CORE
Metadata, citation and similar papers at core.ac.uk
Provided by Online Research Database In Technology

Technical University of Denmark

Fault tree and cause consequence analysis for control software validation

Forskningscenter Risø, Roskilde

Publication date:
1982

Document Version
Publisher's PDF, also known as Version of record

Link back to DTU Orbit

Citation (APA):
Taylor, J. R. (1982). Fault tree and cause consequence analysis for control software validation. (Risø-M; No. 2326).

DTU Library
Technical Information Center of Denmark

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal

If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

RISØ-M-2326

FAULT TREE AND CAUSE CONSEQUENCE ANALYSIS FOR CONTROL SOFTWARE VALIDATION

J.R. Taylor

Abstract. A theory underlying application of automatic fault tree analysis to computer programs is described.

INIS Descriptors: COMPUTER CODES; FAULT TREE ANALYSIS; FAILURE MODE ANALYSIS.

UDC 681.3.06:519.248

January 1982

Risø National Laboratory, DK-4000 Roskilde, Denmark

ISBN 87-550-0819-4
ISSN [illegible]

Risø repro 1982

TABLE OF CONTENTS

Page

INTRODUCTION 5

FAULT TREE ANALYSIS 6

FAULT TREE ANALYSIS FOR SOFTWARE 8

ASSIGNMENT STATEMENT 9

LOOPS AND INDUCTION 10

WHILE STATEMENT 11

FEATURES OF THIS APPROACH 12

TREATMENT OF FUNCTIONS AND SUBROUTINES 13

PROCEDURE CALL 14

WAIT AND SIGNAL STATEMENTS 14

SHARED VARIABLES IN CRITICAL SECTIONS 15

FORK AND JOIN 16

PREDICATE TRANSFORMER RULES FOR STRONGEST POST

CONDITIONS OF $x = x_o$ 17

INTERRUPTS 18

REFERENCES 20

FAULT TREE AND CAUSE CONSEQUENCE ANALYSIS FOR CONTROL SOFTWARE VALIDATION

Introduction

In many areas of computer application, safety depends on freedom from software error. Examples are elevator control, control of chemical plants, railway signalling and control, nuclear reactor safety systems and aircraft landing systems. For systems such as these it is important to be able either to guarantee freedom from error, or to be able to reduce the probability of failure in operations. For example a nuclear reactor shutdown system should fail at a rate which is typically less than once per 10^{15} program executions.

Path domain testing [1], [2], [3], [4], comes close to satisfying the need for a guarantee of freedom from error. If along each path through a program a single variable polynomial is computed of maximum order n, then n + 1 tests per path will reveal all computation errors. A similar number of tests of domain boundaries will detect most control errors, if conditional statement predicates involve only single variables. If several variables are involved in path predicates or in path computations then n tests will not necessarily reveal all errors, since an actual path predicate or path computation may correspond coincidentally to the correct predicate or computation for all test cases. This will happen if the hypersurfaces for the actual and correct polynomials touch or cut each other at each test point. The probability of this will generally be very low, if more than just a few test points are chosen at random.

The remaining group of errors undetected by thorough path domain testing are missing program paths corresponding to special input data cases. To find these, path domain predicates may be compared with formal program specifications [5].

The main advantage of path domain testing, when compared with proof of program correctness is avoidance of the need for a formal specification. It is known that provision of formal specifications is extremely error prone [6]. Additionally, even informal specifications are subject to error, accounting for some 50% of failures in operating software. Systematic testing overcomes some of these problems, by allowing the programmer to deal with program performance on a case by case basis.

One way of checking the correctness of test outputs in path domain testing is to execute the program along with a model of the plant to be controlled. The model is checked to ensure that its performance in response to computer outputs is satisfactory. This leads to the arrangement shown in fig. 1. Here the plant model provides the specification of correct performance.

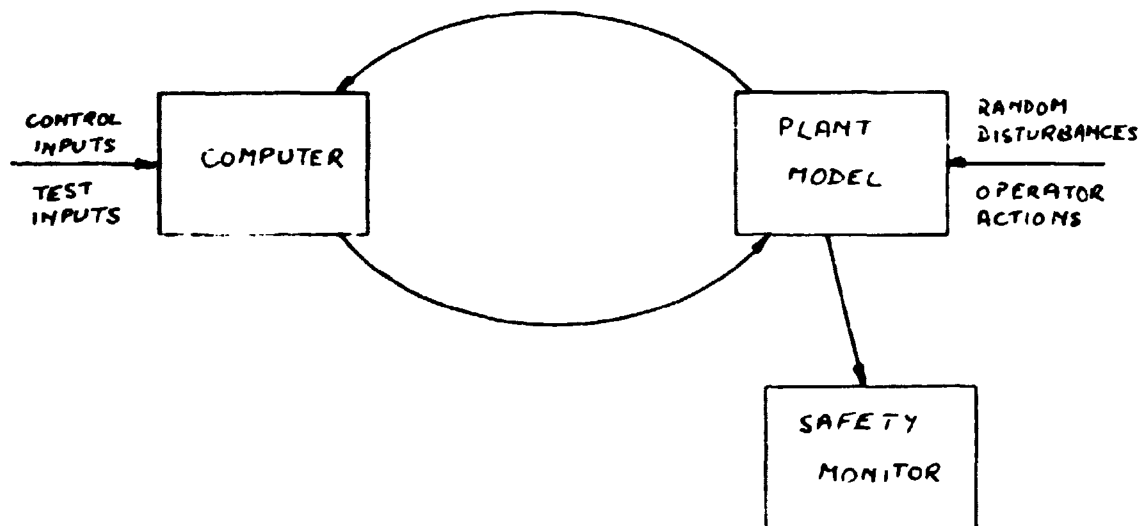


Fig. 1. Arrangement for software testing with a model.

