M Ű E G Y E T E M 1 7 8 2

Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

Towards tensor-based reliability analysis of complex safety-critical systems

Scientific Students' Association Report

Author:

Dániel Szekeres

Advisor:

Kristóf Marussy

2019

Contents

Kivonat

A biztonságkritikus komplex kiberfizikai rendszerek helytelen működése emberi életet veszélyeztet, vagy nagy mértékű üzleti kárt okozhat. Az ilyen rendszerek tervezésekor különösen fontos a funkcionális követelményeken túl a különböző extra-funkcionális követelményeket is szem előtt tartani, mint

a biztonság, a megbízhatóság, vagy a rendelkezésre állás. Ezen követelmények alapvetően kvantitatívak, azaz különböző mérőszámokra határoznak meg elérendő célértékeket. Teljesülésük biztosításával már a rendszertervezés fázisában is foglalkozni kell, amikor még nem áll rendelkezésre a működő rendszer, csak annak modellje.

A megbízhatósági és rendelkezésre állási metrikák meghatározásához a rendszer viselkedésére jellemző véletlenszerűséget tartalmazó, sztochasztikus modelleket használunk. A munkámban egy elterjedt sztochasztikus modellezési formalizmust, a hibafákat vizsgálom. Ez a formalizmus a rendszerszintű hiba dekomponálásán alapul: az egyes elemi komponensek meghibásodásának egy logikai függvényével írja azt le.

A modellből a szükséges megbízhatósági mutatók, mint as első meghibásodásig tartó várható időtartam számításához egy alacsonyabb szintű, matematikailag kezelhető analízis modellt kell származtatni. Az analízis modell elkészítésekor és elemzésekor felvetődő probléma az állapottérrobbanás: bár a magas szintű mérnöki modell még kezelhető méretű, a hozzá tartozó analízis modell mérete ennek exponenciális függvénye. Így az elterjedt explicit elemzési módszerek csak korlátozottan skálázhatóak.

A probléma egy lehetséges megoldása, hogy az elemzés során megoldandó lineáris egyenletrendszert tenzorreprezentációs módszerek segítségével, tömör közelítő formában tároljuk, és a megoldást is ebben a formában keressük. Munkám során a Tensor Train (TT) formátumú reprezentáció alkalmazhatóságát vizsgálom a hibafák elemzésére, melyet a szimbolikus modellellenőrzésben elterjedt döntési diagram állapottér reprezentáció segítségével állítok elő.

A számítások elvégzéséhez több különböző iteratív algoritmus áll rendelkezésre TT formátumú egyenletrendszerek megoldására. Munkám során ezeket hasonlítom össze, valamint egészítem ki a hibafákból származó problémák megoldására.

Az elméleti eredmények helyességét különböző, nagy méretű hibafákon végzett mérésekkel támasztom alá, nemcsak az iteratív TT algoritmusok, hanem a széles körben alkalmazott explicit módszerek felhasználásával is.

Abstract

The failure of complex safety-critical cyber-physical systems can endanger human lives, or cause a high amount of financial loss. Throughout the development of such systems, taking extra-functional requirements - like safety, reliability and performance - into account is of utmost importance. These kinds of requirements are mostly quantitative, meaning that they give target values for some metrics of the system that must be achieved. Achieving these values must be assured already in the design phase, when no usable instance of the system under development is available for measurement, only a model describing it.

The reliability and availability metrics are derived using a stochastic model explicitly describing the randomness inherent in the behavior of the system. In my work, I focus on a widely used stochastic modeling formalism called fault trees. This formalism is based on the decomposition of the system-level error: it describes the system-level error as a logic function of the error of the system's elementary components.

To calculate the necessary metrics from the model, like mean time to first failure, a lower level analysis model must be derived from it, that can be handled mathematically. When creating and analyzing this low-level model, the problem of state-space explosion arises: even though the high-level engineering model is of tractable size, the size of the corresponding analysis model is exponential in the original one's size. Because of this, the scalability of widespread explicit analysis methods is limited.

A possible solution for this problem is storing the linear equation system that needs to be solved during the analysis in a concise approximate form using tensor representation methods, and seeking the solution in the same format. I examined the applicability of Tensor Train (TT) methods for fault tree in my work. I use decision diagram-based state space representation for the derivation of the compressed form, which is widely used in symbolic model checking.

There are several different iterative algorithms for solving equation systems in the TT format to perform the necessary calculations. Throughout my work I compare and extend these methods for the solution of problems arising in the analysis of fault trees.

I verify the correctness of the theoretical results through measurements performed on large fault trees, using not just the iterative TT algorithm, but widely used explicit methods as well.

Chapter 1

Introduction

The failure of complex safety-critical cyber-physical systems can endanger human lives, or cause a high amount of financial loss. Throughout the development of such systems, taking extra-functional requirements – like safety, reliability and performance – into account is of utmost importance. These kinds of requirements are mostly quantitative, meaning that they give target values for some metrics of the system that must be achieved. Achieving these values must be assured already in the design phase, when no usable instance of the system under development is available for measurement, only a model describing it. Even if an operational prototype of the the system were available, the requirements imposed on safety-critical systems by various laws, regulations and standards are so strict that statistically measuring the reliability is impossible. For atomic components like electronic parts, accelerated life-time analysis is an option for determining the reliability characteristics, but complex systems are not suitable for this.

The reliability and availability metrics in such cases can be derived using a stochastic model explicitly describing the randomness inherent in the behavior of the system. There are lots of different modeling formalisms available, e.g. stochastic Petri nets and its extensions, stochastic autamata networks, stochastic process algebra, etc. [35, 7]. In my work, I focus on a widely used stochastic modeling formalism called *fault trees* [36]. This formalism is based on the decomposition of the system-level error: it describes the system-level error as a Boolean function of the failure of the system's elementary components. Fault trees are mostly used as a high-level modeling technique, so these elementary components are usually small subsystems, not directly atomic parts.

Several different extensions have been developed for fault trees which allow the modeling of complicated behaviour, like constraining the order in which components can fail or dependencies between components. Such models are called dynamic fault trees, while the original simple formalism is called static. This work focuses only on static fault trees for now.

The computation scheme used in this work can be adapted for a lot of different reliability or performance metrics, like mean time to failure, mean time to first failure, mean time between failures, etc. For now, I focused on *mean time to first the (MTFF)*. MTFF is the expected amount of time until the system enters a failure state. This can often be used when the model is made for probabilistic verification of the system's safety goals. In this case, the individual components of the system can be replaced or repaired, but once the system itself enters a failure state there is no way back. In modeling safety goals, the system-level failure is often interpreted as the event that the system's faulty behaviour caused a catastrophe, like two trains colliding in the case of railway interlocking systems, or a plane crash in case the modeled system is an airplane. These are events that should not happen at all, but assuring this is physically impossible (as hardware components can randomly fail from stress or fatigue), so the regulations and standards prescribe a very-very low non-zero probability for these. Another case when MTFF is a useful metric is the case when no repairs are done, the system is replaced once it enters a failure state.

Once the model is built, there are principally two ways for calculating the system's metrics: Monte-Carlo simulations and mathematical analysis. Simulation is a simple and general way for obtaining the necessary characteristics of the system, but the results of these methods are often unreliable, and too many independent simulations of the system must be run in order to get a good enough convergence. When system-level failure is a very unlikely event, as it mostly is in the case of well-designed safety-critical systems, basic Monte-Carlo simulations have too high variance to converge to a good estimate of the real value, often underestimating the probability of failure. Multi-level Monte-Carlo methods intentionally designed for rare event simulations are a better alternative if the system is too complex for non-statistical analysis, but direct mathematical analysis is still more accurate, so it is still better to use if possible. This work deals with the direct mathematical analysis of the model.

To calculate the necessary metrics from the model, like MTFF, a lower level analysis model must be derived from it, that can be handled mathematically. For static fault trees – similarly to a lot of other stochastic models used in performance and reliability analysis – this model is a (usually continuous-time) Markov chain. When creating and analyzing this low-level model, the problem of *state-space explosion* arises: even though the high-level engineering model is of tractable size, the size of the corresponding analysis model is exponential in the size of the original one, making it potentially too large to handle. Because of this, the scalability of widespread explicit analysis methods is limited.

In this report, I propose a method for overcoming the state space explosion when calculating metrics for static fault trees. It is based on *structured tensor representations*, specifically the *tensor train decomposition* [24], and representing states the operational state set (or by taking its complement, the failed state set) of the fault tree compactly as a *binary decision diagram*, which is a common technique in symbolic model checking.

Contributions The main contribution of this report is three-fold:

- Two mathematically proven formula are given for the computation of mean time to first failure in a fault tree which can be applied to structured representations. These are usable not only for fault trees, but also other mean time to absorption problems in structured Markov chains.
- A method is given for efficiently deriving the structured representation for fault tree MTFF calculation using BDDs. This is a novel approach, at least to my knowledge.
- The method was benchmarked with different iterative solution methods created for the TT format. Even though on smaller models the proposed method was slower than the state-of-the-art optimized model checker model checker, there was one setting that greatly outperformed the baseline on larger models. This shows that the method could also be used in practice for appropriately structured large models that cannot be analyzed using explicit methods.

Structure of the report Chapter 2 reviews the necessary background in stochastics and reliability modeling, and introduces the Tensor Train (TT) format. Chapter 5 provides a summary of the linear equation solvers applicable to systems given in the TT format. Chapter 4 describes the main contribution of the paper: the structured method proposed for MTTF calculation of fault trees. Chapter 6 presents the benchmarking method used to measure the applicability of the proposed method, and the results of the benchmark. Chapter 7 concludes the work and provides a summary of some potential areas of future research in the topic.

Chapter 2

Background

This chapter reviews the necessary mathematical and modeling background.

Section 2.1 introduces *continuous-time Markov chains*, which are used as the low-level mathematical analysis model for the proposed algorithm. Section 2.2 defines *Phase-type probability distributions*, which serve as an important probabilistic model for the time until system failure in reliability analysis.

Section 2.3 describes the *fault tree* modeling formalism. A new analysis method for such models is the focus of this report.

Section 2.4 presents *binary decision diagrams*, which are an used to derive the structured representations used by the algorithm proposed in this report.

Section 2.5 defines the operations of *Kronecker algebra*, and explains the notion of *multi-index notation.* These are often used throughout the report.

Section 2.6 introduces perhaps the most important background for the proposed approach: the *tensor train* format. This format is the structured tensor representation that the proposed method is based on.

Section 2.7 describes an efficient approximate matrix inversion approach for appropriately structured matrices. This approach is used in some versions of the proposed algorithm.

At the end of the chapter, Section 2.8 provides an overview of related works in the literature.

2.1 Continuous-Time Markov Chains

Markov Chains (MCs) form an important mathematical formalism used in stochastic modeling. Intuitively, a Markov Chain models a stateful system, where state transitions happen randomly, but with the Markov property: the future depends on the past only through the present. Markov Chains can have either a continuous or a discrete state space (although some authors refer only to those with discrete state space as Markov chains, and call continuous ones simply Markovian continuous stochastic processes), and can evolve in continuous or discrete time, leading to different types of MCs.

In reliability and performance modeling, MCs usually have a discrete state space, which can be either finite or countably infinite. For example, when dealing with queuing systems, the size of a queue can be modelled as a Markov chain with a countably infinite number of states, if the queue has infinite capacity. Most models in reliability analysis, like fault trees or Petri-nets (with a finite maximum number of tokens) yield MCs with a finite state space. The infinite case needs a very different approach to analyze than the finite case, so I focused only on MCs with a finite state space in my work.

On the time dimension, both discrete and continuous-time modeling can be a reasonable choice when dealing with extra-functional requirement verification, depending on the problem. For now, I considered only a continuous-time treatment of Fault Trees for the computation of mean time to failure (see section 2.3). In a Continuous-Time Markov Chain, state transitions can happen at any point in time, but the probability of taking a given transition in a give time period depends only on the current state of the system, independent of the previous states and the time spent in the current state before the interval. The formal definition of a CTMC is as follows (from [34]):

Definition 1. A stochastic process $\{X(t), t \geq 0\}$ is a *Continuous-Time Markov Chain (CTMC)* if for all integers n and for any sequence $t_0, t_1, \ldots, t_n, t_{n+1}$ such that $t_0 < t_1 < \cdots < t_{n-1} < t_n$, the following equation holds:

$$\begin{aligned} \mathit{Prob}\{X(t_{n+1}) = x_{n+1} | X(t_n) = x_n, X(t_{n-1}) = x_{n-1}, \ldots, X(t_0) = x_0\} = \\ = \mathit{Prob}\{X(t_{n+1}) = x_{n+1} | X(t_n) = x_n\}. \end{aligned}$$

▪

This definition uses the Markov property in continuous time to define a Markov chain, but inspecting CTMCs only through this property is hardly useful. A more useful characterization of CTMCs comes from 2.1, which is described in the following.

For the sake of correctly identifying the scope of the work and the theorems used, a small aside is needed about time-homogeneity.

Definition 2. [34] A CTMC is called *time-homogeneous* if $\mathit{Prob}\{X(t_{n+1}) = x_{n+1} | X(t_n) = x_n\}$ depends only on x_{n+1}, x_n and the difference $t_{n+1} - t_n$. It is called *time-inhomogenous* otherwise.

▪

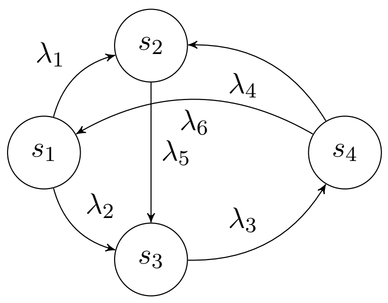
Analyzing time-inhomogenous CTMCs is a challenging task, they are rarely used. The CTMCs derived from Fault Trees and Petri-nets are always time-homogeneous, so I considered only time-homogeneous models in my work. From now on, all CTMCs are implicitly assumed to be time-homogeneous.

We can assemble the *state probability vector* $\boldsymbol{\pi}(t)$, the ith element of which is the probability of being in the ith state at time t. As this is a vector made of all elements of a discrete probability distribution, $|\boldsymbol{\pi}(t)|_1 = 1$, and its elements are non-negative.

With this notation in place, it can be shown (see any textbook on the topic, like [34]), that the time of taking a transition follows an exponential distribution, and the evolution of the probabilities of being in each state can be described by the following linear differential equation, called the *Kolmogorov forward equation*:

$$\frac{d\boldsymbol{\pi}(t)}{dt} = \boldsymbol{\pi}(t)\boldsymbol{Q} \tag{2.1}$$

where $\boldsymbol{\pi}(t)$ is the state probability vector at time t, and $\boldsymbol{Q}$ is called the CTMC's *infinitesimal generator matrix*. It can be computed from the *rate matrix* of the CTMC, in which the diagonal is zero, and the off-diagonal element (i, j) is the rate (parameter) of the exponential distribution describing the amount of time after which the system transitions from state i to state j (0, if transitioning from state i to state j is not allowed). From this



$$\begin{bmatrix} -(\lambda_1+\lambda_2) & \lambda_1 & \lambda_2 & 0 \\ 0 & -\lambda_5 & \lambda_5 & 0 \\ 0 & 0 & -\lambda_3 & \lambda_3 \\ \lambda_6 & \lambda_4 & 0 & -(\lambda_6+\lambda_4) \end{bmatrix}$$

Figure 2.1: CTMC example represented by its state graph and the corresponding infinitesimal generator matrix

matrix, $\boldsymbol{Q} = \boldsymbol{R} - \operatorname{diag}\{\boldsymbol{R}\cdot\boldsymbol{1}\}$. Each row of the infinitesimal generator sums to zero, so it is always singular. An example of a CTMC with its state graph and infinitesimal generator matrix can be seen on figure 2.1.

Markov Chains are widely used in reliability and performance analysis, as several metrics and characteristics of the system can be derived by analysing them. Some examples are:

- Transient analysis, which describes the time evolution of the system, answering questions like what is the probability that the system fails throughout its lifetime, or how long can the system be operated without maintenance if a given maximum failure probability is prescribed
- Mean time to absorption, which will be described in more detail in Section 2.2, is used when calculating the mean time to failure, mean time to failure, mean time to repair, or mean time between failures metrics
- Steady-state analysis tells us how the system approximately behaves after it is left on its own for a long amount of time
- Reward-based analysis can be used to compute metrics like expected energy consumption of a system

2.2 Phase-type Distributions

Phase-type distributions is a family of continuous probability distributions related to Markov chains. They are useful when modeling a random variable representing the amount of time passed until some event happens.

