Solving BFL Queries: From BDDs to Quantified SAT

CAZ SAALTINK, University of Twente, the Netherlands

Fault trees are used to analyse the propagation of faults in a system. They also allow for fault tree analysis, which can be used to analyse the key factors that impact system failure. Recently, Nicoletti et al. introduced a new logic called BFL to reason about fault trees. BFL can be used to formally define properties of fault trees. The paper also provides algorithms for BFL, for example, to check whether properties written in BFL hold. All algorithms make use of BDDs, which can use exponential space and time in the number of basic events of the tree. To combat this problem, this work will aim to use quantified Boolean formulas (QBF) instead of BDDs. The algorithms provided by Nicoletti et al. will be replaced with an algorithm that translates BFL to QBF and then uses a QBF solver. This approach is implemented—also marking the first-ever BFL implementation—and it is demonstrated in a case study. Furthermore, qualitative differences from a BDD-based approach are discussed.

CCS Concepts: • **Theory of computation → Logic and verification**.

Additional Key Words and Phrases: fault trees, QSAT, quantified Boolean formulas, BDD

1 INTRODUCTION

Fault trees are often used in safety-critical systems, such as nuclear power plants, aviation, and autonomous vehicles [17]. In these systems, fault tree analysis (FTA) is utilised to analyse the causes of system failures.

A fault tree (FT) is a model that shows how a system can fail by investigating the possible causes of system failure [20]. Fault trees consist of two elements: *events*, which represent things that can fail in a system, and *gates*, which model the relation between the events. An FT is a rooted directed acyclic graph and the root is called the top-level event (TLE). From the top-level event, the tree recursively specifies the causes of events using gates. The two most common gates are the AND and OR gate, which behave exactly like the Boolean operators: an AND gate means that the parent event occurs iff all its child events occur, and an OR gate means that the parent event occurs iff at least one of the child events occur [17]. Furthermore, an FT can contain two types of events: intermediate events and basic events. Intermediate events are events that have a gate with children, i.e., for those events it is specified what causes those events. Basic events are events that are not further specified and thus form the leaves of the tree.

FTA can be used to identify critical basic events, for example by finding minimal cut sets (MCSs) or minimal path sets (MPSs). A minimal cut set is a minimal set of basic events that cause system failure, i.e., they cause the TLE to occur. On the other hand, a minimal path set is a minimal set of basic events that prevent system failure, i.e., the TLE can not occur when these basic events do not occur. Moreover, FTA can be used to analyse dependencies between events, as well as probabilities of events.

Another aspect of fault tree analysis is reasoning about system failures and system states. For example, an FT can be used to prove that some basic event is always a cause of some other event. In [12], Nicoletti et al. define a logic called Boolean Fault tree Logic (BFL) to formalise such properties of fault trees. BFL allows writing first-order logic statements about fault trees. The atoms in this logic are the events of the fault tree. In addition to operators for negation, conjunction, disjunction, implication, and equivalence, BFL also provides operators for MCSs, MPSs, independence of events, and setting evidence, i.e., only including scenarios where given events have a set value.

The paper on BFL [12] also provides algorithms to (1) check if a BFL formula holds for a given FT and status vector (a vector which indicates whether each basic event failed or not); (2) find all status vectors that satisfy a BFL formula for a given FT; and (3) generate counterexamples if a BFL formula does not hold. If a BFL formula does not hold for some FT and status vector, a counterexample is provided, which is a slightly modified status vector for which the BFL formula does hold. It is important that the original status vector and the counterexample differ as little as possible to make it easy to detect which basic event(s) cause(s) the different outcome.

All algorithms described in [12] rely on binary decision diagrams (BDDs)[1]. A BDD is a graphical representation of a Boolean function, where vertices represent the Boolean variables, and each vertex has a *low* and *high* edge that points to the right subtree based on the value of the variable. Ultimately, after finishing a path through all variables in the formula, you reach the 1 or 0 node, which means, respectively, that the formula was or was not satisfied. While a BDD has the potential to provide a very compact representation of a Boolean function, in the worst case BDDs can grow exponentially in the number of variables.