Fault tree analysis

Automated fault tree analysis [7,8,9] makes use of finite state models of plant components which are very much like predicate transformers used in deriving path predicates. Each component is described by means of a set of statements, termed mini-fault trees, of the form

"if x occurs at the input to a component, and if the state of the component is Y then x' will occur at the output of the component, and the new component state will be Y'"

In building up a fault tree a "top event" is chosen within a component, generally some undesired event, such as an explosion or crash. This event is matched to output events of mini fault trees for the component, and any matching mini-fault trees are added to the overall fault tree for the system. If there are several matching mini-fault trees, they are connected into the overall fault tree via an OR gate.

Having found an initial match, indicating a "first level" cause of the top event, in an input, or internal state change event X within the component, causes of this event are then sought. If X is an internal state change event, mini fault trees with X as a resulting event are sought for. If X is an input event, mini fault trees providing a cause for X are sought in the components connected to that in which X occurs.

This process is iterated, building up chains of events backwards in time. The chains branch via an AND gate whenever both an event X and a state Y are required for an event X' to occur. The chains branch via an OR gate whenever there are two potential ways in which an event X occurs (fig. 2). The process of building up an event chain terminates when either a "spontaneous" event is found (that is one for which no specific further cause is defined), or a "normal" event or state is found, that is one which will occur frequently during normal operation of a plant. The process of building up the fault tree terminates when all event chains have been terminated with "normal" or "spontaneous events".

It is necessary to store the state of components as the fault tree is built up, since if two chains, 'anded' together, arrive at the same component (due to a feedback or feed forward loop in the system) then the two chains must be checked for compatibility. They may, for example, require that the component fulfil

two mutually exclusive conditions at the same time. If such a condition is detected, then the event chains are rejected.

This building of event chains can be carried out in the forward direction, in which case the process is known as consequence analysis [10]. The two processes may be combined into one procedure, cause consequence analysis, in which the causes and consequences of a single "critical event" are sought. Heuristic rules have been devised for systematic selection of critical events in plant safety analysis [11].

Fault tree analysis for software

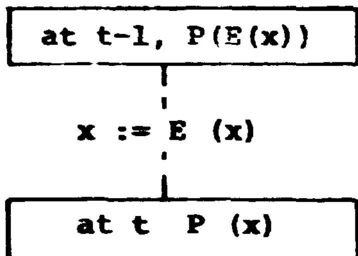
The analogy between state transition functions for hardware components and predicate transformers can be utilised in building up fault trees for combined hardware and software systems. In building up the trees, some unwanted event is identified within the hardware and event chains are traced which can cause this event. When a "computer" component is reached, events at the output registers of the computer will be found. The "causes" of these are register manipulation statements within the software.

Chains of "events" within the software are then sought by tracing changes in program variables from statement to statement using a predicate transformer technique, until program inputs are found which are necessary for the particular event chain. The chains can then be extended once again to hardware, seeking the potential causes of the program input events found.

The mini fault tree notation can be applied to individual program statements, allowing a uniform process of hardware/software fault tree construction. For each statement a set of mini fault trees is generated according to the schemes shown in fig. 3. These are a modified form of weakest precondition predicate transformers used elsewhere in proving program correctness [15].

The "events" in the program mini fault trees are statements of the form

Assignment statement



x is a free variable representing the vector of program variable values at different points in the program.

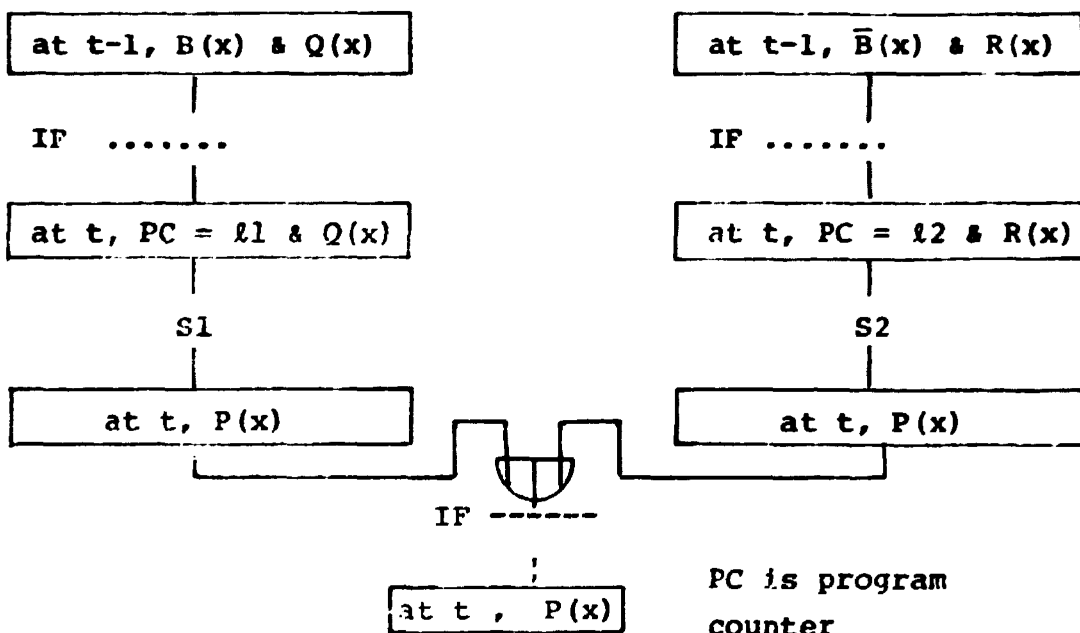
t is a free variable representing a pair -"earliest and latest time".

