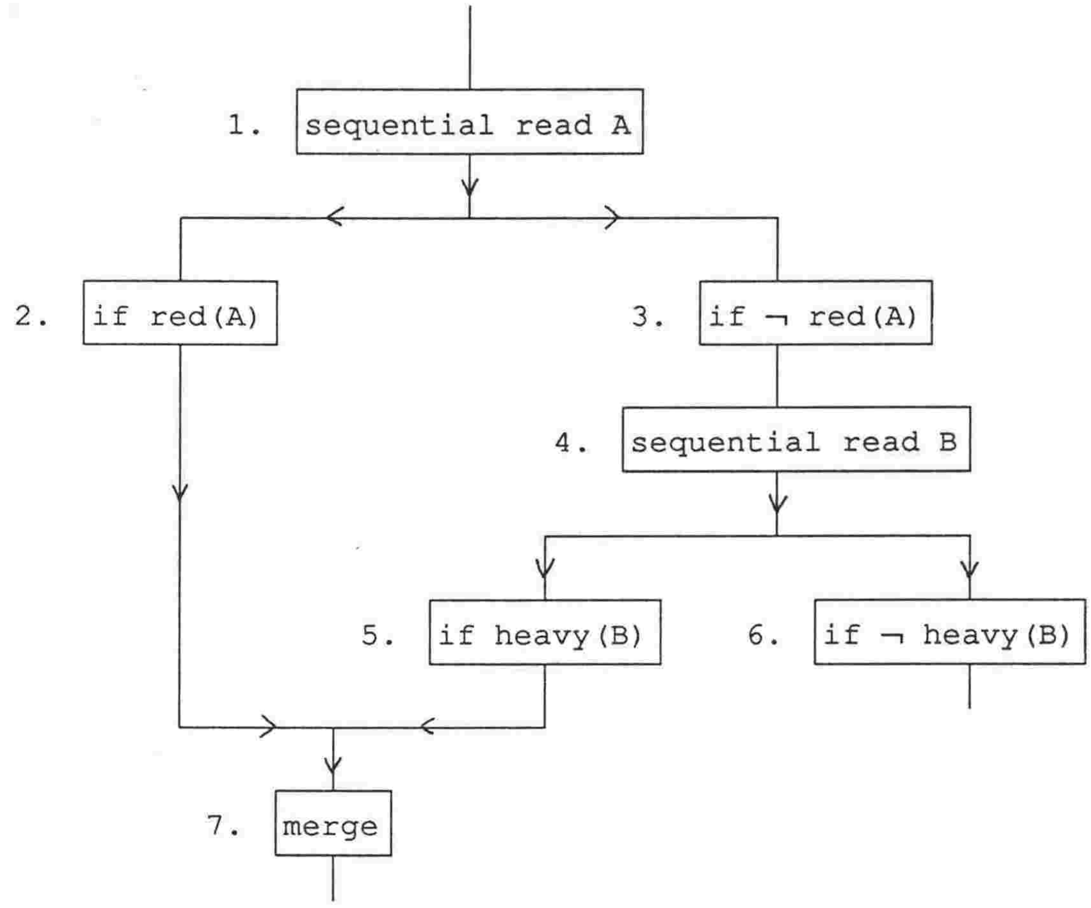


Figure 8-2 Program which Violates the Conditional Branch Restrictions

$(\neg$ red(A) $\wedge$ heavy(B)) $\vee$ red(A)

which simplifies to

heavy(B) $\vee$ red(A).

Since this condition is minimally stronger than the condition to reach statement 1, the subsequence generation model of section 4.3.2 will use it to create a subsequence from the sequence, S, of items from file A, namely

```
(sequence i = 1 to k sequential-object-in-source(i, file(A)))
```

But since this condition contains items not in S or any sequences concurrent with S the subsequence generation model will fail. The conditional branch restrictions are sufficient to avoid this problem, but not necessary. PAN could be improved by either finding weaker conditions which are still sufficient to avoid the problem or by making the subsequence generation model more selective in choosing conditions.

Appendix A
Further Examples

A.1 INTRODUCTION

PAN has been tested on approximately thirty programs, which cannot all be described in this thesis. Program fragments and the examples in section 1.4 have been used to illustrate PAN analyses. The intention of this appendix is to describe a PAN analysis of four programs chosen to illustrate key features of the system.

A.2 EXTENDED SEQUENCE EXAMPLE

The first example program is illustrated in figure A-1. To analyse this program PAN needs to use sequences initially defined in terms of variables, and then to extend such sequences in exit processing.

This program moves the first b+1 objects on the line at angle Φ from pos-a. The ones that are not red are moved to pos-d. The red ones are moved to pos-b if they are in the first a objects from the line and to pos-c otherwise.

PAN Output Specification

```
if

    T

then

    counter = b + 1
    ∀i (1 ≤ i ≤ size(SEQUENCE-4) → position(item-in-sequence(i,
     SEQUENCE-4) = pos-b)
    ∀i (1 ≤ i ≤ size(SEQUENCE-5) → position(item-in-sequence(i,
     SEQUENCE-5) = pos-c)
```

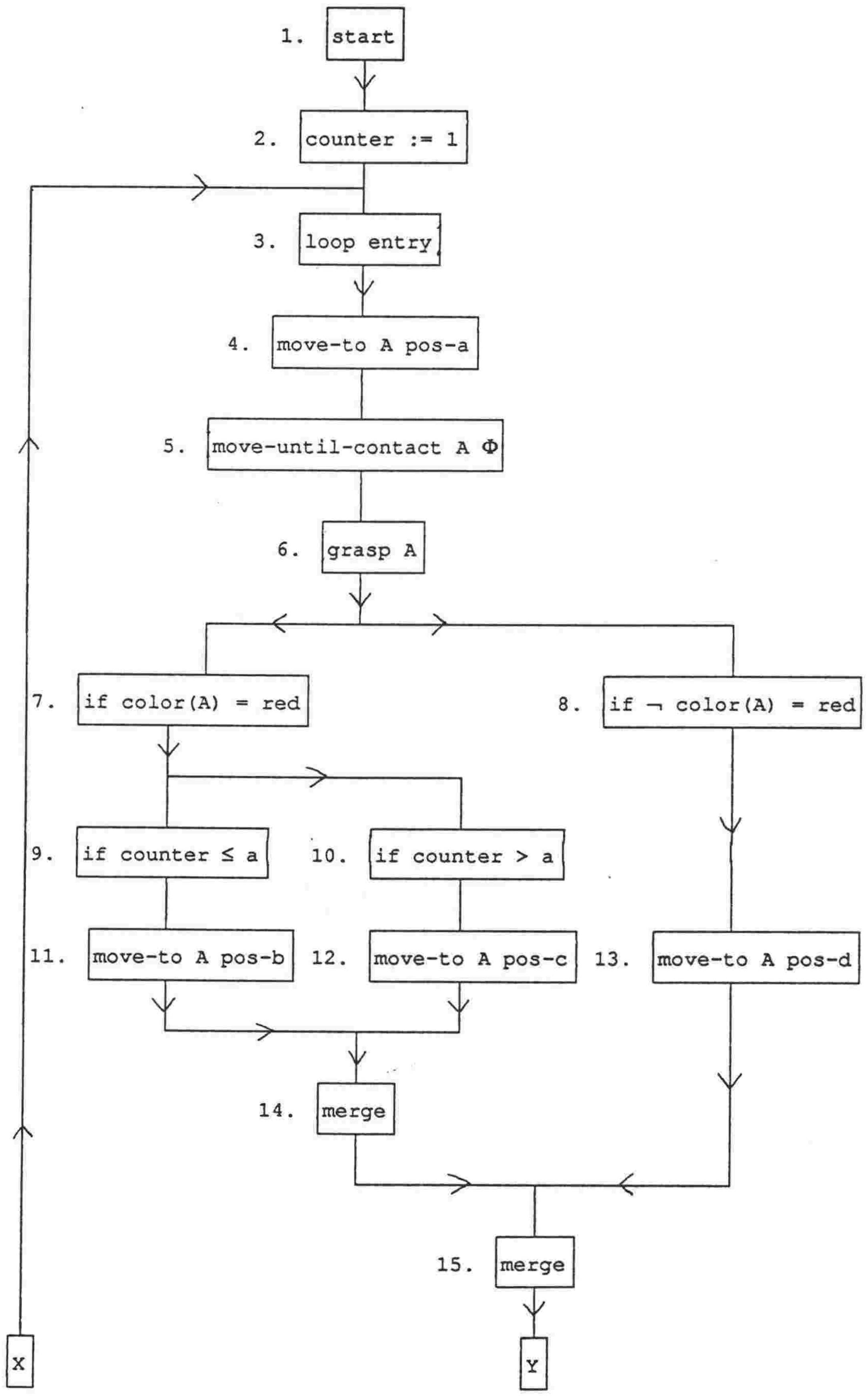


Figure A-1 - Extended Sequences (Part 1)