In [12], BFL formulas are first transformed into BDDs, which are then used in each of the algorithms. Because constructing the BDD *could* take exponential space and time, some problems are infeasible to solve using BDDs. In some of these cases, SAT solvers can find solutions where BDDs could not. Research has shown that BDD- and SAT-based approaches complement each other [1, 2, 10, 11, 21]. This research aims to develop and implement a SAT-based approach to provide a complement to the developed (though not yet implemented) BDD-based approach in [12]. This will be achieved by translating BFL formulas to quantified Boolean formulas (QBF) and then using a QBF solver instead of the algorithms provided in [12], eliminating the need to construct a BDD. Biere et al. [3] have already successfully replaced BDDs with SAT solvers to solve problems that were infeasible to solve with a BDD-based approach.

The approach is described in detail in Section 4 of this paper, and it is implemented in Python. The source code of the implementation can be found at [18]. This is the first implementation of BFL so

TScIT 39, July 7, 2023, Enschede, The Netherlands

© 2023 University of Twente, Faculty of Electrical Engineering, Mathematics and Computer Science.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

[1] Strictly speaking, "BDD" is used as a synonym for reduced ordered BDD in this paper.

this work also provides the opportunity to use BFL in practice. We present a case study in which we test the implementation and it simultaneously serves as an example of how the implementation works. Unfortunately, no BDD-based implementation exists yet, so the QSAT-based approach cannot be experimentally compared to the BDD approach. Despite that, differences between the two approaches are highlighted and discussed in Section 6. The section also emphasizes aspects that should be taken into account if the two approaches will be compared in the future.

This paper will be structured as follows: we start by providing background information about fault trees and BFL in Section 2. In Section 3, we investigate related work and different QBF solvers. The implementation is explained in Section 4. Then, we present a case study in Section 5. In Section 6, we discuss differences from a BDD-based approach and cover considerations for future comparisons to a BDD-based approach. Finally, we conclude in Section 7.

2 BACKGROUND

2.1 Fault Trees

A fault tree (FT) is a directed acyclic graph containing events and gates. The top-level event (TLE) of an FT represents system failure. FTs have two different types of events: intermediate events and basic events. Basic events are low-level faults which are not further refined, i.e., they only exist at the bottom of an FT. In contrast, intermediate events are refined using gates and child events. Gates describe which child events cause the parent event to occur. For example, an AND gate means all child events must occur for the parent event to occur, whereas an OR gate means only one event must occur. A third, often used gate is the voting gate (also called: VOT, VT, k/n, or k-out-of-n gate), which means at least k child events must occur (out of n total child events) to cause the parent event. These three gates are the most common, and the only ones currently supported in BFL. In this paper, the voting gate is extended to use any comparison $\bowtie \in \{<, \leq, =, \geq, >\}$. The original k/n gate would be equal to a $\texttt{VOT}(\geq, k)$ gate with this extension. With the extended voting gate, we can also specify that at most ($\texttt{VOT}(\leq, k)$) or exactly ($\texttt{VOT}(=, k)$) k events must occur in order for the fault to propagate. Many more gate types exist [17], but they are not supported in this paper.

Figure 1 shows a small example tree. In this tree, *NL* is the TLE, *NL* and *LB* are intermediate events, and *L1*, *L2* and *PO* are basic events. The intermediate event *LB* has an AND gate, which means *LB* occurs iff *L1* and *L2* both occur. The TLE has an OR gate, which means it occurs if *LB* or *PO* occurs (or both).

In fault tree analysis, often used concepts are minimal cut sets (MCSs) and minimal path sets (MPSs). A cut set is a set of basic events that, if they occur, cause the TLE to occur. If no events can be removed from a cut set without failing to cause the TLE, it is called an MCS. For example, if we take the tree in Figure 1, $\{L1, L2, PO\}$ is a cut set, though it is not minimal. The MCSs of the tree are $\{L1, L2\}$ and $\{PO\}$. On the other hand, a path set is a set of basic events that prevent the TLE, i.e., if the events in the path set do not occur, the TLE also cannot occur. Similarly, an MPS is a path set where no events can be removed without losing the property of being a path set. In the tree in Figure 1, the path sets are $\{L1, L2, PO\}$, $\{L1, PO\}$, and $\{L2, PO\}$, where the last two are minimal.

