A Linear Time Algorithm to Find Modules of Fault Trees

Yves Dutuit, Antoine Rauzy

Abstract— **A module of a fault tree is a subtree whose terminal events do not occur elsewhere in the tree. Modules, which are independant subtrees, can be used to reduce the computational cost of basic operations on fault trees, such as the computation of the probability of the root event or the computation of the minimal cut sets. This paper presents a linear time algorithm to detect modules of a fault tree, coherent or not, that is derived from the Tarjan's algorithm to find strongly connected components of a graph. We show, on a benchmark of real life fault trees, that our method detects modules of trees with several hundreds gates and events within few milliseconds on a personal computer.**

Keywords— **Fault Trees, Modules.**

I. Introduction

Analysis of large fault tree can be computationally expensive, despite of recent works on the use of Bryant's binary decision diagrams (BDD's for short) [1], as proposed by J.C. Madre and O. Coudert [2] and one of the authors [3] (see also [4]). Furthermore, it is sometimes difficult to understand the physical importance of a large number of minimal cuts. A way to tackle both difficulties is to detect modules and to treat them separately. A module of a fault tree is a subtree whose terminal events do not occur elsewhere in the tree. In a word, modules are independant subtrees. P. Chatterjee [5] and Z.W. Birnbaum and J.P. Esary [6] developed the properties of modules and demonstrated their use in fault-tree analysis. M.O. Locks [7] expanded the concept to non-coherent fault trees and showed its effectiveness in obtaining cut sets.

A number of methods have been proposed to modularize fault trees (see [8], [9], [10], [11] to cite a few). These methods are based on the rewritting of the original formula into an equivalent one, which contains more modules. The algorithm we propose here achieves a less ambitious goal since it only detects modules already existing in the tree under study. However, its efficiency makes it of a special interest as a subprogram of rewritting methods or to design heuristics for variable ordering in BDD's [2].

Rigourous complexity analyses of fault tree analysis algorithms become more important as the size of the studied trees increases. Nowadays, fault trees with several hundreds gates and events are not uncommon, due the generalization of tools that automatically generate them. For such trees, differences of complexity (say linear versus quadratic complexity) are significant in practice, at least for operations that are often repeated. General techniques to achieve a linear complexity are thus of a great interest. Our algorithm is derived from the Tarjan's one to detect strongly connected components of a graph [12]. We believe that this kind of graph techniques can be useful to perform other operations on fault trees or reliability networks.

The remaining of this paper is organized as follows: basics about fault trees are recalled section III. The algorithm is presented section IV. Finally, experimental results are reported section V.

II. Notations

Boolean formulae are terms inductively built over the two (Boolean) constants 0 (False) and 1 (True), a denumerable set of variables $\mathcal{V} = \{e_1, e_2, \ldots\}$, and the usual logical connectives $\wedge$ (and), $\vee$ (or), $\neg$ (not).

A *graph* is given by a set of nodes (also called *vertices*) U together with a set of edges $E \subseteq U \times U$ (edges are thus pairs of nodes). In what follows, we consider only *directed* graphs which means that pairs are ordered.

Let (u, v) be an edge. u is a *parent node* of v. v is a *child node* of u. A node without parent node is called a *root* node, while a node without child node is called a *sink* node. Sink nodes are also called *leaves*.

A *path* in the graph is a sequence of nodes $u_1, \ldots, u_k$ such that $(u_i, u_{i+1}) \in E$ for $i = 1, \ldots, k-1$. The *length* of a path is the number of edges it traverses. A node v is said to be *reachable* from a node u if there exists a path from u to v.

A graph is said *acyclic* if for every node u, u is not reachable from itself through a positive length path.

A strongly connected component of a graph $G = (U, E)$ is a set of nodes $U' \subseteq U$ such that for all u, v in U' there exists a path from u to v and vice-versa.

III. Fault Trees

We assume the reader is familiar with fault trees and their applications (see [13] for reviews on that technique). For the purpose of this paper, *fault trees* are essentially considered as *Boolean formulae*.