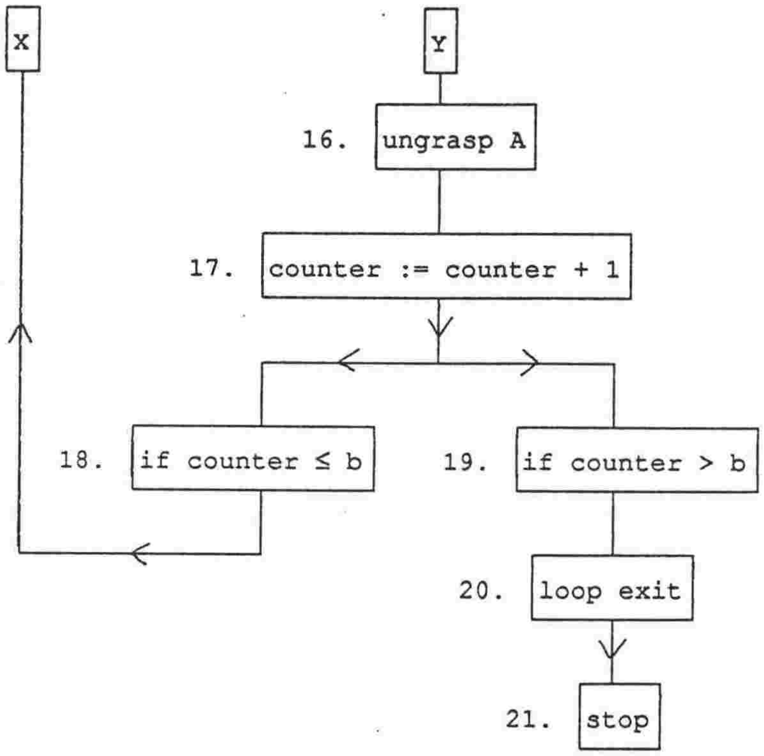


Figure A-1 - Extended Sequences (Part 2).

$\forall i\ (1 \le i \le \text{size(SEQUENCE-3)} \rightarrow \text{position(item-in-sequence(i, SEQUENCE-3)} = \text{pos-d})$

where

SEQUENCE-1 is defined as (sequence i = 1 to b sequential-item-in-source(i, SOURCE-1))

SEQUENCE-2 is defined as (item: item $\in$ SEQUENCE-1 $\wedge$ color(item) = red)

SEQUENCE-3 is defined as (item: item $\in$ SEQUENCE-1 $\wedge$ $\neg$ color(item) = red)

SEQUENCE-4 is defined as (item: item $\in$ SEQUENCE-2 $\wedge$ position-in-sequence(item, SEQUENCE-1) $\le$ a)

SEQUENCE-5 is defined as (item: item $\in$ SEQUENCE-2 $\wedge$ position-in-sequence(item, SEQUENCE-1) > a)

SOURCE-1 is defined as line(pos-a, Φ)

Comments

During loop generalization, PAN generated SEQUENCE-1 using sequence generation model 1, while it generated all other sequences using sequence generation model 2. SEQUENCE-4 and SEQUENCE-5 were initially defined in terms of the variable "counter". However, PAN resolved this variable to position-in-sequence form using model 3 for variables occurring in sequences. PAN also resolved the updated properties of objects in SEQUENCE-3, SEQUENCE-4 and SEQUENCE-5 by the updated properties model, and the value of "counter" at loop entry to size(SEQUENCE-1) + 1 = k+1, using variable at loop entry model 3.

During exit processing PAN extended the sequences. SEQUENCE-1, initially defined as

    (sequence i = 1 to k sequential-object-in-source(i, SOURCE-1))

was extended to

    (sequence i = 1 to k+1 sequential-object-in-source(i, SOURCE-1))

Recall that subsequences of the form $(o: o \in S \wedge P(o))$ are extended by determining whether a new item in S satisfies P. Now in the case of SEQUENCE-4, for example, this involves determining whether the last item in the extended SEQUENCE-2 satisfies

    position-in-sequence(item, SEQUENCE-1) $\leq$ a.

PAN determined the value of position-in-sequence(item, SEQUENCE-1) for the new item in SEQUENCE-1 as size(SEQUENCE-1) = k+1. Thus the new item in SEQUENCE-2 will also be in SEQUENCE-4 if $k+1 \leq a$. For the execution state associated with loop exit that describes the effect of executing the statements on the path passing through statement 9, execution of statement 9 added counter = $k+1 \leq a$ to the path conditions. Thus SEQUENCE-2 will be extended to include the new object. Similarly SEQUENCE-5 and SEQUENCE-3 will be extended in those execution states associated with loop exit that describe the effect of executing statements on paths passing through statements 10 and 8 respectively. The three execution states will now be identical (apart from path conditions). Thus only a single execution state will be allowed to exit. The value of k in this execution state was resolved by simplifying the exit condition to counter = k+2 > b, giving a value of k on exit of b-1. When substituted into the single execution state chosen for exit, counter will have a value of b+1 and SEQUENCE-1 will now be defined as

(sequence i = 1 to b sequential-object-in-source(i, SOURCE-1)).

A.3 KEYED FILE EXAMPLE

The second example is illustrated in figure A-2. It shows PAN's ability to analyse a program updating specified records from a keyed file. Also, several updated properties have been assigned to the correct sequence and the single item from file C has been analysed in the same way as if it were a variable.

This program updates a keyed file of account records in a banking application system. The accounts to update are specified by the account numbers read from sequential file A. The account number is used to retrieve account details from keyed file B. The date-last-processed is updated and if the account balance is not zero, credit or debit interest is increased and date-last-interest is updated. The total of the account balances of all accounts accessed is calculated and written out as a single record to file C.

