ARMY RESEARCH LABORATORY

Fault Tree Representation and Evaluation

by Richard Saucier

ARL-TR-2923 March 2003

NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.

NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.

Army Research Laboratory
Aberdeen Proving Ground, MD 21005-5068

ARL-TR-2923 **March 2003**

Fault Tree Representation and Evaluation

Richard Saucier
Survivability/Lethality Analysis Directorate, ARL

INTENTIONALLY LEFT BLANK.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

A *criticality analysis* is the term used to describe the vulnerability of a combat vehicle from the perspective of an automotive engineer who understands the purpose, function, and design of the vehicle. The most basic unit that is used in its construction is that of the *critical component.** A criticality analysis does not itself describe the vulnerability of its constituent components; there are other tools as well as test firings that are designed for this purpose. Instead, the focus of the criticality analysis is to describe how the components work together as a system to support the various functions of the vehicle. Much of this can be accomplished with the aid of engineering drawings and verified by nondestructive testing, where one simply disconnects wires or components and notes the effect upon vehicle function. The main emphasis is on the logical and engineering relationships of the components, rather than the components themselves. In order to deal with the engineering complexity of a modern combat vehicle, an analyst will usually approach the task in a hierarchical manner. First, critical components are identified and assembled into subsystems; next, subsystems are assembled into systems; and finally, systems are assembled into the various functions of the vehicle, such as mobility, communications, firepower, etc. Fault trees are the basic building blocks used in this construction of a criticality analysis.

This report deals with two aspects of fault trees and the role they play in a criticality analysis: how to *represent* them and how to *evaluate* them. Fault trees have been represented as *deactivation diagrams*, an example of which is shown in Figure 1.

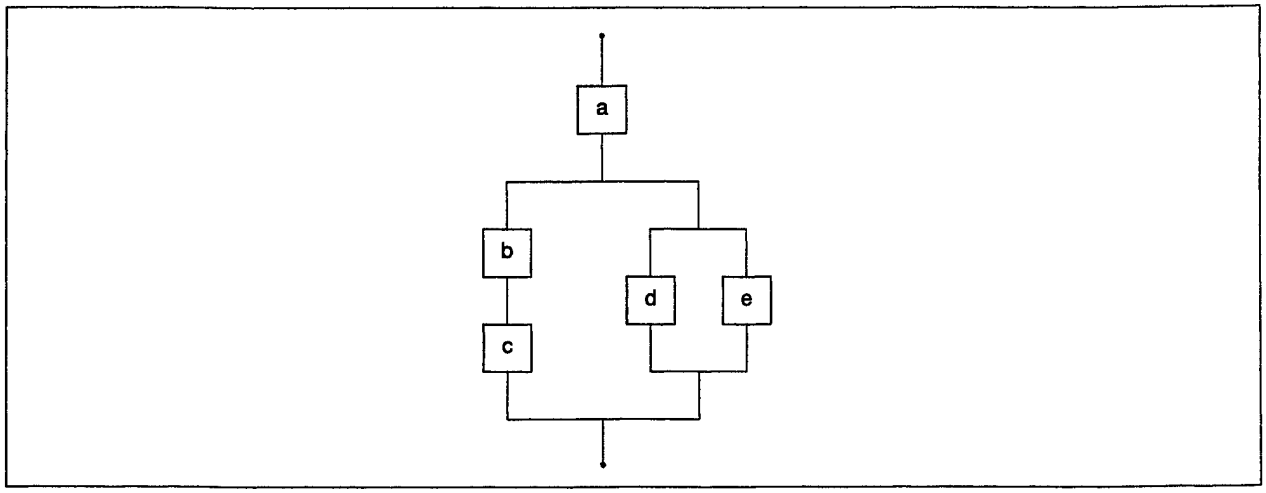


Figure 1. Example deactivation diagram.

This particular deactivation diagram is made up of five critical components, a, b, c, d, and e. The system that this deactivation diagram represents is considered fully functional if and only if there is at least one complete path from the top node to the bottom node of the diagram. If there is no such path, the system is considered completely nonfunctional. The more common term, which we shall use, is "killed." Components are also treated as having only two states: functional or killed. Now notice that this diagram has three possible paths so there is a certain amount of redundancy that helps to reduce the vulnerability. Let's suppose that components b and e are killed. Then we still have a complete path through a and d. As far as the system is concerned, it would still be considered fully functional.

Deactivation diagrams are also commonly referred to as fault trees. In this report we make the distinction that a deactivation diagram will refer to the actual diagram or figure, whereas a fault tree will represent the abstract logical structure that the diagram represents. For example, in Figure 1, components b and c are arranged in *series* and components d and e are arranged in *parallel*. One of the goals of this report is to provide a structure that captures this information without requiring a diagram. Deactivation diagrams are fine for illustrating fault trees, but that is also their limitation. Often they are just figures in a report and cannot be used by other software. Even if they are written in terms of a software drawing program, they describe the layout of the figure, not the logical relationships between components; the emphasis is on appearance rather than logical structure. The other goal of this report is to show that there is a very simple and efficient way to evaluate systems from their fault trees that avoids having to consider every possible path through the deactivation diagram.

* The critical components that are used in a criticality analysis are at the level of a Line Replaceable Unit (LRU).

2. A (VERY) BRIEF INTRODUCTION TO XML

The Extensible Markup Language (XML)* will be used to represent fault trees. Here we give a short synopsis of this language. Actually, XML is better characterized as a meta-language for creating other languages. It achieves this by providing a means of creating *markup* (also called *tags*), which can then be embedded in a document to describe its structure. Unlike HTML (Hypertext Markup Language), which has a fixed set of predefined tags, the tags in XML are open-ended and of our own choosing. The particular meaning (semantics) and rules (grammar) that we give to the tags must be described in a Document Type Definition (DTD), and so this is an ancillary file that must accompany the structured document.†

The idea of using tags to describe a document's structure was the original intention of HTML, but over the years these tags have become more commonly used for controlling the document's appearance in a Web browser. The emphasis in XML is on content rather than appearance since the whole purpose of the tags is to provide a facility for describing document structure, and as such, is independent of how it might appear in a Web browser. In fact, it is only recently that browsers have become capable of displaying XML directly. It turns out, however, that this is not really a practical limitation, since there are other means of displaying XML in Web browsers.

The syntax in XML is stricter than that of HTML. Every opening tag must have a closing tag unless it is an empty tag. An opening tag has the form

```
<element>,
```

where `element` is the name that we give to the tag. A closing tag has the form

```
</element>,
```

and an empty tag has the form

```
<element/>.
```

Tags can also have attributes, in which case they have the form

```
<element attribute="value">.
```

Document content takes on a well-defined meaning when it it sandwiched between an opening and a closing tag, and it is in this way that the document becomes structured. What makes all of this work is something called an *XML Parser*. This is a piece of software that understands tag syntax irrespective of our particular tag semantics. Parsers have been written for many different programming languages. At a minimum, parsers will automatically check for proper tag syntax. Validating parsers go further and check that the document conforms to the grammar embodied in the DTD.

This introduction barely scratches the surface, but it is enough for our purposes. The essential point is that XML allows us to construct a markup language that can be used to describe fault trees. XML, along with its associated technologies, is a burgeoning area in the software industry, and there are many resources available online that provide further information.‡

3. FAULT TREE REPRESENTATION

We describe a Fault Tree Markup Language (FTML) with the following list of capabilities.

- A fault tree written in FTML will be able to generate both
  (1) the deactivation diagram, and
  (2) the *sysdef* code that MUVES [1] requires.
- It can be read (parsed) and used (interrogated and manipulated) by other programs.

---

* The complete specification of the open source language is available at www.w3.org/TR/REC-xml.

† It is also possible to include the DTD directly in the document. In general, it can have both external and internal parts and has the syntax `<!DOCTYPE RootElement SYSTEM ExternalDTD [InternalDTD]>`, where *RootElement* is the actual name of the root element, *ExternalDTD* is the name of the DTD file, and the square brackets contain any DTD definitions not contained in the external file. Note that the square brackets must be present; they do not have their common meaning of indicating an option.

‡ For example, just type in the keyword *XML* at www.google.com.

- It is written as a standard text file—which makes it easy to edit and platform independent.
- It can be verified (checked for proper syntax) and validated (checked for proper grammar).

Of course we also need to write the software that makes use of the parser, extracts the information and then displays it in various ways from deactivation diagrams to sysdef computer code. We will show how this can be done in a Web-scripting language. But our first task is to describe the language itself.

3.1 Fault Tree Markup Language (FTML)

Fault trees are composed of critical components, or other fault trees, that are arranged in a combination of series and/or parallel structure. As such, we need a language that can be used to describe both a fault tree and a component (which will then act as nouns of the language). We also need to describe series and parallel arrangement of components (verbs). Finally, the components and fault trees have certain attributes, such as a name and an identifier (adjectives). XML is ideally suited for this purpose. As we already mentioned, tags defined through XML can be used to describe the content of a document rather than its formatting appearance. In our case the content is precisely the fault tree structure itself.

First, we define the semantics of the language by defining valid tags and attributes, as summarized in Table 1. The actual grammar of the language is embodied in the DTD, to be described later (see section 3.2).

Table 1. Tags and attributes of FTML.

Meaning	Element	Attributes[a]	Value Type	Example

Fault Tree	`ft`	`id`
		`name`
			`string`
			`string`
				`<ft id="5" name="fuel_system">`
				`...`
				`</ft>`

Component	`c`	`ft`
		`id`
		`name`
			`( true | false )`
			`string`
			`string`
				`<c id="1234" name="left_track"/>`
				The default is `ft="false"`
				Notice that this is an empty tag.

Series	`s`	*none*	*not applicable*	`<s>`
				`...`
				`</s>`

Parallel	`p`	*none*	*not applicable*	`<p>`
				`...`
				`</p>`

[a] The order of the attributes is immaterial. For example, `<c id="1" name="a"/>` can also be written as `<c name="a" id="1"/>`.

These four tags and the three attributes are all there is to the language. This makes FTML very easy to learn, and perhaps the quickest way to learn it is to simply code some fault trees. The simplest system is one that contains only one component. An example of this is the supercharger system on the Paladin [2]; its deactivation diagram is depicted in Figure 2.

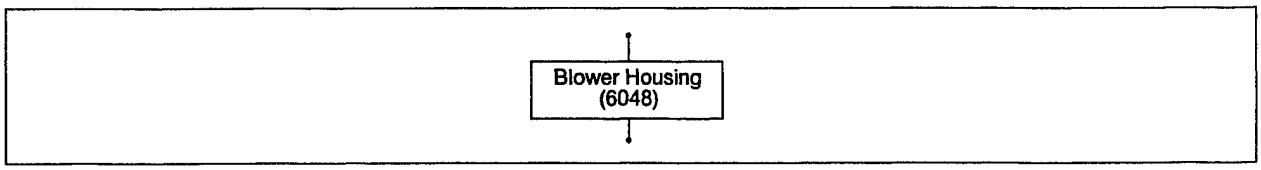


Figure 2. Example of a single-component deactivation diagram.