Fig. 1 depicts a small fault tree, that will be used throughout this paper. r is the *root event* of the tree, the g_i's are *internal events*, and the e_i's are *terminal events*. The Boolean formula associated with r is the following.

$$r \quad = \quad (((e_1 \wedge e_2) \vee (e_3 \wedge e_4)) \wedge ((e_3 \wedge e_4) \vee (e_5 \wedge e_6))) \vee e_7$$

The physical system whose failures are described by such a fault tree as well as each of its parts (elementary or not) are assumed to be in one (and only one) of the two states good or failed. Moreover, terminal events that describe failures of elementary components are assumed to be independent one another from a probabilistic point of view.

Even small formulae, as the above one, can be quite hard to read. This is the reason why one prefers, in general, to write fault trees as sets of boolean equations. Moreover

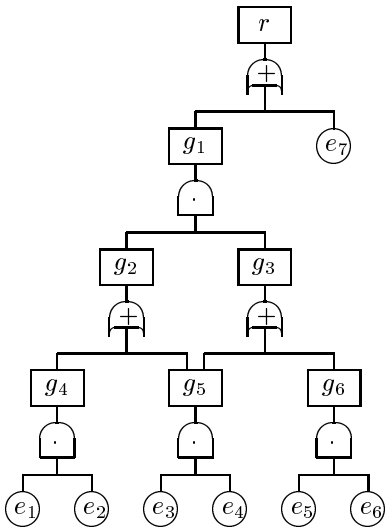

Fig. 1. A Coherent Fault Tree

equations reflect the graphical nature of fault trees. The set of equations describing r is as follows.

$$\begin{array}{lcllcl} r & = & (g_1 \vee e_7) & g_4 & = & (e_1 \wedge e_2) \\ g_1 & = & (g_2 \wedge g_3) & g_5 & = & (e_3 \wedge e_4) \\ g_2 & = & (g_4 \vee g_5) & g_6 & = & (e_5 \wedge e_6) \\ g_3 & = & (g_5 \vee g_6) & & & \end{array}$$

A fault tree can be considered as a directed acyclic graph whose nodes are either events or logical gates. An edge links the vertex v to the vertex w if v takes w as input.

We denote by $Events(v)$ the set of events reachable from the vertex v. For instance, on the tree pictured Fig. 1, $Events(g_2) = \{g_4, e_1, e_2, g_5, e_3, e_4\}$.

A *module* of a fault tree is an internal event whose terminal events do not occur elsewhere in the tree. Formally, an event v is a module if for any other event w, either $w \in Events(v)$ or $Events(v) \cap Events(w) = \emptyset$.

It follows from the definition that the root event and the terminal events are always modules. In what follows, for the sake of conciseness, we do not mention them.

Modules of the tree pictured Fig. 1 are g_1, g_4, g_5 and g_6.

IV. Detecting modules in linear time

A. Depth-first left-most traversals

Our algorithm works "in the spirit" of Tarjan's algorithm to compute strongly connected components of (directed) graphs [12] in that it performs two depth-first left-most traversals of the formula under study, updating counters at each traversal.

A depth-first left-most traversal of the tree pictured Fig. 1 visits the events in the order reported table I.

Note that:

- Each internal event is visited at least twice: the first time when descending from its first parent, the second one when going back from its right-most child.
- The graph under a vertex is never traversed twice. For instance, the graph under g_5 is traversed when coming from g_2 but not the third time g_5 is encountered (i.e. when coming from g_3).

From the two previous remarks it follows that each edge (i.e. link between a node and one of its children) is traversed exactly twice and thus a depth-first left-most traversal of the graph is linear in the number of edges it contains.

B. Algorithm

The principle of the algorithm can be stated as follows. Let v be an internal event, and let t_1 and t_2 be respectively the dates of the first and second visits of v in a depth-first left-most traversal of the graph. Then v is a module if and only if none of its descendants is visited before t_1 or after t_2 during the traversal.

For instance, g_5 is a module because both e_3 and e_4 are visited after 8 (date of the first visit of g_5) and before 11 (date of the second visit of g_5). g_2 is not a module because g_5 is visited at 14, which is later than the date of the second visit of g_2 (here 12). Nor is g_3 because g_5 is visited at 8 which is earlier than the date of the first visit of g_3 (13).

