Construction of a Fault Tree Using Prolog†

Shuichi FUKUDA*

Abstract

This paper shows that the programming language Prolog provides a very useful and versatile tool for constructing a fault tree for the purpose of preventing fractures. As Prolog does not discriminate between program and data and is furnished with quite a strong pattern-matching function, the use of Prolog in constructing a fault tree greatly reduces the time and trouble and a very good man-machine interface is provided.

KEY WORDS: (Fault Tree Analysis) (Fracture Prevention) (Prolog)

1. Introduction

First of all, the concept of Fault Tree Analysis (FTA) is explained briefly. FTA is a kind of graphical representation of the failure logic. In FTA, an undesired event (system failure) is selected as the Top Event, and this Top Event is connected through logic symbols with the lower rank events which are immediate causes of the Top Event. These lower rank events are further connected through logic symbols with the one rank lower events. These operations are repeated rank after rank until they reach Basic Events which cannot be developed any further. In FTA, the Top Event and Intermediate Events are represented by rectangles and Basic Events are represented by circles.

Fault Tree Analysis is known to be one of the most versatile and useful tools for analyzing failures of quite complicated systems, but to apply an FTA to prevent fractures, there are such difficulties as

(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 used more often than not for developing a fault tree.

(2) To describe the structure of the problem with accuracy, it is often necessary to give a detailed account of a node content. Therefore, symbolic manipulation is most essential in this field.

This paper points out that the adoption of Prolog eliminates the above difficulties in constructing a fault tree for this purpose.

2. Construction of a Fault Tree using Prolog: Simple Illustration

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 electrical circuit shown in **Fig. 1** as an example. The graphical representation of this fault tree is given in **Fig. 2.**

: (ASSERT (NO-LIGHT) (NO-CURRENT-IN-A))
: (ASSERT (NO-LIGHT) (NO-CURRENT-IN-B))
: (ASSERT (NO-CURRENT-IN-A) (SWITCH-1-

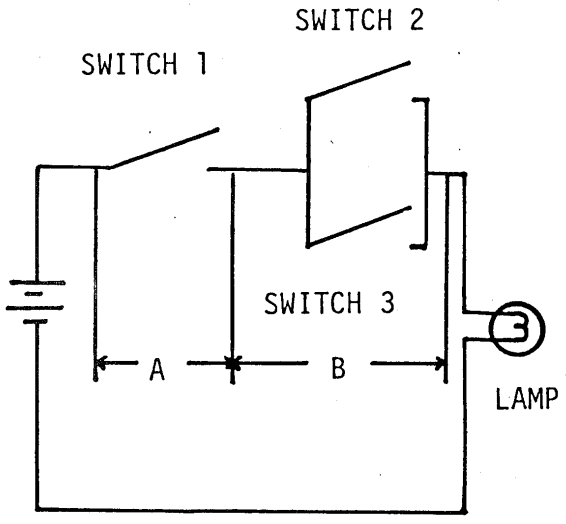

Fig. 1 Sample system

† Received on April 30, 1984
* Associate Professor

Transactions of JWRI is published by Welding Research Institute of Osaka University, Ibaraki, Osaka 567, Japan

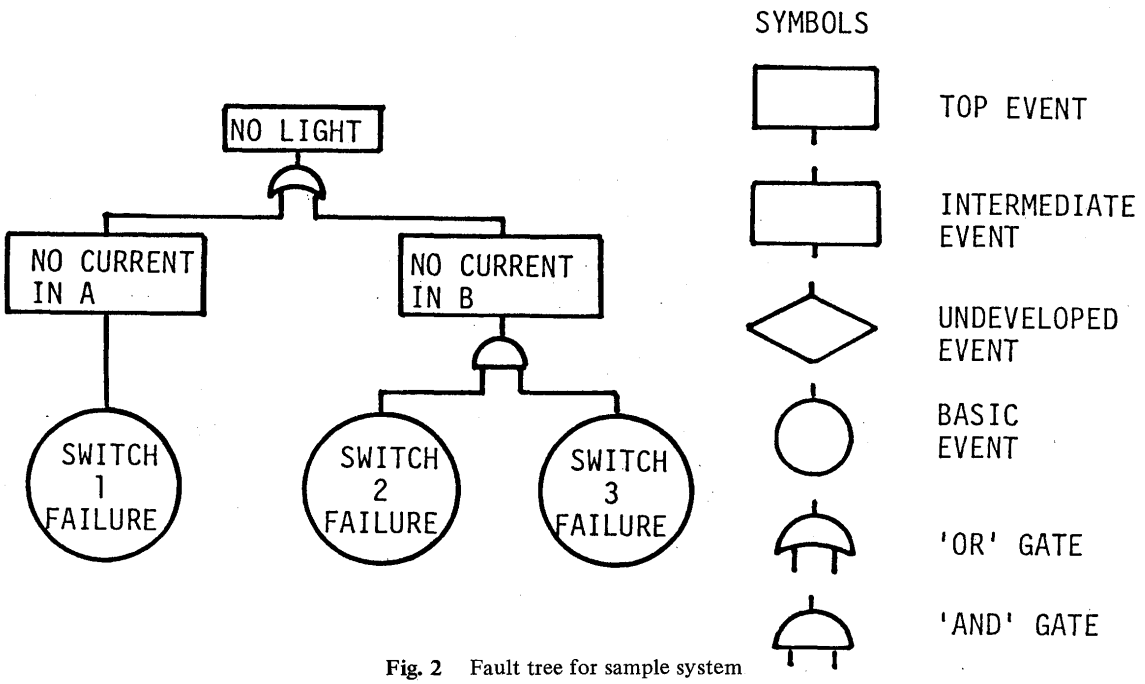

Fig. 2 Fault tree for sample system

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

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

```
: (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))
```