The component's *name* is Blower Housing and the number 6048 is referred to as the component's *ident*, which is a unique identifier. This fault tree expressed in FTML is as follows:

```
<?xml version="1.0"?>
<!DOCTYPE ft SYSTEM "ft.dtd">
<ft id="9" name="supercharger_system">
   <c id="6048" name="blower_housing"/>
</ft>
```

The first two lines of this file are boilerplate and will be the same for all fault trees. The first line indicates which version of XML to use. (Currently, there is only one version.) The second line indicates that the root tag is `ft` and

that the full DTD is located in a file called `ft.dtd`. The `id` of the fault tree should be a unique identifier. A convenient choice—and the one used here—is the figure number of the deactivation diagram as published in the criticality analysis report.

The next simplest system is that of two components in series. An example of this is the left-front shock absorber system of the Paladin, as shown in Figure 3.

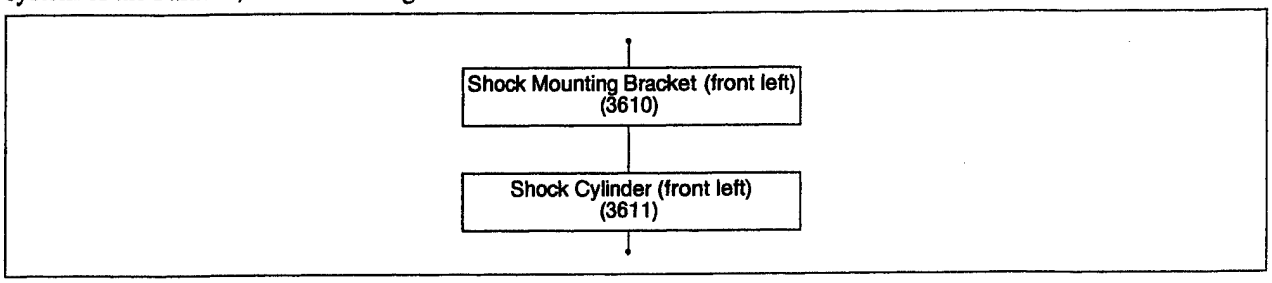


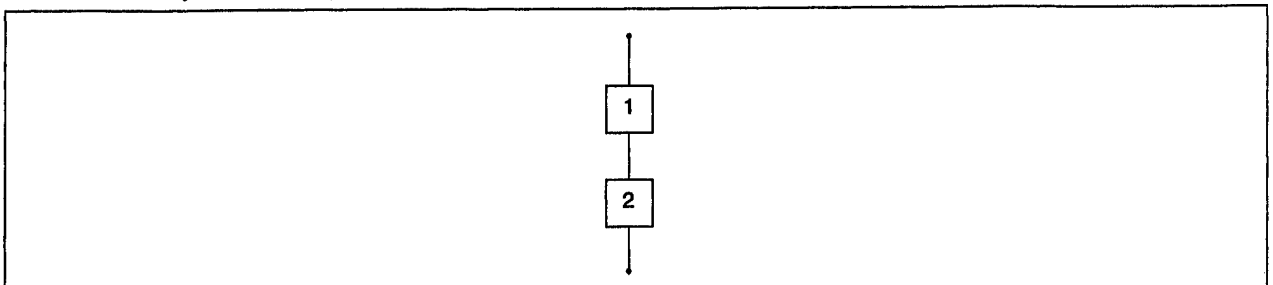
Figure 3. Example of a simple series deactivation diagram.

This fault tree expressed in FTML is as follows:

```
<?xml version="1.0"?>
<!DOCTYPE ft SYSTEM "ft.dtd">
<ft id="16" name="shock_absorber_left_front_system">
   <s>
      <c id="3610" name="shock_mounting_bracket_front_left"/>
      <c id="3611" name="shock_cylinder_front_left"/>
   </s>
</ft>
```

Notice that the two component tags are simply sandwiched between an opening and a closing series tag.

Before we go on, let's make some adjustments in the deactivation diagrams. Notice that the component names can get rather long, which forces us to use larger boxes (or smaller fonts). This can result in diagrams that span more than one page (or have names that are difficult to read). To reduce the size of the diagrams, and to concentrate on the relationship between components, we will use boxes of a standard size and list component names in a table, which will be keyed to the diagram itself. Figure 4 shows the above diagram in this standardized form.



Fault Tree 16: `shock_absorber_left_front_system`

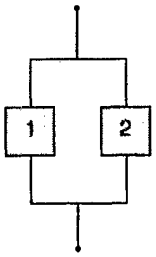
List of Critical Components/Systems

Label	ID	Name
1	3610	shock_mounting_bracket_front_left
2	3611	shock_cylinder_front_left

Figure 4. Simple series deactivation diagram in standard form.

We will also adopt the convention of always using lower case for names and using underscores instead of spaces. Spaces are perfectly acceptable in FTML, but underscores will prove to be convenient later in case we want to use the name as a variable in the sysdef code.

Next, is an example of a system consisting of two components in parallel, as shown in Figure 5.



Fault Tree 66: `fire_control_system`		
List of Critical Components and/or Systems		
Label	**ID**	**Name**
1	69	automatic_fire_control_system
2	67	backup_fire_control_system

Figure 5. Example of a simple parallel deactivation diagram.

Shaded boxes will indicate that the "component" is actually a system with it's own fault tree.*

This fault tree expressed in FTML is as follows:

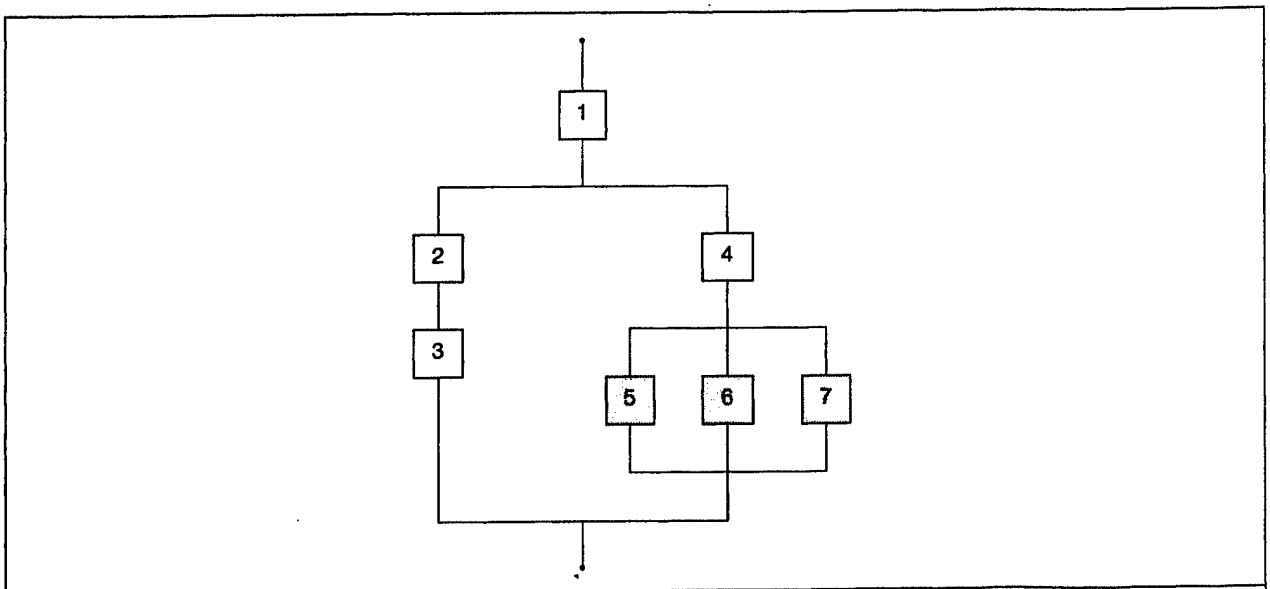
```
<?xml version="1.0"?>
<!DOCTYPE ft SYSTEM "ft.dtd">
<ft id="66" name="fire_control_system">
   <p>
      <c ft="true" id="69" name="automatic_fire_control_system"/>
      <c ft="true" id="67" name="backup_fire_control_system"/>
   </p>
</ft>
```

Notice that the system names are underlined, indicating that these are clickable links when displayed in a Web browser. Clicking on them brings up the corresponding deactivation diagram for that system. Also notice that the attribute `ft` has been set to `true` to indicate that these "components" are systems in their own right. (The default value for this attribute is `false`.) This attribute will prove to be essential when we wish to determine the proper order for evaluating the fault trees.

Notice that the syntactic structure of this fault tree is virtually identical to the series fault tree. The only difference in the FTML is that now the component tags are enclosed between parallel tags instead of series tags. Notice that series and parallel tags have no attributes. It may also be worth mentioning again that the attributes in components and systems can be in any order.

Next we consider a system that has a combination of both series and parallel arrangement of components. An example of this is shown in Figure 6, which was taken from the Paladin criticality analysis [3].

* There is no hard-and-fast rule for the distinction between components and systems, as it depends upon the context. In practice, anything that is represented with a deactivation diagram is considered a system. The authors of the criticality analysis on the Paladin [3] also adopted the convention of using the suffix *system* to aid the reader.



Fault Tree 27: left_segment_board_power_system		
List of Critical Components and/or Systems		
Label	**ID**	**Name**
1	9142	left_segment_board
2	9160	external_receptacle
3	9090	harness_ext_receptacle_to_left_segment_board
4	9027	power_lead_to_left_segment_board
5	26	slave_receptacle_power_system
6	25	battery_supplied_power_system
7	24	engine_suppled_power_system

Figure 6. Example of a deactivation diagram with both series and parallel.

This fault tree expressed in FTML is as follows.

```
<?xml version="1.0"?>
<!DOCTYPE ft SYSTEM "ft.dtd">
<ft id="27" name="left_segment_board_power_system">
   <s>
      <c id="9142" name="left_segment_board"/>
      <p>
         <s>
            <c id="9160" name="external_receptacle"/>
            <c id="9090" name="harness_ext_receptacle_to_left_segment_board"/>
         </s>
         <s>
            <c id="9027" name="power_lead_to_left_segment_board"/>
            <p>
               <c ft="true" id="26" name="slave_receptacle_power_system"/>
               <c ft="true" id="25" name="battery_supplied_power_system"/>
               <c ft="true" id="24" name="engine_suppled_power_system"/>
            </p>
         </s>
      </p>
   </s>
</ft>
```

The knack of coding fault trees comes with experience, but one can take either a top-down or a bottom-up approach, or some combination of the two. For example, with this diagram, we might find it easiest to use a bottom-up approach by first handling the parallel structure of components 5, 6, and 7, then combine that result with component 4 in a series structure, then turn to the left path and combine components 2 and 3 in series, then combine the two parallel paths, and finally combine that result with component 1 in series. The top-down approach, on the other

hand, would first recognize that what we have essentially is component 1 in series with some combination of the rest. The combination is then recognized abstractly as a parallel structure. The parallel structure would then be expanded in terms of two paths, each of which is in series, etc. In the bottom-up approach, we are regrouping and collapsing, whereas in the top-down approach we begin with the overall abstract form and expand out from there. Both methods of coding use a systematic stepwise approach. Which approach to use is simply a matter of taste; the final result is what is important, not how we get there. Later, we will describe tools that will allow us to generate the deactivation diagram, and this will then provide visual confirmation that the fault tree is being coded properly.