E is an expression.

IF statement

IF B(x) THEN ℓ1 : S1 ELSE ℓ2 : S2

S1 and S2 are any statements including compound statements. ℓ1 and ℓ2 are labels.



PC is program counter

Fig. 3. Mini fault trees for basic program statements.

"at time t, predicate P becomes true of the program variables"

There are three main differences in the process of building fault trees for hardware and software. Firstly, instead of there being a simple match between events and mini fault trees, as for hardware, the software mini fault trees may match <u>any</u> predicate. Secondly, the mini fault trees, when matched, must be modified by substituting the output predicate into the mini fault tree, both at the input and output position. Thirdly, it is generally necessary to simplify the predicates representing input and output events in the mini-fault trees. If this simplification results in the predicate FALSE the event chain is abandoned.

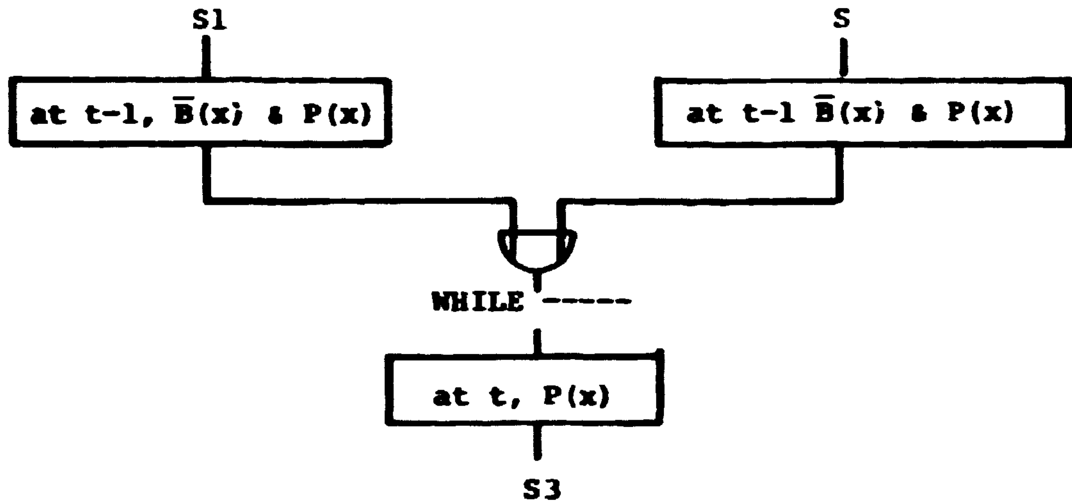
Note that the form of fault tree for IF and WHILE statements provides an OR branch for each direction of branching through the statement. This ensures that all paths through a program will be treated as different branches of the fault tree. This has the advantage of ensuring greater understandability, and of keeping individual predicates relatively simple. It has the disadvantage of producing very large trees in many cases. For this reason it h as been found advisable to insist on quite severe structuring rules for programs to be used with this technique. - "Separate (possibly parallel) programs should be provided for each separate control or safety function within a computer system". This rule ensures at least that the program paths followed are relevant to the analysts safety problem. An alternative formulation of the structuring criterion can be applied systematically. - If there are two program outputs X and Y for which the values are functions of sets of inputs I(Y) and I(Y), then if I(X) ∩ I(Y) is empty, programs producing outputs X and Y should be disjoint.

