# Fault Tree Analysis using Sylvan (multi-core BDDs)

Dennis Aanstoot
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
d.aanstoot@student.utwente.nl

## ABSTRACT

Fault trees are abstractions of real systems that consist of more than one component, and where the reliability of each component can be calculated or estimated. Methods exist that can analyze fault trees. The outcome of the methods can tell the user information about which components are vulnerable for total system failure.

Many computers nowadays have more than one core in its processor, and there are even computers with more processors on the motherboard. This can make the computer a lot faster, but software is not always taking advantage of it.

The purpose of the research described in this paper is creating a tool for static analysis of fault trees using binary decision diagrams that makes use of multiple cores. After that the tool will be benchmarked, to time the speed-up that the ability to use multiple cores can bring.

## Keywords

Fault Trees (FTs), Binary Decision Diagrams (BDDs), multi-threaded.

## 1. INTRODUCTION

In the next paragraphs the theories needed for the research will be discussed.

### 1.1 Fault Trees

A fault tree is a mathematical concept for calculating the risk of failure in complex systems with multiple components.

Fault trees are for example used in aerospace, nuclear reactors and chemical industries to estimate the chances that, the systems they use, will fail. The system can easily be split in components, and the reliability of the components can be tested with experiments. Analyzing the system is for an airplane or reactor of great importance, because of the damage a failure can inflict.

The fault tree represents the system, where the system is divided in its individual components. The leaves of the tree represent the components of the system. The top node represents the system as a whole. Every other node in between the top node and the leaves has a function. Standard functions for Fault Trees are:

| | |
|---|---|
| **AND:** | If all children fail, this node fails. |
| **OR:** | If one of the children fails, this node fails. |
| **k/N:** | If k of the N children fail, this node fails. |
| **INHIBIT:** | The same as AND. Is only used for clarity for readers in some occasions |

When a component of the whole system fails, it depends on the node and its function where it is connected to if it will also fail. If the node also breaks, that will propagate further up, until a node doesn't fail, or the top node is reached. It is also possible that multiple components fail. This means that multiple leaves will propagate a fail signal up, which increases the chance of the whole system to fail.

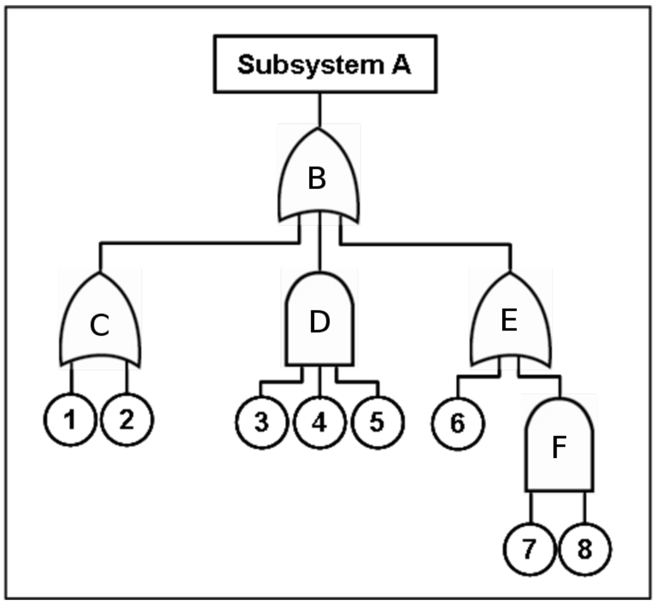

**Figure 1. Example of a Fault Tree**

Figure 1 shows a fault tree representing subsystem A. The nodes B, C and E represent OR nodes, and the nodes D and F represent AND nodes. The numbers 1 to 8 represent the components where subsystem A consists of. If, for example, component 1 fails, node C will fail, and node B will fail. This means the system as a whole will fail.

The nodes in fault trees can have reliabilities, percentages that give the probability the component will fail in a given amount of time.

Multiple methods for analyzing the reliability of the system exist. One of the methods is calculating the Minimal Cut Sets. A cut set is a set of components that together can cause the system to fail. A minimal cut set is a cut sets that is not a subset of the other cut sets.

An other method for analyzing the fault tree is by calculating the Minimal Path Sets. A path set is the set of components such that, is they do not fail, the system will be operational. A minimal path set is a set that is not a subset of the other path sets.