Any mistakes that may arise in coding fault trees are likely to be due to lapses in concentration rather than any lack of understanding of how to do it. This is because the coding process is mostly common sense and doesn't require any specialized knowledge. The rules used in coding fault trees, which are embodied in the DTD, are therefore mostly to avoid these types of errors.

3.2 FTML Content Rules

There are a few simple content rules for writing valid FTML.

- `ft` There must be exactly one `ft` (fault tree) tag per fault tree file. This is the root tag. After the first two lines of boilerplate, there must be an opening `ft` tag and the last line of the file must be the closing `ft` tag. It is a requirement of XML that there be only one root tag.
- `c` There must be at least one `c` (component) tag per fault tree file. If there are two or more `c` tags, they must be enclosed by either an `s` or a `p` tag.
- `s` An `s` (series) tag must enclose either
  - a `c` followed by another `c` or a `c` followed by a `p`, or
  - a `p` followed by either a `c`, `s`, or another `p` tag.
- `p` A `p` (parallel) tag must enclose either
  - a `c` followed by either another `c`, an `s`, or a `p`, or
  - an `s` followed by either a `c`, another `s`, or a `p`, or
  - a `p` followed by either a `c`, an `s`, or another `p`.

The complete set of rules for FTML is embodied in the DTD shown here:

```
<!ELEMENT   ft      ( c | s | p ) >
<!ATTLIST   ft
            id      NMTOKEN     #REQUIRED
            name    CDATA       #REQUIRED
>
<!ELEMENT   s       (
                      ( c, ( c | p )+ ) |
                      ( p, ( c | s | p )+ )
                    )
>
<!ELEMENT   p       (
                      ( c, ( c | s | p )+ ) |
                      ( s, ( c | s | p )+ ) |
                      ( p, ( c | s | p )+ )
                    )
>
<!ELEMENT   c       EMPTY     >
<!ATTLIST   c
            ft      ( true | false )    "false"
            id      CDATA               #REQUIRED
            name    CDATA               #REQUIRED
>
```

This file, although cryptic, concisely and completely describes valid tags, attributes, and content. Once a fault tree has been written in FTML, it can be validated against the DTD. For example, a free service is provided by the Scholarly Technology Group at Brown University (www.stg.brown.edu/pub/xmlvalid/), which allows one to validate an XML document online. Not only does it check for syntax errors, but it will also check that the document conforms to the rules expressed in the DTD. Current versions of Web browsers that have the capability of displaying

XML will also display FTML.* However, they will only check for proper syntax, not the more stringent requirement of conforming to the DTD.

3.3 Criticality Analysis

Once all the fault trees have been coded, they need to be assembled into a criticality analysis file. This is a necessary step before software can analyze dependencies between fault trees. Assuming that the individual fault trees reside in files with names of the form `ft.*.ftml`, then the following shell script, `ft2ca`, will assemble them into a criticality analysis file called `ca.ftml`:

```
#!/bin/sh
# ft2ca: Assemble the criticality analysis file from the individual fault tree
#        files.  The individual fault tree files must have names of the form
#        "ft.*.ftml", where "*" is usually the figure number of the fault tree
#        as it appears in a report.  This script assembles the fault trees in
#        ascending order on figure number and writes to stdout.

echo '<?xml version="1.0"?>'
echo '<!DOCTYPE ca SYSTEM "ca.dtd">'
echo '<ca name="Criticality Analysis of Paladin M109A3E2">'
for file in `ls ft.*.ftml | sort -n -t. +1`
do
   cat $file | grep -v "version" | grep -v "DOCTYPE"
done
echo '</ca>'
```

This sample script is for the Paladin; the name should be changed as appropriate for another vehicle. The criticality analysis DTD is basically the fault tree DTD with the change that the root tag is now `ca` instead of `ft`:

```
<!ELEMENT ca    ( ft )+ >
<!ATTLIST ca
          name CDATA          #REQUIRED
          ref  CDATA          #IMPLIED
          note CDATA          #IMPLIED
>
<!ELEMENT    ft      ( c | s | p ) >
<!ATTLIST    ft
             id      NMTOKEN    #REQUIRED
             name    CDATA      #REQUIRED
>
<!ELEMENT    s       (
                       ( c, ( c | p )+ ) |
                       ( p, ( c | s | p )+ )
                     )
>
<!ELEMENT    p       (
                       ( c, ( c | s | p )+ ) |
                       ( s, ( c | s | p )+ ) |
                       ( p, ( c | s | p )+ )
                     )
>
<!ELEMENT    c       EMPTY     >
<!ATTLIST    c
             ft      ( true | false )   "false"
             id      CDATA              #REQUIRED
             name    CDATA              #REQUIRED
>
```

The `ca` tag has three attributes: `name`, which is required; `ref`, which is an optional reference such as a published report; and `note`, which is an optional note which documents the criticality analysis (such as assumptions or limitations). A criticality analysis must consist of at least one fault tree. Once the `ca.ftml` file has been constructed, the software described in this report no longer has a need for the individual fault tree files and they can be safely deleted.

* Incidentally, the display of FTML by current browsers looks very much like what is in the fault tree file, including indentation (even if indentation is not in the original), and there is a good reason for this: The content of FTML *is* the structure. That is to say, the only structure that the browser knows about is the tag structure based upon tag syntax, and that is precisely the tree structure of the fault tree. This is another reason why fault trees are such a natural application for the use of XML.

3.4 Fault Tree Tools

FTML is an XML-based markup language for describing and storing fault trees. It is a very simple language, consisting of only four tags; and yet these four tags are all that is necessary to code any set of fault trees in a criticality analysis, no matter how complex. As a language, FTML does not require the Web—provided we still have an XML parser! And up to this point, the only use we have made of FTML is to provide a structure so that fault trees can be described and stored as text files. But the fact that XML is so closely associated with the Web means there are substantial benefits if we take advantage of this fact. And so the Web-scripting language of PHP* has been used to design some Web-based tools that extend the usefulness of FTML. These tools are able to display a criticality analysis in a Web browser, as both a deactivation diagram and as MUVES sysdef code. In either display mode, system names are formatted as hyperlinks, in order to facilitate navigating through the criticality analysis. Using the fault tree in Figure 6, here is an example of how this tool displays sysdef code:

```
left_segment_board_power_system =
left_segment_board
|
(
  (
    external_receptacle
    |
    harness_ext_receptacle_to_left_segment_board
  )
  &
  (
    power_lead_to_left_segment_board
    |
    (
      slave receptacle power system
      &
      battery supplied power system
      &
      engine suppled power system
    )
  )
);
```

This Web-based tool also has the capability to analyze a criticality analysis and perform the following functions:

- List all the critical components along with their idents;
- List all the systems along with their IDs;
- Search for a specific component and list all its idents;
- Search for a specific system and list all its components;
- Find all the systems that contain a given component;
- Find all the systems that contain a given system as a subsystem;
- Find all the components that occur more than once along with the systems in which they occur;
- Find all the systems that contain repeated components.

The tool will also analyze dependencies in order to display the systems, and the sysdef, in the proper order for evaluation. If there are cyclic dependencies among the systems, this will be detected and all the systems involved will be listed. Repeated components and cyclic dependencies among systems become important when it comes to actually evaluating fault trees (as we show in the next section).

A Web browser provides an excellent environment for navigating through a set of fault trees in a criticality analysis and should be particularly helpful in the actual construction of the criticality analysis.† However, it is not the best medium if one wants a record of the criticality analysis that can be included in a report. To satisfy this need, the tool was extended to provide the option of saving the criticality analysis as a PDF document (see Figure 15 on p. 22).

* This is an open-source language that is available online at www.php.net. Originally, PHP stood for Personal Home Page but has now taken on the (recursive) meaning of PHP Hypertext Preprocessor.

† Notice that the tools described in this report are all based upon the FTML text file, which must be constructed first before the deactivation diagram can be displayed. Another tool has been developed recently that facilitates the construction of deactivation diagrams by providing a graphical interface, and also has the capability of writing out the FTML [4].

Finally, in order to facilitate the conversion to FTML, another tool has been developed which converts legacy sysdef code to FTML.

4. FAULT TREE EVALUATION

Evaluation of a fault tree means a determination of whether the system it represents is functional (not killed) or non-functional (killed), given the state of each of its constituent components (as either killed or not killed). There are two different ways that a fault tree can be evaluated. One is based on Monte Carlo sampling, and the other is based on probability algebra.

4.1 Monte Carlo Sampling

In Monte Carlo sampling, components and systems are represented as having only two states: either killed or not killed. We will use `1 (true)` to represent a killed state and `0 (false)` to represent a non-killed state. Given that the probability of killing component i is p_i, one performs a Bernoulli trial on each component by drawing a uniform random number r_i between 0 and 1—more precisely, $r_i \in [0, 1)$—and assigning the state k_i of the component according to

$$k_i = \begin{cases} 1 \text{ if } r_i < p_i & \text{(killed)}, \\ 0 \text{ if } r_i \geq p_i & \text{(not-killed)}. \end{cases} \tag{1}$$

Once all the components are assigned, the state of the fault tree itself can be determined. This is accomplished by breaking it down into its arrangement of series and/or parallel subsystems and systematically evaluating each of these. First consider a system consisting of just two components, `a` and `b`, arranged in series. The system will be killed (i.e., no complete path from top node to the bottom node of the deactivation diagram) if and only if either `a` is killed, or `b` is killed, or both are killed. Next consider a system consisting of two components, `a` and `b`, arranged in parallel. This system will be killed if and only if both `a` and `b` are killed. This means that the OR operator, `|`, will represent two components in series and the AND operator, `&`, will represent two components in parallel, as demonstrated in Table 2. For completeness, we also include the XOR (exclusive or) operator, `^`, which is true only when one or the other is true, but not both.

Table 2. Truth table for series, parallel, and exclusive OR.

Component		Series	Parallel	Exclusive OR
`a`	`b`	`a | b`	`a & b`	`a ^ b`
0	0	0	0	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	0

The beauty of this method is its simplicity. For example, the C code to evaluate the fault tree depicted in Figure 1 is

```
system = a | ( ( b | c ) & ( d & e ) );
```

Parentheses perform the function of grouping the components into simple series and parallel subsystems. In this way, it is very straightforward to write the C code to evaluate the most complicated fault tree, since it is nothing more than components arranged in nested series and parallel. Indeed, one of the tools will do this automatically from the FTML description of the fault tree. The downside of this approach is that one needs to sample the components and evaluate the system a number of times from the given component kill probabilities in order to generate statistically significant results. We will continue this discussion of the pros and cons of the Monte Carlo approach after we have first described the probability algebra approach.

4.2 Probability Algebra

In this approach we use the component kill probabilities directly. If p_a represents the probability that component a is killed, and p_b represents the probability that component b is killed, then the kill probabilities for series, parallel, and exclusive OR arrangement are given in Table 3. It should be emphasized that these formulas are strictly valid only when there is independence (i.e., no correlation) among the individual kill probabilities.

Table 3. Kill probabilities for series, parallel and exclusive OR.

