Proceedings of the Twenty-Ninth International
Florida Artificial Intelligence Research Society Conference

Using L-Systems to Generate Fault Trees for Benchmarking and Testing

Jeff Hanes
Applied Research Associates, Inc.
Niceville, FL
jhanes@ara.com

R. Paul Wiegand
Institute for Simulation & Training
University of Central Florida
wiegand@ist.ucf.edu

Abstract

We present the use of Lindenmayer systems to generate fault trees for testing and benchmarking purposes. Our method produces benchmarks that are reproducible, capable of producing fault trees similar properties to real-world designs, and scalable while maintaining predictable structural properties. Our approach will be extremely useful for testing and analyzing different kinds of solvers for fault tree analysis tasks at different scales and under different conditions.

Introduction

This paper demonstrates the utility of using Lindenmayer systems (L-systems) (Lindenmayer 1968) to generate fault trees (FTs) that emulate engineering designs and can be used to test and benchmark different methods of solving FTs at varying degrees of scale.

Fault tree analysis (FTA) is a form of deductive failure analysis for determining how and where complex engineering designs might fail (Clifton 1999). Real world FTs are created top-down, based on a system design, codifying how components in that system depend on one another. A well-described FT can reveal weaknesses in a design before it goes to production or help find ways to improve existing infrastructure. There are a variety of methods for analyzing FTs to find such faults, and many real-world engineering institutions rely on FTA to reveal system problems and also design more robust systems, including Bell Labs (Clifton 1999), NASA (Stamatelatos et al. 2002), the Nuclear Regulatory Commission (Vesely et al. 1981), and the U.S. Department of Defense (Deitz et al. 2009). FTs constructed to address real-world designs are becoming increasingly larger and more complex, sometimes with thousands of components. Effective algorithms for FTA that demonstrate efficiency as systems scale are critical to many modern engineering projects.

Unfortunately, most published research about FT solvers either rely on very small systems — FTs with fewer than 50 components — or are applied to FTs that are not provided because they are proprietary. Further, it is currently difficult to describe large FTs in a compact and reproducible way. The FTA community needs tools to provide compact representations of FTs for testing and benchmarking new algorithms such that generated trees are reproducible, have similar properties to real world FTs, and can be scaled in size.

L-systems are a formal grammar that have compact representations and use re-writing rules to recursively generate tree-like structures. The very purpose of these tools is to "*grow*" a tree that maintains certain properties as it scales, making them suitable as benchmark problem generation tools for FTA.

In this paper, we demonstrate the application of L-systems to the generation of FTs for benchmarking and testing of FTA solvers. The generation tool allows users to encode certain key structural properties that relate to properties in real-world systems, and we show that generated FTs maintain these key structural properties even as the system gets larger. This facilitates performance comparison of FTA methods at varying scales and also provides a means by which benchmark problems can be compactly represented and communicated for the greater community.

Background

Testing & Benchmarking of Fault Trees

Reliability literature describes many algorithms to improve the efficiency and value of solutions to FTs (Clifton 1999; Rauzy 2001). However, these are usually demonstrated by showing their operation on small problems that would not stretch the capabilities of any existing methods or software. Examples found in the literature generally do not exceed 50 components; however, real world problems often contain hundreds or thousands of components (Gauthier, Leduc, and Rauzy 2007). Thus, the problems shown in the literature do not provide researchers with sufficient information to ascertain whether any such methods are useful on a full scale problem.

Clearly, there is a need to generate test sets in a reliable way that can be shown to stress existing methods and motivate the creation of new methods. These test sets can also be used to test the effectiveness of emerging methods.

Realistic problems that exceed a thousand components cause significant difficulties for commonly used methods of solving FTs. Traditional FT solvers are typically exact methods while the task itself is NP-Hard (Ball 1986). Small

problems and solutions for small problem sets are currently the standard for presenting FTA research. There are few researchers besides those mentioned above who discuss methods to solve larger problems. Researchers rarely publish large and complex FTs for at least two very good reasons:

1. A legible illustration of a FT containing 30 or so components will fill a normal 8.5 by 11 page. Thus, a real world problem with hundreds of components can take tens of pages to show legibly.
2. Sometimes access to the information is restricted by the sponsor of the work. This can happen when large development firms do not wish to release design details of their systems. In addition, the military uses FTs to analyze system vulnerabilities (Deitz et al. 2009).