The last analyzing method mentioned in this paper is the Rare Case Approximation, which is the sum of the probabilities of the minimal cut sets failing. It estimates the chance of the whole system failing by analyzing the the most critical sets of components.

More information about fault trees can be found in [7].

## 1.2 Binary Decision Diagrams

A Binary Decision Diagram is an rooted, acyclic, directed graph that represents a boolean function [1]. A boolean function is a function with boolean inputs that will give a true (1) or a false (0) as a result.

Every node also represents a boolean function, and has exactly 2 outgoing edges, one labeled as "1" and one labeled as "0". The outcome of the boolean function of the node depends which edge will be taken.

To construct a BDD from a boolean formula, the Shannon expansion formula can be used to construct the top node [1].

$f(x_1, x_2, ..., x_n) =$
$(x_1 \wedge f(1, x_2, ..., x_n)) \vee (\neg x_1 \wedge f(0, x_2, ..., x_n))$.

BDDs are used in for example model checking, to efficiently represent the state space and transition relation. BDDs are useful for analysis because they can encode fault trees very efficiently.

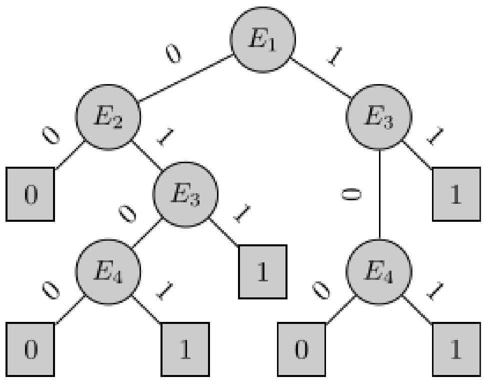

**Figure 2. Example of a Binary Decision Diagram**

A BDD can be evaluated by starting at the root node, and follow the path down by evaluating the boolean functions, following the edge with the solution of each boolean function, and finish at a leaf. This is the outcome of the boolean function of the BDD. Figure 2 gives an example of an BDD. So, if only E1 is failing, the path taken while evaluating is $E1 \rightarrow (1\ edge) \rightarrow E3 \rightarrow (0\ edge) \rightarrow E4 \rightarrow (0\ edge) \rightarrow (0\ leaf)$. This means the system as a whole isn't failing, and will still be running.

The node "E3" and the 1 and 0 leaves can be found multiple times, but they have the same children, so the duplicates are overhead. To save memory, one node "E3" can be removed, along with its children, and all incoming connections can be connected to the other "E3". All connections to a leaf can be connected to the same 0 or 1 leaf. The result of such optimization can be found in Figure 3.

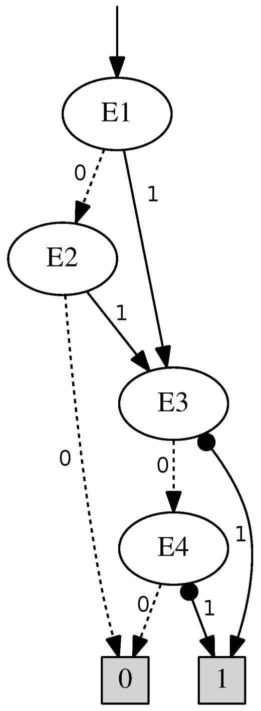

**Figure 3. Example of a Fault Tree converted to a Binary Decision Diagram**

## 1.3 Analysis of Fault Trees

### 1.3.1 Converting a FT to a BDD

A simple way to analyze a fault tree is by converting the fault tree to a binary decision diagram and perform analyse techniques on the BDD.

An efficient way of constructing the BDD is using the Rauzy algorithm[2]. It makes use of the ite function. ite is a function with three arguments, an if statement, a then statement, and an else statement. $(f \wedge G) \vee (\neg f \wedge H)$ results in *ite(f,G,H)* and it can be read as "if f, then G, else H".

By taking a node (N) with children (c1..cn) of a fault tree, the ite function can be used as follows:

If AND node:

$ite(c1, ite(c2, ite(c3, .., false), false), false)$

if OR node:

$ite(c1, true, ite(c2, true, ite(c3, .., false)))$

Figure 4 shows a binary decision diagram and a fault tree that represent the same system. If the nodes of a fault tree have reliabilities, the nodes of the binary decision tree that represent the same component will have the same reliability.