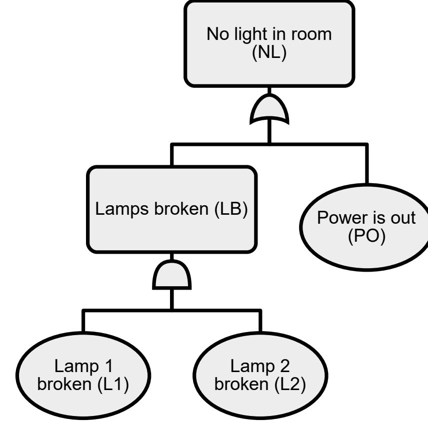


Fig. 1. Small example fault tree with one OR gate and one AND gate.

A status vector is a vector of the statuses of basic events. The vector contains ones and zeroes; a 1 means that the basic event does occur, and a 0 means that it does not. Most often, a status vector is represented as a vector, alongside the corresponding basic events in the same order, for example: $\overline{b} = \langle 1, 0, 1 \rangle$ for basic events a, b, c. Alternatively, a status vector can be treated as a set, in which case it contains all basic events that have a status of 1.

Concretely, a fault tree consists of the following components (adapted from [12]):

- $\texttt{BE}$ is the set of basic events.
- $\texttt{IE}$ is the set of intermediate events.
- $\texttt{E} = \texttt{BE} \cup \texttt{IE}$ is the set of all events.
- $t\colon \texttt{IE} \rightarrow \{\texttt{OR}, \texttt{AND}, \texttt{VOT}(\bowtie, k)\}$ is a function that maps an intermediate event to the type of its gate.
- $ch\colon \texttt{IE} \rightarrow \mathcal{P}(\texttt{E}) \backslash \emptyset$ is a function that maps an intermediate event to its children.

Additionally, a fault tree must have one root, the top-level event, also called e_{top}.

2.2 Boolean Fault tree Logic

Boolean Fault tree Logic (BFL) is a logic to reason about fault trees. With BFL, properties of fault trees can be formulated and checked. BFL can be divided into *formulas* and *queries*. Similar to Boolean formulas, BFL formulas can be combined with Boolean connectives. The atoms in a BFL formula are the events in the fault tree. BFL queries are expressions that yield a result. A summary of BFL queries can be found in the summary at the end of this subsection.

The full[2] grammar of BFL formulas is given below, with $\bowtie \in \{<, \leq, =, \geq, >\}$ and $k \leq n$, where e can be any event in the fault

[2]"Full" because the operators described as "syntactic sugar" [12], i.e., those that can be derived from others, are also included. In [12], a minimal grammar is given, as well as the derivations of the non-minimal operators.

tree.

$$
\begin{aligned}
\phi ::= \;& e \\
& \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \implies \phi \mid \phi \equiv \phi \mid \phi \not\equiv \phi \\
& \mid \phi[e \mapsto 0] \mid \phi[e \mapsto 1] \\
& \mid \mathrm{MCS}(\phi) \mid \mathrm{MPS}(\phi) \\
& \mid \underset{\bowtie k}{\mathrm{Vot}}(\phi_1, \ldots, \phi_n) \\
\psi ::= \;& \exists\phi \mid \forall\phi \mid \mathrm{IDP}(\phi, \phi) \mid \mathrm{SUP}(e)
\end{aligned}
$$

BFL consists of two layers, denoted with ϕ and ψ, respectively. The first layer (ϕ) is the propositional logic layer. For all formulas in ϕ there exists a set (not necessarily non-empty) of status vectors that satisfy the formula. Other than the usual logical connectives, the first layer contains operators for setting evidence, minimal cut/path sets, and voting.

$\phi[e \mapsto 0]$ sets the event e in ϕ to 0 (false), and $\phi[e \mapsto 1]$ to 1 (true). As an example, $(a \vee b)[a \mapsto 1]$ evaluates to true, and $(a \vee b)[a \mapsto 0]$ is equivalent to b.

The MCS and MPS operators express, respectively, the minimal cut sets and minimal path sets of a given formula. While finding MCSs and MPSs most often happens for the TLE, the operators are not that restricted and can be used for any BFL formula in the first layer.

The Vot operator is similar to the VOT gate in a fault tree. It expresses certain combinations of inputs depending on k and $\bowtie$. For example, $\underset{<3}{\mathrm{Vot}}(\phi_1, \ldots, \phi_N)$ is true iff at most 2 of the inputs are true, and $\underset{\geq 4}{\mathrm{Vot}}(\phi_1, \ldots, \phi_N)$ is true iff at least 4 of the inputs are true.