In order to implement the algorithm, one needs three counters per node that will contain respectively the dates of the first, second and last visits.

Then, the algorithm works in three steps:

1. It initializes the counters (this is performed by traversing the list of nodes).
2. It performs a first depth-first left-most traversal of the graph to set counters (note that for leaves first and second dates are identical).
3. It performs a second depth-first left-most traversal of the graph in which it collects, for each internal event v, the minimum of the first dates and the maximum of the last dates of its children. v is a module if and only if the collected minimum and maximum are respectively greater than the first date of v and less than the second one.

Indeed, the second traversal has the same complexity than the first one. It follows that the overall algorithm is linear in the size of the tree, i.e. number of nodes plus number of edges.

For our example, results are summarized table II.

Note that connectives do not influence the algorithm that works consequently on any kind of formulae (including coherent and non coherent fault trees).

V. Experimental results and conclusion

The results reported in the table III have been obtained on a set of real-life fault trees from Dassault Aviation and Électricité de France. The first row gives the numbers of internal events of the tested trees, the second one their numbers of terminal events, the third one the number of modules they contain and the last one the running times in milliseconds on a PC 486 66Mgz (note that our machine does not provide a measure of time finer than 16ms, which explains the regularity of reported times).

As shown by the table III, the algorithm we propose in this paper is very efficient. It is integrated satisfactorily in several tools of the Aralia toolbox [14], to design heuristics

step	1	2	3	4	5	6	7	8	9	10	11
visited node	r	g_1	g_2	g_4	e_1	e_2	g_4	g_5	e_3	e_4	g_5

step	12	13	14	15	16	17	18	19	20	21	22
visited node	g_2	g_3	g_5	g_6	e_5	e_6	g_6	g_3	g_1	e_7	r

TABLE I

DEPTH-FIRST LEFT-MOST TRAVERSAL OF THE FAULT TREE PICTURED FIG. 1.

	r	g_1	g_2	g_3	g_4	g_5	g_6	e_1	e_2	e_3	e_4	e_5	e_6	e_7
first	1	2	3	13	4	8	15	5	6	9	10	16	17	21
second	22	20	12	19	7	11	18	5	6	9	10	16	17	21
last	22	20	12	19	7	14	18	5	6	9	10	16	17	21
minimum	2	3	4	8	5	9	16	-	-	-	-	-	-	-
maximum	21	19	14	18	6	10	17	-	-	-	-	-	-	-
module	yes	yes	no	no	yes	yes	yes	-	-	-	-	-	-	-

TABLE II

RESULTS FOR THE TREE OF FIG. 1.

	1	2	3	4	5	6	7	8
#gates	248	254	323	337	289	439	484	1050
#basic events	283	278	276	306	311	530	548	362
#modules	25	25	70	32	32	30	29	70
#gates largest module	224	230	230	306	258	410	456	981
#basic events largest module	251	246	216	276	272	490	509	240
Running times for :								
module detection	$16ms$	$25ms$	$33ms$	$33ms$	$16ms$	$16ms$	$33ms$	$50ms$
tree processing without m.d.	$9s16$	$11s93$	$1s38$	$95s28$	$39s45$	$1s35$	$2s53$	$69s25$
tree processing with m.d.	$8s67$	$10s45$	$1s06$	$88s95$	$37s14$	$1s27$	$2s47$	$44s70$

TABLE III

EXPERIMENTAL RESULTS

for variable ordering (for BDD's) and as a part of formula simplifiers. Table III also reports running times to compute BDD's encoding fault trees under study with and without module detection. They are not very different because all of the trees but the last one are decomposed into a large module just under the root event and a number of modules grouping only very few terminal events. Despite of that, running times are always improved by module detection. The largest module of the last tree is significantly smaller than the tree itself. As a consequence, module detection improves running times significantly as well.

REFERENCES

[1] K. Brace, R. Rudell, and R. Bryant, "Efficient Implementation of a BDD Package", in *Proceedings of the 27th ACM/IEEE Design Automation Conference*. 1990, pp. 40–45, IEEE 0738.

[2] O. Coudert and J.-C. Madre, "MetaPrime: an Iteractive Fault Tree Analyser", *IEEE Transactions on Reliability*, vol. 43, no. 1, pp. 121–127, March 1994.