### 1.3.2 Quantitative Analysis

Fault Tree Analysis can be divided in Quantitative Analysis and Qualitative Analysis.

Qualitative Analysis provides insight into the structure of the FT and detects system vulnerabilities. Examples of

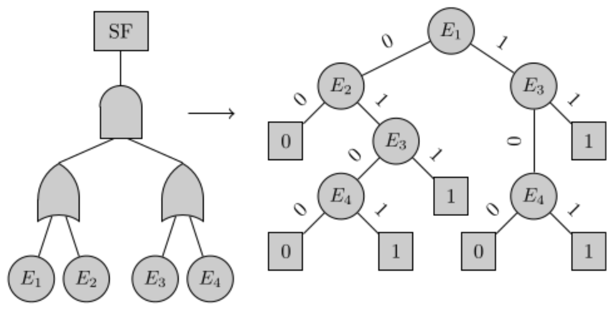

**Figure 4. Example of a Fault Tree converted to a Binary Decision Diagram**

qualitative analyzing are finding the minimal cut sets and minimal path sets in the Fault Tree.

Cut sets are the sets of system components that if they failed, the whole system would fail. Minimal cut sets are cut sets that are not a subset of the other cut sets. The minimal cut sets give insight in which components should need redundancy, or need an increase in reliability, because the components are very important for the system.

For a BDD the minimal cut set can be found by finding the minimal cut sets for all nodes. Getting the minimal cut sets for node N is getting the minimal cut sets of the node at the 1 node and adding N to each set, and combine that set with the minimal cut sets of the node at the 0 edge. What's left is removing cut sets that are not minimal, so sets that are supersets of other cut sets.

Let's take the BDD in Figure 3. For E4 that means that the only cut set is $\{E4\}$. If itself fails, it will result in a 1 leaf. Than for E3 the minimal cut sets can be found by following the 1 edge, that will lead to an 1 leaf, so $\{E3\}$ is a minimal cut set. Combine that will the $\{E4\}$ of the 0 edge and the minimal cut sets for E3 are $[\{E3\}, \{E4\}]$. In this example the minimal cut sets of the whole BDD (E1) are $[\{E1, E3\}, \{E1, E4\}, \{E2, E3\}, \{E2, E4\}]$.

Path sets are the sets of components needed for the system to keep running. The minimal path sets are path sets that are not a subset of other path sets. The knowledge of minimal path sets visualizes which components are important, and shouldn't fail.

The minimal path sets in a BDD can be found the opposite way as the minimal cut sets. The only difference is that the node itself must added to the sets obtained from the 0 edge, and not from the 1 edge. The minimal path sets of the BDD in Figure 3 are $[\{E1, E2\}, \{E3, E4\}]$.

### *1.3.3 Quantitative Analysis*

Quantitative Analysis methods derive relevant numerical values for the Fault Trees.

An example of Qualitative Analysis is Rare Event Approximation. It is the sum of the unavailabilities of all the Minimal Cut Sets. Again, this is more easily evaluated using BDDs.

The rare event approximation is a approximation of the reliability of the fault tree. The reliability of every node can be calculated as follow. For a AND node the formula for the reliability is

$\prod_{C_i \in Children}(C_i)$

For a OR node the formula is

$1 - \prod_{C_i \in Children}(1 - C_i)$

Lets say the reliabilities of the nodes of Figure 1 are:

| | |
|---|---|
| E1: | 0.02 |
| E2: | 0.07 |
| E3: | 0.03 |
| E4: | 0.10 |

Than the unavailability can be calculated.

The OR node above E1 and E2 has a reliability of 1 - (1 - 0.02) * (1 - 0.07) = 0.0886, and the OR node above E3 and E4 have a reliability of 1 - (1 - 0.03) * (1- 0.10) = 0.127. The AND node has a reliability of 0.0886 * 0.127 = 0.0113, so is the reliability of the fault tree as a whole.

In a BDD the rare event approximation gives a approximation of this number. It is the sum of the products of the reliabilities of the members of the minimal cut sets.