Combination	Expression
Series	$1-(1-p_a)(1-p_b)$
Parallel	$p_a p_b$
Exclusive OR	$p_a(1-p_b)+p_b(1-p_a)$

To make these expressions as simple to use and as transparent to verify as the Boolean expressions, we encapsulate the probability algebra into a C++ class,* as summarized in Table 4.

Table 4. Probability algebra with the Prob class.

Operation	Mathematical Notation	Computer Code
Definition	Let p be a probability, where $0 \le p \le 1$	`Prob p;` `assert(0<=p&&p<=1);`[a]
	Let $p = 0.5$	`Prob p( 0.5 );` *or* `Prob p = 0.5;`
Logical OR	$1-(1-p_a)(1-p_b)$	`a | b;`
Logical OR assignment	$p_a \Leftarrow 1-(1-p_a)(1-p_b)$	`a |= b;`
Logical AND	$p_a p_b$	`a & b;`
Logical AND assignment	$p_a \Leftarrow p_a p_b$	`a &= b;`
Logical EXCLUSIVE OR	$p_a(1-p_b)+(1-p_a)p_b$	`a ^ b;`
Logical EXCLUSIVE OR assignment	$p_a \Leftarrow p_a(1-p_b)+(1-p_a)p_b$	`a ^= b;`
Logical NOT	$1-p$	`!p;` *or* `~p;`
Input a probability p	*NA*	`cin >> p;`
Output a probability p	*NA*	`cout << p;`
Assign one probability to another	Let $p_b = p_a$ *or* $p_b \Leftarrow p_a$	`b = a;` *or* `b( a );`
Addition	$p_a + p_b$	`a + b;`
Addition assignment	$p_a \Leftarrow p_a + p_b$	`a += b;`
Subtraction	$p_a - p_b$	`a - b;`
Subtraction assignment	$p_a \Leftarrow p_a - p_b$	`a -= b;`
Multiplication by a scalar s	$p_b = s\,p_a$ *or* $p_b = p_a\,s$	`b = s * a;` *or* `b = a * s;`
Multiplication assignment	$p_a \Leftarrow s\,p_a$ *or* $p_a \Leftarrow p_a\,s$	`a *= s;`
Division by a scalar s	p/s	`p / s;`
Division assignment	$p \Leftarrow p/s$	`p /= s;`
Check for equality	Is it true that $p_a = p_b$?	`a == b;`
Check for inequality	Is it true that $p_a \neq p_b$?	`a != b;`

[a] It is not necessary for the user to explicitly code this assertion, as the Prob class constructor enforces it automatically.

Note: *NA* = not applicable.

---

* Defining a class in C++ effectively extends the language, so that variables that are now declared `Prob` can be used just as easily as native types such as `int` or `bool`.

This C++ class gives us the ability to use the same simple expressions in the computer code regardless of whether the components are being represented by discrete Boolean states or by continuous probabilities. This makes it very easy to verify fault tree coding— where we would otherwise be confused by a sea of parentheses. To take a simple example, the coding of Figure 1 is $p = 1-(1-p_a)(1-(1-(1-p_b)(1-p_c))p_d p_e)$, using parentheses only where they are essential! Using the `Prob` class, `p = a | ( ( b | c ) & ( d & e ) );`.

However, there is an important issue that must be addressed in this probability approach—which is not an issue with the Monte Carlo approach— and that is the problem of repeated components.

4.3 How to Handle Repeated Components

Consider the deactivation diagram shown in Figure 7, where component a has been repeated.

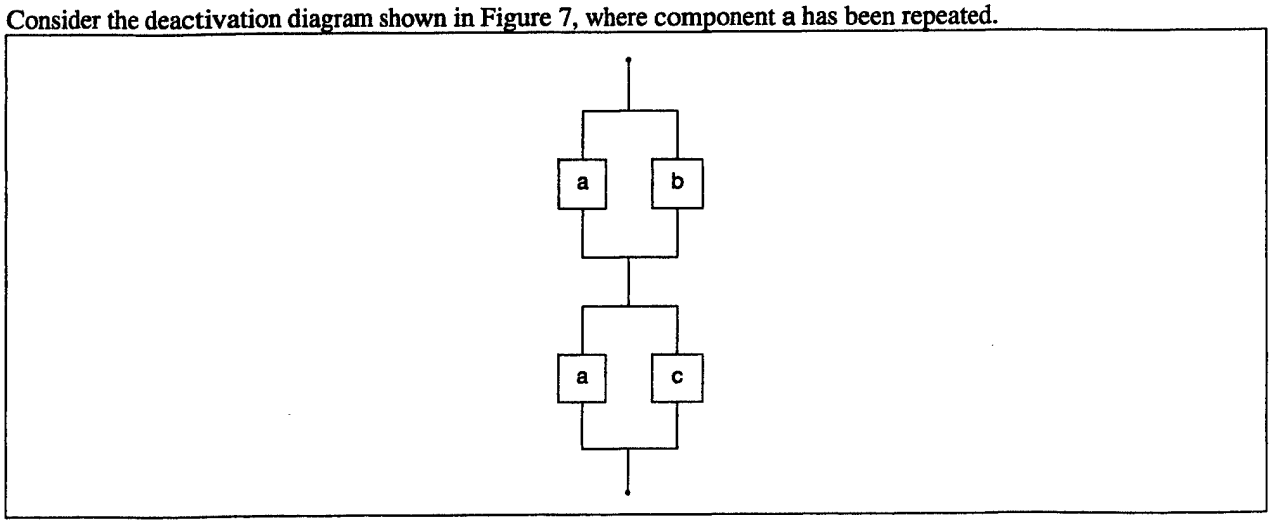


Figure 7. Deactivation diagram with repeated component.

The Bradley A2 degraded states [5] deactivation diagrams are expressed in this manner. It's important to realize that the diagram expresses the logical arrangement of components, and not their actual physical arrangement. One will get the wrong answer if this is not taken into account. If component a gets killed, for instance, then it gets killed everywhere. To properly evaluate fault trees with repeated components, we factor out the repeated component and systematically reduce the fault tree to a sum of simpler fault trees that have no repeated component. We can factor out the dependence on component a, as depicted in Figure 8.

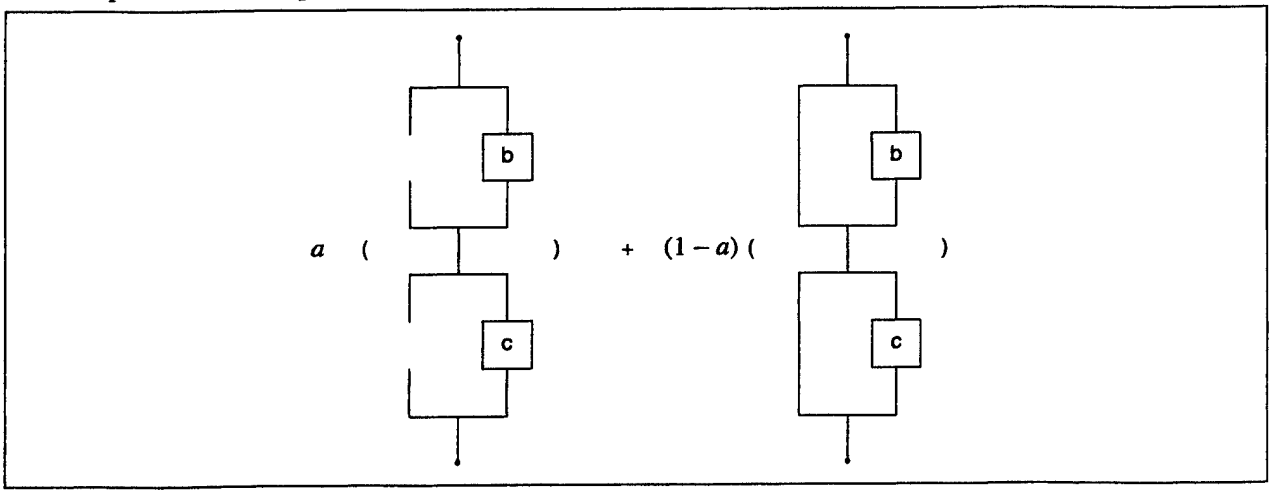


Figure 8. Decomposition of repeated component diagram.

The component a is either killed, in which case we have the diagram on the left, or it is not killed, in which case we have the diagram on the right. The probabilities are additive because this decomposition represents a partition of the probabilities into disjoint cases of either a (with probability a) or !a (with probability $1-a$). The diagram on the left is killed (i.e., no complete path from top node to bottom node) if and only if either b or c is killed. The diagram on

the right, on the other hand, already has a completed path, regardless of the state of b or c, and thus cannot be killed (i.e., its kill state is 0). Therefore, the diagram evaluates to

$$a\,(\,\texttt{b}\mid\texttt{c}\,)+(1-a)\,(\,\texttt{0}\,)\;\Rightarrow\;\texttt{a}\;\&\;(\,\texttt{b}\mid\texttt{c}\,).$$

This final expression shows that the original diagram is equivalent to the one shown in Figure 9.

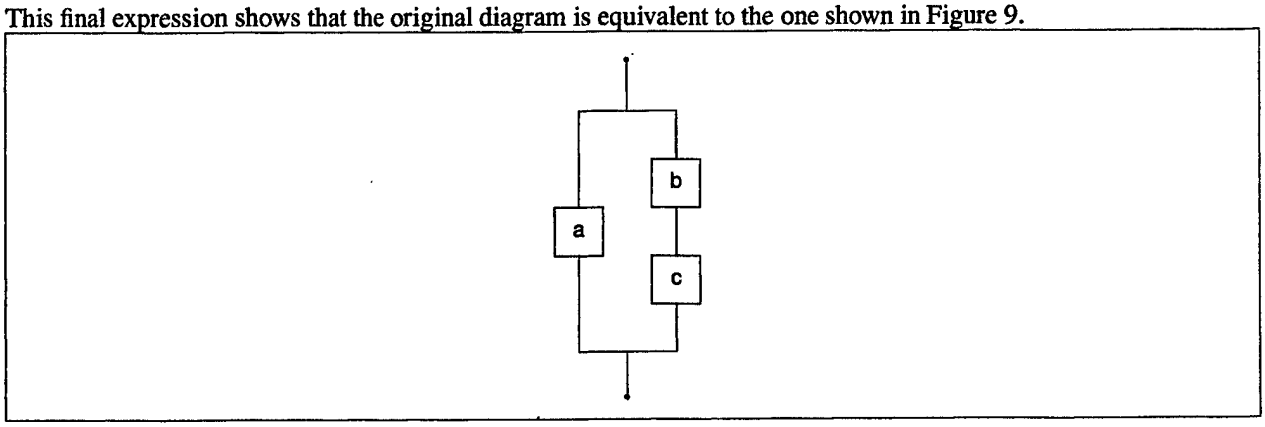


Figure 9. Equivalent diagram without repeated component.