Definition 3. A state of a CTMC is called *absorbing*, if it has no outgoing transitions. This is equivalent for the all the elements of corresponding row in the infinitesimal generator to be zero. ▪

Definition 4. A *phase-type distribution* is the distribution of a random variable representing the time until absorption in a Markov chain with a single absorbing state. ▪

A phase-type distribution can be represented by the infinitesimal generator matrix and initial probability vector of the associated Markov chain. If the states of the CTMC are ordered such that the absorbing state is the last one, the infinitesimal generator and the initial probability vector can be written in the following block form:

$$\boldsymbol{Q} = \begin{bmatrix} \boldsymbol{S} & \boldsymbol{s}_{absorb} \\ \boldsymbol{0}^T & 0 \end{bmatrix}, \ \boldsymbol{\pi}(0) = [\boldsymbol{\sigma} \mid \sigma_{absorb}]$$

With this notation, any moment of the phase-type distributed random variable X can be computed as:

$$\mathbb{E}[X^j] = (-1)^j j! \boldsymbol{\sigma} \boldsymbol{S}^{-j} \mathbf{1} \tag{2.2}$$

Setting j=1 in this formula, we get the formula for the expectation of the random variable:

$$\mathbb{E}[X] = -\boldsymbol{\sigma} \boldsymbol{S}^{-1} \mathbf{1}$$

Computing this formula directly involves taking the inverse of $\boldsymbol{S}$, which is often computationally infeasible if the associated Markov chain has a large number of states. Alternatively, we can decompose this formula into solving the linear system $\boldsymbol{S}\boldsymbol{x} = \mathbf{1}$, and then computing $-\boldsymbol{\sigma}\boldsymbol{x}$, or doing the same in the other direction. Thus, the task of finding the expected value of a phase-type distributed random variable can be reduced to solving a linear system.

Although the original definition of a phase-type distribution involves a Markov chain with a single absorbing state, it can be easily shown that the time until absorption in a CTMC with more than one absorbing state is also a Phase-type distribution.

To see this, we only need to substitute all the absorbing states with a single, abstract absorbing state, and redirect all the transitions originally leading to one of the absorbing states to this state. As there are (by definition) no outgoing transitions from any of the absorbing states, merging them into a single state does not change the behaviour of the Markov chain. The time until absorption in the Markov chain with the merged absorbing state is thus the same as in the original one.

When dealing with a Markov chain with an explicitly represented state space, the absorbing states can easily be abstracted into a single state. In the case of structured representation, such as those I deal with in this work, however, this is not possible, so the extension to multiple failure states is needed. Calculation methods can also be easily extended: instead of eliminating the rows and columns of the single absorbing state, all the rows and columns corresponding to an absorbing state must be eliminated.

2.3 Fault Trees

A Fault Tree (FT) is a model for decomposing the failure event of a complex system into basic failure events. It represents the system Boolean function describing the system failure as a connected directed acyclic graph. The graph has exactly one node which has no outgoing edges, which is called the *top event* of the fault tree.

Nodes without any incoming edges are called *basic events*. A basic event represents the (permanent) failure of a system component, that is considered atomic. Fault trees can also take repairs into consideration, meaning that the component corresponding to a basic event can become working again after a failure. If none of the components are modeled as repairable, the fault tree is called *monotone*, else they are called *non-monotone*.

Nodes with at least one incoming edge (including the top event, if it is not the only node in the graph) are called *gates*. These nodes correspond to the failure of a compound

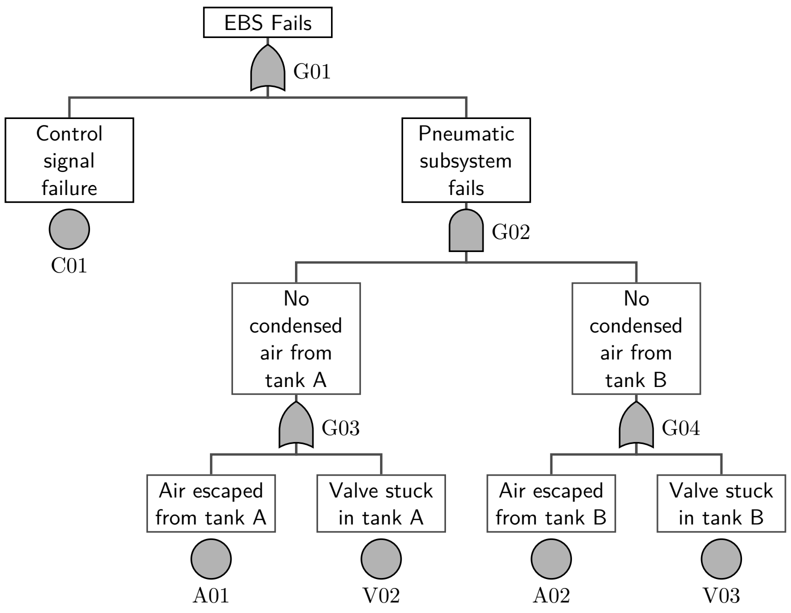


Figure 2.2: Example of a (static) fault tree representing the failure of a pneumatic emergency break system

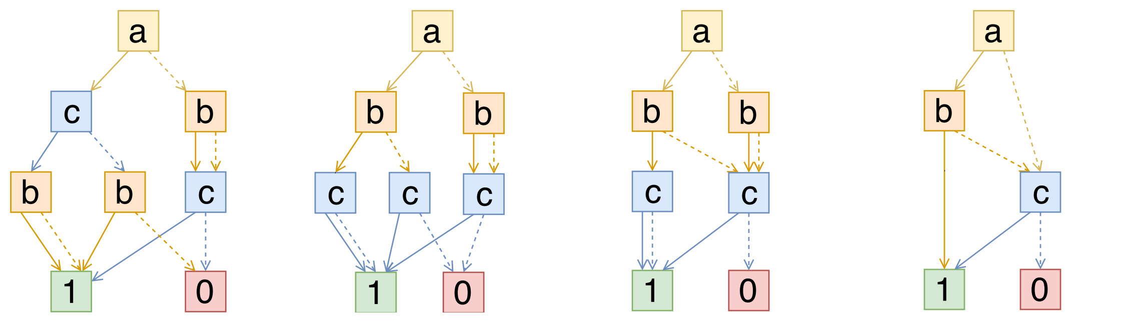
subsystem. Nodes at the end of the incoming edges of a gate are the *input events* of the gate. Gates are typed vertices. In the simplest version of fault trees, called *static fault trees*, there are three types of gates:

- *Or* gates: the subsystem is failed if any of the inputs are failed.
- *And* gates: the subsystem is failed if all of the inputs are failed.
- *Voting* or *K-of-N* gates: these gates have a parameter K, and the corresponding subsystem is failed if at least K of the inputs are failed.

Fault trees can be extended with other, more complicated gates, like dependencies or gates that take the order of failures into account [37]. The probability of the top event of a fault tree with these gates has a more sophisticated time evolution. Such fault trees are called *dynamic fault trees.* In my current work, I focused only on static fault trees. Extending the method to dynamic ones is a potential area for future work. An example of a fault tree can be seen on Figure 2.2.

When analyzing a FT in continuous time, the time until failure or repair of a component is mostly assumed to be an exponentially distributed random variable with failure and repair rates set for each basic event (unrepairable components have 0 repair rate). Combining these exponentially distributed events through static fault tree gates leads to a phase-type distribution for the time until the top event happens.

Mean Time to First Failure (MTFF) is a common metric used in reliability engineering to assess the quality of the system's reliability. It is defined as the expected time until the system's first failure happens. If we model the system failure as the top event of a fault tree, we can compute the MTFF as the expectation of a phase type distributed random variable. As described in section 2.2, this can be done by solving a linear equation system after assembling the infinitesimal generator matrix of the associated Markov chain.



(a) BDD with no constraints **(b)** ordered BDD **(c)** quasi-reduced ordered BDD **(d)** reduced ordered BDD

Figure 2.3: BDDs representing the function $f(a, b, c) = (a \wedge b) \vee c$, with solid lines signifying 1 edges and dashed lines signifying 0 edges.

2.4 Binary Decision Diagrams

Definition 5. A *Binary Decision Diagram* is a rooted DAG with one or two terminal nodes of out-degree zero labeled 0 or 1 and a set of non-terminal nodes of out-degree two. Non-terminal nodes are labeled with a corresponding *variable* (more than one node can be labeled with the same variable), and the edges of a non-terminal node correspond to choosing the value of the node's variable. A path from the root to a terminal node thus means choosing the values of the variables the way the edges on the path specify them.

A BDD is *ordered* if on all paths through the graph, the variables respect a given linear order. In this case, the nodes of the graph are organised into levels, each of the levels corresponding to a *variable*, except for the last one, which consists of the terminal nodes.

An ordered BDD is *reduced*, if it satisfies the following properties (where $u[0]$ and $u[1]$ means the node at the end of the outgoing edge of u labeled 0 and 1, respectively):

- Uniqueness: $(\, u[0] = v[0] \wedge u[1] = v[1] \,) \implies u = v$
- Non-redundant tests: $u[0] \neq u[1]$ ▪

BDDs were originally created to represent Boolean functions in a compact format. The variables of the BDD are the inputs of the function, the edges are the values of the inputs, and the terminal node reached by choosing the appropriate edges for each node is the output of the function. A reduced ordered BDD is a canonical representation of a Boolean function this way.

Another canonical form can be given if the non-redundant tests constraint is dropped, and instead all paths leading from the root to the terminal level are required to go through exactly one node in each level. Such BDDs are called *quasi-reduced*. This practically means that all the input variables are considered in each path, even if their value does not matter. Although this means storing redundant nodes, this form is needed for the structured representations used in this work. The computations can be implemented in such a way that while building the BDD, the more efficient fully-reduced format is used internally. Throughout the report, "BDD" refers to a quasi-reduced ordered binary decision diagram.

An example for different BDD types is given in Figure 2.3.

A subset of a state space can be represented similarly if the states can be decomposed into binary state variables: in this case, a Boolean function can be given which takes the state variables as inputs, and returns 1, if the state is included in the set, and 0 otherwise. Efficient algorithms exist for performing set operations, like union, intersection or difference, on sets represented by BDDs this way.

The set of failure states in a fault tree can be represented as a BDD using the state space decomposition described above, by using the state of the basic events as binary state variables (0 - operational, 1 - failed), and representing the Boolean function encoded by the tree's gates as a BDD.

2.5 Kronecker algebra and multi-index notation

Kronecker algebra defines two operations between matrices: the Kronecker product, denoted by $\boldsymbol{A} \otimes \boldsymbol{B}$, and the Kronecker sum, denoted by $\boldsymbol{A} \oplus \boldsymbol{B}$. These operations are defined as follows:

$$\begin{aligned}(\boldsymbol{A} \otimes \boldsymbol{B})[i_1 r_{\boldsymbol{B}} + i_2, j_1 c_{\boldsymbol{B}} + j_2)] &= \boldsymbol{A}[i_1, j_1] \cdot \boldsymbol{B}[i_2, j_2] \\ \boldsymbol{A} \oplus \boldsymbol{B} &= (\boldsymbol{A} \otimes \boldsymbol{I_B}) + (\boldsymbol{I_A} \otimes \boldsymbol{B})\end{aligned}$$

where $r_{\boldsymbol{B}}$ and $c_{\boldsymbol{B}}$ denote the number of rows and columns of $\boldsymbol{B}$, respectively. The Kronecker product is essentially an every-element-by-every-element product. When selecting an element of the result, it must be specified which element of each input matrix is taken. Because of this, the result has $r_{\boldsymbol{A}} r_{\boldsymbol{B}}$ rows and $c_{\boldsymbol{A}} c_{\boldsymbol{B}}$ columns.

Example 1 (Kronecker product).

$$\boldsymbol{M} = \begin{bmatrix}1 & 2\\3 & 4\end{bmatrix} \otimes \begin{bmatrix}1 & 0\\0 & 2\end{bmatrix} = \begin{bmatrix}1 & 0 & 2 & 0\\0 & 2 & 0 & 4\\3 & 0 & 4 & 0\\0 & 6 & 0 & 8\end{bmatrix}$$

A more natural indexing convention for the result is the *multi-index notation*: instead of a single row index and a single column index, two indices are used for each, specifying the element indices in the input matrices separately. With this a convention, an element of a matrix resulting from a Kronecker product is denoted as follows:

$$\boldsymbol{C}[(i_1, i_2), (j_1, j_2)] = \boldsymbol{C}[i_1 r_2 + i_2, j_1 c_2 + j_2)]$$

This notation can be also be used with more than two indices when the indexed matrix is the Kronecker product of more than two matrices:

$$\begin{aligned}\boldsymbol{C} &= \bigotimes_{k=1}^{N} \boldsymbol{A}_k \\ \boldsymbol{C}[(i_1, i_2, \dots, i_N), (j_1, j_2, \dots, j_N)] &= \prod_{k=1}^{N} \boldsymbol{A}_k[i_k, j_k]\end{aligned}$$

Example 2 (Multi-index notation). *With* $\boldsymbol{M}$ *from the previous example:*

$$\begin{aligned}\boldsymbol{M}[(2,2),(2,2)] &= 4 \cdot 2 = 8\\ \boldsymbol{M}[(2,1),(1,1)] &= 3 \cdot 1 = 3\end{aligned}$$

Considering n long vectors as $n \times 1$ matrices extends the operations naturally to vectors. The multi-index notation can be used in this case, too. Only one index group is used in this case:

$$\boldsymbol{v} = \bigotimes_{k=1}^{N} \boldsymbol{u}_k$$

$$\boldsymbol{w}[(i_1, \dots, i_N)] = \prod_{k=1}^{N} \boldsymbol{u}_k[i_k]$$

Multi-index notation is also used when a vector or a matrix is not the result of a Kronecker operation, but its elements can more naturally be indexed by an index group (two index groups, in case of matrices) instead of a single index. This is the case for most of the vectors and matrices in this work. For example, the state probability vector of the underlying Markov chain of a fault tree can be indexed by specifying for each basic event if the corresponding component is failed or operational, instead of specifying the index of the state directly in the product state space.

Kronecker algebra is used in two ways in this work. First, some linear algebra operations for tensor trains are implemented using Kronecker products. Second, the rate matrix of the Markov chain generated by the basic events of a fault tree can be computed as the Kronecker sum of the 2-by-2 rate matrices of the individual basic events:

$$\boldsymbol{R} = \bigoplus_{i} \begin{bmatrix} 0 & \lambda_i \\ \mu_i & 0 \end{bmatrix}$$

where i ranges through the indices of basic events, λ_i is the failure rate, and μ_i is the repair rate of the ith basic event.

2.6 The Tensor Train Format

The *tensor train (TT)* format has been defined by Oseledets in [24]. The format itself has already been used in the quantum physics community under the name *Matrix Product State*. The decomposition is formally defined originally for *tensors*, which in this area are essentially multi-way arrays. It can also be used to represent vectors and matrices, when indexing the vector or matrix can be done using index groups, like the ones in Section 2.5 used to index the result of Kronecker operations. The vector or matrix does not have to be Kronecker structured, but the closer it is to such a structure, the more compact the tensor train representation is. The places in the index group are called *modes*.

A vector $\boldsymbol{v}$ or a matrix $\boldsymbol{M}$ represented as a tensor train is given in the following form using the multi-index notation:

$$\boldsymbol{v}[(i_1, i_2, \dots, i_n)] = \boldsymbol{V}_1(i_1) \cdot \boldsymbol{V}_2(i_2) \cdot \dots \cdot \boldsymbol{V}_n(i_n)$$
$$\boldsymbol{M}[(i_1, i_2, \dots, i_n), (j_1, j_2, \dots, j_n)] = \boldsymbol{M}_1(i_1, j_1) \cdot \boldsymbol{M}_2(i_2, j_2) \cdot \dots \cdot \boldsymbol{M}_n(i_n, j_n)$$

The components $\boldsymbol{V}_k$ and $\boldsymbol{M}_k$ are called *core tensors* or *cores* (they are also called tensor carriages, hence the name “tensor train”, but core is more common). These are three way tensors, but they can be simply considered a bunch of matrices, and the matrix is chosen by the corresponding index. In some TT articles define the cores as matrix functions, as they take as input an integer (two, in the case of matrices), and return a matrix. In others they are referred to as parameter-dependent matrices. Each core corresponds to a mode. The kth core consists of $r_{k-1} \times r_k$ matrices, with $r_0 = r_n = 1$, so that the result of the multiplications is a scalar. The integeres r_k are called the *ranks* or *TT-ranks* (to distinguish it from the rank commonly used in linear algebra) of the tensor train.

The format comes with efficient algorithms for basic linear algebra operations. The following list presents these. Most of these operations are applicable in the same way to vectors and matrices, so they are not presented separately. The matrix case can be reduced to the vector case by considering the kth row and column index as a single kth index (for example, a row index and a column index with range 1 to 4 is treated as a single index with range 1 to 16).