This is unfortunate because the research highlights the need to develop methods to handle large FTs, but there are no sample FTs for the research community to use to develop more efficient FTA methods. Researchers who do not have access to these large systems through their sponsors are left with little on which to work.

It is necessary to have a means to generate realistic systems that can be used to show the operation of a new FTA method but do not reveal proprietary information. Such a tool would be a great help to researchers who need to publish the methodologies developed in their research, but for various reasons cannot show the problems that actually inspired the research. It is also important to provide a means to compactly display the required information for generating a FT without multi-page illustrations.

Any method developed to create test FTs needs to be: 1) reproducible, 2) capable of producing FTs similar to real world FTs, and 3) scalable in measurable ways from small problems to large problems. Indeed, scalability is crucial for this purpose, since there must be some way to show the impact of the size of the FT without confounding it with the complexity of the tree. In other words, one would like to hold characteristics of the FT constant for various sizes.

It is, of course, possible to randomly generate FTs, but this approach would have difficulties. First, it would be difficult to reproduce, unless one published the random seed and the exact pseudorandom generation algorithm used to create the sequence of numbers. In addition, the rules driving the pseudorandom algorithm would need to be carefully described so that others could try the same approach. Furthermore, the stochastic nature of the random algorithm would not necessarily be able to generate solutions that maintain their properties as they increase in size.

Thus, the researcher using random generation would be reduced to trial and error to find a solution that matched the problem he was really trying to solve while requiring significant effort to guarantee that other researchers are able to reproduce the work. It would not really address the challenge of communicating large FTs, since the author would still need to show the actual system generated by the random algorithm to show that the work was valid — thus, it would end up defeating the utility of creating a generator to compactly define sample FT systems.

Lindenmayer Systems

Biologist Aristid Lindenmayer devised a mathematical approach to generate structures that maintain certain properties at different scales, such as seen in many natural structures like plants and algae (Lindenmayer 1968). These systems transitioned into the computer science community in the mid-80's (Smith 1984) and have become an important tool in computer graphics, artificial life, and biology (Rozenberg and Salomaa 1992). L-systems are particularly well-suited for generating topologies that have modular sub-structures (Frijters and Lindenmayer 1976), which makes them ideal candidates for generating engineered designs.

An L-system is a formal, context-free grammar containing three elements: $G = (V, \omega, P)$, where V is the alphabet of symbols that can be replaced, ω is the starting string of symbols from V, and P are production rules that define how symbols are replaced. A production rule is a mapping from one string of symbols to another. A system is iterated some number of times, and in each iteration the existing symbols in the string are matched to the left-hand-side of the production rules and a new string is produced by replacing matched substrings with the right-hand-side of the matching production rules. The reader is referred to Rozenberg (Rozenberg and Salomaa 1992) for technical details.

A very popular use of L-systems is to generate plant-like computer graphics images; however, L-systems have also been used in a variety of engineering applications. For example, within the machine learning community, L-systems have been used as generative encoding schemes for evolutionary computation based learning methods for neural network structures (Lima de Campos, Limão de Oliveira, and Roisenberg 2015). Within the natural language processing community, they have been used to model text sequences (Liou et al. 2013).

Moreover, L-systems have also been used successfully by engineers to construct benchmark problems for other fields. For example Martin *et al.* (Martin et al. 2010) used functional L-systems within automated scenario generation tools to build up scenarios in simulations and games used for training. Additionally, Ahammed *et al.* (Ahammed and Moscato 2011) use L-systems to design challenging traveling salesman problem (TSP) instances for benchmarking TSP solvers. They are able to demonstrate that increasingly more challenging TSP instances could be produced while maintaining salient properties of existing, known problems at varying scales.