Now, this example is simple enough that we could easily recognize the diagram in Figure 7 could be replaced by the diagram in Figure 9. But the fact remains that this method of factorization of repeated components and systematic decomposition into simpler diagrams can always be carried out. Formally, if A represents the diagram in Figure 7, and $f(a,b,c)$ is the function defined by the expression `( a & b ) | ( a & c )`, then

$$\begin{aligned}
A &= f(a,b,c)\\
&= af(1,b,c)+(1-a)f(0,b,c)\\
&= a[(\,\texttt{1 \& b}\,)\mid(\,\texttt{1 \& c}\,)]+(1-a)[(\,\texttt{0 \& b}\,)\mid(\,\texttt{0 \& c}\,)]\\
&= a[\texttt{b}\mid\texttt{c}]+(1-a)[\texttt{0}\mid\texttt{0}]\\
&= a(\texttt{b}\mid\texttt{c})\\
&\Rightarrow \texttt{a \& (b | c)},
\end{aligned}\tag{1}$$

where we have made use of the correspondence between the Boolean expression and probability algebra summarized in Table 5.

Table 5. Correspondence between Boolean expression and probability algebra.

Code	Boolean Expression	Probability Algebra
a & b	a & b	ab
1 & a	a	a
0 & a	0	0
a | b	a | b	$1-(1-a)(1-b)$
1 | a	1	1
0 | a	a	a
a ^ b	a ^ b	$a(1-b)+b(1-a)$
1 ^ a	!a	$1-a$
0 ^ a	a	a

Next, suppose that the function $f(a,b,c)$ represents a diagram where both components a and b are each repeated. Then, the factoring and decomposition proceeds as follows:

$$\begin{aligned}
f(a,b,c) &= af(1,b,c)+(1-a)f(0,b,c)\\
&= a[bf(1,1,c)+(1-b)f(1,0,c)]+(1-a)[bf(0,1,c)+(1-b)f(0,0,c)]\\
&= abf(1,1,c)+a(1-b)f(1,0,c)+(1-a)bf(0,1,c)+(1-a)(1-b)f(0,0,c)
\end{aligned}$$

resulting in four terms. Finally, suppose that all three components are repeated. This will produce $2^3 = 8$ terms:

$$f(a,b,c) = (1-a)(1-b)(1-c)f(0,0,0)+(1-a)(1-b)cf(0,0,1)+$$

$$(1-a)b(1-c)f(0,1,0) + (1-a)bcf(0,1,1) +$$
$$a(1-b)(1-c)f(1,0,0) + a(1-b)cf(1,0,1) +$$
$$ab(1-c)f(1,1,0) + abcf(1,1,1)\,.$$

This can also be written as

$$f(a,b,c) = \sum_{i=0}^{1}\sum_{j=0}^{1}\sum_{k=0}^{1} a^i(1-a)^{1-i}b^j(1-b)^{1-j}c^k(1-c)^{1-k}f(i,j,k), \tag{2}$$

with an obvious generalization to n components, $a_1, \cdots, a_n$:

$$f(a_1,a_2,\cdots,a_n) = \sum_{i_1=0}^{1}\sum_{i_2=0}^{1}\cdots\sum_{i_n=0}^{1} a_1^{i_1}(1-a_1)^{1-i_1}a_2^{i_2}(1-a_2)^{1-i_2}\cdots a_n^{i_n}(1-a_n)^{1-i_n}f(i_1,i_2,\cdots,i_n)\,. \tag{3}$$

With the Prob class, evaluation of fault trees is simple. For example, suppose that a system is defined by the following function:

```
Prob system( vector< Prob > p ) {

   return
   p[0]        // a
   |
   (
     ( p[1]    // b
       |
       p[2]    // c
     )
     &
     (
       p[3]    // d
       &
       p[4]    // e
     )
   );
}
```

This can be evaluated by simply calling the function

```
system( p );
```

But now consider the case where one or more components occur more than once in the fault tree. Let's say **b** is repeated:

```
Prob system( vector< Prob > p ) {

   return
   p[0]        // a
   |
   (
     ( p[1]    // b
       |
       p[2]    // c
     )
     &
     (
       p[1]    // b again
       &
       p[3]    // d
     )
   );
}
```

We will get the wrong answer unless we account for this repeated component. This prompts us to extend the `Prob` class and develop methods that handle any number of repeated components. The result of these efforts is summarized in Table 6.

Table 6. Probability methods with the Prob class.

Description	Mathematical Notation	Computer Code

Probability of exactly zero kills, given N individual kill probabilities p_i	$\prod_{i=1}^{N}(1-p_i)$	`vector< Prob > p;`
		`zero( p ); or`
		`none( p ); or`
		`prob( p, 0 );`

Probability that all are killed, given N individual kill probabilities p_i	$\prod_{i=1}^{N} p_i$	`vector< Prob > p;`
		`all( p ); or`
		`prob( p, p.size() );`

Probability that exactly (any) n items are killed, given N individual kill probabilities p_i, where $1 \le n \le N$	$\sum_{i_1=0}^{1}\cdots\sum_{i_N=0}^{1} p_1^{i_1}(1-p_1)^{1-i_1}\cdots p_N^{i_N}(1-p_N)^{1-i_N}$ such that $i_1+\cdots+i_n=n$	`vector< Prob > p;`
		`prob( p, n ); or`
		`prob( p, n, ANY );`

Probability that n *or more* items are killed, given N individual kill probabilities p_i, where $1 \le n \le N$	$\sum_{i_1=0}^{1}\cdots\sum_{i_N=0}^{1} p_1^{i_1}(1-p_1)^{1-i_1}\cdots p_N^{i_N}(1-p_N)^{1-i_N}$ such that $n \le i_1+\cdots+i_n \le N$	`vector< Prob > p;` `prob( p, n, OR_MORE);`
Probability that n *or less* items are killed, given N individual kill probabilities p_i, where $1 \le n \le N$	$\sum_{i_1=0}^{1}\cdots\sum_{i_N=0}^{1} p_1^{i_1}(1-p_1)^{1-i_1}\cdots p_N^{i_N}(1-p_N)^{1-i_N}$ such that $1 \le i_1+\cdots+i_n \le n$	`vector< Prob > p;` `prob( p, n, OR_LESS);`
Probability that n *adjacent* items are killed, given N individual kill probabilities p_i, where $1 \le n \le N$	$\sum_{i_1=0}^{1}\cdots\sum_{i_N=0}^{1} p_1^{i_1}(1-p_1)^{1-i_1}\cdots p_N^{i_N}(1-p_N)^{1-i_N}$ such that $i_1+\cdots+i_n=n$ and $i_1,\cdots,i_n$ are consecutive indicies	`vector< Prob > p;` `prob(p,n,ADJACENT);`
Probability that n *non-adjacent* items are killed, given N individual kill probabilities p_i, where $1 \le n \le N$	$\sum_{i_1=0}^{1}\cdots\sum_{i_N=0}^{1} p_1^{i_1}(1-p_1)^{1-i_1}\cdots p_N^{i_N}(1-p_N)^{1-i_N}$ such that $i_1+\cdots+i_n=n$ and $i_1,\cdots,i_n$ are not consecutive indicies	`vector< Prob > p;` `prob(p,n,NON_ADJACENT);`

Probability that a system $f(p_1,p_2,\cdots,p_N)$ is killed, given the individual (non-repeating) kill probabilities	$f(p_1,p_2,\cdots,p_N)$ where no p_i occurs more than once	`Prob(*f)(vector<Prob>);`
		`vector< Prob > p;`
		`f( p ); or`
		`evaluate( f, p );`

Probability that a system $f(p_1,p_2,\cdots,p_N)$ is killed, given that only one index, i, occurs more than once	$f(p_1,p_2,\cdots,p_N)$ where only index i occurs more than once	`Prob(*f)(vector<Prob>);`
		`vector< Prob > p;`
		`evaluate( f, p, i );`

Probability that a system $f(p_1,p_2,\cdots,p_N)$ is killed, given that many indicies occur more than once	$f(p_1,p_2,\cdots,p_N)$ where indicies $i_1,\cdots,i_n$ occur more than once	`Prob(*f)(vector<Prob>);`
		`vector< Prob > p;`
		`vector< int > i;`
		`evaluate( f, p, i );`

Probability that a system $f(p_1,p_2,\cdots,p_N)$ is killed, making no assumptions about repeated indicies	$f(p_1,p_2,\cdots,p_N)$ where any or all of the indicies may occur more than once	`Prob(*f)(vector<Prob>);`
		`vector< Prob > p;`
		`evaluate_all( f, p );`

With these methods, the case considered last is simply

```
evaluate( system, p, 1 );
```

since 1 is the repeated index.

4.4 How to Handle Circular Dependency of Fault Trees

Beyond the problem of lack of independence among components due to their repeated occurrence is the problem of cyclic dependence among fault trees. One finds instances of this in the Abrams M1A2 criticality analysis [6], so it is a real issue that must be dealt with. Consider a simple example, as illustrated in Figure 10.

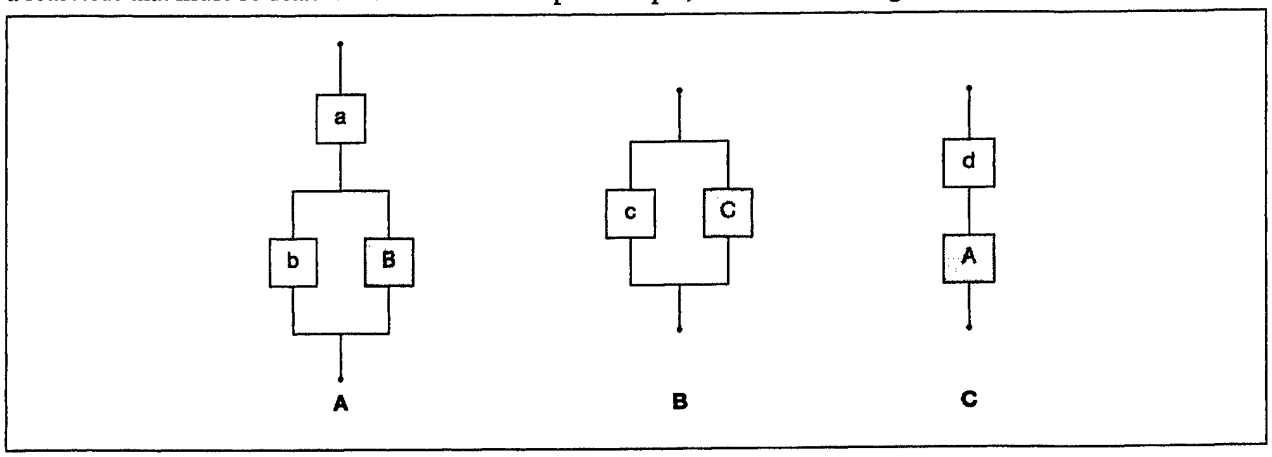


Figure 10. Example of cyclic dependence among fault trees.

In terms of Boolean logic, the state of each diagram is easy to express in code:

```
A = a | ( b & B );
B = c & C;
C = d | A;
```

But in order to evaluate system A, we need to know system B, which involves system C, and that, in turn, involves system A again. Each system is defined recursively, and we don't seem to be able to get started.