[3] A. Rauzy, "New Algorithms for Fault Trees Analysis", *Reliability Engineering & System Safety*, vol. 05, no. 59, pp. 203–211, 1993.

[4] S.A. Doyle, J.B. Dugan, and M. Boyd, "Combinatorial-Models and Coverage: A Binary Decision Diagram (BDD) Approach", in *Proceedings of the IEEE Annual Reliability and Maintainability Symposium, ARMS'95*. 1996, pp. 82–89, IEEE.

[5] P. Chatterjee, "Modularization of fault trees: A method to to reduce the cost of analysis", *Reliability and Fault Tree Analysis, SIAM*, pp. 101–137, 1975.

[6] Z.W. Birnbaum and J.P. Esary, "Modules of coherent binary systems", *SIAM J. of Applied Mathematics*, vol. 13, pp. 442–462, 1965.

[7] M.O. Locks, "Modularizing, minimizing and interpreting the K & H fault tree", *IEEE Transactions on Reliability*, vol. R-30, pp. 411–415, December 1981.

[8] A. Rosenthal, "Decomposition methods for fault tree analysis", *IEEE Transactions on Reliability*, vol. R-29, pp. 136–138, 1980.

[9] L. Camarinopoulos and J. Yllera, "An Improved Top-down Algorithm Combined with Modularization as Highly Efficient Method for Fault Tree Analysis", *Reliability Engineering and System Safety*, vol. 11, pp. 93–108, 1985.

[10] J. Yllera, "Modularization methods for evaluating fault trees of complex technical systems", in *Engineering Risk and Hazard Assessment*, A. Kandel and E. Avni, Eds., vol. 2, chapter 5. CRC Press, 1988, ISBN 0-8493-4655-X.

[11] T. Kohda, E.J. Henley, and K. Inoue, "Finding Modules in Fault Trees", *IEEE Transactions on Reliability*, vol. 38, no. 2, pp. 165–176, June 1989.

[12] R.E. Tarjan, "Depth First Search and Linear Graph Algorithms", *SIAM J. Comput.*, vol. 1, pp. 146–160, 1972.

[13] W.S. Lee, D.L. Grosh, F.A. Tillman, and C.H. Lie, "Fault tree analysis, methods and applications : a review", *IEEE Transactions on Reliability*, vol. 34, pp. 194–303, 1985.

[14] Groupe Aralia, "Computation of Prime Implicants of a Fault Tree within Aralia", in *Proceedings of the European Safety and Reliability Association Conference, ESREL'95*, Bournemouth – England, June 1995, pp. 190–202, European Safety and Reliability Association.

Pr. Yves Dutuit
Dept. Hygiène et Sécurité / LADS,
I.U.T. A, Université Bordeaux I,
33405 Talence Cedex, FRANCE, < AIR Mail >
Work phone: [33] 56845834, Fax: [33] 56845829
Internet (e-mail): dutuit@minuit.iuta.u-bordeaux.fr.

Yves Dutuit was born in 1943 in Morocco. After having received his doctorat-ès-sciences in time domain spectroscopy from Bordeaux university, he devoted himself to teaching of safety and reliability at the Bordeaux institute of technology. His research interests lie in qualitative and quantitative assessements methods in reliability. He belongs to the editorial board of the Reliability Engineering & System Safety and he is a referee for this journal and Performance Evaluation Journal.

Antoine Rauzy
LaBRI, Université Bordeaux I,
351, cours de la Libération,
33405 Talence Cedex, FRANCE, < AIR Mail >
Work phone: [33] 56845834, Fax: [33] 56846669
Internet (e-mail): rauzy@labri.u-bordeaux.fr.

Antoine Rauzy has got his P.H.D. in computer science in 1989 from the university of Marseilles. He joined the computer science laboratory of the university of Bordeaux by the end of 1989 and the French National Center for Scientific Research in 1991. His topics of research are constraint logic programming, automated deduction, validation of distributed system and since 1993 reliability analyses. His activity is focused on Boolean reasoning and he leads the french group RESSAC working on practical resolution of NP-complete problems.