Generating & Measuring Fault Tree Benchmarks

Assuming that it is possible to produce FTs using this technique, there remains the question of whether they are useful for testing solution methods and software. For that to be the case, they should retain similar features to the smaller FTs seen in the literature. These features should not change significantly as they grow larger through more iterations.

To ascertain whether they are similar, it is necessary to define metrics for FTs that can be calculated for large or

small trees. For the current research, several metrics were calculated to determine which were the most useful.

Graph theory provides the notion of degree; for a tree, it is possible to measure the "out degree" for each operator node, that is, the number of edges connected to nodes below it. Calculating the average of the out degree for all operator nodes can give an idea of the "leafiness" of the tree. That is, how much it branches as the generator iterates.

Also, in some cases, systems and components are re-used in portions of the tree. In the real world applications, this can describe something like an automobile's electrical power system that is used by the engine, the onboard radio and several other systems in a car. Thus, the average in-degree and out-degree for all nodes (or their distribution) can be useful to describe the structure of a fault tree.

Another useful metric is the proportion of different kinds of operator nodes in the tree. For traditional FTs, there are three types of nodes: OR, AND and K-of-N. These refer to the number of constituent nodes that need to be damaged to render the system described by the node ineffective. AND means that all nodes must be killed, OR means that only one needs to be killed and K-of-N means that any combination of K out of the N links will kill the system. One way to express this information is to show the percentage of nodes containing each type of operator in the tree. This distribution of the number of possible kill conditions is a useful metric.

These metrics can be adapted for use with any of the extensions created for FTA developed over the past two or three decades. In particular, Dynamic fault trees (Dugan, Bavuso, and Boyd 1992) can be described with minor changes.

The metrics described above provide some initial means to determine whether generated trees are "similar" to those found in the literature — at least "similar" enough to be useful for exercising research methods for solving FTs. One more metric is applied, which is somewhat more complicated to describe.

Number of Minimal Cut Sets

The best means to measure the complexity of a FT is to compute the number of possible minimal cut sets (MCS) for that tree.

Traditional methods for qualitatively solving FTs use the rules of Boolean logic to enumerate all of the possible MCS that defeat the tree. While these methods will produce a complete set of answers, it can be computationally expensive for large trees and will not always be able to generate a complete set of solutions in a useful time, nor to store all of the possible combinations produced. However, it is still useful to know how many possible solutions exist, even if they are not enumerated.

We developed a method to compute this number directly without solving the system. In its current form, it will correctly calculate the number of possible MCS for trees that do not reuse systems or components. That is, it works for acyclic trees. As explained above, many FTs are cyclic in the graph theoretic sense; that is, they repeat the use of some systems or components. This method will *not* work for such systems.

The number of MCS is calculated by propagating the counts of component combinations through the gates of the FT. The calculation is different for each type of operator that defines a node in the tree.

If a node has N elements, each with a number of MCS M_1 through M_N, then the total number of MCS for each type of node is:

$$M_{OR} = \sum_{n=1}^{N} M_n$$

$$M_{AND} = \prod_{n=1}^{N} M_n$$

$$M_{KofN} = f(k, 1)$$

Where $f()$ is a recursive function applied to $(M_1, ..., M_N)$ as follows:

$$f(k, i) = M_i \cdot f((k-1), (i+1)) + f(k, (i+1))$$

The first argument, *k*, indicates the number of elements to be chosen; the second, *i*, indicates the element with which to begin. This function is applied recursively to progressively smaller problems until it arrives at one of two cases. If $k = 1$, this indicates 1 of $(N - i)$ elements, which is equivalent to the *OR* formula. If $k = (N - i)$, this indicates $(N - i)$ of $(N - i)$ elements, which calls for the *AND* formula.

The number of MCS is calculated by executing a depth first traverse of the tree. Since every leaf node has exactly 1 MCS, the computation of MCS for the bottom nodes is trivial. The values for each node are propagated up to progressively higher level nodes until the top node is reached.