PAN Output Specification

```
if

   T

then
```

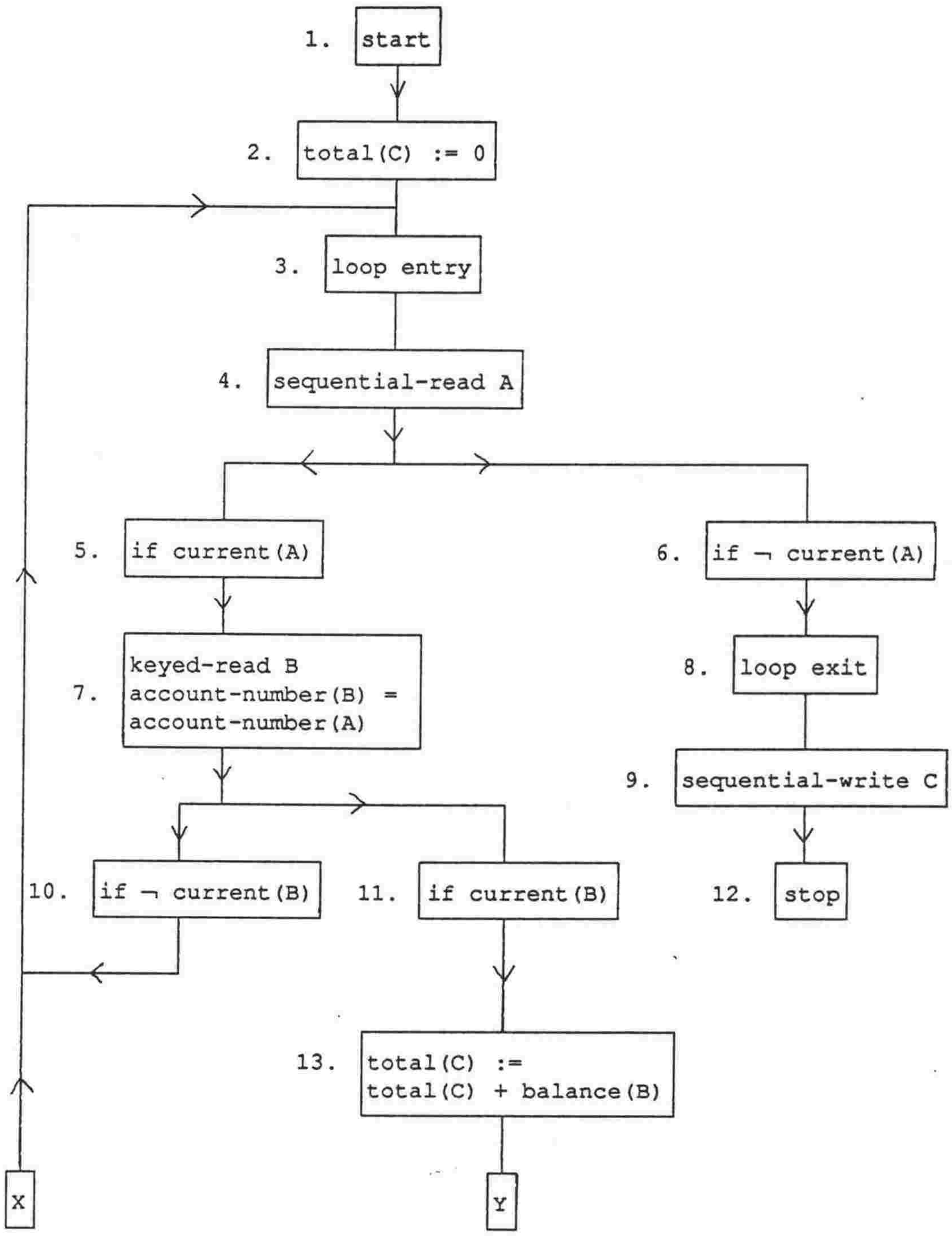


Figure A-2 - Keyed File Example (Part 1)

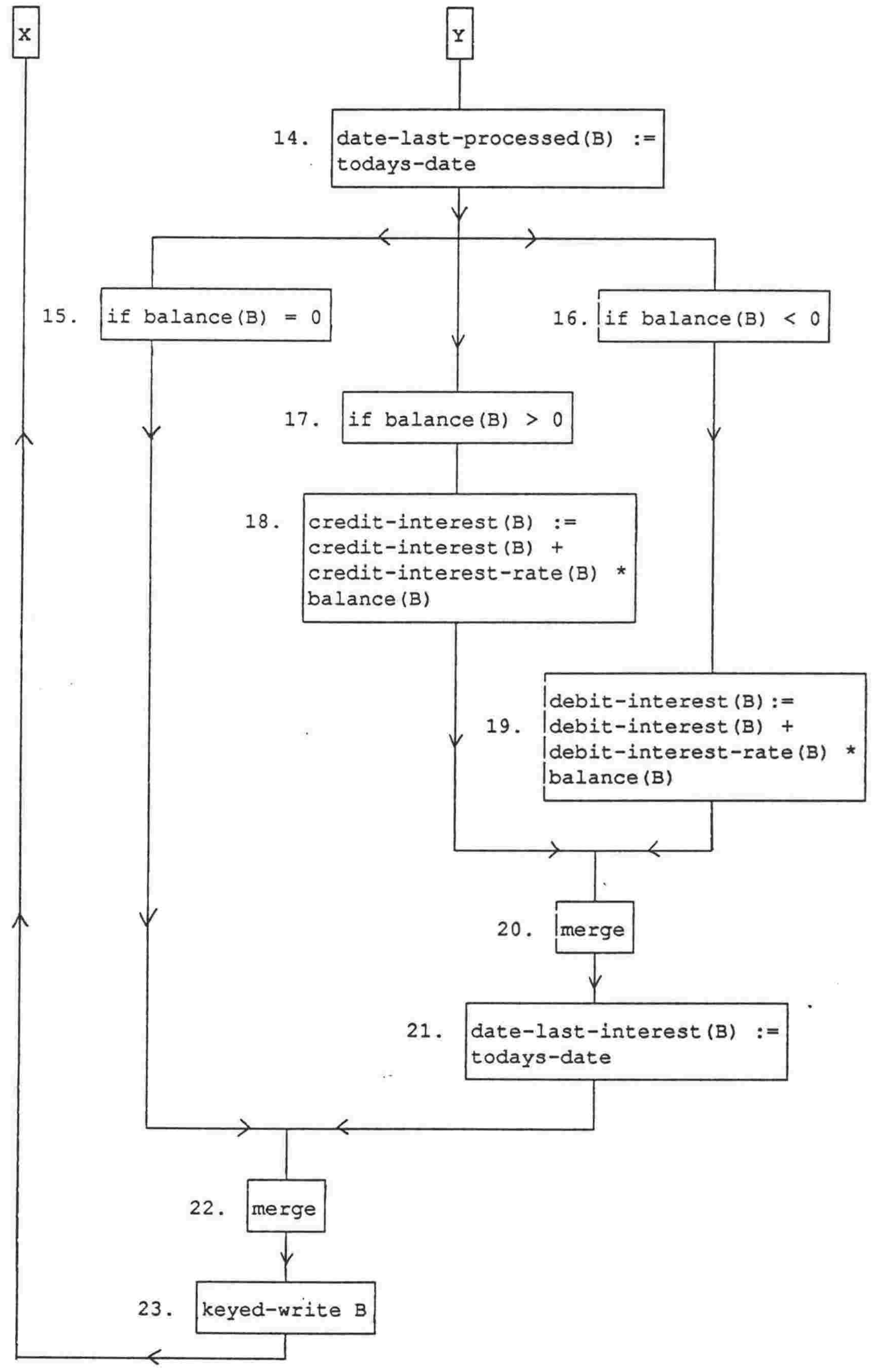

X
Y
14. date-last-processed(B) := todays-date
15. if balance(B) = 0
16. if balance(B) < 0
17. if balance(B) > 0
18. credit-interest(B) := credit-interest(B) + credit-interest-rate(B) * balance(B)
19. debit-interest(B) := debit-interest(B) + debit-interest-rate(B) * balance(B)
20. merge
21. date-last-interest(B) := todays-date
22. merge
23. keyed-write B