But it's also possible to write down the kill probabilities for each diagram:

$$p_A = 1 - (1 - p_a)(1 - p_b p_B),$$
$$p_B = p_c p_C,$$
$$p_C = 1 - (1 - p_d)(1 - p_A).$$

This is simply a system of three equations in three unknowns, which is easily solved:

$$p_A = 1 - \frac{(1 - p_a)[1 - p_b + p_b(1 - p_c)]}{1 - (1 - p_a)p_b p_c(1 - p_d)},$$
$$p_B = p_c - p_c(1 - p_d)(1 - p_1),$$
$$p_C = 1 - (1 - p_d)(1 - p_1).$$

So, it's clear that there is indeed a solution, and the problem is well defined. The problem, though, is that we would rather not have to perform this algebra. This example happens to be very simple; a real criticality analysis can involve many more than three fault trees, where each fault tree can involve many more components, resulting in some very messy algebra.

But the fact that the problem is well defined leads to another approach. Notice that no matter how complicated each system may be, they are still probabilities, and as such, must lie between 0 and 1. Consequently, we could try finding a solution by iteration using either a Newton-Rhapson or bisection algorithm. But a better idea is to simply use recursion. The idea here is that we initially assign kill probabilities to each of the systems involved in the cyclic dependence and then iterate a number of times until each of the system probabilities converges to a solution. It turns out that this seems to work well and is very easy to implement. For example, here is the code for this example:

```
Prob A( 0. ), B( 0. ), C( 0. );   // using 0 probabilities as a starting point
const int N_ITERATIONS = 15;      // number of iterations

for ( int i = 0; i < N_ITERATIONS; i++ ) {

   A = a | ( b & B );
   B = c & C;
   C = d | A;
}
```

By trial and error, fifteen iterations were found to be more than sufficient, not only for this example, but also for many others that were tried, including the Abrams criticality analysis which involved dozens of fault trees.

Monte Carlo sampling, in which components and systems are represented by Boolean variables, can also deal with cyclic dependence using this same principle of recursion:

```
bool A( 0 ), B( 0 ), C( 0 );   // using 0 Booleans as a starting point
const int N_ITERATIONS = 15;   // number of iterations

for ( int i = 0; i < N_ITERATIONS; i++ ) {

   A = a | ( b & B );
   B = c & C;
   C = d | A;
}
```

This effectively primes the systems, and this becomes the starting point for gathering statistics. This procedure amounts to taking $N + 15$ samples, but then performing the statistics on the last N.

4.5 Comparison of Monte Carlo Sampling and Probability Algebra

The coding for normal fault trees, which have no repeated components and no cyclic dependencies, is the same for both methods, only the variable declaration is different (`bool` for the Monte Carlo method and `Prob` for the probability algebra approach). Table 7 summarizes the differences between the two approaches.

Table 7. Comparison of Monte Carlo sampling and probability algebra.

Issue	Monte Carlo Sampling	Probability Algebra
Component Representation	Discrete *killed* or *not-killed*	Continuous kill probability
Accuracy	Increases with the number N of samples, but no better than N^{-1}	Exact
Able to handle dependencies among components?	Yes[a]	No[a]
How to handle repeated components	Done automatically	Identify components and call the appropriate class method
How to handle cyclic dependence	Slight increase in number of samples	Iterate small number of times first
Requirements	Random number generator	`Prob` class
Speed with a normal fault tree	Depends on number of samples, but each sample is very fast	Fast
Speed with repeated components	Not any slower than normal case	Could be significantly slower
Speed with cyclic dependence	Negligibly slower than normal case	Negligibly slower than normal case

[a] The component dependency issue merits some elaboration. Let's suppose that every time component a is killed, component b is not, perhaps due to mutual shielding. So every time that a is 1, b is 0, and vice versa. If the component state vector is a Boolean vector of 1's and 0's, then the Monte Carlo approach would reflect this dependency. However, the component state is usually not (never?) in this form. Rather, it is a *pk* vector of component kill probabilities. This vector contains no information regarding dependencies or correlations among the components, and, consequently, the Monte Carlo method would not be able to extract any. So, for all practical purposes with current MUVES runs, neither method handles dependencies among components because it is not contained in the input. Nevertheless, the Monte Carlo approach is capable of dealing with dependencies, whereas the probability algebra approach described here is not because it assumes independence.

5. FAULT TREE BROWSER

The Web-scripting language of PHP is particularly well-suited for writing applications based on FTML, for two primary reasons:

- PHP was and is designed for the Web, with a rich set of features that support a number of technologies including XML, graphics, and PDF.
- It is also an interpreted language—which means that it doesn't need to be compiled first—and this results in a very interactive environment for the user.

The combination of FTML and PHP makes it possible to design interactive software for displaying in a Web browser or for generating other interactive (PDF) documents. Here we describe one such application, a *Fault Tree Browser* for viewing a criticality analysis.

After the individual FTML files have been assembed into a criticality analysis (as explained in section 3.3), we would like to be able to browse through the complete set of fault trees. This is the purpose of the Fault Tree Browser. The home page for this application is shown in Figure 11.

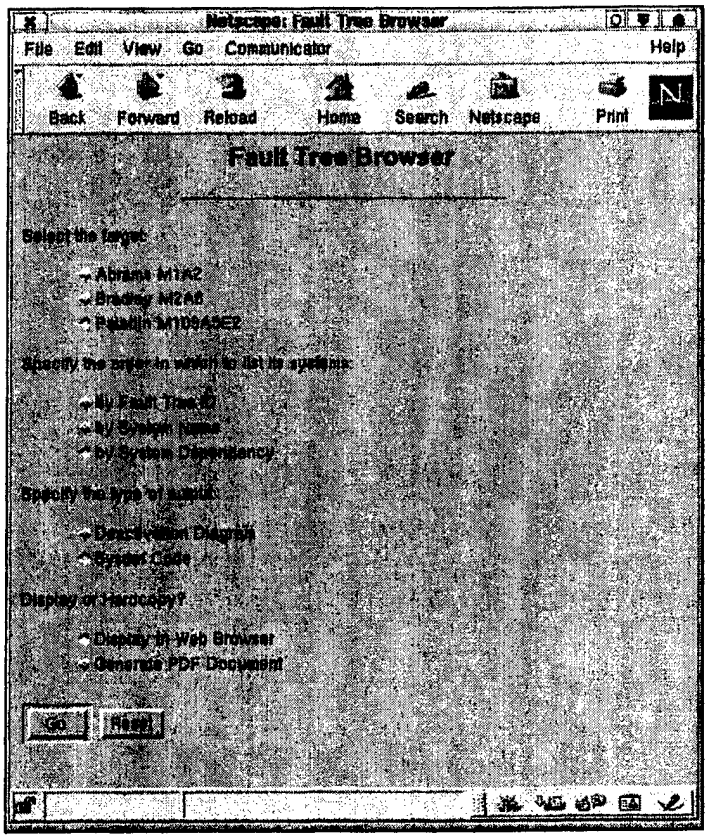


Figure 11. Fault tree browser home page.

The user selects the particular target of interest and the desired order for a listing of its various systems. These include the order as they are documented in an existing report (i.e., by figure number), alphabetical order by system name, or the order in which they depend upon one another. In the last case, the ordering is done by making sure that if system *A* depends upon system *B*, then system *B* is listed before system *A*. This ordering is important for proper initialization, and with the number of dependent systems in a typical criticality analysis, the required ordering is not at all obvious. In some cases, such as the Abrams M1A2, it is even possible to have circular dependencies. The software is able to detect this and isolate all those systems that are involved in circular dependencies. These are reported separately from all the other systems that are either independent of one another or, at most, have sequential dependency. There are two types of output that are supported: *deactivation diagrams* for viewing the fault trees or *sysdef code* that is useful for evaluating the fault tree. Finally, the user has the option of either displaying the results

in a Web browser or generating a report that can be saved or printed. The last two options will be described more fully later, but regardless of which options have been selected, when the user clicks on the Go button, the next page to appear will be similar to that shown in Figure 12.

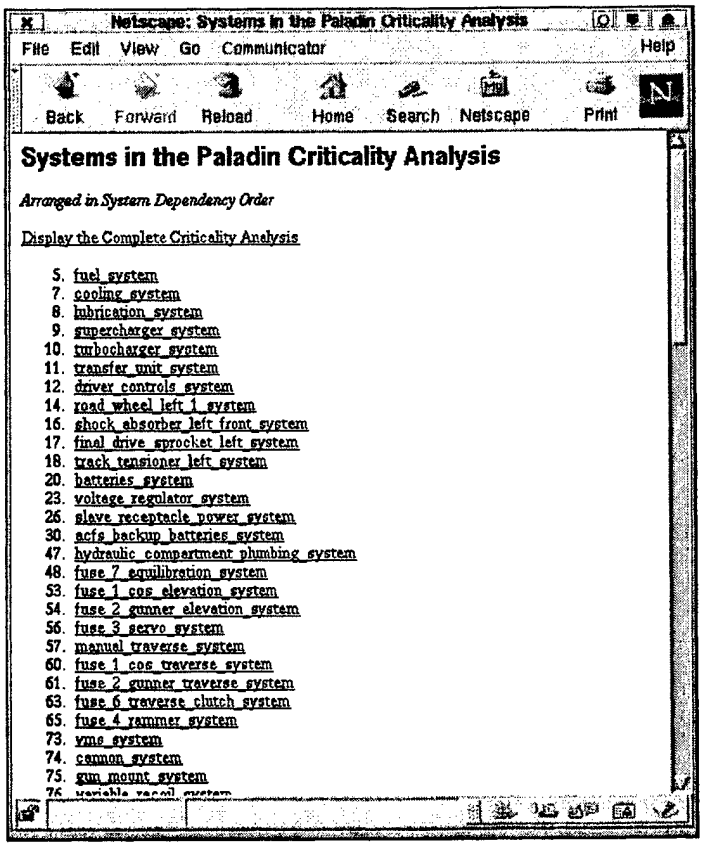


Figure 12. Listing of all the systems in a criticality analysis.

Here we see a listing of all the systems that constitute the criticality analysis. In this case, they are arranged in system dependency order. By clicking on the top link, "Display the Complete Criticality Analysis," the complete sysdef code for all the systems is concatenated together and displayed in the browser. The page can then be saved as a text file and used in other codes that require evaluation of the fault trees. Notice that all systems are links and by clicking on them, the user is automatically taken to the particular sysdef expression for that system, an example of which is shown in Figure 13.

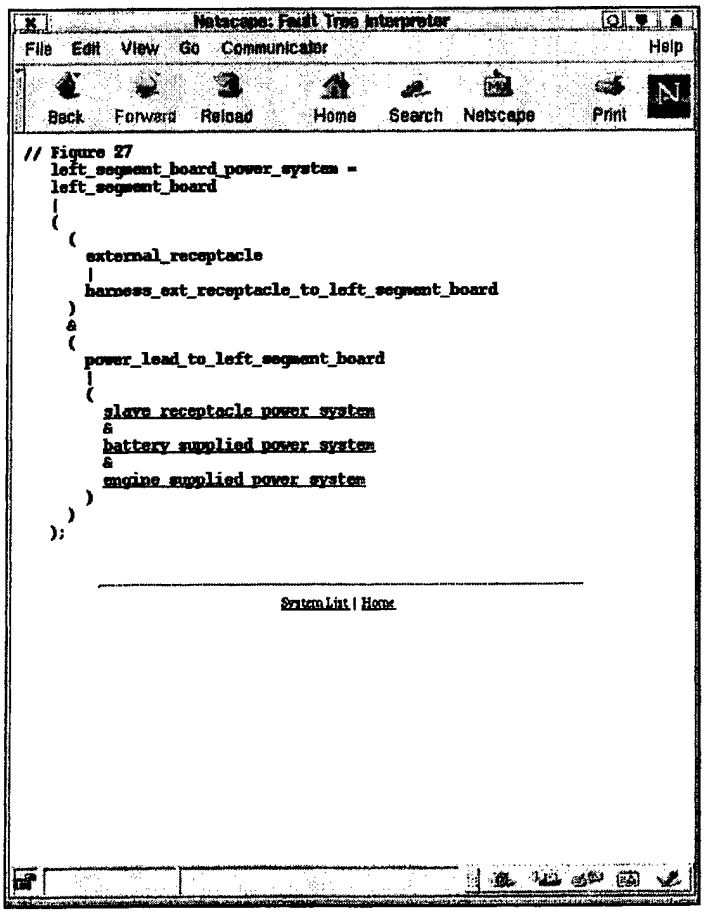


Figure 13. Sysdef code with links.