The accuracy of this formula was verified by comparison with the count of MCS obtained from *xfta* [1] and was proven accurate on systems with up to several million MCS.

Sample Fault Trees from the Literature

An exploration of FTA literature produced a small set of FTs used as examples for various forms of methodology development. The FTs selected for this research are listed here along with a code used to refer to them later in this paper.

Song09:	(Song, Shi, and Li 2009)
Lacey11:	(Lacey 2011)
Sui11:	(Sui and Pan 2011)
Shah12:	(Shahriar, Sadiq, and Tesfamariam 2012)

Although this is far from an exhaustive sample of FTs used in the literature, it does show the types of problems used to illustrate FTA algorithms and methods.

The metrics calculated for these FTs are shown in Table 1. An examination of these metrics reveals some interesting features.

Several systems were found that contain only OR gates. By the laws of boolean algebra, these are the logical equivalent of flat system with a single OR gate of which all components are children. Thus, they can be defeated by killing

[1] *xfta* can be downloaded for free from *www.open-psa.org*

any single component and the number of MCS is exactly equal to the number of components. From an analysis point of view, these are not interesting and were not used in this research.

The FTs gleaned from articles in academic journals tend to be simple not only in number of components, but also in types of operators. None of the FTs cited use the K-of-N operator. Overall, only one FT was found that uses this operator, and it is only used once. Therefore, it was decided to omit this operator from the initial research. They will be addressed in later work.

Finally, only two of the systems have an average in-degree higher than 1.0. An average in-degree of 1.0 indicates that no systems or components are repeated. This is a practice quite common; for example when a power generation system is used by several electrical subsystems.

After discussing the interpreter for L-systems, the Results section will show how these properties can be replicated and then extrapolated using L-systems.

Adapting L-Systems to Generate Fault Trees

To show how L-systems were adapted to generate FTs, consider the following grammar. It has been altered from a normal L-system by adding an operator as the first element of each production rule. In practical terms, when the interpreter executes a rule, it will add the indicated operator to the node and create a number of child nodes below it equal to the number of letters that follow.

Example 1 *L-System Fault Tree Example*

```
start:  A
    A:  (OR) CBB
    B:  (AND) ACC
    C:  (T)
```

Executing the interpreter on the grammar above produces the follow results for two iterations:

Iteration 0:	A
Iteration 1:	(OR) CBB
Iteration 2:	(OR) {(T)} {(AND) ACC} {(AND) ACC}

This grammar produces no leaf nodes until the interpreter reaches the final iteration; consequently, all components will be at the bottom level of the tree. Experience and FT literature shows that most FTs have leaf nodes at many levels of the tree. Therefore, an operator was created to generate leaf nodes through out the system. The operator looks like this: (T) with no letters following.

When the interpreter reaches the number of iterations defined by the user, it will convert all of the leaf nodes into components. This is easier to understand by visualizing the process as illustrated in Figure 1 for the same grammar shown in Example 1.

The algorithm for interpreting the L-system grammar is straightforward. It comprises a file parser, a tree library and a file writer. The resulting FTs were stored in the Open PSA Model Exchange format [2]. This provides a means to analyze them using algorithms developed by the larger reliability research community.

[2]see *www.open-psa.org*

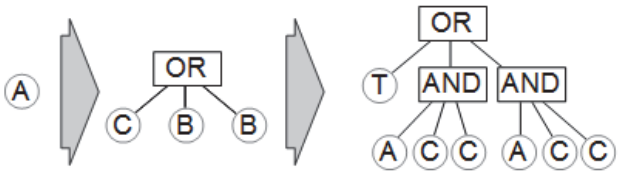

Figure 1: Fault tree generated using simple L-system grammar in Example 1.