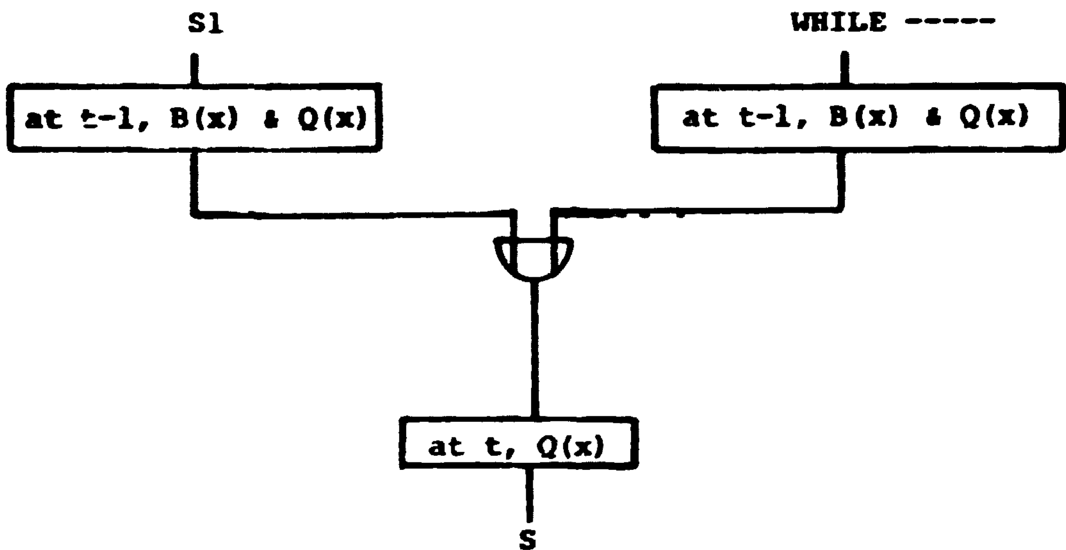
<u>Loops and induction</u>

With loops in a program, the size of the fault trees produced may grow very large, or may even grow indefinitely. For practical purposes, the number of times the fault tree iterates round a

WHILE statement

S1; WHILE B(x) DO S ; S3





program loop must be limited.

Induction can be applied to the fault tree to show that no possible program inputs lead to a dangerous event, in the following way.

1. It is shown that no dangerous output is produced from any input causing zero iterations round a loop.

2. It is shown that no dangerous outputs are produced from any input causing one iteration round a loop.

3. It is shown that if no dangerous output has been produced with n iterations around a loop, then no dangerous output will be produced with n + 1 iterations.

This type of proof is aided by the graphic presentation of the fault tree. The fault tree branch corresponding to the $n + 1^{th}$ iteration is generally similar in form or is a simple systematic modification of the form of the branch corresponding to the nth iteration. If the forms can, by logical manipulation, be made identical apart from the numerical constant n, then the induction proof is completed.

Features of this approach

The main advantage of the approach outline here is that software testing is integrated into plant functional analysis and failure analysis. The overall performance of plant and control system can be investigated.

The problem of finding an unequivocal decision about which program outputs are "correct", which limits the usefulness of path domain testing, are solved with this approach by referring to the 'correct' or 'safe' performance of the plant model.

Problems of specification errors are to a large extent avoided with this approach. No formal specification of program require-

ments need be made, and the "testing" is independent of the program specification. The plant component models used are drawn from a library of standard models which are tested over a long series of analyses. The models are derived by means of a standardised procedure from physical equations for the components [4]. In other words the test is very thorough and is independent of the programming process.

Parallel processing is accounted for by the fault tree method in a natural way.

The major problems with the fault tree approach to software validation are the need for inductive proofs for programs with loops, (these must be provided by the analyst), and the very large size of fault trees formed. It is doubtful whether any practical program could be validated as a whole. Rather, it seems necessary to break down programs hierarchically, into subroutines or modules.

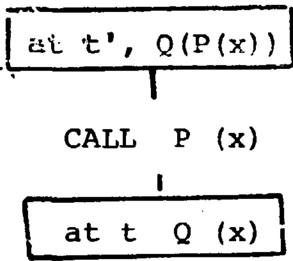
Given that a proof of freedom from program error is carried through using a fault tree approach, one might ask whether any errors can remain in the program. Such weak points must arise due to some common error in both the programming and plant modelling process. Three obvious sources are errors in the physical description of the plant (plant flow sheet), errors in the plant construction so that i does not accord with plant design, and common misconceptions, shared by the programmer and the plant model builder, about the way the plant works.

Treatment of functions and subroutines

Given the need for a hierarchical decomposition of the analysis task, a question arises, 'how?'. One approach is to analyse functions and subroutines separately, and to incorporate these directly into the program descriptions.

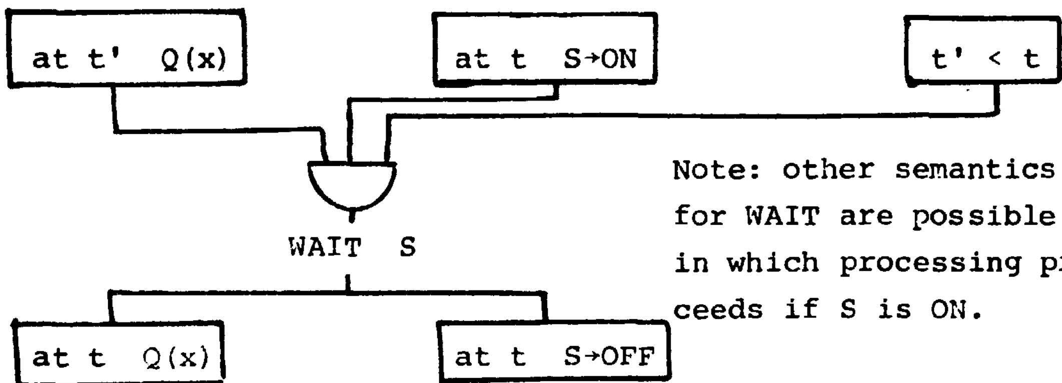
One way of doing this is to evaluate the function which a subroutine computes and to make use of the functional description

Procedure call

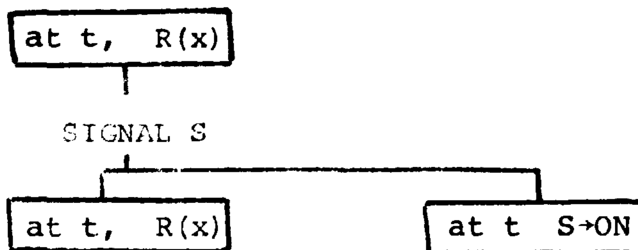


Note: It is necessary to evaluate the function calculated by P. See fig. 4.

