CONSTRUCTION OF A FAULT TREE USING PROLOG

S. Fukuda

Welding Research Institute, Osaka University 11-1, Mihogaoka, Ibaraki-City, Osaka 567, Japan

ABSTRACT

This paper shows that the programming language Prolog provides a very useful and versatile tool for constructing a fault tree for failure and fracture-related problems. Prolog not only reduces the time and trouble in developing a fault tree, but it also injects inferring ability into a computer. Therefore, it provides a very good man-machine interface for making inferences and arriving at an adequate conclusion.

KEY WORDS

Fault tree analysis; fracture prevention; artificial intelligence; knowledge engineering; Prolog

INTRODUCTION

As machines and structures are being operated under more and more severe conditions and are becoming more complex, multifaceted and gigantic than ever before, the prevention of their failures is becoming one of the most important problems.

Fault tree analysis is known to be one of the most versatile and useful tools for analyzing failures of quite complicated systems. But in the case of fracture-related problems there are certain difficulties in constructing a fault tree; two such major difficulties are

(1) It is not so straightforward to develop a fault tree as in the case of control systems for mechanical failures, since the structure of the problem is not so well defined mathematically. Hence, a trial and error method has to be more often used for developing a fault tree.

(2) To well define the structure of the problem, it is necessary to describe the content of a node more accurately than in other fields. Therefore, symbolic manipulation is most essential in this field.

This paper points out that by adopting the progamming language Prolog, we can eliminate the above difficulties in constructing a fault tree in the case

of failure and fracture-related problems and can make a knowledge engineering approach to the problem.

KNOWLEDGE ENGINEERING AND PROLOG: SIMPLE EXPLANATION

Apart from a very rigorous definition, Knowledge Engineering (Feigenbaum, 1977) may be said to be one field of applied artificial intelligence where a man's knowledge (which includes experience) is implemented as data on a computer and intelligent processing of information is carried out.

To carry out intelligent processing of information, an adequate selection of the tool, i.e., the programming language is inevitable. Therefore, we have to use the programming language suited for manipulating symbols. Although Fortran is otherwise useful, it is developed for processing numerical values and is not suited for this purpose. Prolog (Kowalski, 1977) is one of the most useful tools for manipulating symbols and is considered in Japan as a promising language for the computers of the coming generation, i.e., the computers of the fifth generation (Motooka, 1981).

Prolog is based on the first order predicate logic. An example of such logic is given below.

Major premise: Every human is mortal
Minor premise: Socrates is a human
Conclusion: Socrates is mortal

In other words, if the following two sentences are given,

$\forall x.$ human $(x) \longrightarrow$ mortal (x)
human (Socrates)

then a computer infers that

mortal (Socrates)

In Prolog/KR developed by Nakashima (1983), these statements are expressed as follows;

```
:(ASSERT (MORTAL *X)(HUMAN *X))
(ASSERT (MORTAL *X-0000)(HUMAN *X-0000))
:(ASSERT (HUMAN SOCRATES))
(ASSERT (HUMAN SOCRATES))
:(MORTAL *X)
(MORTAL SOCRATES)
```

The symbol : is a prompting from a computer which appears on a CRT. The sentences after the symbol : are inputs to the terminal, whereas the following sentences are outputs from the computer, assuring that those input sentences have been accepted. The third input sentence is a question as to whether it is true or not that if Socrates is a human, he is mortal. As the value Socrates is assigned to the variable *X, (MORTAL *X) means the question whether Socrates is mortal or not. As it is already asserted in the first input sentence, the answer that it is true is returned. The last sentence (MORTAL SOCRATES) means that the computer returned the "TRUE" answer. If the answer is "NO", then the computer returns "NIL".

FAULT TREE ANALYSIS USING PROLOG: SIMPLE ILLUSTRATION

A fault tree is a graphical technique that provides a systematic description of the combinations of possible occurrences in a system, which can result in a 'fault' or 'undesired event'. It is a schematic representation of the inter-relationships between the different 'basic events' and the 'undesired event'. The inter-relationships are shown using Boolean logic symbols.

A rectangle represents an 'event block', which describes that the event is caused by the combination of fault causes through the input gate. A diamond represents an 'undeveloped event' and the circle represents a 'basic event'.

An 'OR' gate shows a logical relationship, that the output event will happen if and only if one or more of the input events happen. An 'AND' gate shows a logical relationship, that the output event will happen if and only if all the input events happen.