One more ability was required to emulate real world FTs: repeating systems from other parts of the tree. The sample systems in (Sui and Pan 2011) and (Shahriar, Sadiq, and Tesfamariam 2012) show this behavior. Adding this capability required a two-fold change to the grammar. First, a "*" before a letter indicates that a system/component may be reused. Second, an (RS) operator indicates that the operator should be replaced with a reused system, while an (RC) indicates the operator should be replaced with a reused component. It is possible for a system to be reused more than once – this is a deliberate design choice since real world systems can exhibit this structure. It is important for the implemention to check to prevent cycles in the tree; otherwise the resulting tree can contain infinite recursion.

Results

To establish the ability of L-systems to match desired metrics, a series of examples was created that match the characteristics of the sample FTs. The characteristics of the reference systems were matched using a trial and error process of defining an initial grammar, executing the parser, checking the metrics to see how close they matched, then adjusting the grammar until the desired behavior was achieved.

The grammars defined using this approach are shown in the examples below and the resulting metrics are shown in Table 1. The reference FTs and their metrics are shown in the same table to facilitate comparison.

Example 2 *L-System to match Song09*

```
start:  A          | C:  (AND) FE
    A:  (OR) BE    | E:  (OR) AF
    B:  (OR) FCF   | F:  (T)
```

Example 3 *L-System to match Lacey11*

```
start:  A          |
    A:  (OR) BE    | E:  (OR) CZ
    B:  (AND) ACE  | F:  (AND) BF
    C:  (OR) FE    | Z:  (T)
```

Example 4 *L-System to match Sui11*

```
start:  A          |
    A:  (OR) *BCD  | E:  (OR) ZZ
    B:  (OR) ACE   | F:  (AND) RE
    C:  (AND) *DRE | R:  (RS)
    D:  (OR) CDZ   | Z:  (T)
```

Table 1: *Metrics for Matching Fault Trees – each reference FT is shown with its matching L-system FT. "i" is number of L-system iterations and "c" is number of components.*

source	*i*	*c*	MCS	OR	AND	out	in
Song09	-	8	7	0.83	0.17	2.17	1
Ex. 2	3	7	6	0.80	0.20	2.20	1
Lacey11	-	17	15	0.71	0.29	2.14	1
Ex. 3	4	17	31	0.71	0.29	2.14	1
Sui11	-	20	24	0.75	0.25	2.83	1.03
Ex. 4	3	20	17	0.73	0.27	2.82	1.03
Shah12	-	42	92	0.70	0.30	2.83	1.15
Ex. 5	5	40	68	0.70	0.30	2.81	1.15

Example 5 *L-System to match Shah12*

```
start:  A
    A:  (OR) BCD     |  F:  (AND) RE
    B:  (OR) *ACZZ   |  G:  (OR) RD
    C:  (AND) *DER   |  H:  (AND) ACE
    D:  (OR) FRZ     |  R:  (RS)
    E:  (OR) GZZ     |  Z:  (T)
```

Although the metrics are not always exact matches for those calculated for the reference systems, they are close and provide useful surrogates for the FTs cited.

To show how L-system FTs scale with further iterations, Example 3 was selected to be grown further. The resulting size and metrics are shown in Table 2. A system with no system reuse was selected because it is possible to calculate the number of MCS analytically. Given the dramatic increase in the number of MCS for the larger FTs, it is not possible, even on very large HPCs, to evaluate all of these cut sets in a useful time.

It should be noted that this system matches the metrics well at a depth of 4, but the metrics asymptotically approach other values at greater depths. The researcher can target a size at which to match the metrics, but any given grammar will not match the metrics for all sizes.

Observing the rapid growth in the number of MCS in comparison with the number of components demonstrates why some methods could encounter performance issues for large problems. Another factor worth noting is that the other metrics converge to within a percent after 5 to 7 iterations. This shows that the method is stable and maintains consistent characteristics as the systems grow larger.