- Product with a scalar: computing $\alpha\boldsymbol{A}$ can be simply done by multiplying each matrix in the first core of $\boldsymbol{A}$ by α
- Addition: given two vectors $\boldsymbol{a}$ and $\boldsymbol{b}$ with the same dimensions in the TT format as $\boldsymbol{a}[(i_1, i_2, \ldots, i_n)] = \boldsymbol{A}_1(i_1)\boldsymbol{A}_2(i_2)\ldots\boldsymbol{A}_m(i_m)$ and $\boldsymbol{b}[(i_1, i_2, \ldots, i_n)] = \boldsymbol{B}_1(i_1)\boldsymbol{B}_2(i_2)\ldots\boldsymbol{B}_m(i_m)$, their sum $\boldsymbol{c} = \boldsymbol{a} + \boldsymbol{b}$ can be computed in the TT format by setting its cores to:

$$\begin{aligned} \boldsymbol{C}_1(i_1) &= \begin{bmatrix} \boldsymbol{A}_1(i_1) & \boldsymbol{B}_1(i_1) \end{bmatrix} \\ \boldsymbol{C}_k(i_k) &= \begin{bmatrix} \boldsymbol{A}_k(i_k) & 0 \\ 0 & \boldsymbol{B}_k(i_k) \end{bmatrix} \; for \, k = 2, \ldots, m-1 \\ \boldsymbol{C}_m(i_m) &= \begin{bmatrix} \boldsymbol{A}_m(i_m) \\ \boldsymbol{B}_m(i_m) \end{bmatrix} \end{aligned}$$

- Hadamard (element-wise) product: given two vectors of the same dimensions $\boldsymbol{a}$ and $\boldsymbol{b}$ in the TT format as $\boldsymbol{a}[(i_1, i_2, \ldots, i_n)] = \boldsymbol{A}_1(i_1)\boldsymbol{A}_2(i_2)\ldots\boldsymbol{A}_m(i_m)$ and $\boldsymbol{b}[(i_1, i_2, \ldots, i_n)] = \boldsymbol{B}_1(i_1)\boldsymbol{B}_2(i_2)\ldots\boldsymbol{B}_m(i_m)$, their Hadamard product $\boldsymbol{c} = \boldsymbol{a} \circ \boldsymbol{b}$, the cores of $\boldsymbol{c}$ in the TT format are

$$\boldsymbol{C}_k(i_k) = \boldsymbol{A}_k(i_k) \otimes \boldsymbol{B}_k(i_k),$$

where $\otimes$ is the Kronecker product.

- The scalar product of two vectors of the same size, defined as

$$\langle \boldsymbol{a}, \boldsymbol{b} \rangle = \sum_{i_1, i_2, \ldots, i_m} \boldsymbol{a}[(i_1, i_2, \ldots, i_m)]\boldsymbol{b}[(i_1, i_2, \ldots, i_m)]$$

can be efficiently computed in the TT format by using the following computations:

$$\begin{aligned} \boldsymbol{\Gamma}_k &= \sum_{i_k} \boldsymbol{A}_k(i_k)\boldsymbol{B}_k(i_k) \\ \boldsymbol{v}_k &= \boldsymbol{v}_{k-1}\boldsymbol{\Gamma}_k \text{ with } \boldsymbol{v}_1 = \boldsymbol{\Gamma}_1 \end{aligned}$$

then $\boldsymbol{v}_m$ is a scalar, and $\langle \boldsymbol{a}, \boldsymbol{b} \rangle = \boldsymbol{v}_m$.

- The Euclidean norm of a vector – which generalizes to the Frobenius norm in the matrix case – can be computed using the previously mentioned method for the scalar product:

$$||\boldsymbol{a}||_F = \sqrt{\langle \boldsymbol{a}, \boldsymbol{a} \rangle}$$

- Matrix-vector product: Given a matrix $\boldsymbol{M}[(i_1, i_2, \ldots, i_n)] = \boldsymbol{M}_1(i_1)\boldsymbol{M}_2(i_2)\ldots\boldsymbol{M}_m(i_m)$, and a vector $\boldsymbol{x}[(i_1, i_2, \ldots, i_n)] = \boldsymbol{X}_1(i_1)\boldsymbol{X}_2(i_2)\ldots\boldsymbol{X}_m(i_m)$ in the same tensor structure, their product $\boldsymbol{y} = \boldsymbol{M}\boldsymbol{x}$ can be computed in the TT format by computing its cores as

$$\boldsymbol{Y}_k(i_k) = \sum_{j_k}(\boldsymbol{M}_k(i_k, j_k) \otimes \boldsymbol{X}_k(i_k))$$

- Transposition: transposition can be done by simply rearranging the matrices in a core, so that the originally $(i_k * columns + j_k)$th matrix of the core tensor becomes the $(j_k * rows + i_k)$th one.

- Outer product of vectors: the outer product $\boldsymbol{X} = \boldsymbol{v}\boldsymbol{w}^T$ of two vectors represented using the above format can be computed in the TT matrix format described above using the following cores:

$$\boldsymbol{X}_k(i \cdot columns + j) = \boldsymbol{V}_k(i) \otimes \boldsymbol{W}_k(j)$$

- Matrix-matrix product: this operation should only be used when it is really necessary and unavoidable, as the cores of the resulting tensor train will be very large if used in an iteration step. The cores for the resulting matrix $\boldsymbol{C} = \boldsymbol{A}\boldsymbol{B}$ are:

$$\boldsymbol{C}_k(i, j) = \sum_{p}(\boldsymbol{A}_k(i, p) \otimes \boldsymbol{B}_k(p, j))$$

Apart from being able to represent structured matrices and vectors compactly, another advantage of the tensor train format is that the size of the representation can be automatically reduced even further if the representation does not need to be exact. This is done through the tensor train rounding algorithm. This algorithm takes as input a tensor train and a rounding tolerance, and returns a tensor train that has TT-ranks less than or equal to those of the original, and which represents a matrix/vector not further from the original one in Frobenius norm than the specified tolerance.

Given a vector $\boldsymbol{a}$ in the TT format a compressed version $\tilde{\boldsymbol{a}}$ with ε relative tolerance in Frobenius norm, meaning that $\frac{||\boldsymbol{a}-\tilde{\boldsymbol{a}}||_F}{||\boldsymbol{a}||_F} \leq \varepsilon$ can be computed using the following steps:

1. compute truncation parameter $\delta = \frac{\varepsilon}{\sqrt{m-1}}$ where m denotes the number of cores in $\boldsymbol{a}$

2. orthogonalize the tensor using QR decomposition of the cores

3. compute truncated SVD of the unfolding matrix of each core by dropping singular vector with corresponding singular value $\sigma_k < \delta$. For each core from left to right, with the truncated SVD decomposition denoted as $\tilde{\boldsymbol{U}}\tilde{\boldsymbol{\Sigma}}\tilde{\boldsymbol{V}}$, keep $\tilde{\boldsymbol{U}}$ folded back as the new core, and merge $\tilde{\boldsymbol{\Sigma}}\tilde{\boldsymbol{V}}$ into the matrices in the next core before computing the next SVD.

For details and pseudo code of the algorithm, see [24].

Most iterative algorithms for solving systems of linear equations can be implemented using only these operations. The rounding operation is needed because the size of the cores of the solution vector grows in each iteration (see the formula for addition and matrix-vector product), so the iteration vector needs to be compressed using the rounding algorithm described above.

The tensor train decomposition can be considered an extension of decomposing a vector or matrix into a Kronecker product of smaller components, as a Kronecker product can be represented as a tensor train with all ranks 1. To convert the Kronecker product into a tensor train, the components of the product must be simply flattened into cores consisting of 1×1 matrices. If the Kronecker product of the matrices $\boldsymbol{C}_k$ is computed, then the (i, j)th matrix in the kth core is just $\boldsymbol{C}_k[i, j]$.

2.7 Approximate matrix inversion through exponentiation

A method for approximating the inverse of a non-singular matrix given as a Kronecker sum was used in [29]. It is based on the following identity, which gives an efficient computation method for exponentiating a large matrix if it has a Kronecker sum structure:

$$e^{\boldsymbol{A} \oplus \boldsymbol{B}} = e^{\boldsymbol{A}} \otimes e^{\boldsymbol{B}}$$

This means that when a large matrix has a Kronecker sum structure, then its exponential can be obtained by computing the exponentials of the Kronecker sum terms, which are usually small, and then computing the Kronecker product of the results. Approximating the function $1/x$ is possible through a sum of exponentials:

$$\boldsymbol{A}^{-1} = \sum_{i=0}^{\infty} \alpha_i e^{\beta_i \boldsymbol{A}}$$

with some scalars α_i and β_i. These scalars can be computed in various ways. Similarly to [29], I used a method based on sinc quadrature. The inverse can be approximated by truncating the infinite sum. This approximation is used in the computation of the MTFF for inverting $\boldsymbol{R} + \gamma \boldsymbol{I}$, where $\boldsymbol{R}$ is the rate matrix of the CTMC generated by the basic events of a fault tree, and the diagonal term is added to make it non-singular (see Chapter 4 for details).

2.8 Related works

Calculating reliability and performance metrics of complex system by exploiting the inherent structure has been an area of ongoing research for a long time. A well-studied method of this principle is using Kronecker algebra for decomposing the infinitesimal generator matrix of a Markov chain into smaller matrices, and using iterative linear equation system solution methods without assembling the full matrix [8, 21].

The main disadvantages of such methods are that they are not applicable generally, only for system having exactly the necessary structure, they cannot be used when the state space is so large that vectors used in the analysis must also be compressed, and they can store the system only exactly, no memory reduction can be made even when only an approximate solution is sought. Often, the potential state space of the system has the necessary structure, and so some analysis e.g. steady-state computation is possible through Kronecker methods, but this structure is destroyed when failure states must be considered absorbing, like in the case of MTFF computation.

Tensor representation methods, like the Hierarchical Tucker decomposition (HTD) or the Tensor Train decomposition (TT) can overcome these problems. These methods exploit that state probabilities and rate matrices/infinitesimal generator matrices of a CTMC can be treated as a high-dimensional tensor (essentially, a multi-dimensional array) by decomposing the state space into domains of state variables. These tensors can then be represented using structured formats (HTD or TT), which can represent any matrix or vector exactly, although if it is not well structured, the decomposition might have a large size. There are approximation and rounding algorithm which can reduce the size of these representations, if exact representation is not required, making a trade-off between size and accuracy possible. Steady-state computation methods have been proposed using these formats in several works [6, 19, 5], and the TT format was also used for mean time to absorption computation when the system has only a single absorbing state in [29].

Several different methods have been proposed for quantitative analysis of fault trees, for a survey of these see [30]. Using BDDs for fault tree analysis was proposed first in [26]. This method is based on computing a BDD representing the Boolean function encoded by the fault tree's gates. The approach is used to speed up both qualitative analysis of the tree, where the most important calculation is determining the minimal sets of events that lead to a system-level failure, and quantitative analysis, like computing the probability of the top event given the probability of the individual basic events. Since then, a lot of methods were built on this idea, e. g. [4].

Approximation techniques in fault tree analysis are often based on dropping failure cases where more than a given number of components are needed for the system level failure, because they are considered unlikely enough. The method proposed here relies on approximating the solution of the linear equation system arising when calculating the mean time to first failure instead. This means that no failure cases are totally ignored, which leads to higher fidelity to the specified system architecture, and a tunable accuracy parameter.

The state-of-the-art probabilistic model checking tool Storm [9], which is used as a baseline for the benchmarks in this report, uses a state-space generation technique based on several optimization methods for fault trees described in [37].

Chapter 3

Overview of the proposed method

The main difficulty in computing the MTFF of a large fault tree is that it can involve the solution of an intractably large linear equation system. This system can often not be explicitly stored in a computer's memory. If the number of basic events is sufficiently large, there may be cases when even the solution vector cannot be stored explicitly. Most state-of-the-art methods for MTFF computation try to address this through computing only the reachable state space through some state space exploration method. However, the reachable state space can still be exponentially large in the number of basic events, and the state space exploration can take a lot of time.

A possible solution for this problem is using structured tensor representations instead of the explicit representation. There are several different decomposition techniques that can be used as such representations [13, 18, 24]. However, a lot of them still suffer from the fact that they cannot represent high dimensional cases compactly enough, or that no efficient algorithm is available for computing at least a quasi-minimal representation of a matrix efficiently. The tensor train format, however, does not have these problems. It is also a very simple format with most linear algebraic operations also straightforward to implement. It was chosen as the structured representation used for the proposed method for these reasons.

The central idea of the proposed method is computing the mean time to first failure metric using vectors and matrices represented in the tensor train format. The method proposed mainly for stochastic automata networks in [29] used the same principal idea, but worked only for a single failure state, which makes it unusable for fault trees. The method proposed here extends this to multiple failure state. The most novel part of the method is using BDDs in the process of creating the necessary tensor train representations.

The method tries to exploit the inherent structure in both the potential state space and the reachable state space, by using a tensor train representation of both the infinitesimal generator matrix of the original Markov chain induced by the basic events, and the indicator vector of the operational states.

The representation of the indicator vector is derived by first encoding the Boolean function represented by the fault tree's gates as a binary decision diagram, and then assembling the cores based on the BDD. The ranks of the TT created this way depend on the number of nodes in each level of the BDD. The BDD computation uses operations that guarantee, that the number of nodes is minimal, if the variable ordering is fixed, so this is an efficient method of creating a TT representation with small ranks.

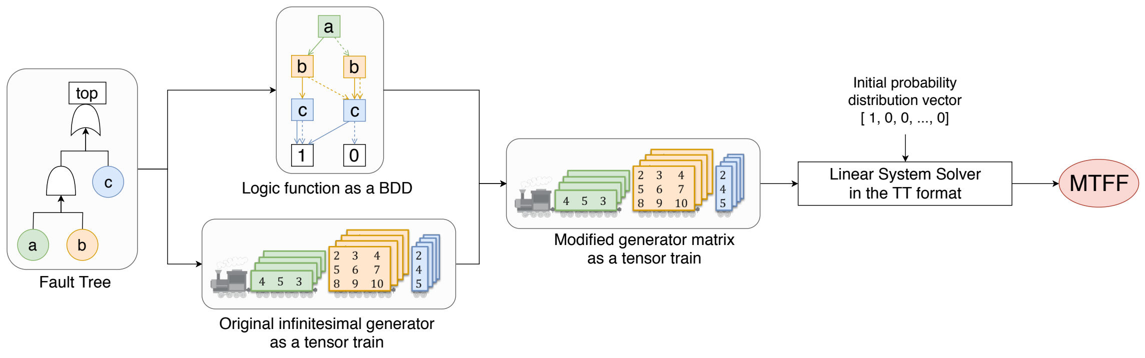

top
a
b
c
Fault Tree
a
b
b
c
c
1
0
Logic function as a BDD
Original infinitesimal generator as a tensor train
Modified generator matrix as a tensor train
Initial probability distribution vector [1, 0, 0, ..., 0]
Linear System Solver in the TT format
MTFF

Figure 3.1: High-level overview of the proposed MTFF computation process from a fault tree

The alternative would be to use either the SVD-based tensor train creation algorithm (TT-SVD) from the explicit vector, or the black-box decomposition based on cross approximation [23] (TT-cross). TT-SVD would need firs explicitly storing the large vector, which might not be even possible. TT-cross is a black-box tensor approximation method, meaning that it needs only a function which returns the value of an element given its position, so it does not need explicit storage of and computations on a potentially intractably large vector. However, it is only an iterative approximation, and even if it converges, it might be stuck in a local minimum, and the ranks of the tensor train must be determined a priori, which also limits its applicability.

Another limitation that both algorithms suffer from is that they can calculate only a quasi-minimal representation. The direct result of the BDD-based approach is also not guaranteed to be minimal, even though the BDD itself is minimal, as tensor trains can use numeric operations (addition and multiplication) and decimal values for the representation, which are not available to BDDs. However, a TT-rounding with 0 tolerance (or, even better, a non-zero value of the scale of machine precision) can be applied after the computation. This way, the minimization capabilities of the BDD and the TT-SVD algorithm (which TT-rounding is based on) are combined in the result.

The computation consists of the following steps:

- A binary decision diagram is built representing the operational states of the system. A tensor train can easily be derived from this representation of the state set.
- The infinitesimal generator matrix or the rate matrix of the Markov chain induced by the basic events is assembled directly in the tensor train format using the failure and repair rates of each basic event.
- A modified version of the generator matrix is computed using the indicator vector of the operational state set based on one of the theorems in Section 4.1, which is suitable for MTFF computation.
- A linear equation system is created in the TT format from this modified matrix and the initial probability vector of the underlying Markov chain. This system is solved using one of the solution methods described in Chapter 5.
- The MTFF metric is finally computed by computing the scalar product of the solution vector obtained in the previous step and a vector of 1s. As the solution is given in the tensor train format, and the vector of 1s can be represented as a tensor

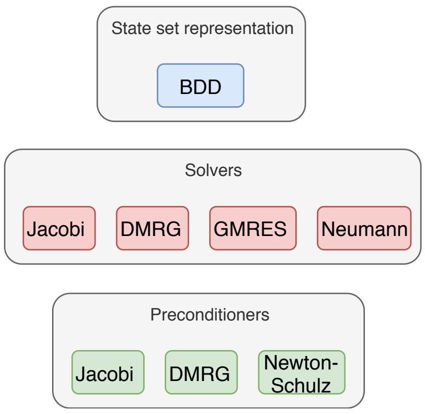


Figure 3.2: Overview of the implemented algorithms

train with all TT-ranks 1, the scalar product can be computed efficiently without evaluating each element of the vector.

A graphical overview of the proposed MTFF computation process can be seen on Figure 3.1. Figure 3.2 provides an overview of the algorithms that were implemented. The multiple solvers and preconditioners provide several options for solving the linear system arising in the proposed approach.

Chapter 4

Creating tensor trains for MTFF computation

This chapter describes the proposed structured MTFF computation method in detail.