$\sum_{M_i in mcs}(\prod_{N_j in M_i} N_j)$

For the BDD in Figure 3, which represents the FT in Figure 3 the rare event approximation is:

$0.02 \times 0.03 + 0.02 \times 0.10 + 0.07 \times 0.03 + 0.07 \times 0.10 = 0.0117$

## 1.4 Multi-threaded

Modern computers have multiple processing units, but they are not always utilized by the computer. This requires the programs that run on the pc to be programmed in a way it can take advantage of the cores. In a recent study a library called Sylvan has been developed for manipulating BBDs that can take advantage of multiple cores [3].

Because CPU's won't get much faster in aspects like clock speed, but instead more cores will be added to CPU's to give them more processing power. To make use of all cores, the processes that run on the computer should be written in a way the tasks can be divided to all cores. This applies only to processes that can be divided. This isn't possible for all processes.

A parallel solution for Fault Tree Analysis will be beneficial to multi-core computers in this modern world.

## 1.5 Sylvan

Sylvan is a parallel (multi-core) BDD library in C developed by the University of Twente. It can store BDDs and perform operations on them. It uses the work-stealing framework Lace to do this multi-threaded. This means it wil make use of all cores of a computer, if it has more than one. More information about Sylvan can be found here [3].

## 1.6 Lace

Lace is a work-stealing framework. Work-stealing is a method for load balancing task-based parallel programs. Workers will be make, which can perform a task, and return the result at a later moment. The framework can be fed tasks, that will be divided among the initiated workers. To use Lace the most efficiently, the amount of workers should be equal to the amount of processing units of the computer and every worker will be kept busy with tasks at all time.

## 2. THE PROGRAM

The main function of the program is to first open a file in the CP format. The CP format defines a Fault Tree. The file is read and a model for the Fault Tree is created. The Fault Tree is then converted to a BDD model in the Sylvan library.

The program has functions to find the minimal cut set, the minimal path set and the rare case approximation. The functions are written with the work stealing framework Lace to parallelize the function and take advantage of the multiple cores of the computer. Moreover, Sylvan already has functions which run in parallel.

## 2.1 Approaches

### *2.1.1 FT to BDD*

A Fault Tree can be converted to a BDD using Rauzy's method[2]. A BDD type can be a node or a leaf. The resulting BDD will represent the top node of the BDD. ithvar will create a node, and gives it the same number as the FT leaf, so we have mapped the FT component with the BDD component.

```
def node_to_BDD(node) {
  retval = invalid;

  if (node.type == OR) {
    for(child in node.childlist) {
      if (child.type == LEAF
          && retval == invalid) {
        retval = ithvar(child.value);
      }
      else if (child.type == LEAF) {
        other = ithvar(child.value);
        retval = ite(other,
          true, current);
      }
      else if (retval == invalid) {
        retval = node_to_BDD(child);
      }
      else {
        other = node_to_BDD(child);
        retval = ite(other,
          true, current);
      }
    }
  }

  else if (node.type == AND) {
    for(child in node.childlist) {
      if (child.type == LEAF
          && retval == invalid) {
        retval = ithvar(child.value);
      }
      else if (child.type == LEAF) {
        other = ithvar(child.value);
        retval = ite(other,
          retval, false);
      }
      else if (retval == invalid) {
        retval = node_to_BDD(child);
      }
      else {
        other = node_to_BDD(child);
        retval = ite(other,
          retval, false);
      }
    }
  }

  return retval;
}
```

### *2.1.2 Analysing a BDD*

The function FT_minimal_cs will create a task for the workers of Lace, so it will run in parallel.The result of the function will be an array of minimal cut sets. Each minimal cut set will be encoded as an array of BDD nodes (bdd).

```
def FT_minimal_cs(bdd) {
  high = get_high_node(bdd);
  low = get_low_node(bdd);
  if(high == true) {
    array[];
    array.add(bdd);
    high_results.add(array);
  }
  else {
    high_results = FT_minimal_cs(high);
    add_node(high_results,bdd);
  }
  if(low != sylvan_false) {
    low_results = FT_minimal_cs(low);

    // combine results
    // and remove not-minimals
    for(low_array in low_results) {
      for(high_array in high_results){
        if(low_array.
            is_subset(high_array){
          high_results.remove(high_array);
          high_results.add(low_array);
          done = true;
          break;
        }
        else if(high_array.
            is_subset(low_array)) {
          done = true;
          break;
        }
      }
      if(!done) {
        high_results.add(low_array);
      }
    }
  }
  return high_results;
}

void add_node(result, bdd){
  for(array in result) {
    array.add(bdd);
  }
}
```