The terminal symbols in the grammar are the events in the fault tree. We define the function VARS on any BFL formula to get the set of Boolean variables representing these events.

The semantics of BFL is given by the satisfaction relation $\models$, which expresses whether a status vector $\overline{b} = \langle b_1, \ldots, b_k \rangle$ and an FT T satisfy a formula ϕ. We write $\overline{b}, T \models \phi$ iff $\overline{b}$ satisfies ϕ with tree T. For example, if we take the fault tree in Figure 1 as T, and the status vector is given in the order $L1, L2, PO$, we have $\langle 1, 1, 0 \rangle, T \models NL$ and $\langle 1, 0, 0 \rangle, T \not\models NL$. In this paper, we will—among other things—present algorithms to check whether the satisfaction relation holds. Checking whether a satisfaction relation holds is one of the BFL queries.

If—for some tree T—a status vector does not satisfy a formula, [12] provides an algorithm to generate a counterexample. If $\overline{b}, T \not\models \phi$, a counterexample is a new status vector $\overline{c}$ such that $\overline{c}, T \models \phi$. Furthermore, the counterexample must only have the minimal necessary changes needed to ensure satisfaction, i.e., there is no c_i with $c_i \neq b_i$ in $\overline{c}$ that can be replaced with b_i without invalidating the satisfaction relation. Concretely, for all c_i with $c_i \neq b_i$, we have $\langle c_1, \ldots, c_{i-1}, b_i, c_{i+1}, \ldots, c_n \rangle, T \not\models \phi$.

BFL offers a means to get the *satisfaction set* of a formula ϕ. The satisfaction set is the set of all status vectors that satisfy ϕ, and it is represented by $[\![\phi]\!]$.

The second layer (ψ) adds quantifiers and the IDP and SUP operators. All formulas in this layer evaluate to either true or false given a tree T. $T \models \exists\phi$ is true iff there exists a status vector that satisfies ϕ with given T. $T \models \forall\phi$ is true iff all status vectors, i.e., all possible combinations of basic events, satisfy ϕ with given T.

The IDP operator can be used to check whether two formulas are independent under a given tree, i.e., the two formulas do not share any dependent variables.

A formula's dependent variables are those that can influence the result of the formula. Most variables are dependent, but a formula may also contain independent variables. For example, c is an independent variable in $(a \wedge b) \vee (a \wedge b \wedge c)$ because it can never influence the result of the formula; the formula is only satisfied iff a and b are true, in which case it does not matter what the value of c is.

The SUP operator is used to check whether an event e in a tree T is superfluous, i.e., its value can not influence the TLE.

In summary, BFL allows us to write three different kinds of queries. Firstly, a formula in the second layer (ψ) can be written and it can be checked whether it is satisfied by a tree, returning either true or false. Secondly, the satisfaction relation ($\models$) can be used to check whether a given status vector and tree satisfy a formula from the first layer. Lastly, to find all satisfying status vectors for a formula ϕ, the satisfaction set of ϕ can be calculated.

3 RELATED WORK

Because the paper on BFL [12] is quite new, no work built on top of it has been published yet. Therefore, no work related to translating BFL to QBF exists. However, replacing the use of BDDs with other solutions happens in other applications as well [3, 7]. These works all aim to solve the *space explosion problem*: the problem that arises because BDDs could take up exponential space in the order of the number of inputs. Biere et al. and Clarke et al.—similarly to us—harness the power of SAT solvers by constructing propositional formulas [3, 7]. Instead of using BDDs for model checking, they construct a propositional formula which is satisfiable iff some property about the model holds. This is very similar to our use case, except we will be checking BFL formulas instead of finite-state models.

3.1 QBF Solvers

As we will be using a QBF solver in this project, it is appropriate to discuss some popular QBF solvers and some of their differences. QBFLIB contains a large collection of QBF solvers [9]. On top of that, they also host evaluations of QBF solvers. In their evaluations, there are conjunctive normal form (CNF) tracks, and non-CNF tracks. All tracks are in prenex normal form (PNF). They provide two formats: QDIMACS [14], which only allows CNF and PNF, and QCIR [13], which allows non-CNF and non-PNF.