Figure A-2 - Keyed File Example (Part 2)

$$\text{total(OBJECT-1)} = \sum_{\text{i-1}}^{\text{size(SEQUENCE-3)}} \text{balance(item-in-sequence(i, SEQUENCE-3))}$$

```
∀i (1 ≤ i ≤ size(SEQUENCE-3) →
 date-last-processed(item-in-sequence(i, SEQUENCE-3) =
 todays-date)
∀i (1 ≤ i ≤ size(SEQUENCE-4) →
 credit-interest(item-in-sequence(i, SEQUENCE-4)) =
 credit-interest(item-in-sequence(i, SEQUENCE-4)) +
 credit-interest-rate(item-in-sequence(i, SEQUENCE-4)) *
 balance(item-in-sequence(i, SEQUENCE-4)))
∀i (1 ≤ i ≤ size(SEQUENCE-5) →
 debit-interest(item-in-sequence(i, SEQUENCE-5)) =
 debit-interest(item-in-sequence(i, SEQUENCE-5)) +
 debit-interest-rate(item-in-sequence(i, SEQUENCE-5)) *
 balance(item-in-sequence(i, SEQUENCE-5)))
∀i (1 ≤ i ≤ size(SEQUENCE-6) →
 date-last-interest(item-in-sequence(i, SEQUENCE-6) =
 todays-date)
```

where

```
OBJECT-1 is defined as sequential-object-in-source(1, SOURCE-3)
SEQUENCE-1 is defined as (sequence i = 1 to size (SOURCE-1)
 sequential-object-in-source(i, SOURCE-1))
SEQUENCE-2 is defined as (item: item ∈ SEQUENCE-1 ∧ ∃object
 (object ∈ SOURCE-2 ∧ account-number(object) = account-number
 (item)))
SEQUENCE-3 is defined as (sequence i = 1 to size(SEQUENCE-2)
 keyed-object-in-source(account-number(object) = account-number
 (item-in-sequence(i, SEQUENCE-1)), SOURCE-2))
SEQUENCE-4 is defined as (item: item ∈ SEQUENCE-3 ∧ balance
 (item) > 0)
SEQUENCE-5 is defined as (item: item ∈ SEQUENCE-3 ∧ balance
 (item) < 0)
SEQUENCE-6 is defined as (item: item ∈ SEQUENCE-3 ∧ (balance
 (item) < 0) ∨ balance(item) > 0)))
```

SOURCE-1 is defined as file(A)

SOURCE-2 is defined as file(B)

SOURCE-3 is defined as file(C)

Comments

In this example, note that the size of both SEQUENCE-1 and SEQUENCE-3 are expressed in terms of the size of another item, SOURCE-1 and SEQUENCE-2 respectively. PAN generated SEQUENCE-3 directly using sequence generation model 1, which can express size either as the iteration count, or the size of a previously generated sequence.

We now describe how the size of SEQUENCE-1 was determined. PAN initially set the size of SEQUENCE-1 to the iteration count k using sequence generation model 1. In loop generalization PAN also set the number of retrievals from SOURCE-1 = file A to k. When symbolic execution continued after loop generalization it attempted another retrieval from SOURCE-1, at statement 4, for a total of k+1 retrievals attempted. To exit from the loop, statement 6 was executed, which had the effect of putting k+1 = (number of retrievals attempted from SOURCE-1) > size(SOURCE-1) in the path conditions. Thus exit processing used k+1 > size(SOURCE-1) to determine the value for k to exit as

minimum({j: j+1 > size(SOURCE-1)}) = size(SOURCE-1).

This will then replace k in the definition of SEQUENCE-1.

During loop generalization PAN also generalized the updated properties of the objects in SEQUENCE-3 and each of its subsequences, SEQUENCE-4, SEQUENCE-5 and SEQUENCE-6. In each case the updated property has been described in terms of the most general sequence. Also the credit and debit interest property update for SEQUENCE-4 and SEQUENCE-5 varies for each item in the sequence.

Correct analysis of date-last-interest, updated in statement 21, required the generation of SEQUENCE-6, containing all items with an updated value for date-last-interest. This, in turn, required that the statement conditions contain balance(object) < 0 v balance(object) > 0, provided by the merge process described in section 3.3.2.

Finally, the single object in file C, which acts as if it were a variable, was correctly analysed by variables model 4 to be the total of the balances in SEQUENCE-3.

A.4 BRUTE FORCE ANALYSIS OF UPDATED PROPERTY

The third example program is illustrated in figure A-3. In this program items in a sequence are updated depending on their position in another sequence. This cannot be handled by the models in Chapter 4 and requires the brute force search.

The program splits a line of different coloured blocks into two lines, one of red blocks and one of non-red blocks. It moves all objects on the line of length l at angle Φ from pos-a. The ith such object is moved to (X1 , Y1+i) if it is red and to (X2 , Y2+i) otherwise.

<u>PAN Output Specification</u>

```
if

    T

then

    counter = size(SOURCE-1)
    ∀i (1 ≤ i ≤ size(SEQUENCE-2) → position(item-in-sequence(i,
     SEQUENCE-2) = (X1 , Y1 +
     position-in-sequence(item-in-sequence(i, SEQUENCE-2),
     SEQUENCE-1))))
```

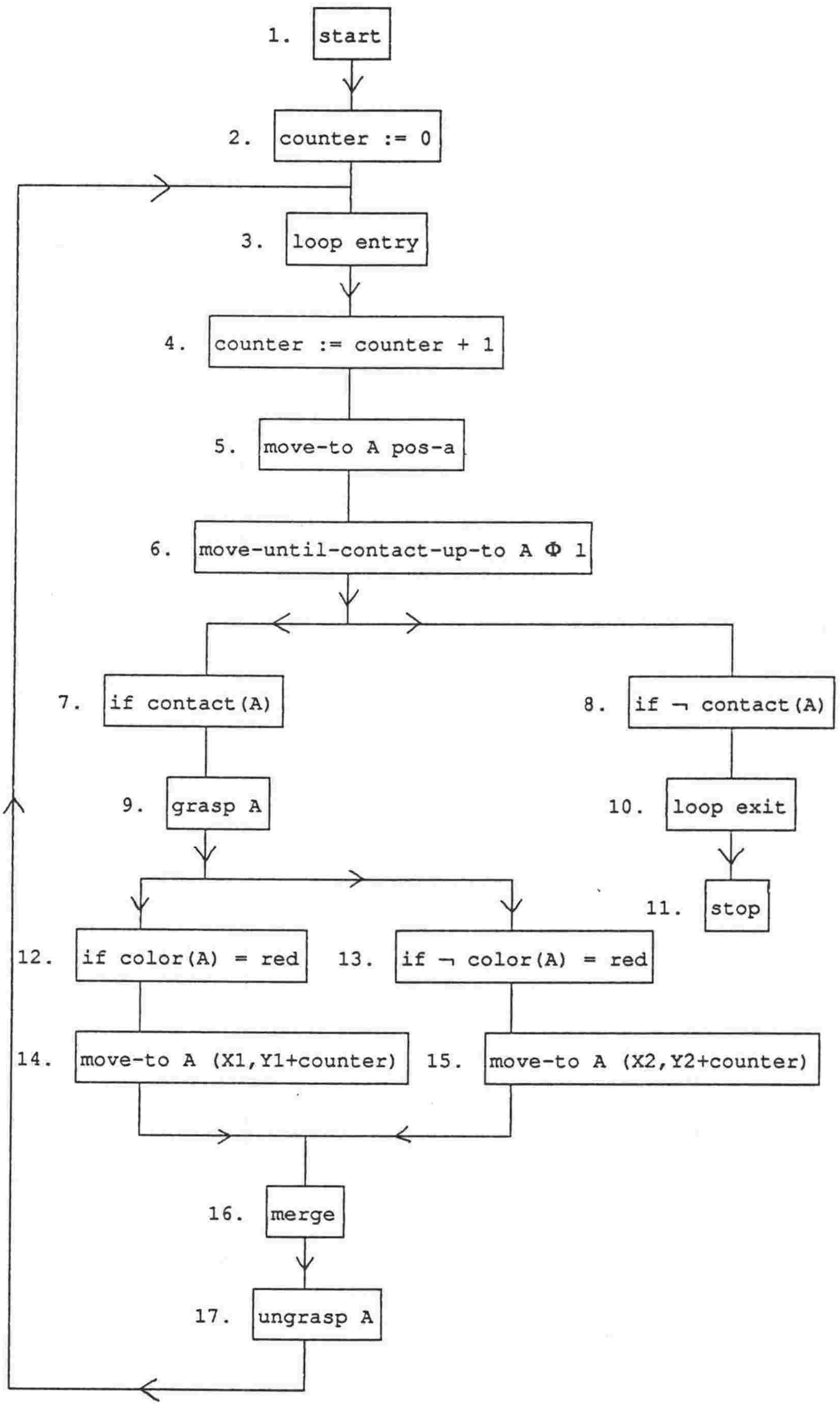


Figure A-3 - Brute Force Search Analysis of Updated Property.