The function FT_minimal_ps will create a job for the workers of Lace, so it will run in parallel. The result of the function will be an array of minimal path sets. Each minimal path set will be encoded as an array of BDD nodes (bdd).

```
def FT_minimal_ps(bdd) {
  high = get_high_node(bdd);
  low = get_low_node(bdd);
  if(low == false) {
    array[];
    array.add(bdd);
    low_results.add(array);
  }
  else {
    low_results = FT_minimal_ps(low);
    add_node(low_results,bdd);
  }
  if(high != sylvan_true) {
    high_results = FT_minimal_ps(high);
```

```
      // combine results
      // and remove not-minimals
      for(high_array in high_results) {
        for(low_array in low_results) {
          if(high_array.
              is_subset(low_array) {
            low_results.remove(low_array);
            low_results.add(high_array);
            done = true;
            break;
          }
          else if(low_array.
              is_subset(high_array)) {
            done = true;
            break;
          }
        }
        if(!done) {
          low_results.add(high_array);
        }
      }
    }
    return low_results;
}

void add_node(result, bdd){
    for(array in result) {
      array.add(bdd);
    }
}
```

The Rare Event Approximation can be found by summing the probability of all Minimal Cut Sets.

```
def FT_rare_event_approximation(bdd) {
      mcs = FT_minimal_cs(bdd);
      result = 0.0;
      for(set in mcs) {
           temp = 1.0;
           for(node in set) {
             temp *= node.get_reliability();
           }
           result += temp;
      }
      return result;
}
```

## 3. TEST SETUP

The tests will be performed on a 48 core computer so that a large amount of Lace workers can be tested on parallelism. 2 implemented methods with the use of Sylvan (minimal cut sets, minimal path sets) will be timed on 1, 2, 4, 8, 16, 32, and 48 cores with models by Rauzy [4]. In the most optimal situation, the processing time will decrease to the time used by one processing unit divided by the amount of processing units.

To compare the rare event approximation and the real reliability, they will both be calculated. Because the models used don't provide probabilities, they will be generated. Each node will be given a random number lower than 0.01.

## 4. TEST RESULTS

Table 1 contains the durations of calculating the minimum cut sets for different fault tree models on different amount of CPUs. In Figure 5 the data can be found in a chart.

**Table 1. This table shows the time taken to calculate the minimal cut sets on different amount of CPUs**

| Model | 48 CPU | 32 CPUs | 16 CPUs |
|---|---|---|---|
| ftr10.cp | 8.585 | 12.071 | 22.544 |
| das9202.cp | 72.872 | 59.084 | 55.488 |
| das9203.cp | 48.110 | 38.503 | 33.367 |
| das9204.cp | 36.341 | 28.692 | 26.917 |
| das9205.cp | 103.223 | 81.746 | 71.933 |
| das9206.cp | 476.522 | 493.905 | 772.005 |

| Model | 8 CPUs | 4 CPU | 2 CPUs |
|---|---|---|---|
| ftr10.cp | 46.404 | 86.968 | 165.090 |
| das9202.cp | 55.317 | 66.466 | 104.061 |
| das9203.cp | 31.898 | 31.200 | 31.284 |
| das9204.cp | 30.208 | 36.104 | 53.126 |
| das9205.cp | 68.541 | 66.764 | 65.877 |
| das9206.cp | 1337.704 | 2524.314 | 4866.159 |

| Model | 1 CPUs | Speedup 1/48 |
|---|---|---|
| ftr10.cp | 298.758 | 34.799 |
| das9202.cp | 174.931 | 2.400 |
| das9203.cp | 33.035 | 0.686 |
| das9204.cp | 97.463 | 2.681 |
| das9205.cp | 65.987 | 0.639 |
| das9206.cp | 9345.069 | 19.610 |

ftr10 and das9206 can keep a steady decrease of processing time. das9202 and das9204 have a decrease till 4 CPUs, where after the processing time will stagnate decreasing. das9203 and das9205 don't get any speedup at all.

Table 2 contains the durations of calculating the minimum path sets for different fault tree models on different amount of CPUs. In Figure 6 the data can be found in a chart. All models show a low to high speed-up, with the exception of das9205.

Table 3 shows the rare event approximation compared with the real reliability of the fault tree. The approximation is for most models close. The rare event approximation has for every model overestimated the real reliability.