QuAbS (Quantified Abstraction Solver) [19] is an interesting and well-performing (arguably the best) non-CNF solver. It even accepts non-PNF formulas which is convenient because our translation of BFL may contain quantifiers between variables. A non-PNF solver means we do not have to rewrite the resulting formulas to PNF before solving them. CAQE [15] is perhaps the best CNF solver, created partially by the same author as [19]. However, it unfortunately only accepts formulas written in CNF. As rewriting a formula to CNF can be a complex task, it is unlikely we will be using a CNF solver.

Another interesting solver is Z3 [8]. Z3 is an SMT solver with support for quantifiers, making it a QBF solver as well. We do not know how its performance compares to that of [19]. However, what makes Z3 a very appealing option is the API it provides to easily write formulas inside different programming languages. With Z3, it would not be necessary to write in QCIR format (the format used by [19]), which makes implementation much simpler.

Additionally, Z3 can be used to calculate all satisfying solutions of a formula, which is not possible to do efficiently for any of the other mentioned solvers. In Z3, it is possible to add additional constraints to the solver after it found a solution. This can be used to force the solver to find another solution if a constraint is added that makes the current solution invalid. Adding a constraint does not reset all learned clauses in the solver so the solver can efficiently continue searching for a new solution. Since this is not possible for QuAbs and CAQE, it would be very inefficient to let those solvers find all solutions: it would require searching from scratch for each new solution.

4 IMPLEMENTATION

At the centre of the implementation lies the translation from BFL formulas to QBF. After the translation, depending on the BFL query, the QBF solver is used in different ways to determine an answer to the query.

4.1 From BFL to QBF

Before we can translate BFL into QBF, we need to be able to translate a fault tree event to a Boolean formula. Let **(Q)BF** be the set of all (quantified) Boolean formulas.

Definition 4.1. (Adapted from [12].) The translation function of an FT T is a function $\Psi_T\colon \mathsf{E} \to \mathbf{(Q)BF}$ that takes as input an element $e \in \mathsf{E}$. With $e' \in ch(e)$, we can define Ψ_T:

$$\Psi_T(e) = \begin{cases} \overline{\mathbf{B}}(e) & \text{if } e \in \mathtt{BE} \\ \bigvee_{e' \in ch(e)} \Psi_T(e') & \text{if } e \in \mathtt{IE} \text{ and } t(e) = \mathtt{OR} \\ \bigwedge_{e' \in ch(e)} \Psi_T(e') & \text{if } e \in \mathtt{IE} \text{ and } t(e) = \mathtt{AND} \\ \textsc{AtMost}_{k-1}(\Psi_T(ch(e))) & \text{if } e \in \mathtt{IE} \text{ and } t(e) = \mathtt{VOT}(<, k) \\ \textsc{AtMost}_{k}(\Psi_T(ch(e))) & \text{if } e \in \mathtt{IE} \text{ and } t(e) = \mathtt{VOT}(\leq, k) \\ \textsc{AtMost}_{k}(\Psi_T(ch(e))) & \\ \quad \wedge \textsc{AtLeast}_{k}(\Psi_T(ch(e))) & \text{if } e \in \mathtt{IE} \text{ and } t(e) = \mathtt{VOT}(=, k) \\ \textsc{AtLeast}_{k}(\Psi_T(ch(e))) & \text{if } e \in \mathtt{IE} \text{ and } t(e) = \mathtt{VOT}(\geq, k) \\ \textsc{AtLeast}_{k+1}(\Psi_T(ch(e))) & \text{if } e \in \mathtt{IE} \text{ and } t(e) = \mathtt{VOT}(>, k) \end{cases}$$

Where $\overline{\mathbf{B}}(e)$ is a Boolean formula consisting of only a variable representing e.

$\textsc{AtMost}_k$ and $\textsc{AtLeast}_k$ are Z3 functions that return true iff at most k inputs and at least k inputs are true, respectively.

To aid the translation of BFL into QBF, we will create some definitions.

Negating atoms. We define $\boldsymbol{N}\colon \mathbf{(Q)BF} \to \mathbf{(Q)BF}$, a function that negates all atoms in a Boolean formula. For example, $\boldsymbol{N}((a \wedge b) \vee c) = (\neg a \wedge \neg b) \vee \neg c$. It is used in the translation of the MPS operator.