Table 2: *Metrics for Fault Trees derived from Example 3 for 10 iterations. In-degree is 1.0 for all iterations.*

depth	# comps	# MCS	OR	AND	out-degree
2	5	3	0.67	0.33	2.33
3	9	11	0.86	0.14	2.14
4	17	31	0.71	0.29	2.14
5	31	181	0.65	0.35	2.15
6	59	1873	0.60	0.40	2.16
7	113	5.20e+5	0.58	0.42	2.18
8	217	1.58e+10	0.58	0.42	2.17
9	417	1.40e+18	0.58	0.42	2.18
10	799	4.11e+31	0.58	0.42	2.18

One more set of computations serves to illustrate the need for examples of this size. The FTs shown in Table 2 were solved qualitatively using *xfta*, mentioned earlier. The time to evaluate the solutions and the number of MCS enumerated were recorded. In this case, the computations were stopped after the time to compute the MCS passed 60 seconds. While this is not much time, the size of files produced was between .3 and .6GB – the computations were cut short to preserve disk space as much as time. The results are shown in Table 3.

Table 3: *Time to compute and number of MCS found in t seconds for generated fault trees showing p, the proportion of possible results found*

i	*c*	MCS	MCS found	*t*	*p*
5	31	181	181	<1	1
6	59	1873	1873	1	1
7	113	520235	520235	18	1
8	217	1.58e+10	2074513	70	0.00013
9	417	1.40e+018	3136138	142	2.2e-12
10	799	4.11e+031	1925378	114	4.7e-26

The interesting thing to note in Table 3 is the last column, which shows the proportion of MCS found compared to the computed number of possible MCS. Up to more than one hundred components, *xfta* has no trouble evaluating all possible solutions. This is almost three times the largest FT found in the literature. However, as the FTs grow larger, the proportion of solutions found compared to all possible solutions becomes miniscule. At 800 components, the total proportion of solutions found is miniscule, even though they take up .3 GB of disk space.

It is true that for many applications, a subset of the smallest MCS is enough to solve the problem at hand. However, for other applications the solutions must be compared with other criteria. In these cases, for large FTs, it is virtually guaranteed that there will be useful solutions in the vast space of unexplored options. This clearly shows the importance of large test sets for exploring new methods for solving FTs that stress existing methodologies.

Discussion

The results above show that L-systems can be used to create FTs that emulate specific metrics and to grow very large FTs that show characteristics similar to benchmark smaller systems. The approach described in this paper meets the desired behavior defined in the introduction. That is to say it is:

- reproducible – Once a grammar interpretation algorithm is defined, any given grammar will produce the same result, which guarantees that other researchers can reproduce the same FT and verify methodology innovations.
- capable of producing FTs similar to real world FTs – Table 1 shows that L-systems can be defined that produce FTs that closely match metrics for FT systems defined in academic sources or elsewhere.
- scalable in measurable ways from small problems to large problems – Table 2 shows that the grammar in Example 3

scales to much larger systems while demonstrating metrics similar to those seen in the smaller systems. These metrics are seen to converge after only a few iterations.

Ideally, the metrics would be invariant as the systems grow larger; instead, most of them asymptotically approach a static value in 3 to 6 iterations. This is still useful for the intended purpose, since researchers can target specific metrics at desired system sizes.

An additional benefit is that the compact representation can readily be used in academic articles, since it can fully describe a FT generated from L-systems in just a few lines, even if it has hundreds of components.

Future Work

The work described in this paper shows that it is possible to use L-Systems to develop FTs that match metrics for example systems. The method used to generate new systems is manual and somewhat tedious; an important extension for this work will be to develop a generalized algorithm for creating such systems. Another viable approach to matching desired FTs would be to apply genetic algorithms or other progressive refinement techniques to create L-system grammars that match the reference systems. This has the potential to provide very good matches for specific FT metrics at required sizes.

In addition, more refined metrics may help better monitor the characteristics of generated FTs. On the other hand, this would also increase the difficulty of matching the desired behaviors using a trial and error method as applied in this research. This argues more heavily for an automated approach for creating matching FTs.

The motivation for this research was to provide test cases for approaches to solving large FTs constrained by physical proximity. These solution methods will be explored in future papers.