$\forall$i (1 $\leq$ i $\leq$ size(SEQUENCE-3) $\rightarrow$ position(item-in-sequence(i, SEQUENCE-3) = (X2 , Y2 + position-in-sequence(item-in-sequence(i, SEQUENCE-3), SEQUENCE-1))))

where

SEQUENCE-1 is defined as (sequence i = 1 to (size SOURCE-1) (sequential-object-in-source i SOURCE-1))
SEQUENCE-2 is defined as {item: item $\in$ SEQUENCE-1 $\wedge$ (color item) = red}
SEQUENCE-3 is defined as {item: item $\in$ SEQUENCE-1 $\wedge$ $\neg$ (color item) = red}
SOURCE-1 is defined as line(pos-a, Φ, 1).

Comments

This program is deceptive. At first sight it appears as if the model for updated properties should be able to handle the updated position of SEQUENCE-2 and SEQUENCE-3. As stated in section 4.6, this model can handle the case of a sequence with an updated property whose value, for a given item, depends on the position of that item in the sequence. However, in this case the updated property of items in SEQUENCE-2 and SEQUENCE-3 depends on the position of the items in a different sequence, SEQUENCE-1. This is not a common situation, and the brute force search was invoked to complete the analysis.

The initial formulas for the brute force expansion for one execution state included

SEQUENCE-1 = {OBJECT-1 OBJECT-2 OBJECT-3}
SEQUENCE-2 = {OBJECT-1 OBJECT-3}
SEQUENCE-3 = {OBJECT-2}
color(OBJECT-1) = color(OBJECT-3) = red
$\neg$ color(OBJECT-2) = red
position(OBJECT-1) = (X1, Y1 + 1)
position(OBJECT-2) = (X2, Y2 + 2)
position(OBJECT-3) = (X1, Y1 + 3)

From position(OBJECT-1) = (X1, Y1 + 1), the expansion process generated:

```
position(OBJECT-1) = (X1, Y1 + position-in-sequence(OBJECT-1,
                                  SEQUENCE-1)
```

by the Position in Sequence rule, and then

```
position(item-in-sequence(1, SEQUENCE-2)) =
  (X1, Y1 + position-in-sequence(OBJECT-1, SEQUENCE-1)
```

by the Item in Sequence rule, and then

```
position(item-in-sequence(1, SEQUENCE-2)) =
  (X1, Y1 + position-in-sequence(item-in-sequence(1, SEQUENCE-2),
  SEQUENCE-1))
```

by another application of the Item in Sequence rule.

Similarly, from position(OBJECT-3) = (X1, Y1 + 3), the same rules generated

```
position(item-in-sequence(2, SEQUENCE-2)) =
  (X1, Y1 + position-in-sequence(item-in-sequence(2, SEQUENCE-2),
  SEQUENCE-1)).
```

Then, since SEQUENCE-2 = {OBJECT-1 OBJECT-3}, the Property of All Items in a Sequence rule generated

```
∀i (1 ≤ i ≤ size(SEQUENCE-2) → position(item-in-sequence(i,
  SEQUENCE-2) = (X1, Y1 + position-in-sequence(item-in-sequence(2,
  SEQUENCE-2), SEQUENCE-1))).
```

Since this formula was generated in all execution states which have any items in SEQUENCE-2, it was included in the generalized execution state. Similarly for the updated properties of SEQUENCE-3.

8.5 LOOP WITHIN A LOOP

The fourth example program is illustrated in figure A-4. This program has nested loops with multiple exits from the inner loop.

The program moves objects from the a+1 lines at angle Φ from (X0,Y0), (X0+X,Y0+Y),...,(X0+aX,Y0+aY). Objects are moved from each line to pos-a until a red or blue object is found. If this object is red it is moved to pos-b, otherwise to pos-c, then the program proceeds to the next line.

PAN Output Specification

```
if

    color(item-in-sequence(a+1, SEQUENCE-6)) = red

then

    position(A) = pos-b
    grasping(A) = F
    object-contacted(A) = item-in-sequence(a+1, SEQUENCE-6)
    (Note: the above are robot status variables).
    counter = a+1
    ∀i (1 ≤ i ≤ size(SEQUENCE-5) →
     ∀j (1 ≤ j ≤ size(item-in-sequence(i, SEQUENCE-5) →
     position(item-in-sequence(j, item-in-sequence(i, SEQUENCE-5))) =
     pos-a))
    ∀i (1 ≤ i ≤ size(SEQUENCE-7) →
     position(item-in-sequence(i, SEQUENCE-7)) = pos-b)
    ∀i (1 ≤ i ≤ size(SEQUENCE-8) →
     position(item-in-sequence(i, SEQUENCE-8)) = pos-c)

where
```

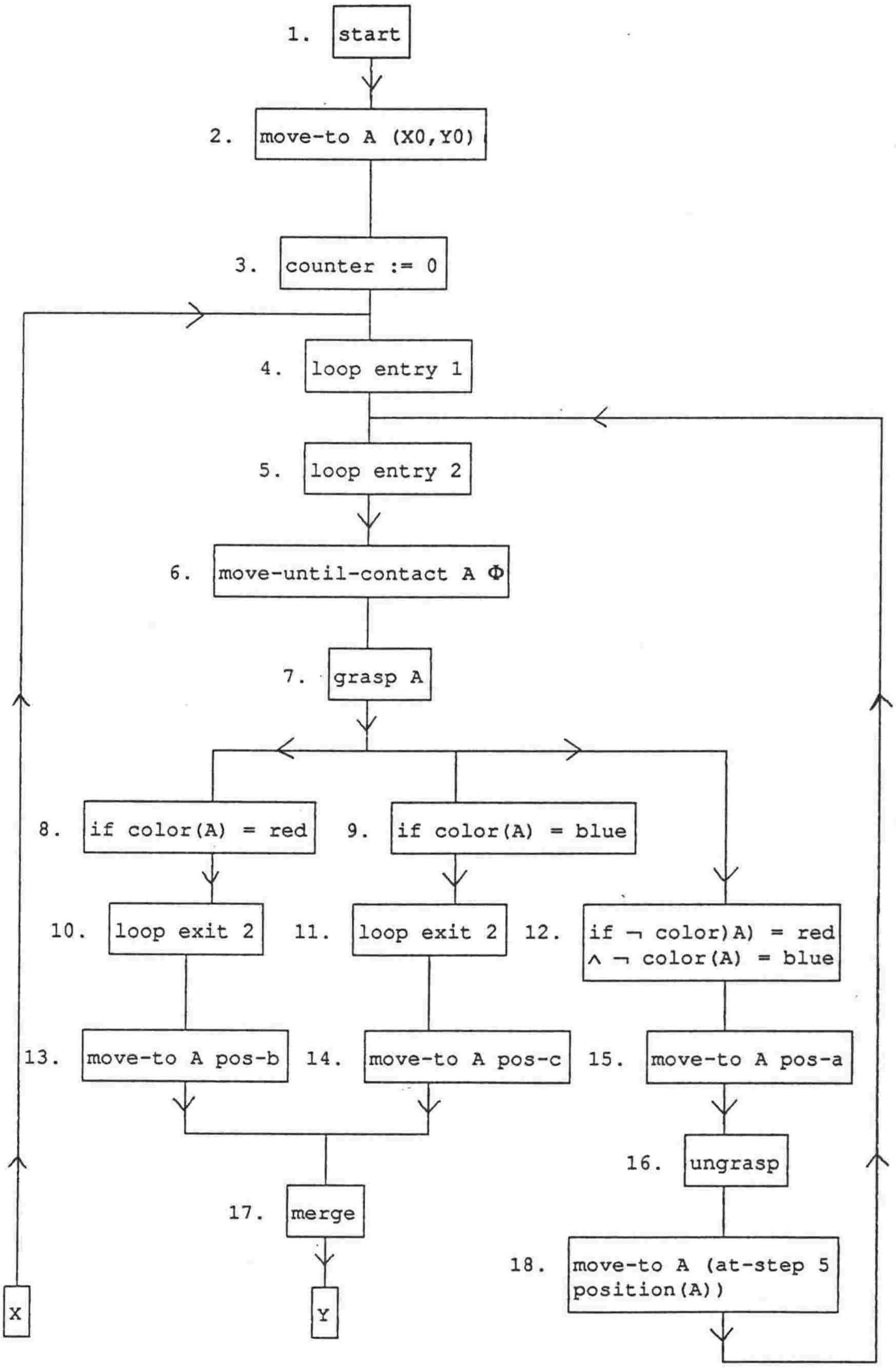


Figure A-4 - Loop within a Loop (Part 1)