By clicking on the links and the Back button of the browser, it is easy to navigate through the criticality analysis.

If we return to the home page and select the *Deactivation Diagram* option, then we are again presented with a list of systems similar to the one in Figure 12. But now, clicking on a system will bring up the corresponding deactivation diagram. An example of which is shown in Figure 14.

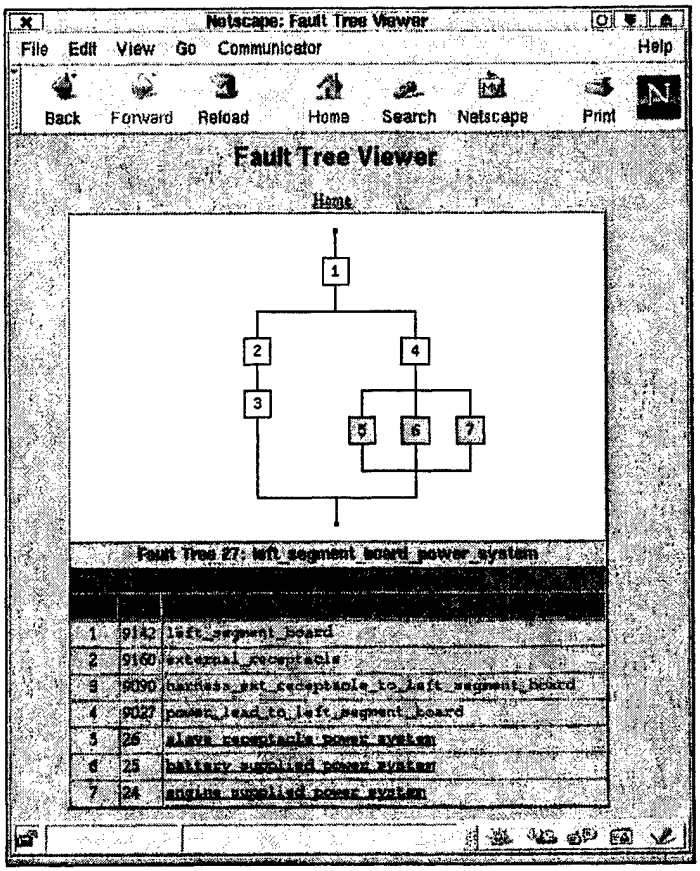


Figure 14. Deactivation diagram in fault tree viewer.

Empty boxes are critical components, and shaded boxes are systems. By clicking on the system name in the table, the user is automatically taken to that particular deactivation diagram.

Finally, by returning to the home page and selecting *Generate PDF Document*, a PDF document is constructed that has one deactivation diagram per page. An example is shown in Figure 15.

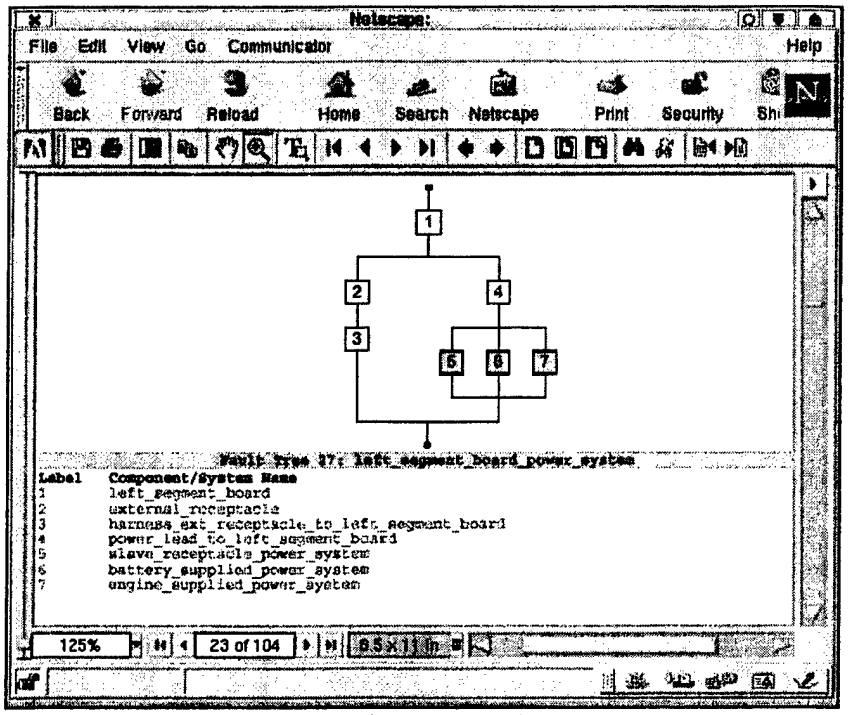


Figure 15. Page of generated PDF document.

Here, the Adobe Acrobat Reader is installed as a plug-in so that the resulting PDF document is loaded directly in the browser. If the user has not installed the plug-in, then the browser would bring up a dialog box requesting the user to save the PDF file. Notice that the system boxes are shaded in the PDF document. These are links embedded directly in the PDF document and can be used to navigate through the criticality analysis.

It should be emphasized here that all of this is being done on the fly from just the FTML. Deactivation diagrams, sysdef code, and PDF documents are constructed and displayed as the FTML is being parsed; nothing has been pre-processed ahead of time.

6. REFERENCES

[1] Hanes, P. J., S. L. Henry, G. S. Moss, K. R. Murray, and W. A. Winner. "Modular UNIX-Based Vulnerability Estimation Suite (MUVES) Analyst's Guide." BRL-MR-3954, U.S. Army Ballistic Research Laboratory, Aberdeen Proving Ground, MD, December 1991.

[2] Kinsler, R. E. "Criticality Analysis of the M2A1 Bradley for the New Close Combat Methodology (NCCM)." ASI International, 15 March 1989.

[3] Muehl, T. T., and J. L. Robertson. "Criticality Analysis of the M109A3E2 155-mm Self-Propelled Howitzer." ARL-TR-179, U.S. Army Research Laboratory, Aberdeen Proving Ground, MD, August 1993.

[4] Christy, T. Personal Communication. U.S. Army Research Laboratory, Aberdeen Proving Ground, MD, October 2002.

[5] Koffinke, R. A., Jr. "Criticality Analysis and Damage Assessment List for the Bradley M2A3." U.S. Army Research Laboratory, Aberdeen Proving Ground, MD, to be published.

[6] Koffinke, R. A., Jr. "Criticality Analysis and Damage Assessment List for the Abrams M1A2 Tank 2000." U.S. Army Research Laboratory, Aberdeen Proving Ground, MD, to be published.