## 5. DISCUSSION

The results in Table 1 and Figure 5 show that the parallelism can have good speed-ups, but in some cases no speed-up at all. The fact that the models das9203 and das9205 weren't faster with multiple processing units was probably due to the uneven distribution of OR and AND nodes. The das9203 and das9205 had respectively only 1 and 2 AND nodes and many OR nodes. Analyzing this kind of fault trees led to fast termination of a thread because nodes tend to lead to a leaf quickly. Fault trees that had a more even distribution of OR nodes as AND nodes did well, and had a good speed-up.

The results in Table 2 and Figure 6 show for all models, with the exception of das9205, a lower calculation time when using more processing units to do the calculation. The overall durations where low for the das series because a low amount of AND nodes, which led to a low amount of path sets, so less time was spend on combining the two lists of path sets.

## 6. FURTHER WORK

The program didn't take advantage of a important aspect of BDDs, namely node sharing. While analyzing nodes, intermediate results aren't stored. When a node is passed

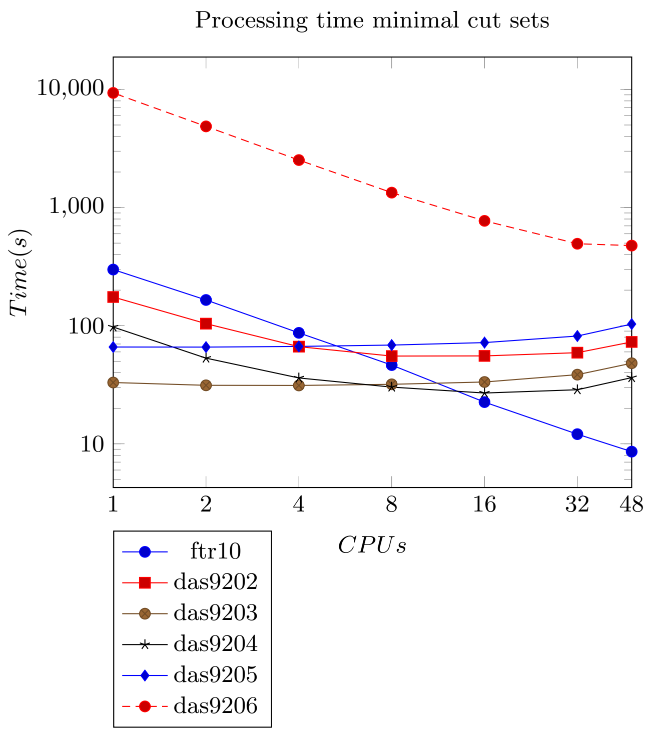

**Figure 5. This plot shows the time taken in seconds to calculate the minimal cut sets on different amount of CPUs**

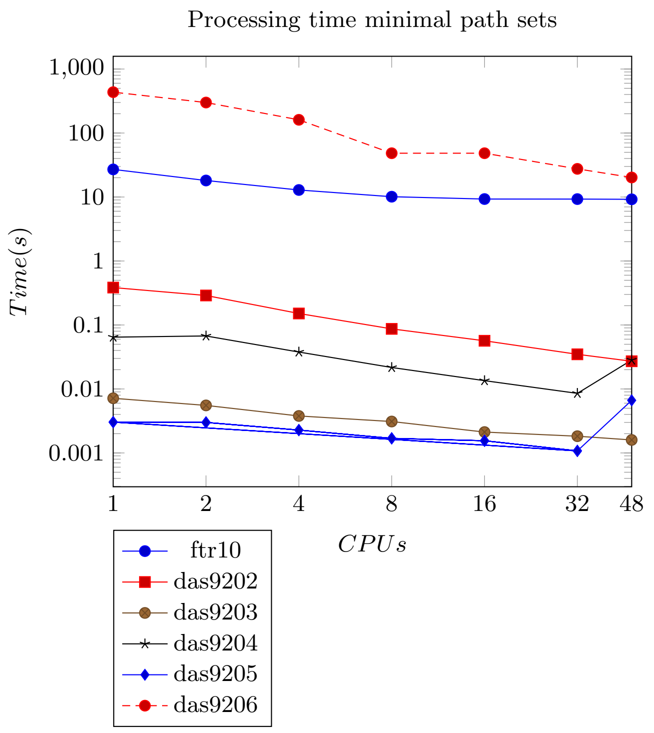

**Figure 6. This plot shows the time taken in seconds to calculate the minimal path sets on different amount of CPUs**

**Table 2. This table shows the time taken to calculate the minimal path sets on different amount of CPUs**

| Model | 48 CPU | 32 CPUs | 16 CPUs |
|---|---|---|---|
| ftr10.cp | 9.203 | 9.277 | 9.292 |
| das9202.cp | 0.0271 | 0.0348 | 0.0566 |
| das9203.cp | 0.0015 | 0.0018 | 0.0021 |
| das9204.cp | 0.0283 | 0.0085 | 0.0134 |
| das9205.cp | 0.0066 | 0.0010 | 0.0015 |
| das9206.cp | 20.200 | 27.531 | 48.334 |

| Model | 8 CPUs | 4 CPU | 2 CPUs |
|---|---|---|---|
| ftr10.cp | 10.106 | 12.872 | 18.114 |
| das9202.cp | 0.0869 | 0.1512 | 0.2891 |
| das9203.cp | 0.0030 | 0.0037 | 0.0055 |
| das9204.cp | 0.0216 | 0.0377 | 0.0673 |
| das9205.cp | 0.0016 | 0.0022 | 0.0029 |
| das9206.cp | 87.273 | 160.863 | 299.534 |

| Model | 1 CPUs | Speedup 1/48 |
|---|---|---|
| ftr10.cp | 26.995 | 2.933 |
| das9202.cp | 0.3843 | 14.178 |
| das9203.cp | 0.0071 | 4.481 |
| das9204.cp | 0.0645 | 2.278 |
| das9205.cp | 0.0030 | 0.456 |
| das9206.cp | 435.109 | 21.539 |

| Model | Rare event approximation | Real reliability |