WAIT and SIGNAL statements



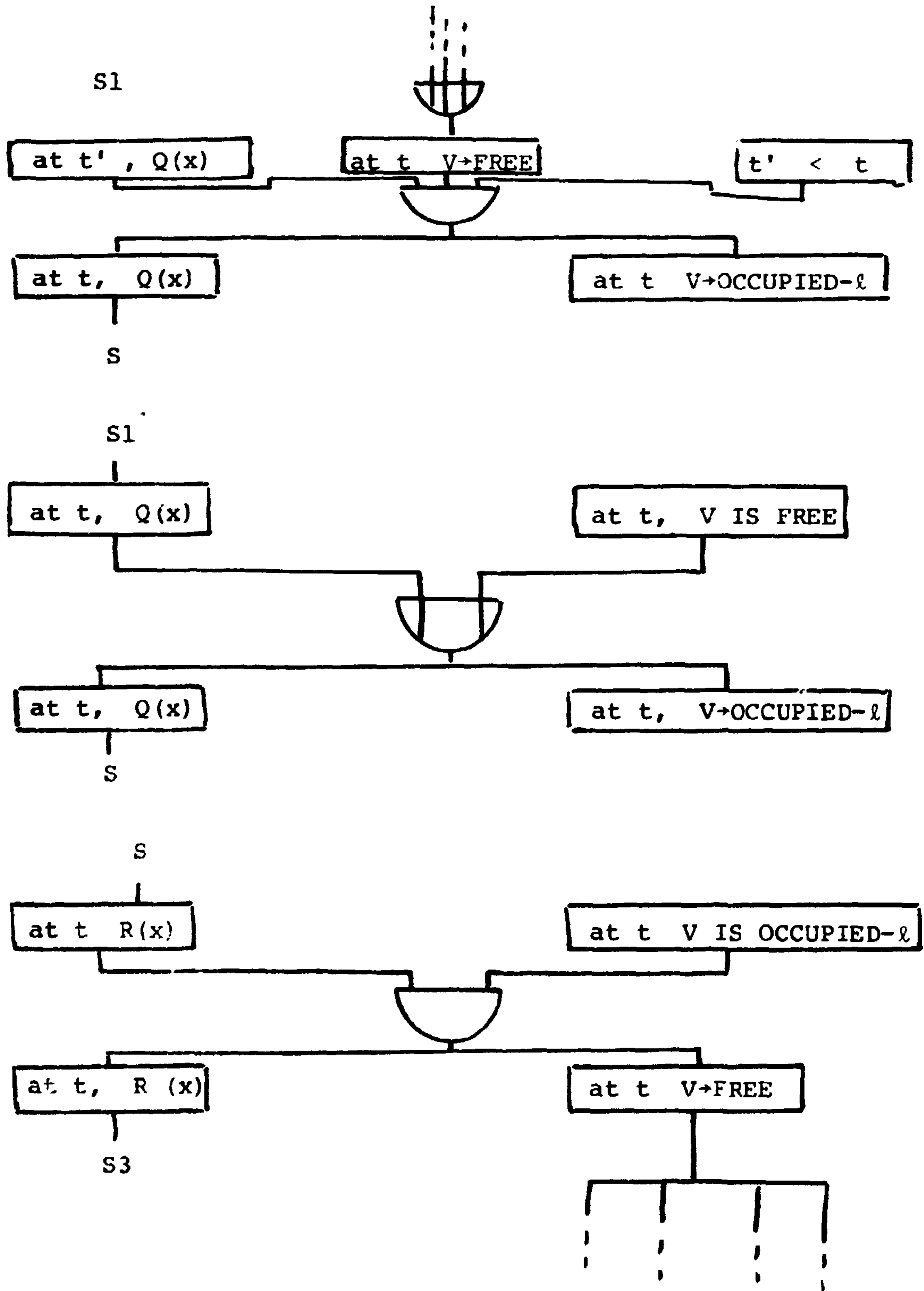
Note: other semantics for WAIT are possible in which processing proceeds if S is ON.



Note: No sharing of variables is assumed here - each program has its own variables.