The following knowledge provides the basic tool for constructing a fault tree using Prolog. The relations on the left hand side are expressed correspondingly in Prolog on the right hand side.

```
If B occurs, then A occurs.        ------ (ASSERT (A) (B))
If B and C occur, then A occurs.   ------ (ASSERT (A) (B) (C))
If B or C occurs, then A occurs.   ------ (ASSERT (A) (B))
                                          (ASSERT (A) (C))
```

And the fact that the event B really happens is expressed: (ASSERT (B))

We will illustrate how the conversation is carried out on a computer by taking a simple eletrical circuit shown in Fig. 1 (a) as an example. The graphical representation of this fault tree is given in Fig. 1 (b).

```
:(ASSERT (NO-LIGHT)
(NO-CURRENT-IN-A))
(ASSERT (NO-LIGHT)
(NO-CURRENT-IN-A))
:(ASSERT (NO-LIGHT)
(NO-CURRENT-IN-B))
(ASSERT (NO-LIGHT)
(NO-CURRENT-IN-B))
:(ASSERT (NO-CURRENT
-IN-A)(SWITCH-1-
FAILURE))
(ASSERT (NO-CURRENT-
IN-A)(SWITCH-1-FAILURE))
:(ASSERT (NO-CURRENT-
IN-B)(SWITCH-2-FAILURE)
(SWITCH-3-FAILURE))
(ASSERT (NO-CURRENT-
IN-B)(SWITCH-2-FAILURE)
(SWITCH-3-FAILURE))
```

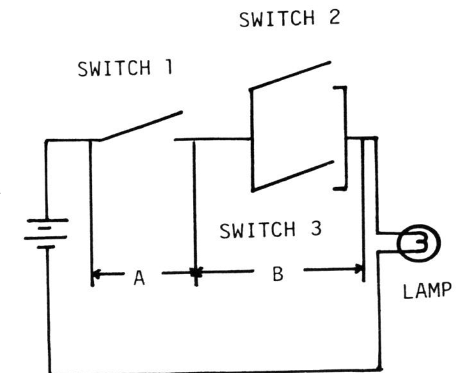

Fig. 1 (a). Sample system

Now, the fault tree of Fig. 1 (b) is constructed on the computer. We will start asking.

```
:(ASSERT (SWITCH-1-FAILURE))
```

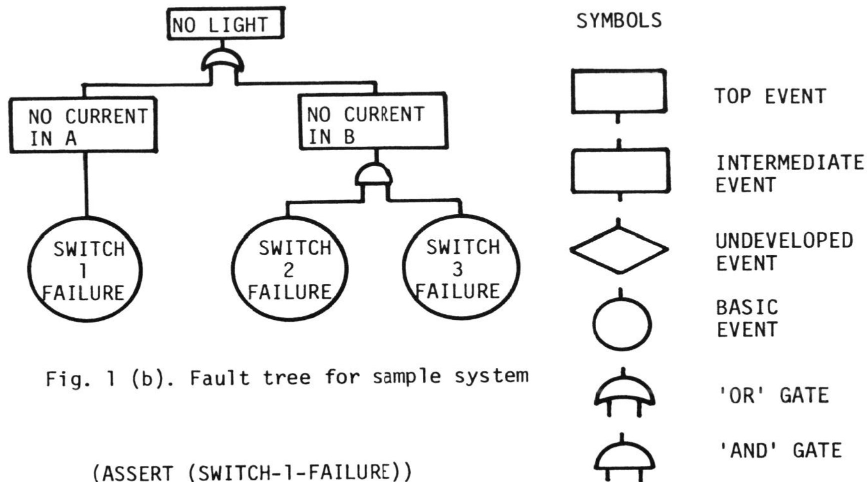

Fig. 1 (b). Fault tree for sample system

```
(ASSERT (SWITCH-1-FAILURE))
:(NO-LIGHT)
(NO-LIGHT)
```

Thus, the computer returns the "TRUE" answer to the question whether it is true or not if the switch 1 fails, the light will not be on. We will examine another case. But before we start another question, we have to withdraw the assertion that the event "Switch 1 fails" occurs. Otherwise this assertion is still valid and unintentionally we will be examining the case we do not wish to analyze. The assertion is easily withdrawn as follows.

```
:(RETRACT (SWITCH-1-FAILURE))
(RETRACT (SWITCH-1-FAILURE))
```