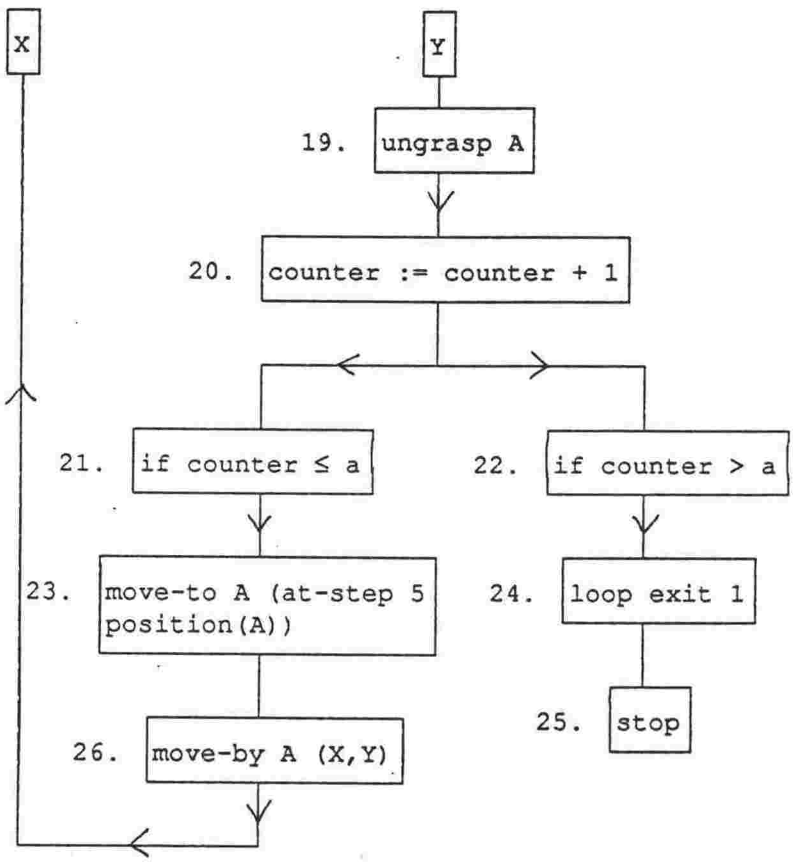


Figure A-4 Loop within a Loop (Part 2)

```
SEQUENCE-4 is defined as
 (sequence i = 1 to a+1 (line(X0 + (i-1)X , Y0 + (i-1)Y), Φ))
SEQUENCE-5 is defined as
 (sequence i = 1 to a+1 (sequence j = 1 to n(i)
  sequential-object-in-source(j,item-in-sequence(i,SEQUENCE-4))))
SEQUENCE-6 is defined as
 (sequence i = 1 to a+1
  sequential-object-in-source(n(i)+1,
   item-in-sequence(i,SEQUENCE-4)))
SEQUENCE-7 is defined as
 (item: item ∈ SEQUENCE-6 ∧ color(item) = red)
SEQUENCE-8 is defined as
 (item: item ∈ SEQUENCE-6 ∧ color(item) = blue)
```

```
    n(i) =
            minimum({j:
             color(sequential-object-in-source(j+1,
               (item-in-sequence(i, SEQUENCE-4)))) = red
             v
             color(sequential-object-in-source(j+1,
               (item-in-sequence(i, SEQUENCE-4)))) = blue}).

if

    color(item-in-sequence(a+1, SEQUENCE-6)) = blue

then

    position(A) = pos-c
    grasping(A) = F
    object-contacted(A) = item-in-sequence(a+1, SEQUENCE-6)
    (Note: the above are robot status variables).
    remainder of this case is as above.
```

Comments

This is the most complex example presented and requires a more extensive explanation. We first deal with the use of 'at-step' in statements 18 and 23. Although not described in PAN's input language in chapter 2, PAN programs are actually allowed to refer to the value of variables or object properties at previous statements. This is a feature of the extended Noddy system of which PAN forms a part. Since this is equivalent to the use of additional variables, (in which these values could have been stored) it did not seem necessary to further complicate the description of the input language by including this feature.

Before PAN generalizes the outer loop, it will be executed some specified number of times. Each execution of the outer loop will include executing, generalizing and exiting from the inner loop. Once PAN has generalized the outer loop, exit from this loop will involve another execution, generalization and exit from the inner loop. PAN scheduling will ensure that these processes are performed in this order.

The first generalization of the inner loop will produce

```
SOURCE-1 = line((X0, Y0), Φ))
SEQUENCE-1 = (sequence i = 1 to k
 sequential-object-in-source(i, SOURCE-1))
∀i (1 ≤ i ≤ size(SEQUENCE-1) →
 position(item-in-sequence(i, SEQUENCE-1)) = pos-a)
counter = 0
number of iteration attempted from SOURCE-1 = k.
```

When symbolic execution is continued after loop generalization, an execution state will be associated with each exit from the inner loop, each having a new object, OBJECT-1, defined as sequential-object-in-source(k+1, SOURCE-1) and exit conditions of color(OBJECT-1) = red and color(OBJECT-1) = blue respectively. Exit processing will attempt to include OBJECT-1 in SEQUENCE-1, but this will fail as OBJECT-1 is not at position pos-a. The value of k on exit is then determined by

```
k_exit =
     minimum({j:
       color(sequential-object-in-source(j+1, SOURCE-1)) = red
       ∨
       color(sequential-object-in-source(j+1, SOURCE-1)) = blue})
```

This value for k will then be substituted wherever it occurs in the execution states. Execution will then continue from these execution states until loop entry is reached, during which process, OBJECT-1 will be moved to either pos-b or pos-c and counter will be incremented to 1.

Thus, after one iteration of the outer loop, the following states will be available for generalization:

Execution State 1:

```
    SOURCE-1 = line((X0, Y0), Φ)
    SEQUENCE-1 = (sequence i = 1 to n1
     sequential-object-in-source(i, SOURCE-1))
    ∀i (1 ≤ i ≤ size(SEQUENCE-1) →
     position(item-in-sequence(i, SEQUENCE-1)) = pos-a)
    OBJECT-1 = sequential-object-in-source(n1+1, SOURCE-1)
    color(OBJECT-1) = red
    position(OBJECT-1) = pos-b
    counter = 1
```

Execution State 2:

```
    SOURCE-1 = line((X0, Y0), Φ)
    SEQUENCE-1 = (sequence i = 1 to n1
     sequential-object-in-source(i, SOURCE-1))
    ∀i (1 ≤ i ≤ size(SEQUENCE-1) →
     position(item-in-sequence(i, SEQUENCE-1)) = pos-a)
    OBJECT-1 = sequential-object-in-source(n1+1, SOURCE-1)
    color(OBJECT-1) = blue
    position(OBJECT-1) = pos-c
    counter = 1
```

where

```
    n1 =
         minimum({j:
          color(sequential-object-in-source(j+1, SOURCE-1)) = red
          ∨
          color(sequential-object-in-source(j+1, SOURCE-1)) = blue})
```

After the second and third iterations of the outer loop have been performed, we will have addition execution states:

Execution State 3:

```
    SOURCE-2 = line((X0 + X, Y0 + Y), Φ)
    SEQUENCE-2 = (sequence i = 1 to n2
     sequential-object-in-source(i, SOURCE-2))
    ∀i (1 ≤ i ≤ size(SEQUENCE-2) →
     position(item-in-sequence(i, SEQUENCE-2)) = pos-a)
    OBJECT-2 = sequential-object-in-source(n2+1, SOURCE-2)
    color(OBJECT-2) = red
    position(OBJECT-2) = pos-b
    counter = 2
```

Execution State 4:

```
    SOURCE-2 = line((X0 + X, Y0 + Y), Φ)
    SEQUENCE-2 = (sequence i = 1 to n2
     sequential-object-in-source(i, SOURCE-2))
    ∀i (1 ≤ i ≤ size(SEQUENCE-2) →
     position(item-in-sequence(i, SEQUENCE-2)) = pos-a)
    OBJECT-2 = sequential-object-in-source(n2+1, SOURCE-2)
    color(OBJECT-2) = blue
    position(OBJECT-2) = pos-c
    counter = 2
```

where

```
    n2 =
         minimum({j:
          color(sequential-object-in-source(j+1, SOURCE-2)) = red
          ∨
          color(sequential-object-in-source(j+1, SOURCE-2)) = blue})
```

and

Execution State 5:

```
    SOURCE-3 = line((X0 + 2*X, Y0 + 2*Y), Φ)
    SEQUENCE-3 = (sequence i = 1 to n3
     sequential-object-in-source(i, SOURCE-3))
    ∀i (1 ≤ i ≤ size(SEQUENCE-3) →
     position(item-in-sequence(i, SEQUENCE-3)) = pos-a)
    OBJECT-3 = sequential-object-in-source(n3+1, SOURCE-3)
    color(OBJECT-3) = red
    position(OBJECT-3) = pos-b
    counter = 3
```

Execution State 6:

```
    SOURCE-3 = line((X0 + 2*X, Y0 + 2*Y), Φ)
    SEQUENCE-3 = (sequence i = 1 to n3
     sequential-object-in-source(i, SOURCE-3))
    ∀i (1 ≤ i ≤ size(SEQUENCE-3) →
     position(item-in-sequence(i, SEQUENCE-3)) = pos-a)
    OBJECT-3 = sequential-object-in-source(n3+1, SOURCE-3)
    color(OBJECT-3) = blue
    position(OBJECT-3) = pos-c
    counter = 3
```

where

```
    n3 =
         (minimum {j:
          color(sequential-object-in-source(j+1, SOURCE-3)) = red
          ∨
          color(sequential-object-in-source(j+1, SOURCE-3)) = blue})
```

PAN will then generalize sources in the outer loop into the sequence

```
    SEQUENCE-4 = (sequence i = 1 to k
     (line (X0 + (i-1)*X, Y0 + (i-1)*Y), Φ)).
```

As specified in the sequence generation algorithm, the definition of SOURCE-1, SOURCE-2 and SOURCE-3 occurring in SEQUENCE-1, SEQUENCE-2, SEQUENCE-3, OBJECT-1, OBJECT-2 and OBJECT-3 will now be replaced by item-in-sequence(1, SEQUENCE-4), item-in-sequence(2, SEQUENCE-4) and item-in-sequence(3, SEQUENCE-4), allowing SEQUENCE-1, SEQUENCE-2 and SEQUENCE-3 to be generalized to

```
SEQUENCE-5 = (sequence i = 1 to k (sequence j = 1 to n(i)
 sequential-object-in-source(j, (item-in-sequence(i,
 SEQUENCE-4)))
```

where

```
n(i) =
      minimum({j:
       color(sequential-object-in-source(j+1,
         (item-in-sequence(i, SEQUENCE-4)))) = red
       ∨
       color(sequential-object-in-source(j+1,
         (item-in-sequence(i, SEQUENCE-4)))) = blue}).
```

OBJECT-1, OBJECT-2 and OBJECT-3 will be generalized to

```
SEQUENCE-6 = (sequence i = 1 to k
 sequential-object-in-source(n(i),
  item-in-sequence(i, SEQUENCE-4)))
SEQUENCE-7 = (item: item ∈ SEQUENCE-6 ∧ color(item) = red)
SEQUENCE-8 = (item: item ∈ SEQUENCE-6 ∧ color(item) = blue)
```

Next the values of counter will be generalized to size(SEQUENCE-4), which is k, by variables at loop entry model 3. Since the items in SEQUENCE-5 (SEQUENCE-1, SEQUENCE-2 and SEQUENCE-3) each have an updated property of

```
∀j (1 ≤ j ≤ size(SEQUENCE-i) →
 position(item-in-sequence(j, SEQUENCE-i)) = pos-a)
```

for i = 1, 2 and 3, the updated properties model will produce an updated property for SEQUENCE-5 of

$$\begin{array}{l}
\forall i\ (1 \le i \le \text{size(SEQUENCE-5)} \rightarrow \\
\quad \forall j\ (1 \le j \le \text{size(item-in-sequence(i, SEQUENCE-5))} \rightarrow \\
\quad \text{position(item-in-sequence(j, item-in-sequence(i, SEQUENCE-5)))} = \\
\qquad \text{pos-a))}
\end{array}$$

The objects in SEQUENCE-6, SEQUENCE-7 and SEQUENCE-8 also have updated properties, which will be generalized to

$$\begin{array}{l}
\forall i\ (1 \le i \le \text{size(SEQUENCE-7)} \rightarrow \\
\quad \text{position(item-in-sequence(i, SEQUENCE-7))} = \text{pos-b)}
\end{array}$$

$$\begin{array}{l}
\forall i\ (1 \le i \le \text{size(SEQUENCE-8)} \rightarrow \\
\quad \text{position(item-in-sequence(i, SEQUENCE-8))} = \text{pos-c)}
\end{array}$$

Finally, the robot position will be generalized to (X0 + k*X, Y0 + k*Y).

Symbolic execution will now continue, beginning with the generalized execution state from the outer loop. To reach the exit of the outer loop, however, it is necessary to reenter the inner loop. Thus, another set of iterations and generalization of the inner loop will be performed. This process will produce a generalized execution state which now contains:

$$\begin{array}{l}
\text{SEQUENCE-4} = \text{(sequence i = 1 to } k_{\text{outer}} \\
\quad \text{line((X0 + (i-1)*X, Y0 + (i-1)*Y), } \Phi\text{)).} \\
\text{SEQUENCE-5} = \text{(sequence i = 1 to } k_{\text{outer}} \\
\quad \text{(sequence j = 1 to n(i)} \\
\qquad \text{sequential-object-in-source(j, (item-in-sequence(i,} \\
\qquad\quad \text{SEQUENCE-4)))} \\
\text{SEQUENCE-6} = \text{(sequence i = 1 to } k_{\text{outer}} \\
\quad \text{sequential-object-in-source(n(i),} \\
\qquad \text{item-in-sequence(i, SEQUENCE-4)))}
\end{array}$$

```
SEQUENCE-7 = (item: item ∈ SEQUENCE-6 ∧ color(item) = red)
SEQUENCE-8 = (item: item ∈ SEQUENCE-6 ∧ color(item) = blue)
∀i (1 ≤ i ≤ size(SEQUENCE-5) →
 ∀j (1 ≤ j ≤ size(item-in-sequence(i, SEQUENCE-5)) →
 position(item-in-sequence(j, item-in-sequence(i, SEQUENCE-5))) =
  pos-a))
∀i (1 ≤ i ≤ size(SEQUENCE-7) →
 position(item-in-sequence(i, SEQUENCE-7)) = pos-b)
∀i (1 ≤ i ≤ size(SEQUENCE-8) →
 position(item-in-sequence(i, SEQUENCE-8)) = pos-c)
counter = $k_{outer}$
```

previously generated, with the addition of

```
SEQUENCE-9 = (sequence i = 1 to $k_{inner}$
 sequential-object-in-source(i, SOURCE-4))
SOURCE-4 = line(X0 + $k_{outer}$*X, Y0 + $k_{outer}$*Y), $\Phi$)
∀i (1 ≤ i ≤ size(SEQUENCE-9) →
 position(item-in-sequence(i, SEQUENCE-9)) = pos-a)
```