Section 4.1 presents two theorems, which provide formulas for MTFF computation which are suitable for use with structured representations. Here, the theorems are described in a quite general manner using general Markov chains, so that the theorems can be used in other contexts. In the case of computing the MTFF of a static fault tree, the Markov chain that is generated by the basic events without taking the system level failure into account plays the role of the *original Markov chain* in the structured computation theorems, and making the states where the whole system is failed absorbing results in the *modified Markov chain*.

Section 4.2 explains most details of the computation process.

Section 4.3 describes the formulas and algorithms used for tensor train creation for the computations.

4.1 Structured MTFF Computation

This section presents and proves the central theorems which make the proposed method possible. Structured representations make computing the expectation of a phase-type distribution by using the generator matrix without the rows and columns corresponding to absorbing states impossible, so another formula is needed. Although these theorems will be used in this report for static fault tree analysis, they are presented in a much more general setting for Markov chains, so that they can be used in other contexts, too.

Let us consider a CTMC with a (possibly empty) set $\mathcal{O}$ of absorbing states, rate matrix $\boldsymbol{R}$, and initial probability vector $\boldsymbol{\pi_0}$, and call this the *original Markov chain*. Take another CTMC with set of absorbing states $\mathcal{F} \supseteq \mathcal{O}$, created from the original Markov chain by deleting the outgoing transitions of the new absorbing states, and having the same initial probability vector $\boldsymbol{\pi}_0$ and call this the *modified Markov chain*. Our aim is to compute the time until absorption in the modified Markov chain using the rate matrix of the original Markov chain. It is also assumed that the initial probability of being in a state that is an element of $\mathcal{F}$ is 0. An example can be seen on Figure 4.1.

The following notations are used in this section:

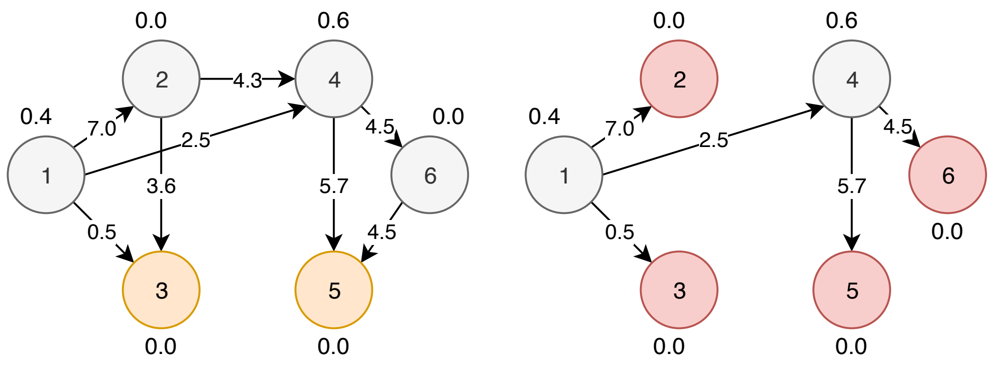


Figure 4.1: Example of an original and modified Markov chain pair, as defined in this section. Initial state probabilities are written next to the circle denoting the given state. States that are elements of $\mathcal{O}$ are colored orange in the original MC. States that are elements of $\mathcal{F}$ are colored red in the modified one.

- $\mathbf{1}_{\mathcal{S}}$ refers to the *indicator vector* of the set $\mathcal{S}$ consisting of indices, defined as:

$$\mathbf{1}_{\mathcal{S}}[i] = \begin{cases} 1 & \text{if } i \in \mathcal{S} \\ 0 & \text{if } i \notin \mathcal{S} \end{cases}.$$

- $\text{diag}\{\boldsymbol{v}\}$ (where $\boldsymbol{v}$ is a vector) is a diagonal matrix with the diagonal's elements defined by $\boldsymbol{v}$
- $\text{diag}\{\boldsymbol{M}\}$ (where $\boldsymbol{M}$ is a matrix) is a vector created from the diagonal of $\boldsymbol{M}$
- $\mathcal{S}^{\complement}$ is the complement of the set $\mathcal{S}$

The following lemma will be needed for the proof of the theorems:

Lemma 1. Let $\hat{\boldsymbol{Q}}_O$ denote the matrix obtained by omitting the rows and columns corresponding to states in $\mathcal{F}$ from the generator matrix of the original Markov chain, and $\hat{\boldsymbol{Q}}_M$ denote the matrix obtained similarly from the generator matrix of the modified Markov chain. Then, $\hat{\boldsymbol{Q}}_O = \hat{\boldsymbol{Q}}_M$. ▪

Proof. The most important part to see to prove this equality is that the two rate matrices differ only in the rows corresponding to states in $\mathcal{F}$. Thus, after obtaining the generators from the rate matrix by subtracting the sum of the appropriate row from each diagonal element, the rows of the states in $\mathcal{F}^{\complement}$ are still equivalent in the two matrices. As only these rows are kept after the omission, the resulting matrices cannot be different. ▪

Theorem 1 (Structured MTFF Computation 1). Define the *perturbed rate matrix* $\bar{\boldsymbol{R}}$ and the *modified generator matrix* $\bar{\boldsymbol{Q}}$ of the original Markov chain as:

$$\bar{\boldsymbol{R}} = \boldsymbol{R} + \gamma \cdot \boldsymbol{I}$$
$$\bar{\boldsymbol{Q}} = \bar{\boldsymbol{R}} \cdot \text{diag}\{\mathbf{1}_{(\mathcal{F}^{\complement})}\} - \text{diag}\{\bar{\boldsymbol{R}} \cdot \mathbf{1}\}$$

where γ is an arbitrary non-zero constant. If $\boldsymbol{\pi}_0^T \mathbf{1}_{\mathcal{F}} = 0$ (that is, the probability of starting in an absorbing state is 0), then using this modified generator matrix, the time until absorption in the modified Markov chain can be calculated as:

$$\mathbb{E}\{t_{absorb}\} = -\boldsymbol{\pi}_0 \bar{\boldsymbol{Q}}^{-1}\mathbf{1}$$ ∎

Proof. Let us start by reordering the states of the Markov chains, such that the states in $\mathcal{F}$ are the last states, and the ordering among the states in $\mathcal{F}$ and $\mathcal{F}^{\complement}$ is preserved, and denote the corresponding permutation matrix $\boldsymbol{P}$. The reordered original rate matrix is denoted by $\boldsymbol{R'} = \boldsymbol{P}^T\boldsymbol{R}\boldsymbol{P}$, and all the other reordered vectors and matrices are denoted similarly by adding a prime to the non-reordered notation. Then, the reordered modified generator matrix can be written in a block matrix form by dividing the rows and columns between $\mathcal{F}^{\complement}$ and $\mathcal{F}$:

$$\bar{\boldsymbol{Q}}' = \begin{bmatrix} \hat{\boldsymbol{Q}} & \mathbf{0} \\ \boldsymbol{C} & \boldsymbol{D} \end{bmatrix}$$

The upper right block is the zero matrix, as multiplication by $\mathrm{diag}\{\mathbf{1}_{(\mathcal{F}^{\complement})}\}$ zeros out all the columns corresponding to states in $\mathcal{F}$, and the subtracted term puts elements only into the diagonal, which stay in the diagonal after the reordering, and thus are contained in $\boldsymbol{D}$. $\hat{\boldsymbol{Q}}'$ is exactly the matrix that could be used in the original method for the mean time to absorption computation: it is the generator matrix of the modified Markov chain with the rows and columns corresponding to the absorbing states removed. This can be seen directly from Lemma 1. $\boldsymbol{D}$ is a non-singular diagonal matrix, whose diagonal values are $-(\lambda_i^{\text{exit}} + \gamma)$, where the λ_i^{exit} are the exit rates in the original Markov chain (sum of the outgoing transitions from the given state). $\boldsymbol{C}$ is an arbitrary matrix, which contains the rate of transitions from $\mathcal{F}$ to $\mathcal{F}^{\complement}$. Using the inversion formula for 2-by-2 block matrices based on the Schur complement, the inverse of $\bar{\boldsymbol{Q}}'$ is:

$$\begin{aligned}\bar{\boldsymbol{Q}}'^{-1} &= \begin{bmatrix} \boldsymbol{I} & \mathbf{0} \\ -\boldsymbol{D}^{-1}\boldsymbol{C} & \boldsymbol{I} \end{bmatrix} \begin{bmatrix} (\hat{\boldsymbol{Q}} - \mathbf{0}\cdot\boldsymbol{D}\cdot\boldsymbol{C})^{-1} & \mathbf{0} \\ \mathbf{0} & \boldsymbol{D}^{-1} \end{bmatrix} \begin{bmatrix} \boldsymbol{I} & -\mathbf{0}\cdot\boldsymbol{D}^{-1} \\ \mathbf{0} & \boldsymbol{I} \end{bmatrix} = \\ &= \begin{bmatrix} \boldsymbol{I} & \mathbf{0} \\ -\boldsymbol{D}^{-1}\boldsymbol{C} & \boldsymbol{I} \end{bmatrix} \begin{bmatrix} \hat{\boldsymbol{Q}}^{-1} & \mathbf{0} \\ \mathbf{0} & \boldsymbol{D}^{-1} \end{bmatrix}\end{aligned}$$

As $\boldsymbol{\pi}_0^T \cdot \mathbf{1}_{\mathcal{F}} = 0$, the reordered initial probability vector $\boldsymbol{\pi}_0'$ can be partitioned as $[\hat{\boldsymbol{\pi}}_0 \,|\, \mathbf{0}^T]$, by dividing between states in $\mathcal{F}^{\complement}$ and $\mathcal{F}$. Using these identities, the statement of the theorem can be shown:

$$\begin{aligned}\boldsymbol{\pi}_0\bar{\boldsymbol{Q}}^{-1}\mathbf{1} &= (\boldsymbol{\pi}_0'\boldsymbol{P}^T)\cdot(\boldsymbol{P}\bar{\boldsymbol{Q}}'\boldsymbol{P}^T)^{-1}\cdot\mathbf{1} = \boldsymbol{\pi}_0'\boldsymbol{P}^T\cdot\bar{\boldsymbol{Q}}'^{-1}\cdot\mathbf{1} = \\ &= [\hat{\boldsymbol{\pi}}_0 \,|\, \mathbf{0}^T] \begin{bmatrix} \boldsymbol{I} & \mathbf{0} \\ -\boldsymbol{D}^{-1}\boldsymbol{C} & I \end{bmatrix} \begin{bmatrix} \hat{\boldsymbol{Q}}^{-1} & \mathbf{0} \\ \mathbf{0} & \boldsymbol{D}^{-1} \end{bmatrix}\mathbf{1} = \\ &= [\hat{\boldsymbol{\pi}}_0 \,|\, \mathbf{0}^T] \begin{bmatrix} \hat{\boldsymbol{Q}}^{-1} & \mathbf{0} \\ \mathbf{0} & \boldsymbol{D}^{-1} \end{bmatrix}\mathbf{1} = \hat{\boldsymbol{\pi}}_0\hat{\boldsymbol{Q}}^{-1}\mathbf{1}\end{aligned}$$ ∎

The last expression is exactly the original formula for the mean time to absorption in the modified Markov chain, and so the statement is proven.

Theorem 2 (Structured MTFF Computation 2). Define the *modified generator* as:

$$\bar{\boldsymbol{Q}} = \boldsymbol{Q} - \boldsymbol{Q}\cdot\mathrm{diag}\{\mathbf{1}_{\mathcal{F}}\} - \mathrm{diag}\{\mathbf{1}_{\mathcal{F}}\}\cdot\boldsymbol{Q} + \gamma\cdot\mathrm{diag}\{\mathbf{1}_{\mathcal{O}}\}$$

where γ is some non-zero constant. If $\boldsymbol{\pi}_0^T\mathbf{1}_{\mathcal{F}} = 0$, then the time until absorption in the modified Markov chain can be calculated as:

$$\mathbb{E}\{t_{absorb}\} = \boldsymbol{\pi}_{\mathbf{0}}\bar{\boldsymbol{Q}}^{-1}\mathbf{1}$$ ∎

Proof. First, let us examine how the second and the third term affects the result. Let us call these the *column mask* and the *row mask* respectively. Considering only one of them, the mask would make all the elements of the columns or rows corresponding to states in $\mathcal{F}$ zero. However, when both terms are applied, the terms that were in both masks are subtracted twice, so there additive inverse is present in the result. The effect of the masks on the elements can be summarized as follows:

- row $\notin \mathcal{F}$, column $\notin \mathcal{F}$: unaffected
- row $\notin \mathcal{F}$, column $\in \mathcal{F}$: zeroed out
- row $\in \mathcal{F}$, column $\notin \mathcal{F}$: zeroed out
- row $\in \mathcal{F}$, column $\in \mathcal{F}$: multiplied by -1

The last term is only there so that $\bar{\boldsymbol{Q}}$ is non-singular.

From here, the proof goes similarly to the previous one, starting with the same reordering. The block form looks like following in this case:

$$\bar{\boldsymbol{Q}}' = \begin{bmatrix} \hat{\boldsymbol{Q}} & \mathbf{0} \\ \mathbf{0} & \boldsymbol{D} \end{bmatrix}$$

as described above. Here, $\boldsymbol{D}$ is no longer diagonal, but still non-singular because of the last term in the definition of $\bar{\boldsymbol{Q}}$. The statement can now be proven more simply:

$$\begin{aligned} \boldsymbol{\pi}_0\bar{\boldsymbol{Q}}^{-1}\mathbf{1} &= (\boldsymbol{\pi}_0'\boldsymbol{P}^T)\cdot(\boldsymbol{P}\bar{\boldsymbol{Q}}'\boldsymbol{P}^T)^{-1}\cdot\mathbf{1} = \boldsymbol{\pi}_0'\boldsymbol{P}^T\cdot\bar{\boldsymbol{Q}}'^{-1}\cdot\mathbf{1} = \\ &= [\hat{\boldsymbol{\pi}}_0 \,|\, \mathbf{0}^T] \begin{bmatrix} \hat{\boldsymbol{Q}} & \mathbf{0} \\ \mathbf{0} & \boldsymbol{D} \end{bmatrix} \mathbf{1} = \hat{\boldsymbol{\pi}}_0\hat{\boldsymbol{Q}}^{-1}\mathbf{1} \end{aligned}$$

■

4.2 Description of the MTFF computation scheme

Based on Theorems 1 and 2 in Section 4.1, the expected time until absorption of a Markov chain can be computed using a structured representation, like the TT format. Using one of the provided formulas, the time until absorption in a CTMC can be computed without really dropping columns and rows of the infinitesimal generator matrix, and they are usable even if the infinitesimal generator is given only for a version of the Markov chain where a subset of the states that should be absorbing still have non-zero outgoing transition rates. This makes it possible to use structured representations like the TT format to compute the MTFF. Although the formula of Theorem 1 might produce a TT with smaller rank directly, the result of the other formula might have a better compression rate when the TT rounding is applied, which cannot be determined without running numerical experiments on both.

The state change of the basic events can be described as a Markov chain, which will be referred to as the *original Markov chain* of the fault tree. The MTFF of the fault tree is the time until absorption in a version of the original Markov chain where all the outgoing transitions from the failure events have been dropped. This will be referred to as the *modified Markov chain*. These Markov chains have the same role as the original and modified Markov chains in Section 4.1, respectively. The rate matrix $\boldsymbol{R}$ of the original Markov chain can be represented as a Tensor Train as given in Section 4.3. From that, the infinitesimal generator can be computed as $\boldsymbol{Q} = \boldsymbol{R} - \operatorname{diag}\{\boldsymbol{R} \cdot \mathbf{1}\}$, which involves only operations that can be efficiently implemented with tensor trains. Section 4.3 also describes how the indicator vector of the failure states can be represented as a tensor train, which means that all the vectors and matrices are available in the TT format that are needed to compute the modified generator matrix that can be used in the MTFF computation defined by either Theorem 1 or 2.

After calculating the matrices and vectors as defined in any of the structured MTFF computation theorems, the solution of a linear system is needed. In case of Theorem 1, the TT ranks of the coefficient matrix of the system to solve can be reduced if the perturbed rate matrix $\bar{\boldsymbol{R}}$ is lifted out of the addition, leading to the formula MTFF $= -\boldsymbol{\pi}_0(\operatorname{diag}\{\mathbf{1}_{(\mathcal{F}^\complement)}\} - \bar{\boldsymbol{R}}^{-1}\operatorname{diag}\{\bar{\boldsymbol{R}} \cdot \mathbf{1}\})^{-1}\bar{\boldsymbol{R}}^{-1}\mathbf{1}$.

The TT-ranks of this is likely smaller than of the original, as the ranks of $\operatorname{diag}\{\mathbf{1}_{(\mathcal{F}^\complement)}\}$ can be large, making the ranks of the result even larger if this is multiplied by $\bar{\boldsymbol{R}}$, while the ranks of subtracted term are usually small. However, this leaves its inverse before the second term, so a matrix inversion for this low-rank matrix must also performed. As this matrix can be written as a Kronecker sum, this inversion can be done using the exponential sum approximation described in Section 5.4.