Priming. A prime $'$ can be applied to a set of events, a status vector, or a BFL formula. For a set of events A, we have $A' = \{e' \mid e \in A\}$. For a status vector $\overline{b} = \langle b_1, \ldots, b_k \rangle$, we have $\overline{b'} = \langle b'_1, \ldots, b'_k \rangle$ When applied to a BFL formula, all atoms will be replaced by a primed version. For example, if we have $\phi = a \vee b$, then $\phi' = a' \vee b'$. Primes are used in the translation of the MCS and MPS operators. If multiple MCS/MPS operators are used in a single BFL formula, priming will still guarantee uniqueness in the whole formula, e.g., if we have $\phi = \rho \wedge \tau$ with $\rho = a \vee b$ and $\tau = a \implies b$, then $\rho' \wedge \tau' = (a' \vee b') \wedge (a'' \implies b'')$.

Smaller status vector. A status vector K is a subset of a status vector L if K and L consist of the same basic events and K contains fewer basic events that have a status of 1. Using priming for easier notation, a subset relation $A' \subset A$ between two status vectors can be encoded as a Boolean formula as follows: $A' \subset A = (\bigwedge_{a \in A} a' \implies a) \wedge (\bigvee_{a \in A} a' \not\equiv a)$. This can be read as: status vector A' is a subset of A iff all true (having status 1) basic events in A' are also true in A, and the status vectors have at least one basic event where the status is different, which means there is at least one false basic event in A' that is true in A.

Free variables. A Boolean variable is called *free* if it is not bound by a quantifier. For example, in $\exists a.\ a \vee b$, b is a free variable and a is a bound variable. Let $\mathbf{B}$ be the set of all Boolean variables. We define a function $\textsc{FreeVars}\colon \mathbf{(Q)BF} \to \mathbf{B}$ that returns all free variables in a given Boolean formula.

With these definitions, we can translate BFL to QBF with the recursion scheme below. The implementation uses a recursive approach. Each function call is stored in a cache to follow dynamic programming standards. Note that the IDP and SUP operators are not covered in the recursion scheme. Their implementations are covered in Section 4.2.

Algorithm 1 Translate BFL to QBF

Input: FT T, BFL formula χ
Output: Boolean formula
Method: Translate χ to QBF using the recursion scheme below.

Recursion scheme:

$B(T, e)$:	$\Psi_T(e)$
$B(T, \neg\phi)$:	$\neg(B(T, \phi))$
$B(T, \phi_1 \wedge \phi_2)$:	$B(T, \phi_1) \wedge B(T, \phi_2)$
$B(T, \phi_1 \vee \phi_2)$:	$B(T, \phi_1) \vee B(T, \phi_2)$
$B(T, \phi_1 \implies \phi_2)$:	$B(T, \phi_1) \implies B(T, \phi_2)$
$B(T, \phi_1 \equiv \phi_2)$:	$B(T, \phi_1) \equiv B(T, \phi_2)$
$B(T, \phi_1 \not\equiv \phi_2)$:	$B(T, \phi_1) \not\equiv B(T, \phi_2)$
$B(T, \phi[e_i \mapsto 0])$:	$\forall e_i.\, \neg e_i \implies B(T, \phi)$
$B(T, \phi[e_i \mapsto 1])$:	$\forall e_i.\, e_i \implies B(T, \phi)$
$B(T, \mathrm{MCS}(\phi))$:	$B(T, \phi) \wedge \left(\neg\exists \mathtt{BE}'.\ \mathtt{BE}' \subset \mathtt{BE} \wedge B(T, \phi')\right)$
$B(T, \mathrm{MPS}(\phi))$:	$\boldsymbol{N}(B(T, \neg\phi)) \wedge \left(\neg\exists \mathtt{BE}'.\ \mathtt{BE}' \subset \mathtt{BE} \wedge \boldsymbol{N}(B(T, \neg\phi'))\right)$
$B(T, \exists\phi)$:	$\exists \textsc{FreeVars}(B(T, \phi)).\, B(T, \phi)$