|---|---|---|
| ftr10.cp | 3.942 | 0.979 |
| das9202.cp | 0.061315 | 0.000004 |
| das9203.cp | 0.106 | 0.025 |
| das9204.cp | 0.000001 | 0 |
| das9205.cp | 0.000509 | 0.000195 |
| das9206.cp | 1.511 | 0.790 |

**Table 3. This table shows the rare event approximation compared to the real reliability of the fault trees**

multiple times, the minimal cut set or minimal path set for that node is calculated the same amount of times. This is time spending and unnecessary. A addition to the the program will be intermediate result storage, so a benefit can be taken from node sharing.

Another improvement will be an other ordering technique for constructing the BDD from the FT. R. Remenyte-Prescott and J.D. Andrews have proposed an ordering (top-down left-right) that probably will result in lower computation times [6].

## 7. CONCLUSION

This paper gives result of a multi-threaded solution of performing analyzing techniques on fault trees. The tool analyzes fault trees that have been converted to BDDs and gives speedup when using more than 1 CPU core. The speed-up can vary depending on the fault tree. For some fault trees the speed-up will keep increase, for some fault trees the speed-up will stagnate after an amount of cores, and for some fault trees there won't be a speed-up. The method isn't always beneficial for the performance when analyzing a fault tree, but it is also never destructive for the performance.

## 8. REFERENCES

[1] S.B. Akers. *Binary Decision Diagrams.* IEEE TRANSACTIONS ON COMPUTERS, VOL. c-27,

NO. 6, JUNE 1978

[2] Andrews, J.D. and Rementye, R *Fault tree conversion to binary decision diagrams.* Loughborough University Institutional Repository

[3] T. van Dijk and J. van de Pol *Sylvan: Multi-Core Decision Diagrams.* Formal Methods and Tools, University of Twente, The Netherlands

[4] Antoine Rauzy *Aralia Fault Trees* http://www.itu.dk/research/cla/externals/clib/Aralia.zip

[5] K.A. Reay and J.D. Andrews. *A Fault Tree Analysis Strategy Using Binary Decision Diagrams.* Loughborough University, Loughborough, Leicestershire, LE11 3TU.

[6] R. Remenyte-Prescott, J.D. Andrews *An enhanced component connection method for conversion of fault trees to binary decision diagrams* Loughborough University, Loughborough LE11 3TU, UK

[7] Enno Ruijters and Mariëlle Stoelinga *Fault tree analysis: A survey of the state-of-the-art in modeling, analysis and tools.* Formal Methods and Tools, University of Twente, The Netherlands