Using the built-in math library of the JVM, the number of terms in the exponential sum has an upper limit, as the computation of the coefficients results in NaNs when a higher number of terms is used because of the numerical limits of the exponential function's implementation. This limits the accuracy of the inversion achievable with this algorithm. The result of this approximation can be fine-tuned using a few sweeps of the DMRG-based inversion scheme, though, so this inverse can be efficiently approximated using these two steps.

When the iterative solvers are used with rounding of the iteration vector, a rounding tolerance must be given. In general, it's hard to determine this tolerance based on the requested residual norm, as it is mostly unknown how much a given perturbation in the solution vector affects the residual. In our case, however, an upper bound on the spectral radius of the system matrix is known based on the Gerschgorin circles [31], which is in turn an upper bound of the ratio of the norm of the solution's perturbation and the norm of the residual's change.

Lemma 2.

$$\lambda_{out}^{max} = \sum_i \max(\lambda_i, \mu_i)$$

where λ_i and μ_i are the failure and repair rates of the ith basic event, and λ_{out}^{max} is the highest exit rate in the original Markov chain, before removing any transitions. $2 \cdot \lambda_{out}^{max}$ is an upper bound for the spectral radius of the modified generator matrices of Theorems 1 and 2. ▪

Proof. In any state of the original Markov chain, an outgoing transition with rate λ_i exists, if the ith component is operational in the given state, and an outgoing transition with μ_i exists, if the ith component is down. There are no other transitions, and these two are mutually exclusive. The means, that in any state, the exit rate is given by $\sum_i \nu_i$, where

$$\nu_i = \begin{cases} \lambda_i, & \text{if the } i\text{th component is operational} \\ \mu_i & \text{if the } i\text{th component is down} \end{cases}$$

This sum is maximal when for each i, the higher of the two rates is chosen. It is also easy to see that this state exists in the original Markov chain. From here, the upper bound on the spectral radius directly follows from the Gerschgorin circles theorem for the original Markov chain's generator: each circle's center and radius is the corresponding state's exit rate. The Gerschgorin circles corresponding to the rows of the non-failure state in the modified generators have the same center, and a smaller radius, so no eigenvalue falling into these circles can have a larger absolute value than the original chain's spectral radius. The remaining Gerschgorin circles have a center that is chosen by the dummy value written into the diagonal of the modified generator matrix, and the radius is smaller than the radius of the corresponding circle in the original chain's generator. The dummy constant can be chosen small, so that it does not modify the spectral radius. The radius of the circles λ_{out}^{max} is also the infinity norm of the diagonal of the generator matrix, which is needed for the solution method based on exponential sums. ▪

To compute the MTFF of a system, the initial state probability vector is needed, as can also be seen from the computation formulas. In our case, this is $[1, 0, \ldots, 0]$, as the system starts from a state where all components are operational, which is the first element of the whole system's state space. This can be represented by a tensor train with all TT-ranks 1: in each core, the matrix with index 0 is 1, and the one with index 1 is zero.

After the linear system is built with the coefficient matrix and the right hand side vector represented as tensor trains, the system has to be solved, returning the solution also as a tensor train. Some possible solvers available for tensor trains are described in Chapter 5. Once the solution is obtained, it scalar product must be computed with the vector of 1s. This vector can be represented by a tensor train with TT-ranks $\mathbf{1}$, which make it efficient to compute the scalar product. The result of this scalar product times -1 is then the mean time to first failure of the fault tree.

4.3 Tensor Trains from Fault Trees

4.3.1 Representing the original rate matrix

As seen from the theorems in Section 4.1, the MTFF of a fault tree can be calculated using the rate matrix (or the infinitesimal generator matrix, which can be derived from the rate matrix) of the CTMC induced by the basic events. This matrix has a structure that makes tensor trains a remarkably efficient representation for it. For each basic event, a state variable can be defined, which is set to 1, if the corresponding component is in a failed state, and 0 if it is operational. Therefore, the whole state space of this CTMC is $\times_{i=1}^{N} \{0, 1\}$, where N is the number of basic events.

The rate matrix represented as a tensor train can be obtained using the following rules:

- Two events (failure or repair) happen at the same time with zero probability, so a transition from state A to state B has rate 0, if A and B differ in at least to state variables
- In any other case, we know that exactly one event happened. This transition has rate equal to the basic event's failure rate or repair rate, depending on which type of event happens (failure means changing the state variable from 0 to 1, repair means the opposite).

The matrix can be directly computed as a Tensor Train, by choosing its cores so that they satisfy the above rules. To be able to do this, two accumulators are needed between the cores: one for the transition rate, and one that is used to avoid a second event happening. Therefore, each inner TT rank is 2. λ_i and μ_i are the failure and repair rates of the component corresponding to the ith basic event respectively. The cores are chosen as follows:

- The first core:

$$\boldsymbol{R}_1(0,0) = \begin{bmatrix} 0 & 1 \end{bmatrix} \qquad \boldsymbol{R}_1(0,1) = \begin{bmatrix} \lambda_1 & 0 \end{bmatrix}$$
$$\boldsymbol{R}_1(1,0) = \begin{bmatrix} \mu_1 & 0 \end{bmatrix} \qquad \boldsymbol{R}_1(1,1) = \begin{bmatrix} 0 & 1 \end{bmatrix}$$

- Middle cores:

$$\boldsymbol{R}_k(0,0) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \qquad \boldsymbol{R}_k(0,1) = \begin{bmatrix} 0 & 0 \\ \lambda_k & 0 \end{bmatrix}$$
$$\boldsymbol{R}_k(1,0) = \begin{bmatrix} 0 & 0 \\ \mu_k & 0 \end{bmatrix} \qquad \boldsymbol{R}_k(1,1) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

- Last core:

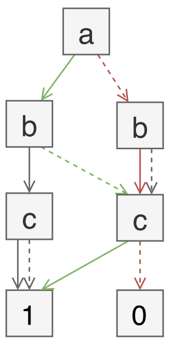
$$\boldsymbol{R}_N(0,0) = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \qquad \boldsymbol{R}_N(0,1) = \begin{bmatrix} 0 \\ \lambda_N \end{bmatrix}$$
$$\boldsymbol{R}_N(1,0) = \begin{bmatrix} 0 \\ \mu_N \end{bmatrix} \qquad \boldsymbol{R}_N(1,1) = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

4.3.2 Calculating indicator vectors using decision diagrams

The indicator vector of the failed state (or the operational ones) are also needed for the structured MTFF computation. This vector can be easily computed from a BDD (see Section 2.4) representing the operational states. The first application of binary decision diagrams for fault tree analysis was proposed in [26], and it has been a widely used approach since then.

BDDs can be used as an efficient way to represent sets of states, when the states can be decomposed into binary state variables: the levels of the BDD in this case correspond to binary state variables, the labeled edges represent the value of the variable of the previous level, and the set consists of the states that end with the terminal 1 node if a path with appropriate labels is chosen from the root. In this case, storing the paths going to the terminal 0 node is redundant, so they are omitted in practice. Efficient algorithms exist for performing set operations on sets represented by BDDs. The set of operational or failed states in a fault tree can be represented as a BDD by using the state space decomposition described above.

The size of a BDD largely depends on the ordering of the variables. A lot of static heuristics and dynamic minimization methods have been proposed for finding a good ordering, as finding the best one is generally too hard. A survey and benchmark results of some static methods can be found e.g. in [28]. Based on the results published in that thesis, I chose the *bottom-up weighted ordering* scheme. This algorithm assigns weights to the nodes of the fault tree calculated in a bottom-up manner based on the type of the node and its



$$\begin{array}{cc} \text{Core[0]} & \text{Core[1]} \\ \begin{bmatrix} 0 & 1 \end{bmatrix} & \begin{bmatrix} 1 & 0 \end{bmatrix} \\ \begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix} & \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \\ \begin{bmatrix} 1 \\ 0 \end{bmatrix} & \begin{bmatrix} 1 \\ 1 \end{bmatrix} \end{array}$$

$$\begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = 1$$

$$\begin{bmatrix} 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = 0$$

Figure 4.2: Example of the conversion between BDD and tensor train with two paths in the BDD and the corresponding element computation in the tensor train highlighted.

inputs. After computing the weights, it orders the basic events in the order they are found through a depth-first traversal of the tree with higher priority given to nodes with higher weights. The chosen ordering must also be used when constructing the tensor train for the rate matrix or the infinitesimal generator so that it is compatible with the indicator vector.

The tensor train for the indicator vector of the set of failed states can be assembled from the computed BDD by using the adjacency matrices between two levels as the matrices in the tensor cores. Each core of the indicator vector has mode length 2. The matrix with index 0 in the core is the adjacency matrix of the 0-edges between corresponding levels, and the matrix with index 1 is the adjacency matrix of the 1-edges. The last core corresponds to the edges between the last variable's nodes and the terminal node 1.

Lemma 3. The tensor train whose cores are the adjacency matrices of the BDD representing the set $\mathcal{S}$ as described above represent the vector $\mathbf{1}_\mathcal{S}$ ▪

Proof. The computation of an element of the represented vector specified using the multi-index convention is done by going through the cores from left to right, choosing the matrix from the ith core according to the ith index in the index group, and multiplying these matrices. Going from left to right, the result of each multiplication before the last one has exactly one 1 in it, and the other elements are zero, because each node has exactly one 1-edge and exactly one 0-edge.

The place of the 1 in each result specifies the index of the node chosen at that level. As the last core specifies which nodes of the last non-terminal level are connected to the 1 terminal node, the result of the last multiplication is 0 if the chosen path ends at the 1 terminal node and 0 otherwise. Figure 4.2 shows an example of this conversion. ▪

Using the conversions described in this section, a tensor train representation is obtained for all the vectors and matrices that are needed for the formula in any of the structured MTFF computation theorems. As described in Section 2.6, an algorithm is provided for each linear algebraic operation needed for the computations that takes its operands as tensor trains and the result is also in this format. Using these algorithms, the whole process has to deal only with tensor trains after the conversions.

Chapter 5

Linear system solvers

The structured MTFF computation formulas presented in Section 4.1 involve computing the inverse of the modified infinitesimal generator matrix of a Markov chain, and then multiplying it from the left with the initial distribution vector and from the right with a vector of ones: MTFF $= -\boldsymbol{\pi}_0\bar{\boldsymbol{Q}}\mathbf{1}$. This is equivalent to solving the linear system $\bar{\boldsymbol{Q}}^T\boldsymbol{x} = \boldsymbol{\pi}_0$ and then computing MTFF $= -\boldsymbol{x}^T\mathbf{1}$. Solving a linear system is much easier than computing the inverse of a matrix and then performing a matrix-vector product with it, so this is the approach that is followed in the proposed method.

By using the tensor train creation methods described in section 4.3, everything is available for this computation in the tensor train format, which makes it possible to store and solve the linear equation system in the structured representation, instead of the intractable explicit solution.

As the system is to be solved in a structured format, direct methods like Gauss elimination or methods using LU decomposition cannot be used. These would need operations that manipulate individual elements, or at least individual rows/columns, which cannot be done efficiently in the decomposed form. The alternative is using iterative linear system solvers [31], which need only those linear algebra operations that are available to tensor trains as described in Section 2.6.

The difficulty of using such solvers with tensor trains is that the TT ranks of the direct result are higher than the TT ranks of the operands. Because of this, the TT rounding algorithm must be applied in each iteration, but keeping in mind that this has consequences: the convergence of the method can be broken if the rounding tolerance is too high, or orthogonality of vectors – which some methods rely on – might be changed.

Several different solvers are available for the tensor train format in the literature. I provide an overview in this chapter of some of them which I have implemented and evaluated their applicability as part of the proposed method through benchmarks.

5.1 Jacobi iteration

Stationary solvers are simple to implement iterative solver methods. They compute the approximation by using an operation that has the true solution as a fix point on the current approximate solution. A commonly used method of this family is the Jacobi iteration. This method updates the ith variable (ith element of $\boldsymbol{x}_k$) by solving the ith equation in the system (the equation defined by the ith row of $\boldsymbol{A}$ and the ith element of $\boldsymbol{b}$), with the other variables fixed at the previous approximation. The update can be written in the elementwise form:

$$\boldsymbol{x}_{k+1}[i] = \frac{1}{\boldsymbol{A}[i,i]}(\boldsymbol{b}[i] - \sum_{j\neq i} \boldsymbol{A}[i,j]\boldsymbol{x}_k[j]) \tag{5.1}$$

This equation can be written in the following matrix form:

$$\boldsymbol{x}_{k+1} = \boldsymbol{D}^{-1}\boldsymbol{b} - \boldsymbol{D}^{-1}\boldsymbol{R}\boldsymbol{x}_k = \boldsymbol{D}^{-1}(\boldsymbol{b} - \boldsymbol{R}\boldsymbol{x}_k) \tag{5.2}$$

where $\boldsymbol{D}$ is the diagonal matrix constructed from the diagonal of $\boldsymbol{A}$, and $\boldsymbol{R} = \boldsymbol{A} - \boldsymbol{D}$. Using this form, we do not need any element-wise operations (not even reading an element), if we can compute the diagonal without them. The only operations needed are:

- computation of the diagonal of $\boldsymbol{A}$
- matrix addition
- vector addition
- matrix-by-vector product

The rate of convergence of this method is determined by the eigenvalues of the matrix $\boldsymbol{D}^{-1}\boldsymbol{R}$ [31].

The TT version of this algorithm can be implemented as follows. First, $\boldsymbol{D}^{-1}$ needs to be computed. $\boldsymbol{D}$ can easily be computed in the TT format from the cores of $\boldsymbol{A}$ by setting the slices corresponding to non-diagonal indices to the zero matrix, or we can get the diagonal as a vector if we delete these slices.

Similarly to [6], the element-wise inverse of this vector can be computed using a version of the Newton-Schulz iteration, originally used to iteratively compute the inverse of a matrix. A version of this method for vectors uses the following formula for the iteration:

$$\tilde{\boldsymbol{d}}_k = \tilde{\boldsymbol{d}}_{k-1} + \tilde{\boldsymbol{d}}_{k-1} \circ (\boldsymbol{1} - \boldsymbol{d} \circ \tilde{\boldsymbol{d}}_{k-1})$$

where $\tilde{\boldsymbol{d}}$ is an approximate element-wise inverse of d.

For the original matrix inversion method, it has been proven in [16], that the iteration converges if for the initial approximation $\tilde{\boldsymbol{A}}_0$, the absolute values of all the eigenvalues of the matrix $\boldsymbol{I} - \tilde{\boldsymbol{A}}_0\boldsymbol{A}$ are less than 1. For a diagonal matrix $\boldsymbol{D}$ with positive values, this condition can be satisfied by setting the initial approximation to $\frac{\boldsymbol{I}}{||\boldsymbol{D}||_F}$. If the diagonal matrix has only negative elements, the same initial approximation can be used with a negative sign.

After convergence of this method, the computed approximation to the vector form of $\boldsymbol{D}^{-1}$ can be extended into a matrix by adding zero matrices to the cores. The computations in the Newton-Schulz iteration and all the other computations in the Jacobi iteration can than be implemented directly using the Tensor Train operations described in section 2.6.

After each iteration (both in the Newton-Schulz inversion and in the Jacobi), the current solution vector needs to be compressed using the TT rounding algorithm, so that the size of the cores does not grow out of control.

5.2 TT-GMRES

5.2.1 GMRES

The *General Minimum Residual* method is a Krylov subspace method, which are special projection methods for solving systems of linear equations [31]. Projection methods for solving a the system $\boldsymbol{Ax} = \boldsymbol{b}$ extract an approximate solution $\boldsymbol{x}_m$ from an affine subspace $\boldsymbol{x}_0 + \mathcal{K}_m$ of dimension m by imposing the Petrov-Galerkin condition

$$\boldsymbol{b} - \boldsymbol{x}_m \perp \mathcal{L}_m$$

where $\mathcal{L}_m$ is another subspace of dimension m. This method is said to be a *projection method onto* $\mathcal{K}_m$ *orthogonal to* $\mathcal{L}_m$.

Krylov subspace methods are projection methods where $\mathcal{K}_m$ is a special subspace of $\mathbb{R}^n$ called a Krylov subspace.

Definition 6. The m dimensional *Krylov subspace* generated by the matrix $\boldsymbol{A} \in \mathbb{R}^{n\times n}$ and the vector $\boldsymbol{v}$ is the subspace

$$\mathcal{K}_m(\boldsymbol{A}, \boldsymbol{v}) = \mathit{span}\{\boldsymbol{v}, \boldsymbol{Av}, \boldsymbol{A}^2\boldsymbol{v}, \dots, \boldsymbol{A}^{m-1}\boldsymbol{v}\}$$

■

When solving a system of linear equations using Krylov subspaces, the Krylov subspaces used are generated by the coefficient matrix $\boldsymbol{A}$ and the initial residual vector $\boldsymbol{r}_0 = \boldsymbol{b} - \boldsymbol{Ax}_0$. In each iteration, we use add one new dimension to the subspace, so in the kth iteration, the Krylov subspace has k dimensions. If the dimension of the subspace is maximal, the iteration stops, as we run into a "lucky breakdown". In this case, the solution at this iteration is already exact, if the system has a solution.