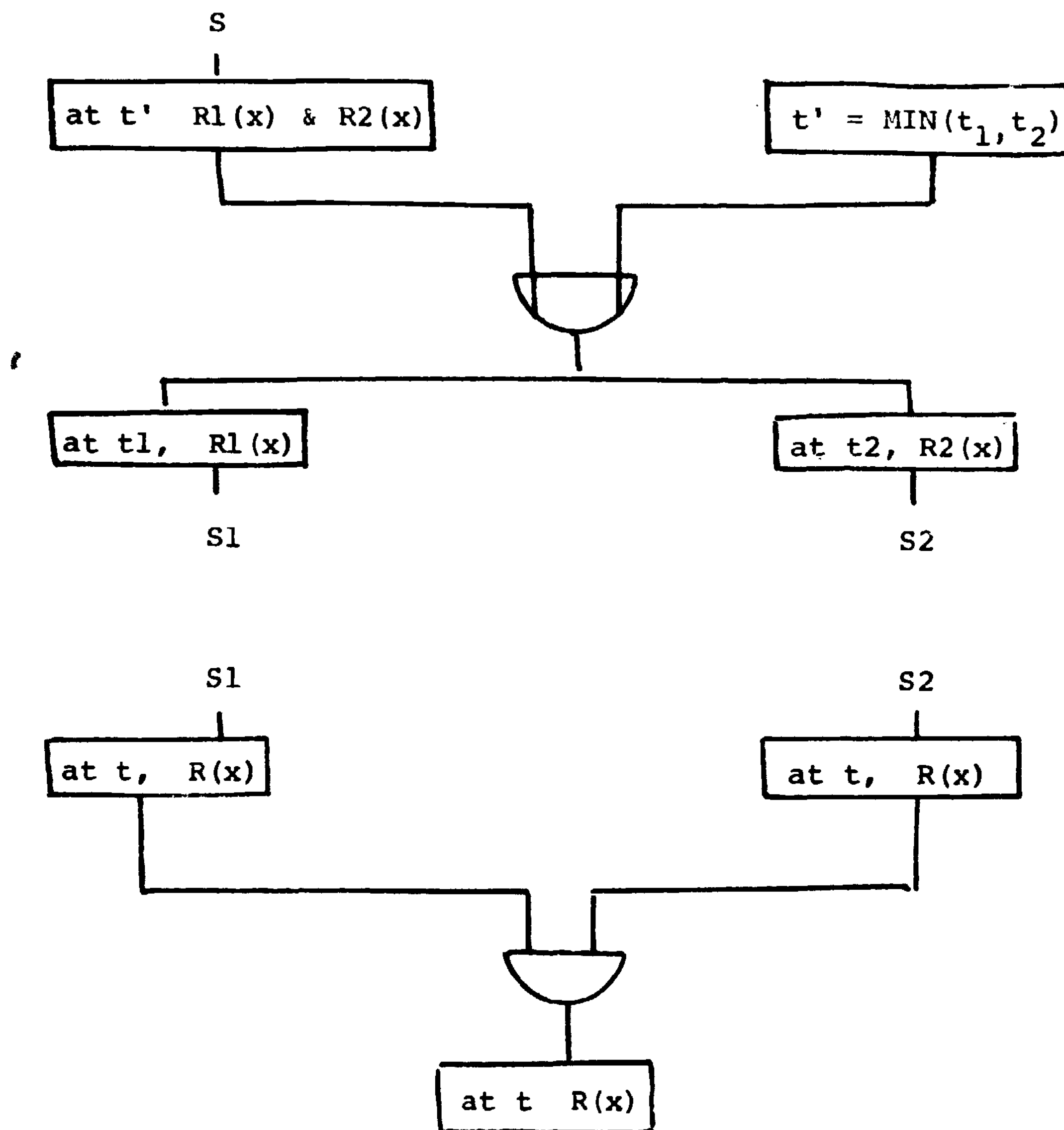
<u>Shared variables in critical sections</u>