And then another case is examined,

```
:(ASSERT (SWITCH-2-FAILURE))
(ASSERT (SWITCH-2-FAILURE))
:(NO-LIGHT)
3(STANDARD-ERROR-HANDLER "UNDEFINED PREDICATE" SWITCH-1-FAILURE))
S: C
3(STANDARD-ERROR-HANDLER "UNDEFINED PREDICATE" SWITCH-3-FAILURE))
S: C
NIL
```

The first message after (NO-LIGHT) means that (SWITCH-1-FAILURE) is not asserted. As this event is already retracted, we input C for the prompt S: to continue the search. Although the event (SWITCH-2-FAILURE) is asserted, the event (SWITCH-3-FAILURE) is not asserted yet. Therefore, we obtain the second message that (SWITCH-3-FAILURE) is not asserted. As "NIL" is returned to the input C for the prompt S:, it is known that the event "NO-LIGHT" will not occur even if the switch 2 fails.

```
:(ASSERT (SWITCH-3-FAILURE))
(ASSERT (SWITCH-3-FAILURE))
```



```
:(NO-LIGHT)
3(STANDARD-ERROR-HANDLER "UNDEFINED PREDICATE" SWITCH-1-FAILURE)
S: C
(NO-LIGHT)
```

This means that as the assertion "SWITCH-2-FAILURE" is still valid, the event "NO-LIGHT" will occur if the switch 2 and 3 fail.

Thus, we can construct a fault tree, change its structure, and study what will happen under the given situation quite easily without any difficulty or trouble if we use the Prolog predicate function (ASSERT) and (RETRACT).

CONSTRUCTION OF A FAULT TREE FOR TRANSVERSE WELD CRACK USING PROLOG

As an example of a practical application, we will consider the construction of a fault tree for weld cracking which occurs during the manufacturing process of a pressure vessel.

Very strict control is carried out in welding a very thick section of a low alloy steel such as 2 1/4 Cr- 1 Mo steel to prevent the initiation of a transverse weld crack. This is because the structural integrity of a pressure vessel is greatly endangered by the presence of this kind of crack. Therefore, an intermediate post weld heat treatment is usually carried out to prevent the occurrence of such a crack, although it requires a great amount of time and energy.

Figure 2 shows an example of a fault tree for this case. The contents of the Top Event and the Basic Events are shown in Table 1 and the contents of the Intermediate Events are shown in Table 2 respectively.

It can easily be observed from the figure that once a fault tree becomes very large, it is quite difficult to follow what is happening, although it is generally said that a fault tree provides good visibility. Furthermore it is quite difficult to write the content of each event in the figure because visibility will be impaired more, although the description of the content is necessary in such a fault tree for failures.

The Prolog version of this fault tree is as follows;

```
:(ASSERT (TRANSVERSE-WELD-CRACK)(EXCESSIVE-HAZ-HARDENING))
:(ASSERT (TRANSVERSE-WELD-CRACK)(EXCESSIVE-STRESSES))
:(ASSERT (TRANSVERSE-WELD-CRACK)(EXCESSIVE-HYDROGEN))
:(ASSERT (EXCESSIVE-HAZ-HARDENING)(IMPROPER-THERMAL-CYCLE)
(MATERIAL-HARDENABILITY))
:(ASSERT (EXCESSIVE-STRESSES)(EXCESSIVE-INTERNAL-CONSTRAINT)
(EXCESSIVE-EXTERNAL-CONSTRAINT))
:(ASSERT (EXCESSIVE-HYDROGEN)(HYDROGEN-DIFFUSION)(EXCESSIVE-
HYDROGEN-CONTENT))
:(ASSERT ( ..........
..........
..........
:(ASSERT (EXCESSIVE-HYDROGEN-CONTENT)(IMPROPER-FLUX)(EXCESSIVE-
HUMIDITY))
```

The outputs from the computer are omitted to save space. Thus, a fault tree is defined. And such predicates as (MATERIAL-HARDENABILITY), (HYDROGEN-DIFFUSION), etc. means that such problems related with material hardenability

or hydrogen diffusion occur.