As it can be seen from definition 6, the method calculates an approximation of the error as the product of a polynom $p(\boldsymbol{A})$ of the coefficient matrix $\boldsymbol{A}$ and the initial residual $\boldsymbol{r}_0$. In the case of a non-singular $\boldsymbol{A}$, this can be seen as implicitly approximating $\boldsymbol{A}^{-1}$ by $p(\boldsymbol{A})$.

Different methods differ in how they choose $\mathcal{L}_m$, the space that the residual is orthogonal to. The method is called an *orthogonal* projection method, if $\mathcal{L}_m = \mathcal{K}_m$, and an *oblique* one otherwise.

One common way to choose the projector of an oblique method is setting $\mathcal{L}_m = \boldsymbol{A}\mathcal{K}_m$, which is motivated by the following theorem.

Theorem 3. [31] Given a system of linear equations $\boldsymbol{Ax} = \boldsymbol{b}$ with an arbitrary square matrix $\boldsymbol{A}$, the vector $\hat{\boldsymbol{x}}$ is the result of an oblique projection method onto $\mathcal{K}$ orthogonal to $\boldsymbol{A}\mathcal{K}$ if and only if minimizes the 2-norm of the residual vector $\boldsymbol{b} - \boldsymbol{A}$ over $\boldsymbol{x} \in \boldsymbol{x}_0 + \mathcal{K}$ i.e. if and only if

$$||\boldsymbol{b} - \boldsymbol{A}\hat{\boldsymbol{x}}||_2 = \min_{\boldsymbol{x}\in\boldsymbol{x}_0+\mathcal{K}} ||\boldsymbol{b} - \boldsymbol{Ax}||_2$$

■

Choosing the orthogonal subspace this way for a Krylov subspace method leads to one of the most commonly used techniques called the *Generalized Minimum Residual Method (GMRES).*

The first part of GMRES is the *Arnoldi loop*, which is used to compute an orthonormal basis of $\mathcal{K}_m$. Let $\{\boldsymbol{v}_1, \boldsymbol{v}_2, \dots, \boldsymbol{v}_m\}$ be an orthonormal basis of $\mathcal{K}_m(\boldsymbol{A}, \boldsymbol{r}_0)$. The Arnoldi algorithm starts by setting $\boldsymbol{v}_1 = \frac{\boldsymbol{r}_0}{||\boldsymbol{r}_0||_2}$, and then for $i = 2, 3, \dots m$: $\tilde{\boldsymbol{v}}_i = \frac{\boldsymbol{Av}_{i-1}}{||\boldsymbol{Av}_{i-1}||_2}$, and computes $\boldsymbol{v}_i$ by orthogonalizing $\tilde{\boldsymbol{v}}_i$ against the previous basis vectors. This can be done by using any orthogonalization algorithm, like the (modified) Gram-Schmidt or Householder. Throughout the Arnoldi process, a Hessenberg matrix $\boldsymbol{H}_m$ is generated, which is used in the update formula.

Let $\boldsymbol{V}_m = [\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_m]$, the matrix whose columns consist of the computed orthonormal basis vectors of $\mathcal{K}_m(\boldsymbol{A}, \boldsymbol{r}_0)$. Any vector $\boldsymbol{x} \in \boldsymbol{x}_0 + \mathcal{K}_m(\boldsymbol{A}, \boldsymbol{r}_0)$ can be written as

$$\boldsymbol{x} = \boldsymbol{x}_0 + \boldsymbol{V}_m \boldsymbol{y}$$

for some vector $\boldsymbol{y} \in \mathbb{R}^m$. The residual can be written as:

$$\begin{aligned}\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x} &= \boldsymbol{b} - \boldsymbol{A}(\boldsymbol{x}_0 + \boldsymbol{V}_m \boldsymbol{y}) \\ &= \boldsymbol{r}_0 - \boldsymbol{A}\boldsymbol{V}_m \boldsymbol{y} \\ &= \beta \boldsymbol{v}_1 - \boldsymbol{V}_{m+1} \bar{H}_m \boldsymbol{y} \\ &= \boldsymbol{V}_{m+1}(\beta \boldsymbol{e}_1 - \bar{\boldsymbol{H}}_m \boldsymbol{y})\end{aligned}$$

where $\boldsymbol{e}_i$ is the ith column of the identity matrix with the appropriate size. Since the columns of $\boldsymbol{V}_{m+1}$ are orthonormal,

$$||\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}||_2 = ||\beta \boldsymbol{e}_1 - \bar{\boldsymbol{H}}_m \boldsymbol{y}||_2 \tag{5.3}$$

From theorem 3 we know, that the GMRES approximation $\boldsymbol{x}_m$ is the unique vector in $\boldsymbol{x}_0 + \mathcal{K}_m(\boldsymbol{A}, \boldsymbol{r}_0)$ that minimizes the residual norm. According to equation 5.3, this can be done by calculating

$$\begin{aligned}\boldsymbol{y}_m &= \underset{\boldsymbol{y} \in \mathbb{R}^m}{\arg\min} \, ||\beta \boldsymbol{e}_1 - \bar{\boldsymbol{H}}_m \boldsymbol{y}||_2 \\ \boldsymbol{x}_m &= \boldsymbol{x}_0 + \boldsymbol{V}_m \boldsymbol{y}\end{aligned}$$

Calculating $\boldsymbol{y}$ is inexpensive, as it needs the solution of an $(m + 1) \times m$ least-squares problem, where m is typically small. As $\bar{\boldsymbol{H}}_m$ is an upper-Hessenberg matrix, the least-squares problem can be solved by using Givens rotations to transform it to an upper triangular matrix, and dropping the last row. See [31] for details.

As the computational complexity of the calculations depends on the dimension of the Krylov subspace, it is desirable to keep m small. To achieve accurate results while keeping the subspaces used for calculation small, GMRES is mostly used in a restarted manner: after some iterations, the algorithm is restarted with the initial guess set to the latest approximation.

5.2.2 The TT version

Adapting GMRES for solving linear systems in the TT format has been proposed in [10], where the resulting algorithm has been given the name *TT-GMRES*. As all of the operations needed in the GMRES algorithm are available for matrices and vectors represented by tensor trains, the algorithm is mostly the same as the general GMRES applied to TT-matrices and TT-vectors.

The only modification needed is the introduction of rounding in each iteration, so that the TT ranks do not become intractably large. In the article, the rounding accuracy needed for convergence is specified according to the results of *inexact Krylov theory*, presented in [32].

Although this rounding does not make the method divergent, it can still slow down convergence of the classical GMRES algorithm, and for large Krylov subspaces, the base vectors computed in the Arnoldi process lose the orthogonality, making the fast residual norm computation used in GMRES incorrect. The ranks of the Krylov vectors can also grow large if a high-dimensional Krylov subspace is sought as the rounding might not be enough.

Because of these problems, the algorithm must be used with restarts, with only a small number of inner iterations.

5.2.3 Preconditioning

A problem with iterative methods is their lack of robustness. The convergence of these methods is highly dependent on the spectrum of the coefficient matrix. Therefore, even if we know that the method converges at all (and we do not know even this surely for projection methods for arbitrary coefficient matrices), the convergence might be unacceptably slow.

Preconditioning is a technique that tries to overcome this disadvantage. It means transforming the system in such a way that the new system has the same solution, but is easier to solve with an iterative solver. When preconditioning is applied to a system of linear equations, it is done by finding a non-singular preconditioning matrix $\boldsymbol{M}$, and then transforming the original system using it in a way that preserves the system's solution.

The most basic form of this is *left preconditioning*, when the preconditioned system has the form $\boldsymbol{MAx} = \boldsymbol{Mb}$. In this work only this form of preconditioning is used. $\boldsymbol{M}$ should be constructed in a way such that the spectrum of the coefficient matrix of the preconditioned system of close to that of the identity matrix.

As the proposed method needs the linear system and the solution in the TT format, the preconditioner must also be a structured matrix that can be easily computed in this format. There are some general methods available to compute a potential preconditioner directly through its TT representation.

Jacobi preconditioner The Jacobi (or diagonal) preconditioner simply uses the inverse of the coefficient matrix's diagonal. This can be computed in the TT format as described in Section 5.1. Using this technique can be efficient in the case of diagonally dominant matrices:

Definition 7. A matrix $\boldsymbol{A} \in \mathbb{R}^{n \times n}$ is *diagonally dominant* if for all $i = 1, 2, \dots, n$

$$|\boldsymbol{A}(i,i)| \geq |\sum_{j \neq i} \boldsymbol{A}(i,j)|$$

▪

In the case of calculating the MTTF for a fault tree, this condition always holds for the coefficient matrix: the absolute value of each diagonal element of the infinitesimal generator matrix for the corresponding absorbing Markov chain equals the sum of the other elements in the appropriate row, and we drop the column and row of the absorbing state, thereby removing positive non-diagonal elements from the remaining rows.

The modifications made by the formula of Theorem 2 does not change this, as the absolute value of the elements in the diagonal does not change, and the change in the off-diagonal elements is always a change from a positive value to zero. This still holds if the MTFF is computed through the formula of Theorem 1 when the absolute value of the dummy value written into the diagonal of failure states is large enough.

Newton-Schulz preconditioner Another way we can approximate the inverse of the coefficient matrix is running some iterations of an iterative matrix inversion procedure, and using the result as the preconditioner. One option for this is the Newton-Schulz iteration, already mentioned for the inversion of the diagonal. The formula of this iteration for a matrix $\boldsymbol{A}$ is the following:

$$\boldsymbol{Ainv}_k = \boldsymbol{Ainv}_{k-1} + \boldsymbol{Ainv}_{k-1}(\boldsymbol{I} - \boldsymbol{A}\boldsymbol{Ainv}_{k-1})$$

To satisfy the convergence condition, the initial approximation can be set to $\frac{\boldsymbol{A}^T}{||\boldsymbol{A}^T\boldsymbol{A}||_F}$, as the matrix $\boldsymbol{A}^T\boldsymbol{A}$ is positive semi-definite for any matrix $\boldsymbol{A}$, thus having only positive or zero eigenvalues, whose absolute value can be bounded from above by the Frobenius norm.

Although this method can greatly reduce the iterations needed for the final iterative solution for the linear system, this comes at a very high cost: we need to use matrix-matrix multiplication in the iteration, which increases the size of the cores by a large extent, and this often cannot be overcome by rounding in the Newton-Schulz iteration.

Experiments have shown, that although after applying the Newton-Schulz preconditioner to the system, the resulting preconditioned system can be compressed to a reasonable size, and the solution needs much less iterations, than the one without preconditioning, the creation of the preconditioner through this iteration takes much more time than the amount spared in the final solution.

A faster version for this algorithm has been proposed in [20], which uses a differential geometric integration method to approximate the iteration directly on a fixed-rank TT manifold. This approach might lead to a more useful preconditioning method.

DMRG preconditioner A third approach to TT-structured preconditioner creation is proposed in [25]. The DMRG scheme used for linear system solution (see Section 5.3 for details) can also be used as an iterative matrix inversion approach directly in the TT format. Running this algorithm for some iterations can be used to compute an approximation to the matrix inverse, which can be used as a preconditioner.

5.3 ALS and DMRG

The previously mentioned linear system solver methods were based on general techniques, not exploiting the structure of tensor trains. In contrast to them, the *Alternating Least Squares (ALS)* method for tensor trains directly uses this structure. It was proposed using a few different derivation in different papers [14, 17, 25], my implementation is based on [25].

The method was originally formulated as the solution of an optimization problem: solving the system $\boldsymbol{Ax} = \boldsymbol{b}$ is equivalent to minimizing $\boldsymbol{x}^T\boldsymbol{Ax} - 2\boldsymbol{b}^T\boldsymbol{x}$, if $\boldsymbol{A}$ is symmetric positive definite (SPD). It starts from an initial guess in the tensor train format, and iteratively updates it by optimizing each core individually. One of the cores is chosen for optimization, and the others are fixed.

A single core enters into this optimization linearly, which leads to a local linear system that is much smaller than the global system would be if solved explicitly. The local system can be solved using either simple direct solvers if it is small enough, or some iterative solver otherwise (see e.g. [31] for such algorithms).

The optimization interpretation is valid only for SPD matrices, but the algorithm can be used with non-SPD matrices, when its steps can be interpreted as a projection method instead: the system is projected in each step onto the subspace that can be described by varying only the optimized core, fixing the others. The optimization perspective provides a convergence guarantee for SPD matrices, while – like for most projection methods – the convergence is not guaranteed in non-SPD cases. However, the usefulness of the algorithm has been empirically proven.

The greatest limitation of the ALS algorithm is that the ranks of the tensor train must be chosen in advance, as the ranks cannot change when only a single core is varied. For the matrix multiplications to make sense, the number of rows of the optimized core must be equal to the number of columns of the previous one, and the number of columns of the optimized core must be equal to the number of rows of the next one. As the previous and next cores as fixed, the number of rows and columns (which define the TT rank) are fixed.

However, a good guess for the TT ranks of the solution is usually not available. Underestimating the TT ranks of the solution makes it impossible for the iteration to converge to a solution accurate enough, while overestimation makes the problem ill-conditioned. To overcome this problem, adaptive versions of the algorithm have been proposed [25, 11].

One adaptive version of ALS is the *Density Matrix Renormalization Group (DMRG)* method [25], originating from quantum physics. This algorithm makes changes in the TT ranks possible by optimizing over supercores instead of single cores: a supercore is created by merging to subsequent cores. The output of the local optimization is the supercore, which is then split into two cores by computing its SVD, which can be considered as a diadic decomposition, so the two cores can be assembled from it. Growth of the TT ranks can be controlled by using here only a truncated SVD, dropping components which correspond to small singular values.

5.4 Exponential sums and the Neumann expansion

A TT-based method for mean time to absorption (MTTA) computation was proposed in [29]. In the original paper, the method was proposed for models where the underlying Markov chain has only a single failure state. In my work, I extended this method to multiple failure states, so that it can be used to compute the MTTF for systems modeled using a fault tree. This extended version can be devised using Theorem 2, which will be described in detail along with the extension approach in Chapter 4.

This MTTF computation scheme is based on two iterative matrix inverse approximation procedures: exponential sums (see Section 2.7) and the Neumann expansion. In the experiments, these sums turned out to be very well suited for compression.

The Neumann expansion can be used to approximate the inverse of a matrix of the form $\boldsymbol{I} - \boldsymbol{M}$, where the spectral radius of $\boldsymbol{M}$ is less than 1. If this requirement is satisfied, then $(\boldsymbol{I} - \boldsymbol{M})^{-1} = \sum_{k=0}^{\infty} \boldsymbol{M}^k$. Here, again, the approximation is done through truncating the infinite sum.

The original paper suggested the method for stochastic automata networks, where the infinitesimal generator of the underlying CTMC can be decomposed as

$$\boldsymbol{Q} = \boldsymbol{R} + \boldsymbol{W} + \boldsymbol{\Delta}$$

where $\boldsymbol{R} = \bigoplus_i \boldsymbol{R}_i$ is a Kronecker sum of the local contributions of individual automata, $\boldsymbol{W}$ is the matrix representing the synchronization contributions, and $\boldsymbol{\Delta} = -\mathrm{diag}\{(\boldsymbol{R}+\boldsymbol{W})\cdot\mathbf{1}\}$ is the diagonal part of the infinitesimal generator.

The algorithm was devised for stochastic automata networks where each automata has a single absorbing state, so the whole network also has only a single absorbing state: when every automata is in its own absorbing state. Therefore, the underlying Markov chain has a single absorbing state, and as shown in the article, mean time to absorption can be computed using an additive perturbation $\boldsymbol{S}$ which has TT rank $\mathbf{1}$, if the probability of starting in the absorbing state is 0:

$$\mathrm{MTTA} = -\boldsymbol{\pi}_{\mathbf{0}}(\boldsymbol{Q} - \boldsymbol{S})^{-1}\mathbf{1}$$

Splitting $\boldsymbol{Q}$ into $\boldsymbol{Q}_1 + \boldsymbol{Q}_2$, the inverse can be expressed as:

$$(\boldsymbol{Q}-\boldsymbol{S})^{-1} = (\boldsymbol{Q}_1+\boldsymbol{Q}_2-\boldsymbol{S})^{-1} = (\boldsymbol{I}+\boldsymbol{Q}_1^{-1}(\boldsymbol{Q}_2-\boldsymbol{S}))^{-1}\boldsymbol{Q}_1^{-1} = \sum_{j=0}^{\infty}(-1)^j(\boldsymbol{Q}_1^{-1}(\boldsymbol{Q}_2-\boldsymbol{S}))^j\boldsymbol{Q}_1^{-1}$$

The MTTA can be approximated by using a truncated form of this expression to approximate $-\boldsymbol{\pi}_0(\boldsymbol{Q}-\boldsymbol{S})^{-1}$, and then the approximation can be multiplied by the vector of ones. If $\boldsymbol{Q}_1$ is a Kronecker sum, then its inverse can be efficiently computed using the exponential sum approximation described above. This, and the spectral radius requirement of the Neumann sum can be satisfied by splitting $\boldsymbol{Q}$ as follows:

$$\begin{aligned}\boldsymbol{Q}_1 &= \boldsymbol{R} - \gamma\boldsymbol{I}\\ \boldsymbol{Q}_2 &= \boldsymbol{W} + \boldsymbol{\Delta} + \gamma\boldsymbol{I}\end{aligned}$$

where γ is a constant such that $\gamma \geq ||\boldsymbol{\Delta}||_\infty$. This constant influences the convergence of both the exponential sums and the Neumann series.