where

```
n(i) =
       minimum({j:
        color(sequential-object-in-source(j+1,
          (item-in-sequence(i, SEQUENCE-4)))) = red
        ∨
        color(sequential-object-in-source(j+1,
          (item-in-sequence(i, SEQUENCE-4)))) = blue}).
```

and, since we now have two unresolved iteration counts, they have been distinguished by k_{inner} and k_{outer}. Two execution states will be associated with the exit of the inner loop, each having a new object, OBJECT-4, defined as sequential-object-in-source(k_{inner}+1, SOURCE-4) and exit conditions of color(OBJECT-4) = red and color(OBJECT-4) = blue respectively. Again, an attempt will be made to include OBJECT-4 into SEQUENCE-9, which will fail because OBJECT-4 is not at pos-a. The exit conditions will then be used to resolve k_{inner} to

$$k_{inner} = \text{minimum}(\{j: P(j)\})$$

where

$$P(i) = \text{color}(\text{sequential-object-in-source}(j+1, \text{SOURCE-4})) = \text{red} \vee \text{color}(\text{sequential-object-in-source}(j+1, \text{SOURCE-4})) = \text{blue}.$$

This will then be substituted wherever k_{inner} occurs in the execution states including the path conditions which contain color(sequential-object-in-source(k_{inner}+1,SOURCE-4)) = red and color(sequential-object-in-source(k_{inner}+1,SOURCE-4)) = blue.

Symbolic execution now continues from these execution states to the exit from the outer loop. In doing so, the counter in incremented to $k_{outer}+1$ and the path conditions have counter $=k_{outer} + 1 > a$ added to them. Thus, two execution states will be associated with the exit from the outer loop. The new items created during the exit path are SEQUENCE-9, SOURCE-4 and OBJECT-4. These will be successfully included in SEQUENCE-5, SEQUENCE-4 and SEQUENCE-6. Since these execution states contain exit conditions of

$$k_{outer}+1 > a \wedge \text{color}(\text{sequential-object-in-source}(k_{inner}+1, \text{SOURCE-4})) = \text{red}$$

and

$$k_{outer}+1 > a \wedge \text{color}(\text{sequential-object-in-source}(k_{inner}+1, \text{SOURCE-4})) = \text{blue}$$

we obtain a value of k_{outer} of

k_{outer} = minimum({j:
(j+1 > a ∧
color(sequential-object-in-source(k_{inner}+1, SOURCE-4)) = red)
∨
(j+1 > a ∧
color(sequential-object-in-source(k_{inner}+1, SOURCE-4)) = blue)
}).

Simplification using the rule (x ∧ y) ∨ (x ∧ z) → x ∧ (y ∨ z) will then produce

k_{outer} = minimum({j: j+1 > a ∧
(color(sequential-object-in-source(k_{inner}+1, SOURCE-4)) = red
∨
color(sequential-object-in-source(k_{inner}+1, SOURCE-4)) = blue)
}).

But, by the definition of P, above, this shows

k_{outer} = minimum({j: j+1 > a ∧ P(k_{inner})}).

But, since k_{inner} = minimum({j: P(j)}), simplification will produce P(k_{inner}) = T, so that

k_{outer} = minimum({j: j+1 > a}) = a.

When this is substituted wherever k_{outer} occurs in the execution states, we will obtain the final form as output by the interpretation process. Finally, we note that the two execution states will not be merged as they are not quite identical. In one the robot is now at pos-b, while in the other it is at pos-c. In other words, the final position of the robot depends on the color of the very last object moved. Note that in PAN's output interpretation, this last object is now defined as the last object in SEQUENCE-6.

REFERENCES

Andreae, P.M. [1984], Constraint Limited Generalization: Acquiring Procedures from Examples, *Proc 3rd AAAI.*

Andreae, P.M. [1985], Justified Generalization: Acquiring Procedures from Examples, (PhD Thesis) MIT/AI/TR-834

Brotsky, D. [1984], An Algorithm for Parsing Flow Graphs, (M.S. Thesis), MIT/AI/TR-704.

Cheatham, T.E., G.H. Holloway and J.A. Townley [1979], Symbolic Evaluation and the Analysis of Programs, *IEEE Trans Software Eng,* SE-5, 4, pp 403-418.

Clarke, L.A. and D.J. Richardson [1981], "Symbolic Evaluation Methods for Program Analysis" in *Program Flow Analysis,* S.S. Muchnick and N.D. Jones (eds), Prentice Hall.

Cohen, D [1983], Symbolic Execution of the Gist Specification Language, *Proc. 8th IJCAI.*

Dannenberg, R.B. and G.W. Ernst [1982], Formal Program Verification using Symbolic Execution, *IEEE Trans Software Eng,* SE-8, 1, pp 43-52

Dershowitz, N. [1985], Synthetic Programming, *Artificial Intelligence,* 25, pp 323-373

Dershowitz, N. and Z. Manna [1981], Inference Rules for Program Annotation, *IEEE Trans Software Eng,* SE-7, 2, pp 207-222

Dietterich, T.G. and R.S. Michalski [1985], Discovering Patterns in Sequences of Events, *Artificial Intelligence,* 25, pp 187-232.

Dijkstra, E. W. [1975], Guarded Commands, Nondeterminacy and Formal Derivation of Programs, *Communications of the ACM,* vol 18, no 8.

Dunlop, D.D. and V.R. Basili [1984], A Heuristic for Deriving Loop Functions, *IEEE Trans Software Eng,* SE-10, 3, pp 275-285

Goosens, D. [1979], Meta-Interpretation of Recursive List-Processing Programs, *Proc. 6th IJCAI.*

Goosens, D. [1981], The Conceptual Calculus for Automatic Program Understanding, *Proc. 7th IJCAI.*

Johnson, W.L. and E. Soloway [1985], Proust, *Byte* pp 179-190

Kemmerer, R.A [1985], Testing Formal Specifications to Detect Design Errors, *IEEE Trans Software Eng,* SE-11, 1, pp 32-43

Laubsch, J. and M.E. Eisenstadt [1981], Domain Specific Debugging Aids for Novice Programmers, *Proc. 7th IJCAI.*

Liu, P. and R. Chang [1987], A New Structural Induction Scheme for Proving Properties of Mutually Recursive Concepts, *Proc. 6th AAAI.*

Micalski, R. S. [1983], A Theory and Methodology of Inductive Learning, in *Machine Learning*, Michalski, Carbonell and Mitchell (eds), Tioga.

Richardson, D.J. and L.A. Clarke [1985], Partition Analysis: A Method of Combining Testing and Verification, *IEEE Trans Software Eng*, SE-11, 12, pp 1477-1490.

Rich, C., H.E. Shrobe and R.C. Waters [1979], An Overview of the Programmer's Apprentice, *Proc. 6th IJCAI.*

Shrobe, H.E. [1979], Dependency Directed Reasoning for Complex Program Understanding, (PhD Thesis), MIT/AI/TR-503.

Soloway, E.M., B. Woolf, E. Rubin and P. Barth [1981], Meno-II: An Intelligent Tutoring System for Novice Programmers, *Proc. 7th IJCAI.*

Steier, D. and E. Kant [1985], Symbolic Execution in Algorithm Design, *Proc. 9th IJCAI.*

Waters, R.C. [1979], A Method for Analysing Loop Programs, *IEEE Trans Software Eng*, SE-5, 3, pp 237-247

Waters, R.C. [1982], The Programmer's Apprentice: Knowledge Based Program Editing, *IEEE Trans Software Eng*, SE-8, 1, pp 1-12.

Waters, R.C. [1985], Program Translation via Abstraction and Reimplementation, *IEEE Trans Software Eng*, sumbitted Nov 1985.

Weld, D.S. [1986], The Use of Aggregation in Causal Simulation, *Artificial Intelligence*, 30, pp 1-34.

Zelinka, L. [1986], Automated Program Recognition, (M.S. Thesis), MIT.