S1 ; CRITICAL SECTION ℓ WITH V DO S ; S3



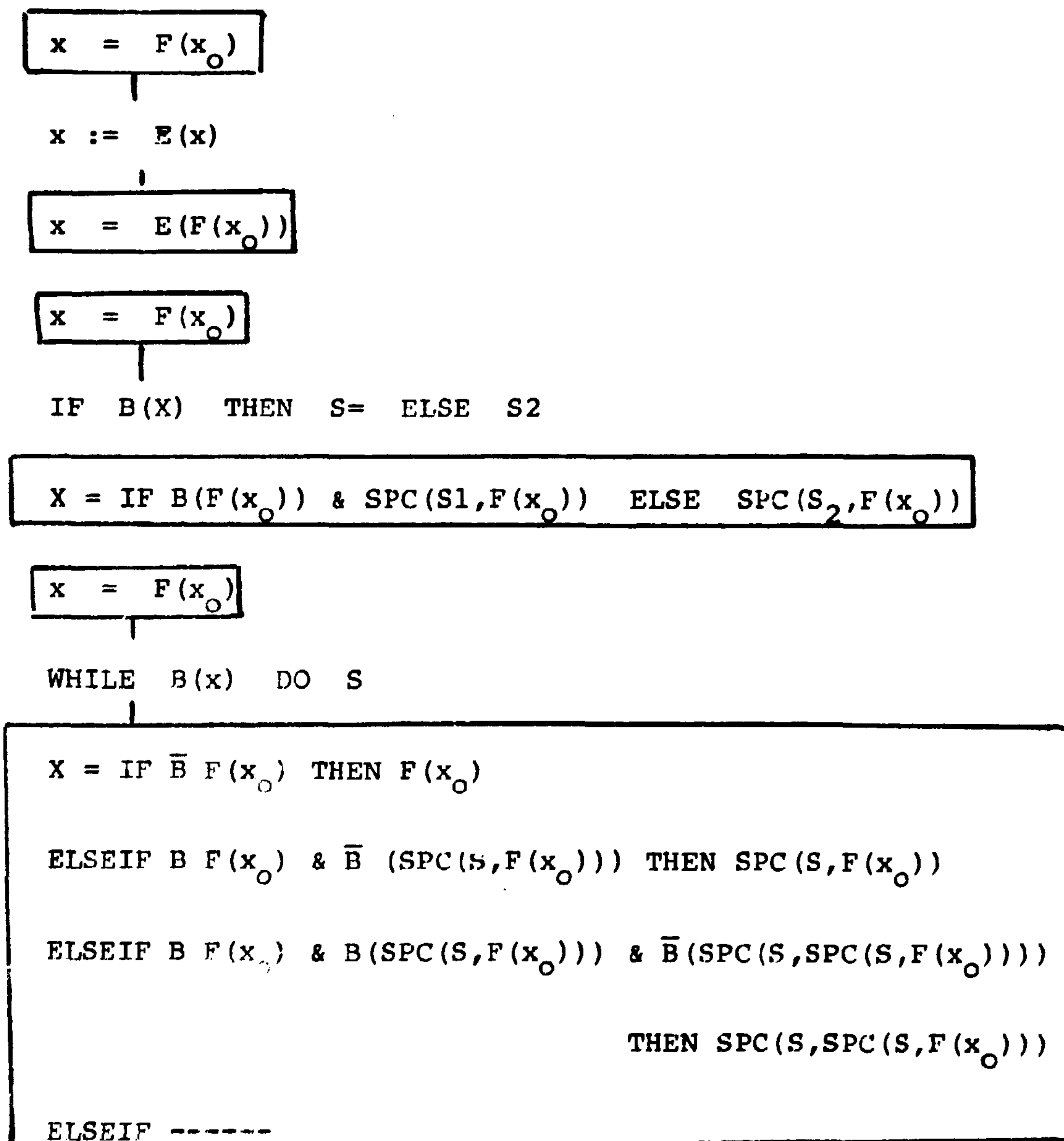
FORK and JOIN

S ; FORK S1 ; S2 JOIN ; S3



of the subroutine in building up the fault tree. A mini fault tree for procedure calls without side effects is shown in fig. 3. The function calculated by a procedure can be found by consequence analysis/symbolic execution, applying the rules shown in fig. 4. These work by finding the value of the program variable vector after execution of a program statement, as a function of the program variable vector prior to execution of the statement. Rules for finding these functions for individual statements are then applied iteratively.

<u>Predicate transformer rules for strongest post conditions of $x = x_o$</u>



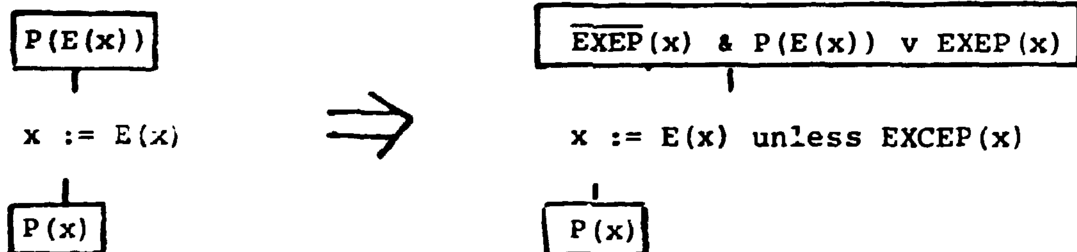
Interrupts

Three kinds of interrupts can be distinguished

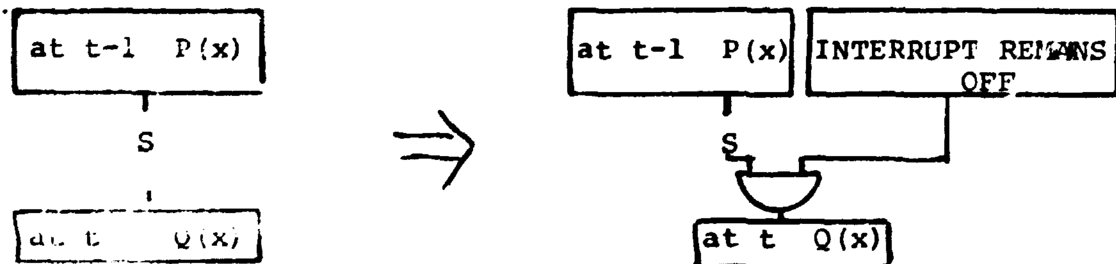
- SIGNALS in which one process waits for an interrupt from another before restarting
- EXCEPTIONS in which a program halts for some error recovery action in case some exceptional condition is met
- ATTENTIONS in which a running program is halted by another process, and is either STOPped, MODIFIEd, or SUSPENDed awaiting a further RESUME command.

The first of these can be treated in the same way as WAIT and SIGNAL described earlier.

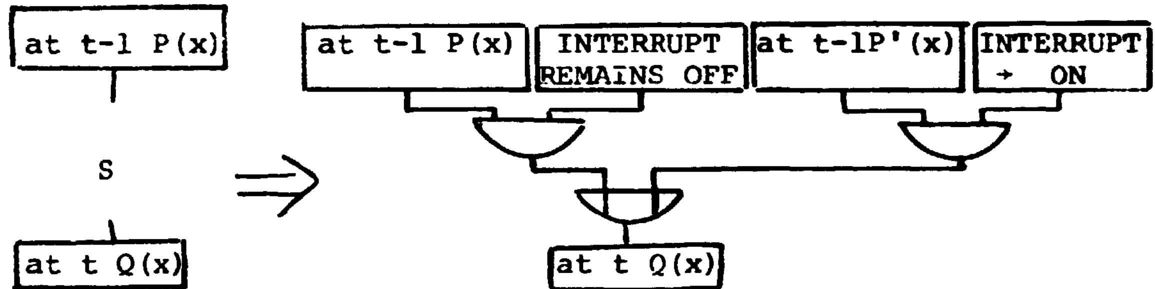
EXCEPTIONS can be treated readily in mini-fault trees by attaching an additional predicate to each tree as follows.