Even though no general recipe is provided for determining this constant, for the algorithm present here, it was observed that $\gamma = ||\boldsymbol{\Delta}||_\infty$ provides the best convergence for the overall method both by the authors of the original paper and by me throughout the implementation. However, during the benchmarking process, I observed that this value can lead to numerical instability in case repairs are present in the fault tree.

Although the spectrum of the large matrix is contained in the appropriate interval, the individual components can still possess negative eigenvalues, which makes the exponentiation unstable: each component $\tilde{\boldsymbol{R}}_i = \boldsymbol{R}_i + \gamma/N\boldsymbol{I}$ is involved in computations of the form $e^{-b\cdot\tilde{\boldsymbol{R}}_i}$, where b is a positive scalar, which means that in case b is large and $\tilde{\boldsymbol{R}}_i$ has a negative eigenvalue, the result can contain infinity and NaN values in the numerical representation.

To solve this problem, *gamma* is set to $max(||\boldsymbol{\Delta}||_\infty, -N\cdot\nu_{min})$, where ν_{min} is the smallest (meaning largest in absolute value) negative eigenvalue among the eigenvalues of the Kronecker sum components, if it exists, and 0 otherwise. This way, any of the eigenvalues of the components is ≥ 0, which gets rid of the blow-up in the exponentiation. The same principle is used when $\hat{\boldsymbol{R}}$ is computed for the formula of Theorem 1.

In the article, another version of the algorithm is proposed based on reordering the Neumann terms and repeatedly squaring $\boldsymbol{M}$ in the series, which in this case equals $-\boldsymbol{Q}_1^{-1}(\boldsymbol{Q}_2 - \boldsymbol{S})$. This provides a much better convergence rate, but in the case of multiple absorbing states, the matrix $\boldsymbol{S}$ is replaced by a matrix with potentially relatively large TT ranks (see Chapter 4 for details), which makes repeated squaring intractable.

5.4.1 Adaptation to fault trees

The algorithm can be adapted to static fault trees using Theorem 2 by setting $\boldsymbol{S}$ to the additive modification matrix described in the theorem. As the rate matrix of the original Markov chain consists of only a Kronecker sum, $\boldsymbol{W}$ is just the zero matrix, and $\boldsymbol{R}$ is the whole rate matrix.

Changing $\boldsymbol{S}$ to the multiple absorbing state form introduces a problem, though: in the original version, the TT-ranks of $\boldsymbol{S}$ were always 1. In this version, however, the ranks can be much larger, depending on the Boolean function encoded by the tree. This can lead to a quick rank growth of the Neumann terms. Because of this, the algorithm is no longer guaranteed to be more efficient than other solvers, so the performance of this method must be compared to the others.

Chapter 6

Evaluation

I measured the applicability of the structured MTTF computation method to Fault Trees by running benchmarks with the two different structured computation formulas (theorems 1 and 2), and the different solver methods. The numerical experiments were conducted to answer the following research questions:

- How do the running time of various Tensor Train-based MTFF analysis methods compare on synthetic and real-world fault tree models?
- How does the running time of Tensor-Train-based MTFF analysis compare to state-of-the-art fault tree analysis methods based on explicit state space generation?

6.1 Implementation and benchmark setup

The main reasons for building a new implementation of all the algorithms instead of using those already built is that the available libraries are mostly built in Matlab, which is not suitable for integration with most modeling tools. My implementation runs on the JVM, so that the library implementing the described algorithms can be easily integrated with JVM-based modeling tools, like those built using the Eclipse Modeling Framework. The project is open-source, the source-code is available on GitHub [2]. This can make progress in the area faster, as the implementation can be used for further research.

For BDD creation and manipulation, a decision diagram library developed at the Fault Tolerant Systems Research Group is used.

In the implementation BiCGStab [31] is used as the local iterative solver for the DMRG scheme. In the first implementation, restarted GMRES was used, but it was prone to stagnation too early and so the global scheme could not converge to a solution accurate enough. The stopping criterion of the Neumann-expansion-based computation method is that the size of the latest Neumann term is less then the required convergence threshold.

I implemented an ANTLR-based parser for a subset of the *galileo* [1] fault tree description language that contains all the language elements needed for static fault trees with exponentially distributed failure events. The state-of-the-art probabilistic model checker Storm, which is used as a baseline for the benchmarks, also uses this input format for fault trees.

The galileo language in itself does not support exponentially distributed repairs, and no extension for this was added in Storm either. I added this feature as an extension to

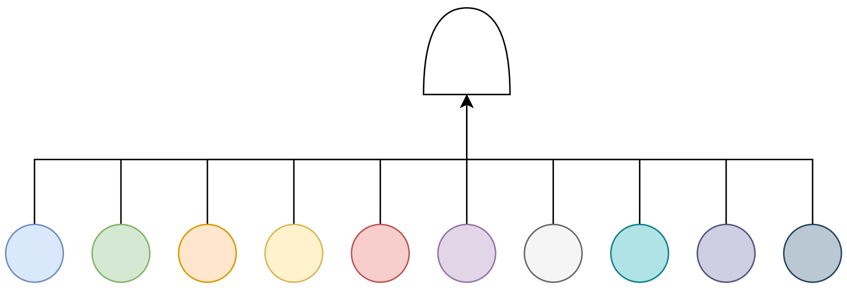


Figure 6.1: AND gate with 10 inputs

the language through a 'repair' property that can be applied to basic events. As this is missing from Storm, unfortunately, the results on non-monotone fault trees could not be compared to it.

6.2 Benchmark model descriptions

The first benchmark is meant to measure the gains of structured representations in the case of fault trees with a simple, highly compressible structure, but for which explicit methods suffer severely from the state-space explosion problem when the number of basic events become large. The best example for this case is a very simple fault tree with a single AND gate as a top event, and a lot of basic events with different failure and repair rates as its inputs. This benchmark is a fault tree with 10 basic events as the inputs of the top AND gate. This model is highly artificial, but it highlights the case where the approach proposed in this work can be the most beneficial. This benchmark is mainly used as a quick comparison between the different solver-preconditioner-formula combinations, to determine which combinations are worth more investigation.

After the AND gate, a group of models that consists of realizations of a scalable model inspired by a SIL4 safety signal transmitter were used as benchmarks. The concept of this model comes from an industrial project. The architecture modeled by the tree is quite general, similar architectures are often used in safety critical systems. The system consists of a central logic module, and at least one IO module attached to it. Each module has three channels, referred to as R, G, B channels, and decisions are made by taking the result of a 2 out of 3 voting of these channels. The voting is performed by a duplicated voter component on each IO module. The information is transmitted between channels through the logic module, so in case the given channel of the logic module fails, that channel is considered failed in every IO module.

The size of the modeled system can be changed by choosing the number of IO modules. Each IO module has 5 corresponding basic events, and the logic module has 3 basic events, so a realization of the fault tree template has $3 + 5k$ basic events, where k is the number of IO modules. Figure 6.2 shows the fault tree template of this benchmark group with two IO modules drawn.

6.3 Experiment setup

The baseline used for the benchmarks is the Storm model checker, which uses state-of-the-art explicit methods for probabilistic model checking. The main difference between the two approaches is that the explicit methods mostly rely on generating the reachable state

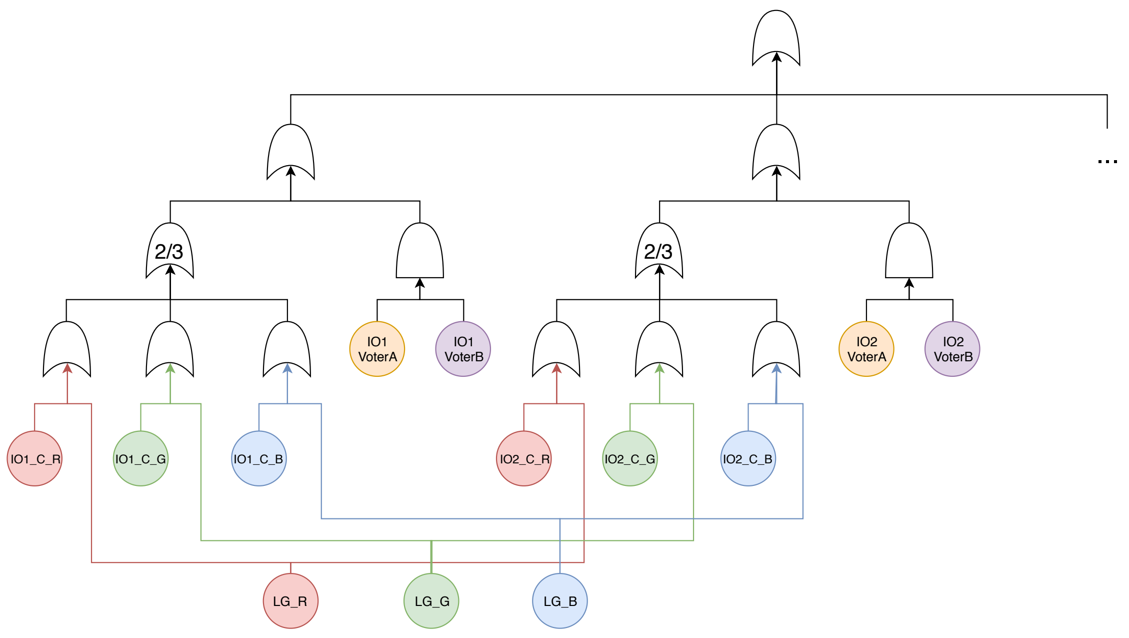


Figure 6.2: Fault tree of the simplified SIL4 Safety Signal Transmitter

space instead of the potential state space. This is often much smaller than the potential state space, but can still be exponentially large in the number of state variables (basic events, in the case of fault trees). The structured approach tries to exploit the structure of the state space instead, which can often lead to better asymptotic behaviour.

The Newton-Schulz preconditioning method with the naive implementation did not make into the benchmark, as the creation of the preconditioner took too long even with a small number of iterations, and the TT-rank growth of the system was too large.

Machine precision was considered 10^{-16} in the benchmarks. Iterative solvers were required to converge to a residual norm $\leq 10^{-7}$. The convergence in TT-GMRES is checked using the faster residual norm computation available in GMRES, which provides the residual norm before the last rounding. Without the rounding, the TT-ranks of the result are too large for further computations, so the real norm cannot be used to efficiently detect the convergence. Therefore, the residual norm after TT-GMRES stops might be a bit above the set threshold because of the last rounding. The benchmarks were run with the OpenJDK version 11.0.4 with 32GB heap size on an AMD EPYC 7551 processor. A timeout of 20 minutes was set for each measurement.

6.4 Results

Meaning of the columns in the benchmark result tables:

- Formula: Thm. 1, if the formula of Theorem 1 is used, and Thm. 2, if the formula of Theorem 2. If 1, then the computation is performed with the formula described in Section 4.1 with $\bar{\boldsymbol{R}}$ lifted out.
- Solver: which of the solvers described in Chapter 5 was used.
- Preconditioner: which structured preconditioner was used, if any. If the solver is DMRG, there are two levels where preconditioning could be performed: at the

global level, by multiplying the large system before DMRG is started, and at the level of the smaller local systems. This preconditioner reported in this column is at the global level, no local preconditioning was implemented for DMRG for the benchmarks here.

- Resnorm: 2-norm of the residual after the iterative solver stopped.
- MTFF: the resulting approximate MTFF value.
- r_{TT}: largest TT-rank of the iteration vector after rounding, throughout the whole iteration process.
- Time: the total time taken by the overall computation, including parsing the fault tree file and creating the linear system. 't.o.' in this field means that the computation was stopped either by a timeout of an out of memory exception.

The results of measurements with a single AND gate with 10 basic events can be seen in Tables 6.1 and 6.2. The Storm baseline is shown only for the version without repairs, as Storm does not support exponentially distributed repairs in fault trees. This benchmark was measured with all formula-solver-preconditioner combinations, but a lot of them timed out even on this size, or stagnated with a very large residual leading to a totally wrong MTFF value, so further measurements were not performed on these.

Formula	Solver	Preconditioner	Resnorm	MTFF	r_{TT}	Time (ms)
	Storm			0.950257		9
Thm. 1	DMRG	DMRG	-	-	-	t.o.
Thm. 1	DMRG	Jacobi	-	-	-	t.o.
Thm. 1	DMRG	-	7.159e-07	0.9502569	12	45697
Thm. 1	GMRES	DMRG	-	-	-	t.o.
Thm. 1	GMRES	Jacobi	-	-	-	t.o.
Thm. 1	GMRES	-	2.446e-07	0.9502569	32	24058
Thm. 1	Jacobi	DMRG	-	-	-	t.o.
Thm. 1	Jacobi	Jacobi	-	-	-	t.o.
Thm. 1	Jacobi	-	-	-	-	t.o.
Thm. 2	DMRG	DMRG	6.174e-09	0.9502570	16	23561
Thm. 2	DMRG	Jacobi	1.981e-08	0.9502569	12	1677
Thm. 2	DMRG	-	7.688e-08	0.9502570	10	1062
Thm. 2	GMRES	DMRG	6.010e-08	~~-0.9999992~~	16	99672
Thm. 2	GMRES	Jacobi	3.508	~~-1992~~	25	74446
Thm. 2	GMRES	-	3.988e-07	0.9502569	32	12324
Thm. 2	Jacobi	DMRG	7.831e-09	0.9502570	16	25871
Thm. 2	Jacobi	Jacobi	-	-	-	t.o.
Thm. 2	Jacobi	-	4.673e-08	0.9502569	18	8351
Thm. 2	Neumann	-	-	0.9509691	16	3170

Table 6.1: Results on the fault tree with a single AND gate with 10 inputs without repairs

For the measurements with the simplified SIL4 signal transmitter, only non-preconditioned DMRG, Jacobi preconditioned DMRG, non-preconditioned GMRES and Neumann iteration were used. These solvers were run on the SIL4 signal transmitter with 1 to 4 IO modules. The results are shown in Table 6.3.

Formula	Solver	Preconditioner	Resnorm	MTFF	r_{TT}	Time (ms)
Thm. 1	DMRG	DMRG	-	-	-	timeout
Thm. 1	DMRG	jacobi	-	-	-	timeout
Thm. 1	DMRG	none	2.593e-4	587.7872191	10	86875
Thm. 1	GMRES	DMRG	-	-	-	timeout
Thm. 1	GMRES	jacobi	-	-	-	timeout
Thm. 1	GMRES	none	5.117e-2	~~20.9718066~~	28	84699
Thm. 1	jacobi	DMRG	-	-	-	timeout
Thm. 1	jacobi	jacobi	-	-	-	timeout
Thm. 1	jacobi	none	-	-	-	timeout
Thm. 2	DMRG	DMRG	-	-	-	t.o.
Thm. 2	DMRG	Jacobi	7.707e-08	587.8045498	12	7489
Thm. 2	DMRG	-	1.632e-05	587.8046084	11	24814
Thm. 2	GMRES	DMRG	-	-	-	t.o.
Thm. 2	GMRES	Jacobi	0.034	~~-1375~~	32	46903
Thm. 2	GMRES	-	0.147	~~-1020~~	31	41403
Thm. 2	Jacobi	DMRG	-	-	-	t.o.
Thm. 2	Jacobi	Jacobi	-	-	-	t.o.
Thm. 2	Jacobi	-	-	-	-	t.o.
Thm. 2	Neumann	-	-	~~135.3470893~~	5	1193025

Table 6.2: Results on the fault tree with a single AND gate with 10 inputs with repairs

In these measurements, the non-preconditioned DMRG seemed to be the overall best, so this was also run on the benchmark with 5 to 8 IO modules. For these benchmarks, the performance of Storm with symmetry reduction turned on was also measured. The results are shown in Table 6.4.

6.5 Discussion of the results

During the measurements with the 10-input AND gate, restarted TT-GMRES displayed a very poor behaviour with the Jacobi preconditioner as it stagnated at a very large residual, which is strange compared to the literature on non-TT restarted GMRES, so there might be an error in the implementation.

Preconditioning through some DMRG inversion iterations was very unreliable, as no global convergence checks were performed on it. DMRG is not guaranteed to converge to the global solution, can get stuck at local optima, which – as observed in this measurement – can lead to even a wrong solution if it is used as a preconditioning method. Heuristic convergence checking can be implemented for the DMRG scheme (see Chapter 7), but even this might not work well for preconditioning as the preconditioner creation should run only for a few sweeps, when it might not be possible to detect if it moves towards the correct solution.