And then another case is examined.

```
: (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 the 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))
: (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).

3. Application to Transverse Weld Crack: An Illustrative Example

As an example of a practical application, we will con-

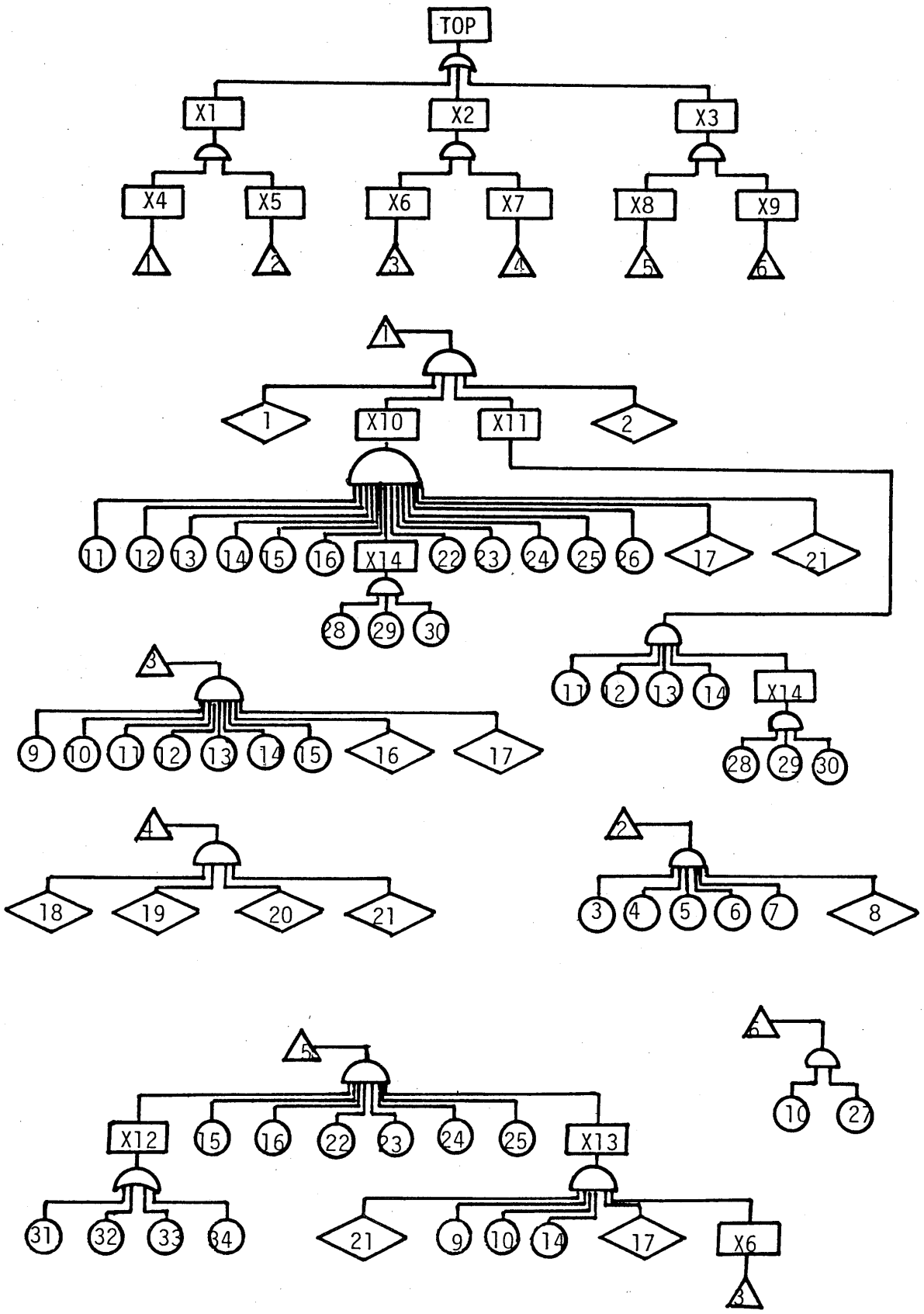

Fig. 3 Fault tree for transverse weld cracking in a heavy section low alloy steel

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

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

sider 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 immediate 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.

Fig. 3 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))
```

Thus, a fault tree is defined. And such predicates as

(MATERIAL-HARDENABILITY), (HYDROGEN-DIFUSION), etc. means that such problems related with material hardenability or hydrogen 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 (LISTING).

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

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 given, the computer carries out the pattern-matching and returns an appropriate answer.

4. Summary

It is discussed that the programming language Prolog provides a very useful and versatile tool for constructing a fault tree for the purpose of preventing fractures, 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. As Prolog does not discriminate between data and program, addition, delection or change of data and/or data structure is quite easy and further as Prolog is furnished with a strong pattern-matching function, the program can be greatly reduced. Thus the advantage of using Prolog is that it not only reduces the time and trouble in developing a fault tree, but it also provides us with a good conversational tool, since we can program easily and communicate with a computer more freely without worrying about any detailed aspects of programming rules or grammars.

References

1) H. Nakashima: A Knowledge Representation System: Prolog/KR, Technical Report METR 83-5, Department of Mathematical Engineering, University of Tokyo.

2) S. Fukuda: An Application of Fault Tree Analysis to Weld Cracking. Trans. JWS, Vol. 11 (1980), No. 1, 57–61.

3) S. Fukuda: An Application of Graph Theory to the Safety and Reliability of a Pressure Vessel. Proc. 4th Int. Conf. Pressure Vessel Technology, London, 1980, 33–36.

4) S. Fukuda: Improvement of the Safety and Reliability of a Welded Structure: an FTA Approach. Proc. Int. Conf. Weld. Res. in the 1980's, 89–94.