Suppose we wish to know what fault events trigger the fault event (IMPROPER-THERMAL-CYCLE). The answer will be immediately and easily given by the input (LIST).

```
:(LISTING IMPROPER-THERMAL-CYCLE)
(ASSERT (IMPROPER-THERMAL-CYCLE)(HOLD-TIME-AT-PEAK-TEMP)
(RAPID-COOLING-RATE)(PEAK-TEMP-TOO-HIGH)(HEATING-RATE))
(LISTING IMPROPER-THERMAL-CYCLE)
```

And further let us suppose that some engineers say that hold time at peak temperature and heating rate are not so influential we better eliminate these factors from the fault tree. Then we simply input (RETRACT) as follows;

```
:(RETRACT (IMPROPER-THERMAL-CYCLE))
```

Then the above assertion is retracted so we assert again

```
:(ASSERT (IMPROPER-THERMAL-CYCLE)(RAPID-COOLING-RATE)(PEAK-TEMP-
TOO-HIGH))
```

In this manner or by utilizing the Prolog editor which is provided with a quite powerful pattern-matching function, we can easily add, eliminate or change any relation at any hierachical level, and by using (LISTING), good visibility is provided, and furthermore we can understand at once what each node represents because its content is fully described. And it should be pointed out that in communicating with a computer, we do not have to worry about the addressing problem as is the case with FORTRAN, BASIC or PASCAL and all we have to do is just simply to input the sentences as we do on a typewriter. Once the situation or the condition is thus given, the computer infers and returns an appropriate answer.

TABLE 1 Top Event and Basic Events

TOP EVENT=TRANSVERSE WELD CRACKING

1=HOLD TIME AT PEAK TEMPERATURE	18=USAGE OF FIXTURE
2=HEATING RATE	19=BOUNDARY CONDITIONS OF JOINT
3=CARBON	20=DIMENSIONS OF MEMBERS
4=MANGANESE	21=TYPE OF JOINT
5=NICKEL	22=PREHEATING
6=CHROMIUM	23=INTERLAYER OR INTERPASS TEMPERATURE
7=MOLYBDENUM	24=POSTHEATING
8=OTHER HARDENABLE ELEMENTS	25=INITIAL TEMPERATURE OF STEEL
9=WIRE	26=THERMAL RADIATION FROM SURFACE
10=FLUX	27=HUMIDITY IN WELDING ENVIRONMENT
11=WELDING SPEED	28=SPECIFIC HEAT
12=WELDING CURRENT	29=THERMAL CONDUCTIVITY
13=WELDING VOLTAGE	30=DENSITY
14=THICKNESS	31=STRUCTURAL DISCONTINUITY
15=NUMBER OF LAYERS OR PASSES	32=GRAIN BOUNDARY
16=DEPOSITION OR WELDING SEQUENCE	33=NONMETALLIC INCLUSION
17=TYPE OF GROOVE	34=LATTICE DEFECT