APPENDIX. SOURCE CODE LISTING OF `Prob` CLASS

```
// Prob.h: Probability Algebra Class for the Evaluation of Fault Trees

#ifndef PROB_H
#define PROB_H

#include <iostream>
#include <vector>
#include <assert.h>
#include <math.h>
using namespace std;

// list of valid conditions

static const int ANY          = 0;
static const int ADJACENT     = 1;
static const int NON_ADJACENT = 2;
static const int NOT_ADJACENT = NON_ADJACENT;
static const int OR_MORE      = 3;
static const int OR_LESS      = 4;

class Prob {

static const bool KILLED     = true;      // meaning of true
static const bool NOT_KILLED = !KILLED;

// friends list of functions

friend
Prob zero( vector< Prob > pk ) {  // probability of exactly zero kills

   double prod = 1.;
   for ( int i = 0; i < pk.size(); i++ ) prod *= !pk[ i ];
   return Prob( prod );
}

friend
Prob none( vector< Prob > pk ) {  // synonymous with zero

   double prod = 1.;
   for ( int i = 0; i < pk.size(); i++ ) prod *= !pk[ i ];
   return Prob( prod );
}

friend
Prob all( vector< Prob > pk ) {  // probability of exactly all killed

   double prod = 1.;
   for ( int i = 0; i < pk.size(); i++ ) prod *= pk[ i ];
   return Prob( prod );
}

// return probability of exactly n_kills under the given condition

friend
Prob prob( vector< Prob > pk,                    // pk vector
           int            n_kills,               // number of kills
           int            condition = ANY ) {    // condition w/ ANY as default

   int n = pk.size();

   if ( n_kills == 0 && condition == ANY ) return zero( pk );
   if ( n_kills == n && condition == ANY ) return all( pk );

   double p_adjacent     = 0.;
   double p_not_adjacent = 0.;
   double p              = exact( pk, n_kills, p_adjacent, p_not_adjacent );

   switch ( condition ) {

      case ( ANY ):
         return Prob( p );
      case ( ADJACENT ):
         if ( n_kills < 2 ) return Prob( 0. );   // requires at least 2
         return Prob( p_adjacent );
      case ( NON_ADJACENT ):
         if ( n_kills < 2  ) return Prob( 0. );   // requires at least 2
         if ( n_kills == n ) return Prob( 0. );   // non-adjacent impossible
         return Prob( p_not_adjacent );
      case ( OR_MORE ):
         for ( int i = n_kills+1; i <= n; i++ )
            p += exact( pk, i, p_adjacent, p_not_adjacent );
         return Prob( p );
      case ( OR_LESS ):
         for ( int i = 0; i < n_kills; i++ )
            p += exact( pk, i, p_adjacent, p_not_adjacent );
         return Prob( p );
      default:
         cerr << "invalid condition: " << condition << endl;
```

```
        exit( 1 );
    }
}

// return probability of exactly n_kills (both adjacent and non-adjacent)

friend
Prob exact( vector< Prob > pk,                  // pk vector
            int            n_kills,             // number of kill
            double&        p_adjacent,          // probability of exactly n_kills, adjacent onl
            double&        p_not_adjacent ) {   // probability of exactly n_kills, non-adjacent only

   int n = pk.size();

   if ( n_kills < 0 ) {
      cerr << "The number of kills of interest, " << n_kills
           << ", must be zero or greater" << endl;
      exit( 1 );
   }
   if ( n_kills > n ) {
      cerr << "Impossible since the number of kills of interest, " << n_kills
           << ", exceeds the number of elements, " << n << endl;
      exit( 1 );
   }
   vector< Prob > p = pk;
   vector< bool > bit( n );
   p_adjacent = 0.;
   p_not_adjacent = 0.;

   for ( unsigned int i = 0; i < pow( 2, n ); i++ ) {

      _decode( i, bit );

      int count = 0;
      for ( int j = 0; j < n; j++ ) count += bit[ j ];

      if ( count != n_kills ) continue;

      bool adjacent = false;
      if ( n_kills == n )
         adjacent = true;
      else {
         for ( int j = 0; j < n - n_kills + 1; j++ ) {
            bool ex = true;
            for ( int k = 0; k < n_kills; k++ ) ex &= bit[ j + k ];
            adjacent |= ex;
         }
      }

      double prod = 1.;
      for ( int k = 0; k < n; k++ ) {
       if ( bit[ k ] ) p[ k ] =  pk[ k ];
         else          p[ k ] = !pk[ k ];
       prod *= p[ k ];
      }
      if ( adjacent ) p_adjacent     += prod;
      else            p_not_adjacent += prod;

   }
   return Prob( p_adjacent + p_not_adjacent );
}

friend
Prob evaluate( Prob( *ft )( vector< Prob > ),   // function pointer to fault tree
               vector< Prob > pk ) {            // pk vector
   return ft( pk );
}

friend
Prob evaluate( Prob( *ft )( vector< Prob > ),   // function pointer to fault tree
               vector< Prob > pk,               // pk vector
               int index ) {                    // one repeated index

   assert( 0 <= index && index < pk.size() );
   vector< Prob > p = pk;

   double e1 =  pk[ index ];
   double e2 = !pk[ index ];

   p[ index ] = KILLED;
   double p1 = e1 * ft( p );

   p[ index ] = NOT_KILLED;
   double p2 = e2 * ft( p );

   return Prob( p1 + p2 );
}

friend
Prob evaluate( Prob( *ft )( vector< Prob > ),   // function pointer to fault tree
               vector< Prob > pk,               // pk vector
               int index1,                      // first repeated index
               int index2 ) {                   // second repeated index
```

```
   assert( 0 <= index1 && index1 < pk.size() );
   vector< Prob > p = pk;

   double e1 =  pk[ index1 ];
   double e2 = !pk[ index1 ];

   p[ index1 ] = KILLED;
   double p1 = e1 * evaluate( ft, p, index2 );

   p[ index1 ] = NOT_KILLED;
   double p2 = e2 * evaluate( ft, p, index2 );

   return Prob( p1 + p2 );
}

friend
Prob evaluate( Prob( *ft )( vector< Prob > ),   // function pointer to fault tree
               vector< Prob > pk,               // pk vector
               vector< int >index ) {           // repeated index vector

   int m = index.size();
   int n = pk.size();
   assert( 0 <= m && m <= n );

   if ( m == 0 ) return evaluate( ft, pk );              // no index occurs more than once
   if ( m == 1 ) return evaluate( ft, pk, index[ 0 ] );  // only one index occurs more than once
   if ( m == n ) return evaluate_all( ft, pk );          // all indices occur more than once

   int index1 = index.back();   // get last index
   assert( 0 <= index1 && index1 < pk.size() );
   index.pop_back();            // decrement the index vector
   vector< Prob > p = pk;

   double e1 =  pk[ index1 ];
   double e2 = !pk[ index1 ];

   p[ index1 ] = KILLED;
   double p1 = e1 * evaluate( ft, p, index );

   p[ index1 ] = NOT_KILLED;
   double p2 = e2 * evaluate( ft, p, index );

   return Prob( p1 + p2 );
}

// evaluate without making any assumptions regarding repeated indices
// gives correct answer even if not all indices are repeated, but does take longer

friend
Prob evaluate_all( Prob( *ft )( vector< Prob > ),   // function pointer to fault tree
                   vector< Prob > pk ) {            // pk vector

   int n = pk.size();
   vector< Prob > p( n ), state( n );
   vector< bool > bit( n );
   double prob = 0.;
   for ( int i = 0; i < pow( 2, n ); i++ ) {
      _decode( i, bit );
      double prod = 1.;
      for ( int j = 0; j < n; j++ ) {
         if ( bit[ j ] ) p[ j ] =  pk[ j ];
         else            p[ j ] = !pk[ j ];
         prod *= p[ j ];
         state[ j ] = Prob( bit[ j ] );
      }
      prob += prod * ft( state );
   }
   return Prob( prob );
}

// overloaded logical operators

   friend Prob operator|( const Prob& a, const Prob& b ) {   // logical OR

      return Prob( a._p + b._p - a._p * b._p );
   }

   friend Prob operator&( const Prob& a, const Prob& b ) {   // logical AND

      return Prob( a._p * b._p );
   }

   friend Prob operator^( const Prob& a, const Prob& b ) {   // logical EXCLUSIVE OR

      return Prob( a._p + b._p - 2. * a._p * b._p );
   }

// overloaded arithmetic operators

   friend Prob operator+( const Prob& a, const Prob& b ) {   // Prob + Prob

      return Prob( a._p + b._p );
   }
```

```
    friend Prob operator+( const Prob& a, double b ) {   // Prob + double
       return Prob( a._p + b );
    }
    friend Prob operator+( double a, const Prob& b ) {   // double + Prob
       return Prob( a + b._p );
    }
    friend Prob operator-( const Prob& a, const Prob& b ) {  // Prob - Prob
      return Prob( a._p - b._p );
    }
    friend Prob operator-( const Prob& a, double b ) {  // Prob - double
      return Prob( a._p - b );
    }
    friend Prob operator-( double a, const Prob& b ) {  // double - Prob
      return Prob( a - b._p );
    }
    friend Prob operator*( const Prob& a, double s ) {   // Prob * double
       return Prob( a._p * s );
    }
    friend Prob operator*( double s, const Prob& a ) {   // double * Prob
       return Prob( a._p * s );
    }
    friend Prob operator/( const Prob& a, double s ) {   // Prob / double
       assert( s >= 0. );
       return Prob( a._p / s );
    }
// overloaded stream operators
    friend istream& operator>>( istream& is, Prob& a ) {   // input Prob
       double p;
       is >> p;   // enter the probability
       if ( 0. <= p && p <= 1. ) {
          a._p = p;
          return is;
       }
       else {
          cerr << "Invalid input: value must be between 0 and 1." << endl
               << "Program Stopped." << endl;
        exit( 1 );
       }
    }
    friend ostream& operator<<( ostream& os, const Prob& a ) {   // output Prob
       return os << a._p;
    }
public:
    Prob( double p = 0. ) {   // constructor with default of zero
       assert( 0. <= p && p <= 1. );   // p must be in the interval [0,1]
       _p = p;
    }
   ~Prob( void ) {   // default destructor
    }
    Prob( const Prob& a ) : _p( a._p ) {    // copy constructor
    }
    Prob& operator=( const Prob& a ) {   // assignment operator
       if ( this != &a ) _p = a._p;
       return *this;
    }
    Prob& operator=( const double& p ) {   // assign a probability
       _p = p;
       return *this;
    }
    Prob operator!( void ) {   // logical NOT
       return Prob( 1. - _p );
    }
```

```
    Prob operator~( void ) {   // logical NOT

        return Prob( 1. - _p );
    }

// overloaded logical operators

    Prob& operator|=( const Prob& a ) {   // logical OR assignment

        _p += ( 1. - _p ) * a._p;

        assert( 0. <= _p && _p <= 1. );
        return *this;
    }

    Prob& operator&=( const Prob& a ) {   // logical AND assignment

        _p *= a._p;

        assert( 0. <= _p && _p <= 1. );
        return *this;
    }

    Prob& operator^=( const Prob& a ) {   // logical EXCLUSIVE OR assignment

        _p += ( 1. - _p - _p ) * a._p;

        assert( 0. <= _p && _p <= 1. );
        return *this;
    }

// overloaded arithmetic operators

    Prob& operator+=( const Prob& a ) {   // addition assignment from Prob

        _p += a._p;
        assert( 0. <= _p && _p <= 1. );
        return *this;
    }

    Prob& operator+=( double p ) {   // addition assignment from double

        _p += p;
        assert( 0. <= _p && _p <= 1. );
        return *this;
    }

    Prob& operator-=( const Prob& a ) {   // subtraction assignment from Prob

        _p -= a._p;
        assert( 0. <= _p && _p <= 1. );
        return *this;
    }

    Prob& operator-=( double p ) {   // subtraction assignment from double

        _p -= p;
        assert( 0. <= _p && _p <= 1. );
        return *this;
    }

    Prob& operator*=( double s ) {   // multiplication assignment

        assert( s >= 0. );
        _p *= s;
        return *this;
    }

    Prob& operator/=( double s ) {   // division assignment

        assert( s > 0. );
        _p /= s;
        return *this;
    }

// conversion operator

    operator double( void ) const {   // return probability value

        return _p;
    }

// access functions

    double p( void ) const {   // return probability value

        return _p;
    }

private:

friend
unsigned int _encode( const vector< bool >& v ) {
```

```
    const unsigned int BITS_PER_BYTE = 8;
    const int N = v.size();
    assert( N <= BITS_PER_BYTE * sizeof( unsigned int ) );
    unsigned int s = ( unsigned int )( v[ 0 ] );
    for ( int i = 1; i < N; s += v[ i++ ] ) s *= 2;
    return s;
}

friend
void _decode( unsigned int s, vector< bool >& v ) {

    const unsigned int BIT = 1;
    const int N = v.size();
    v = vector< bool >( N, false );
    for ( int i = 0; i < N; i++ ) v[ N - 1 - i ] = ( s & ( BIT << i ) );
}

    double _p;     // probability value
};

#endif
```

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED (From - To)
March 2003	Final	September 2001–September 2002

4. TITLE AND SUBTITLE
Fault Tree Representation and Evaluation

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Richard Saucier

5d. PROJECT NUMBER
622618AH80

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
U.S. Army Research Laboratory
ATTN: AMSRL-SL-BE
Aberdeen Proving Ground, MD 21005-5068

8. PERFORMING ORGANIZATION REPORT NUMBER
ARL-TR-2923

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

In order to conduct a vulnerability analysis of a combat vehicle, it is first necessary to perform a criticality analysis, which consists of describing each of its combat functions along with the underlying systems, subsystems, and components required to support the function. The basic building blocks for carrying out this process are the fault trees; they depict the logical arrangement of the components required for the proper functioning of the vehicle. This report describes a method of representing fault trees in XML (Extensible Markup Language) and is accomplished by first defining a Fault Tree Markup Language (FTML) that can be used to describe the essential logical structure of any fault tree. Once the fault tree is described in FTML, it can then be stored as an ordinary text file. Furthermore, software described in this report will then enable one to generate both the deactivation diagram and the C code that is used to evaluate the fault tree. Fault tree evaluation means the determination of whether the system that the fault tree represents is either functional or nonfunctional, given that one or more components are dysfunctional. The second part of this report describes two methods for evaluating fault trees—one based on Monte Carlo sampling and the other on the algebra of probability theory.

15. SUBJECT TERMS
fault tree, criticality analysis, deactivation diagram, XML, FTML

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Richard Saucier
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL	36	19b. TELEPHONE NUMBER *(Include area code)* (410) 278-6721

INTENTIONALLY LEFT BLANK.