For STOP type interrupts the modification is as follows

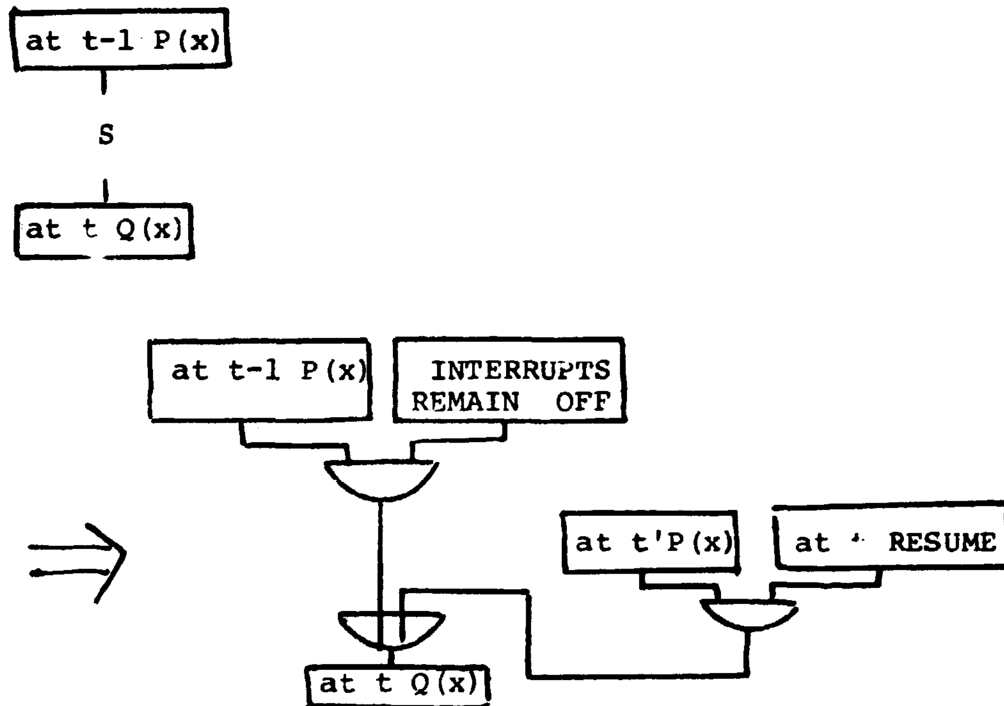


For MODIFY interrupt (in which processing resumes after modification) the transformation is



P(x) = WP(S,Q(x)) P'(x) = WP(S,WP(F,Q(x)) where F is the function computed by the interrupt routine.

For SUSPEND interrupts the modification is as follows



References

1. J.C. Huang, An Approach to Program Testing, ACM Computing Surveys, Vol. 7, No. 3, September 1975.
2. L.J. White and E.T. Cohen, A Domain Strategy for Computer Program Testing, IEEE Trans. Software Eng., Vol. SE6, No. 3, May 1980.
3. W.E. Howden, Methodology for the Generation of Program Test Data, IEEE Trans. Comp., Vol. C-24, May 1975.
4. J.R. Taylor and S. Bologna, Validation of Safety Related Software, IAEA Specialist Meeting on Computerised Control and Safety Systems in Nuclear Power Plants, Pittsburgh, 1977.
5. S. Bologna et al., Deriving Test Data from Specifications, Report RT/ING(78)3, CNEN, Italy, 1978.
6. J.B. Goodenoug and S.L. Gerhardt, Towards a Theory of Test Data Selection, IEEE Trans. on Software Engineering, Vol. SE-1, No. 2, June 1975.
7. J.R. Taylor, Sequential Effects in Failure Mode Analysis. In Reliability and Fault Tree Analysis, Ed. Barlow, Fussel and Singapurwalla, SIAM, 1975.
8. E. Hollo and J.R. Taylor, Experience with Algorithms for Automatic Failure Analysis, in Nuclear Systems Reliability, J.B. Fussel and G.R. Burdidi, SIAM, 1977.
9. J.R. Taylor, An Algorithm for Fault Tree Construction, to be published in IEEE Trans. Reliability.
10. D.S. Nielsen, The Cause-Consequence Method as a Basis for Quantitative Accident Analysis, Report Risø-M-1374, Risø National Laboratory, 1971.
11. J.R. Taylor, A Background to Risk Analysis, Vol. 1 to 4, Risø National Laboratory, 1979.

Risø National Laboratory

Risø - M - 2326

Risø - M - 2326

Title and author(s)

Fault Tree and Cause Consequence Analysis for Control Software Validation

J.R. Taylor

Date January 1982

Department or group

Electronics

Group's own registration number(s)

R-1-82

pages + tables + illustrations

Abstract

A theory underlying application of automatic fault tree analysis to computer programs is described.

Copies to

Available on request from Risø Library, Risø National Laboratory (Risø Bibliotek), Forsøgsanlæg Risø), DK-4000 Roskilde, Denmark
Telephone: (03) 37 12 12, ext. 2262. Telex: 43116