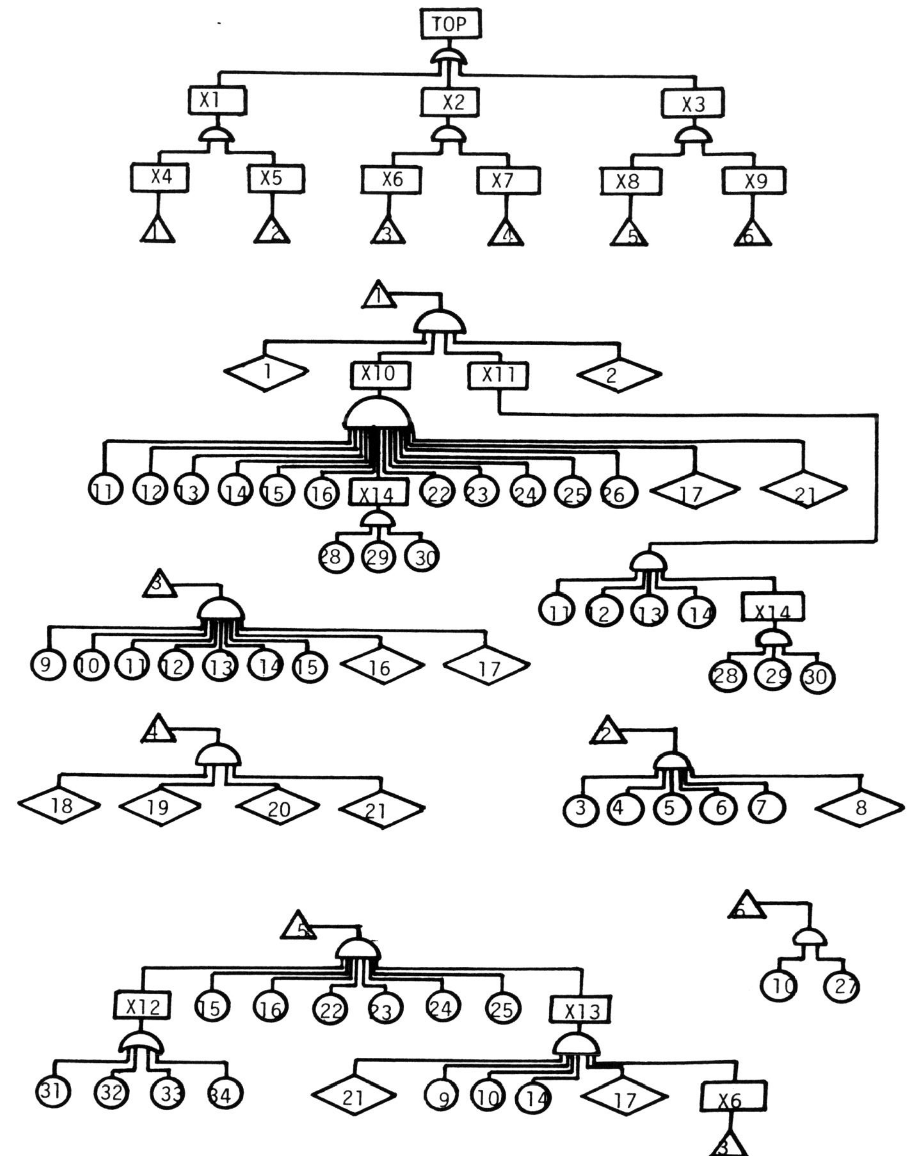

Fig. 2 Fault tree for transverse weld cracking in a heavy section low alloy steel

TABLE 2 Intermediate Events

X1=HARDENING OF HAZ	X8=HYDROGEN DIFFUSION IN WELD ZONE
X2=INTENSITY OF RESTRAINT	X9=HYDROGEN CONTENT IN WELD ZONE
X3=HYDROGEN	X10=COOLING RATE
X4=WELDING THERMAL CYCLE	X11=PEAK TEMPERATURE
X5=HARDENABILITY OF MATERIAL	X12=DEFECT
X6=INTERNAL CONSTRAINT	X13=LOCAL STRESS
X7=EXTERNAL CONSTRAINT	X14=THERMAL PROPERTIES OF MATERIAL

SUMMARY

It is discussed that the programming language Prolog provides a very useful and versatile tool for constructing a fault tree for fracture-related problems, where the process of developing a fault tree often requires a trial and error approach and full descriptions of the contents of nodes are more often than not necessary. The advantage of using Prolog is that we can program easily and communicate with a computer more freely without worrying about any detailed aspects of programming rules or grammars. Thus, Prolog cuts down the time and trouble in developing a fault tree and it provides us with a good conversational tool. But what should be emphasized most is that Prolog injects inferring ability into a computer. Therefore, it provides a very good man-machine interface for making inferences and arriving at an adequate conclusion.

ACKNOWLEDGEMENTS

The author would like to thank Prof. Eiiti Wada, Dr. Hideyuki Nakashima, and Mr. Michio Kimura, Department of Mathematical Engineering, University of Tokyo.

REFERENCES

Barlow, R.E. and F. Proschan (1975). Statistical Theory of Reliability and Life Testing. Holt, Rinehart and Winston, Inc., New York.

Feigenbaum, E. (1977). The art of artificial intelligence:I. themes and case studies of knowledge engineering. IJCAI-5, 1014-1029.

Fukuda, S. (1980). An application of fault tree analysis to weld cracking. Trans. Japan Welding Society., 11-1, 57-61.

Fukuda, S. (1980). An application of graph theory to the safety and reliability of a pressure vessel. Proc. 4th Int. Conf. Pressure Vessel Technology, London, 33-36.

Fukuda, S. (1980). Improvement of the safety and reliability of a welded structure: an FTA approach. Proc. Int. Conf. Weld. Res. Osaka, 89-94.

Fussell, J.B. (1976). Fault tree analysis -- concepts and techniques. In E.J. Henley and J.W. Lynn (Ed.), Generic Techniques in Systems Reliability Assessment, Noordhoff, Leyden, 133-162.

Kowalski, R. (1974). Predicate logic as a programming language. Information Processing - 74. North-Holland, Amsterdam.

Motooka, T. (1981). Fifth Generation Computer Systems. North-Holland, Amsterdam.

Nakashima, H. (1983). A knowledge representation system: Prolog/KR, Technical Report METR 83-5, University of Tokyo.