References

Ahammed, F., and Moscato, P. 2011. Evolving l-systems as an intelligent design approach to find classes of difficult-to-solve traveling salesman problem instances. In *Proceedings of the 2011 International Conference on Applications of Evolutionary Computation - Volume Part I*, 1–11. Springer.

Ball, M. 1986. Computational complexity of network reliability analysis: An overview. *IEEE Transactions on Reliability* R-35(3).

Clifton, E. 1999. Fault tree analysis – a history. In *Proceings of the 17th International System Safety Conference*.

Deitz, P.; Reed Jr., H.; Klopcic, J.; and Walbert, J. 2009. *Fundamentals of Ground Combat System Ballistic Vulnerability/Lethality*. Progress in Astronautics and Aeronautics. Reston, VA: AIAA.

Dugan, J.; Bavuso, S.; and Boyd, M. 1992. Dynamic fault-tree models for fault-tolerant computer systems. *IEEE Transactions on Reliability* 41(3).

Frijters, D., and Lindenmayer, A. 1976. *Automata, languages, development*. North-Holland Pub. Co. chapter Developmental descriptions of branching patterns with paracladial relationships, 57–73.

Gauthier, J.; Leduc, X.; and Rauzy, A. 2007. Assessment of large automatically generated fault trees by means of binary decision diagrams. *Journal of Risk and Reliability, Professional Engineering Publishing* 221.

Lacey, P. 2011. An application of fault tree analysis to the identification and management of risks in government funded human service delivery. In *Proceedings of the 2nd International Conference on Public Policy and Social Sciences held in Kuching*.

Lima de Campos, L.; Limão de Oliveira, R.; and Roisenberg, M. 2015. Evolving artificial neural networks through l-system and evolutionary computation. In *Proceedings of the 2015 International Joint Conference on Neural Networks*.

Lindenmayer, A. 1968. Mathematical models for cellular interaction in development. *Journal of Theoretical Biology* 18:280–315.

Liou, C.-Y.; D.-R., L.; Sinak, A.; and Huang, B.-S. 2013. Syntactic senstive complexity for symbol-free sequence. In *Proceedings of the 2013 International Conference on Intelligence Science and Big Data Engineering*, 14–21. Springer.

Martin, G. A.; Hughes, C. E.; Schatz, S.; and Nicholson, D. 2010. The use of functional l-systems for scenario generation in serious games. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, 6:1–6:5. ACM.

Rauzy, A. 2001. Mathematical foundation of minimal cutsets. *IEEE Transactions on Reliability* 50(4).

Rozenberg, G., and Salomaa, A., eds. 1992. *Lindenmayer Systems: Impacts on Theoretical Computer Science, Computer Graphics, and Developmental Biology*. Springer.

Shahriar, A.; Sadiq, R.; and Tesfamariam, S. 2012. Risk analysis for oil & gas pipelines: A sustainability assessment approach using fuzzy based bow-tie analysis. *Journal of Loss Prevention in the Process Industries* 25(3):505 – 523.

Smith, A. 1984. Plants, factals, and formal languages. In *Procedings of the 1984 Conference on Computer Graphics (SIGGRAPH)*, 1–10. ACM Press.

Song, W.; Shi, H.; and Li, Q. 2009. Application of fault tree knowledge in reasoning of safety risk assessment expert system in petrochemical industry. In *Knowledge Engineering and Software Engineering, 2009. KESE '09. Pacific-Asia Conference on*, 167–170.

Stamatelatos, M.; Vesely, W.; Dugan, J.; Fragola, J.; Minarick, J.; and Railsback, J. 2002. *Fault Tree Handbook with Aerospace Applications*. Washington, DC: NASA Office of Safety and Mission Assurance.

Sui, Y., and Pan, X. 2011. Reliability assessment of urban anti-disasters system based on fuzzy fault tree analysis. In *Emergency Management and Management Sciences (ICEMMS), 2011 2nd IEEE International Conference on*, 159–162.

Vesely, W.; Goldberg, F.; Roberts, N.; and Haasl, D. 1981. *Fault tree handbook, NUREG-0492*. Nuclear Regulatory Commission.