The performance of DMRG was not enhanced by the preconditioners in the monotone case, but it was significantly better with the Jacobi preconditioner in the repaired case. Because of these, preconditioning was only used with DMRG in further measurements, and only the Jacobi preconditioner. When repairs were present in the system, only the Neumann expansion and the DMRG scheme arrived at a correct solution. As the Jacobi solver was too slow, it was omitted in further benchmarks.

Formula	Solver	Preconditioner	Resnorm	MTFF	r_{TT}	Time (ms)
1 IO module (7 basic events)						
	Storm			0.0683632		1357
Thm. 1	DMRG	-	8.849e-09	0.0683632	8	1446
Thm. 1	DMRG	jacobi	-	-	-	t.o.
Thm. 1	GMRES	-	8.963e-08	0.0683632	11	2518
Thm. 1	neumann	-	8.963e-08	0.0683151	160	2619
Thm. 2	DMRG	-	1.793e-08	0.0683632	8	925
Thm. 2	DMRG	jacobi	8.219e-10	0.0683632	8	1698
Thm. 2	GMRES	-	0.135	~~0.3977323~~	10	7390
Thm. 2	neumann	-	-	0.0683151	160	2328
2 IO modules (12 basic events)						
	Storm			0.0479165		1180
Thm. 1	DMRG	-	0.611	~~0.1052902~~	24	759775
Thm. 1	DMRG	jacobi	-	-	-	t.o.
Thm. 1	GMRES	-	1.504e-07	0.0479165	60	140093
Thm. 1	neumann	-	1.504e-07	0.0479536	1032	34000
Thm. 2	DMRG	-	5.9618e-08	0.0479165	32	14268
Thm. 2	DMRG	jacobi	5.792e-10	0.0479165	38	121977
Thm. 2	GMRES	-	1.341e-06	0.0479123	64	495999
Thm. 2	neumann	-	1.341e-06	0.0479536	1032	31741
3 IO modules (17 basic events)						
	Storm			0.0388832		613
Thm. 1	DMRG	-	-	-	-	t.o.
Thm. 1	DMRG	jacobi	-	-	-	t.o.
Thm. 1	GMRES	-	-	-	-	t.o.
Thm. 1	neumann	-	-	-	-	t.o.
Thm. 2	DMRG	-	6.633-08	0.0388814	52	81066
Thm. 2	DMRG	jacobi	-	-	-	t.o.
Thm. 2	GMRES	-	-	-	-	t.o.
Thm. 2	neumann	-	-	-	-	t.o.
4 IO modules (22 basic events)						
	Storm			0.0336233		521
Thm. 1	DMRG	-	-	-	-	t.o.
Thm. 1	DMRG	jacobi	-	-	-	t.o.
Thm. 1	GMRES	-	-	-	-	t.o.
Thm. 1	neumann	-	-	-	-	t.o.
Thm. 2	DMRG	-	2.808-09	0.0336232	58	387948
Thm. 2	DMRG	jacobi	-	-	-	t.o.
Thm. 2	GMRES	-	-	-	-	t.o.
Thm. 2	neumann	-	-	-	-	t.o.

Table 6.3: Results of the SIL4 safety signal transmitter benchmark with 1-4 IO modules

Solver	Resnorm	MTFF	r_{TT}	Time (ms)
5 IO modules (27 basic events)				
Storm		0.0300945		8113
Storm+symred		0.0300945		1111
DMRG	4.171e-8	0.0300945	56	303863
6 IO modules (32 basic events)				
Storm		0.0275177		122141
Storm+symred		0.0275177		10923
DMRG	3.796e-8	0.0275176	51	178066
7 IO modules (37 basic events)				
Storm		-		t.o.
Storm+symred		0.025528		108305
DMRG	3.506e-8	0.0255278	46	166252
8 IO modules (42 basic events)				
Storm		-		t.o.
Storm+symred		0.02393		1015248
DMRG	5.885e-8	0.0239297	45	160701

Table 6.4: Results of the SIL4 safety signal transmitter benchmark with 5-8 IO modules. Only the non-preconditioned DMRG method with the formula of Theorem 2 was used for these models.

This gives a first answer to the question of comparing different solvers on a synthetic example: the most prominent ones are the non-preconditioned and the Jacobi-preconditioned DMRG, the Neuman expansion-based computation, and the non-preconditioned TT-GMRES.

The combinations that seemed competent based on the results of the 10-input AND gate were run with the more realistic SIL4 safety signal transmitter benchmark. Among these, only the non-preconditioned DMRG solver with the formula from Theorem 2 was able to deal with the larger models. For this reason, measurements continued only with this single setting on realizations with more modules.

This refines the comparison of the solvers, by providing the result for a real-world fault tree: the most useful one according to these measurements is the non-preconditioned DMRG.

With 6 modules, it was close to the performance of Storm without symmetry reduction. With 7 modules, Storm was terminated by 25min timeout when used without symmetry reduction, and even with symmetry reduction, it did not outperform the proposed method significantly. With 8 modules, the proposed method finally came out as the winner: Storm without symmetry reduction was terminated by a 25min timeout, symmetry reduction reduced the running time to 16min, while the proposed method with the DMRG solver and formula 2 came to a result in less than 3 minutes. These cases needed 3 DMRG sweeps, and converged to a residual norm $\leq 10^{-7}$.

After this success, another measurement was performed to see if better convergence was also achievable in the same time-span, but unfortunately, the 4th sweep timed out for the 8-module tree. Results of the measurements can be seen in Table 6.3.

So to answer the second research question, which was about comparing the proposed method to state-of-the-art fault tree analysis methods based on explicit state space generation it can be said that:

- On smaller trees, the methods based on explicit state space generation outperform the proposed method
- But the structured computation has much better scalability, and given the appropriate structure, it can be significantly faster on large trees than explicit state space generation techniques.

Overall, a lot of the measurements have shown a very poor performance, but for large models the DMRG solver without a global preconditioner using the formula from Theorem 2 outperformed the state-of-the-art probabilistic model checker Storm by a large margin if convergence to a residual norm $\leq 10^{-7}$ is enough. This shows that this setting is superior to the current implementation of the others, but most of the methods can still be enhanced (see Chapter 7).

The DMRG method was often unreliable, though, as it failed to move towards the global optimum. Some methods are proposed in the literature to detect such cases and restart the DMRG iteration from another random guess [22, 25]. The implementation is still in progress, the benchmark has shown that implementing this check is a top priority feature.

From the two proposed formulas used for structured MTFF computation, the one from Theorem 2 seems to be more useful from the results. However, the disadvantages of the other formula can stem from the fact that only the form which introduced an additional matrix inversion was measured. Therefore, further measurements are still needed, focused on this aspect.

The MTFF computed by those methods that converged to the required threshold for the given model, and by Storm where it was available were sufficiently cause to further confirm the correctness of the proposed method.

Chapter 7

Conclusion and future work

This report proposed a method for overcoming the problem of state space explosion when computing reliability metrics for fault trees. As a proof of concept, the method was developed for the computation of mean time to first failure from a static fault tree. After the explanation of the proposed structured computation method and the description of the corresponding publicly available implementation, the results of some benchmarks performed to evaluate the method were presented.

From analysing these benchmarks, it can be seen that most of the currently implemented solvers preformed very poorly, and the preconditioners did not enhance the performance of the solvers. One solver method, however, the DMRG solver performed quite well, especially on larger models. On larger models, the state-of-the-art model checker used as a baseline performed significantly worse than the proposed method. A lot of measurements still need to be conducted in order to really evaluate the capabilities of the structured representation for MTFF calculation, but it could be seen even from the results presented in this report that there are cases where methods used until now did not make it possible to correctly evaluate large systems, and the proposed method can change this.

Even if the method is not generally enough for all kinds of complex systems, it can lead us closer to the solution of the problem. If the system has an appropriate structure, and some approximation threshold can be accepted, the proposed method can perform very well. The presented approach outperformed a state-of-the-art stochastic model checkers on a large case study based on an industrial project, indicating its real-world applicability. The approximation capability also does not seem to be limiting, as the results of the method agreed with the results computed by the baseline to the accuracy the baseline returned.

Contributions of the report can be summarized as follows:

- Mathematically proven formulas were presented which can be used to compute the mean time to absorption in Markov chains with multiple absorbing states using representations which do not make dropping individual columns and rows possible. Structured tensor representations are such representations.
- A method for using BDD-based and TT-based techniques together was given in Chapter 4, which can lead to further research into how the advantages of these approaches can be combined.
- Benchmarks were performed with various solvers and results were reported in Chapter 6, which shows the applicability of different TT-based linear system solvers to static fault trees.

- A JVM-based publicly available implementation was made, which makes it easier to use the tensor train format and the implemented algorithms in JVM-based applications.

7.1 Future work

The performance of the DMRG method can be enhanced in a lot of different ways. Other TT-based solvers should be tested, or the currently implemented ones improved, e.g. trying out other iterative solvers for DMRG (like [33]), or finding a good local preconditioner. Memory and computation requirements of the method depend on the size of the created BDD, so reducing its size could also speed up the method. Techniques like exploiting "don't care" values in the encoded function [15], or using a conservative approximation of the model [27] and then refining this until a given target value is met are some possible directions for this.

Apart from enhancing the method's performance, the algorithm could also be extend to more general cases. It could be adapted for the analysis of dynamic fault trees, or (generalised) stochastic Petri nets. An even more general modeling formalism would be general stochastic transition systems, described for example through the language of PRISM [3]. Some of these models contain proper non-determinism, which leads to the need for the TT-based analysis of Markov Decision Processes before the proposed method can be applied to them. Some work has been done in this area for example in [12].

Besides introducing other analyzable modeling formalisms, the method could be adapted also to the computation of other metrics. Mean time to failure or mean time between failures can be computed similarly to MTFF, after computing the steady-state distribution of the model. Although the reliability after a fixed time period of a static fault tree is easy to compute, this is much harder for more general models. It needs the transient analysis of the underlying Markov chain, which might also be performed with the help of tensor trains, for example using the numerical integration algorithm developed for tensor trains in [20].

Acknowledgements

Supported by the ÚNKP-19-1 new national excellence program of the ministry for innovation and technology.

I want to say an e^{huge} thank you to my advisor, Kristóf Marussy, for all the help he provided. I am also grateful to everyone in the Fault Tolerant Systems Research Group who aided my work in any way. I would like to thank Katalin Friedl and László Kabódi for their insights on the topic. I want to express my gratitude to Péter Lantos for providing me the opportunity of an intership at Prolan Co., where I could see the practical side of reliability analysis through a real-world project.

I also want to thank God for giving me the opportunity for this research, leading me to the people I had to meet to make this happen, and making me capable of this work.

Bibliography

[1] Galileo dft analysis tool. URL `https://www.cse.msu.edu/~cse870/Materials/FaultTolerant/manual-galileo.htm`.

[2] Implementation of the proposed algorithm. URL `github.com/szdan97/tensortrain`.

[3] Prism model checker. URL `https://www.prismmodelchecker.org/`.

[4] Suprasad V Amari and Jennifer B Akers. Reliability analysis of large fault trees using the vesely failure rate. In *Annual Symposium Reliability and Maintainability, 2004-RAMS*, pages 391–396. IEEE, 2004.

[5] Matthias Bolten, Karsten Kahl, and Sonja Sokolovic. Multigrid methods for tensor structured markov chains with low rank approximation. *SIAM Journal on Scientific Computing*, 38(2):A649–A667, 2016.

[6] Peter Buchholz, Tugrul Dayar, Jan Kriege, and M Can Orhan. On compact solution vectors in kronecker-based markovian analysis. *Performance Evaluation*, 115:132–149, 2017.

[7] Allan Clark, Stephen Gilmore, Jane Hillston, and Mirco Tribastone. Stochastic process algebras. In *International School on Formal Methods for the Design of Computer, Communication and Software Systems*, pages 132–179. Springer, 2007.

[8] Tugrul Dayar. Analyzing markov chains based on kronecker products. *MAM*, pages 279–300, 2006.

[9] Christian Dehnert, Sebastian Junges, Joost-Pieter Katoen, and Matthias Volk. A storm is coming: A modern probabilistic model checker. In *International Conference on Computer Aided Verification*, pages 592–600. Springer, 2017.

[10] Sergey V Dolgov. Tt-gmres: solution to a linear system in the structured tensor format. *Russian Journal of Numerical Analysis and Mathematical Modelling*, 28(2): 149–172, 2013.

[11] Sergey V Dolgov and Dmitry V Savostyanov. Alternating minimal energy methods for linear systems in higher dimensions. *SIAM Journal on Scientific Computing*, 36 (5):A2248–A2271, 2014.

[12] Alex A Gorodetsky, Sertac Karaman, and Youssef M Marzouk. Efficient high-dimensional stochastic optimal motion control using tensor-train decomposition. In *Robotics: Science and Systems*, 2015.

[13] Lars Grasedyck. Hierarchical singular value decomposition of tensors. *SIAM Journal on Matrix Analysis and Applications*, 31(4):2029–2054, 2010.

[14] Sebastian Holtz, THORSTEN Rohwedder, and Reinhold Schneider. The alternating linear scheme for tensor optimisation in the tt format. *Preprint*, 71, 2011.

[15] Youpyo Hong, Peter A Beerel, Jerry R Burch, and Kenneth L McMillan. Sibling-substitution-based bdd minimization using don't cares. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(1):44–55, 2000.

[16] Harold Hotelling. Some new methods in matrix calculation. *The Annals of Mathematical Statistics*, 14(1):1–34, 1943.

[17] Boris N Khoromskij and Ivan V Oseledets. Dmrg+ qtt approach to computation of the ground state for the molecular schrödinger operator. 2010.

[18] Tamara G Kolda and Brett W Bader. Tensor decompositions and applications. *SIAM review*, 51(3):455–500, 2009.

[19] Daniel Kressner and Francisco Macedo. Low-rank tensor methods for communicating markov processes. In *International Conference on Quantitative Evaluation of Systems*, pages 25–40. Springer, 2014.

[20] Christian Lubich, Ivan V Oseledets, and Bart Vandereycken. Time integration of tensor trains. *SIAM Journal on Numerical Analysis*, 53(2):917–941, 2015.

[21] Kristóf Marussy, Attila Klenik, Vince Molnár, András Vörös, István Majzik, and Miklós Telek. Efficient decomposition algorithm for stationary analysis of complex stochastic petri net models. In *International Conference on Application and Theory of Petri Nets and Concurrency*, pages 281–300. Springer, 2016.

[22] Ivan Oseledets. Dmrg approach to fast linear algebra in the tt-format. *Computational Methods in Applied Mathematics Comput. Methods Appl. Math.*, 11(3):382–393, 2011.

[23] Ivan Oseledets and Eugene Tyrtyshnikov. Tt-cross approximation for multidimensional arrays. *Linear Algebra and its Applications*, 432(1):70–88, 2010.

[24] Ivan V Oseledets. Tensor-train decomposition. *SIAM Journal on Scientific Computing*, 33(5):2295–2317, 2011.

[25] Ivan V Oseledets and SV Dolgov. Solution of linear systems and matrix inversion in the tt-format. *SIAM Journal on Scientific Computing*, 34(5):A2718–A2739, 2012.

[26] Antoine Rauzy. New algorithms for fault trees analysis. *Reliability Engineering & System Safety*, 40(3):203–211, 1993.

[27] Kavita Ravi, Kenneth L McMillan, Thomas R Shiple, and Fabio Somenzi. Approximation and decomposition of binary decision diagrams. In *Proceedings of the 35th annual Design Automation Conference*, pages 445–450. ACM, 1998.

[28] Karen A Reay. *Efficient fault tree analysis using binary decision diagrams.* PhD thesis, © Karen Ann Reay, 2002.

[29] Leonardo Robol and Giulio Masetti. Tensor methods for the computation of mtta in large systems of loosely interconnected components. *arXiv preprint arXiv:1907.02449*, 2019.

[30] Enno Ruijters and Mariëlle Stoelinga. Fault tree analysis: A survey of the state-of-the-art in modeling, analysis and tools. *Computer science review*, 15:29–62, 2015.

[31] Yousef Saad. *Iterative methods for sparse linear systems*, volume 82. siam, 2003.

[32] Valeria Simoncini and Daniel B Szyld. Theory of inexact krylov subspace methods and applications to scientific computing. *SIAM Journal on Scientific Computing*, 25 (2):454–477, 2003.

[33] Gerard LG Sleijpen and Martin B Van Gijzen. Exploiting bicgstab(l) strategies to induce dimension reduction. *SIAM journal on scientific computing*, 32(5):2687–2709, 2010.

[34] William J Stewart. *Probability, Markov chains, queues, and simulation: the mathematical basis of performance modeling*. Princeton university press, 2009.

[35] Kishor S Trivedi and Andrea Bobbio. *Reliability and availability engineering: modeling, analysis, and applications*. Cambridge University Press, 2017.

[36] William E Vesely, Francine F Goldberg, Norman H Roberts, and David F Haasl. Fault tree handbook. Technical report, Nuclear Regulatory Commission Washington DC, 1981.

[37] Matthias Volk, Sebastian Junges, and Joost-Pieter Katoen. Advancing dynamic fault tree analysis-get succinct state spaces fast and synthesise failure rates. In *International Conference on Computer Safety, Reliability, and Security*, pages 253–265. Springer, 2016.