$$
\begin{aligned}
&B(T, \forall\phi): && \forall \text{FREEVARS}(B(T,\phi)).\, B(T,\phi)\\
&B(T, \underset{<k}{\text{Vot}}(\phi_1,\dots,\phi_n)): && \text{ATMOST}_{k-1}(B(T,\phi_1),\dots,B(T,\phi_n))\\
&B(T, \underset{\leq k}{\text{Vot}}(\phi_1,\dots,\phi_n)): && \text{ATMOST}_{k}(B(T,\phi_1),\dots,B(T,\phi_n))\\
&B(T, \underset{=k}{\text{Vot}}(\phi_1,\dots,\phi_n)): && \text{ATMOST}_{k}(B(T,\phi_1),\dots,B(T,\phi_n))\\
& && \wedge\ \text{ATLEAST}_{k}(B(T,\phi_1),\dots,B(T,\phi_n))\\
&B(T, \underset{\geq k}{\text{Vot}}(\phi_1,\dots,\phi_n)): && \text{ATLEAST}_{k}(B(T,\phi_1),\dots,B(T,\phi_n))\\
&B(T, \underset{>k}{\text{Vot}}(\phi_1,\dots,\phi_n)): && \text{ATLEAST}_{k+1}(B(T,\phi_1),\dots,B(T,\phi_n))
\end{aligned}
$$

4.2 Solving BFL Queries With QSAT

With the knowledge of how to translate BFL to QBF, we can investigate how different queries can be solved.

Quantified queries. The BFL formulas from the second layer that start with the existential or universal quantifier are the quantified queries. These queries can be translated to QBF after which they can be solved with Z3. The result of this query will be $\top$ iff the Boolean formula is satisfiable.

Algorithm 2 Check whether two BFL formulas ϕ_1 and ϕ_2 are independent.

Input: Two BFL formulas ϕ_1, ϕ_2 and FT T
Output: $\top$ iff the formulas are independent
Method:

```
function GetDependentVariables(φ, vs)
    res ← ∅
    for all v ∈ vs do
        if Substitute_φ(v, ⊤) ≠ Substitute_φ(v, ⊥) then
            res ← res ∪ {v}
        end if
    end for
    return res
end function

f1 ← B(T, φ1)
f2 ← B(T, φ2)
v1 ← Vars(f1)
v2 ← Vars(f2)
s ← v1 ∩ v2
dv1 ← GetDependentVariables(f1, s)
dv2 ← GetDependentVariables(f2, s)
if dv1 ∩ dv2 = ∅ then
    return ⊤
else
    return ⊥
end if
```

IDP query. To determine whether two BFL formulas are independent, we test if the two formulas do not share any dependent variables (see Algorithm 2). First, the BFL formula must be translated to QBF. Then, for the variables that are in both formulas, it is checked whether they are dependent variables. We substitute the variable with $\top$ and test whether it is equal to a formula where we substitute the variable with $\bot$. If they are unequal, it is concluded that the variable is a dependent variable in the formula. If the formulas do not share any dependent variables, they are independent.

The SUBSTITUTE function takes a formula and two values. It returns a new formula where the first value is substituted for the second value. For example, $\text{SUBSTITUTE}_{a\wedge b}(b, \top) = a \wedge \top$.

SUP query. This query is solved in the same way as the IDP query: $\text{SUP}(\phi) = \text{IDP}(\phi, \Psi_T(e_{top}))$.

Satisfaction relation query. To check whether a status vector satisfies a formula, the formula is first translated to QBF, after which Z3 is used to check whether the formula is satisfiable with the given status vector. To do this, the status vector must also be translated to a Boolean formula. Let us define $\Gamma(\overline{b})$ to translate a status vector $\overline{b}$ to a Boolean formula. A status vector is simply a conjunction of basic events, e.g., if we have a status vector $\overline{s} = \langle 1, 0, 1\rangle$ for basic events a, b, c, then $\Gamma(\overline{s}) = a \wedge \neg b \wedge c$. Consequently, $\Gamma(\langle 1\rangle) = d$ and $\Gamma(\langle 0\rangle) = \neg d$ if the status vector represents only event d. Concretely, $\overline{b}, T \models \phi$ iff $B(T,\phi) \wedge \Gamma(\overline{b})$ is satisfiable.

Satisfaction set query. We can use Z3 to find all satisfying status vectors for a formula. After using Algorithm 1, we can let Z3 compute all satisfying status vectors using the approach in [4, Section 5.1].

4.3 Counterexamples

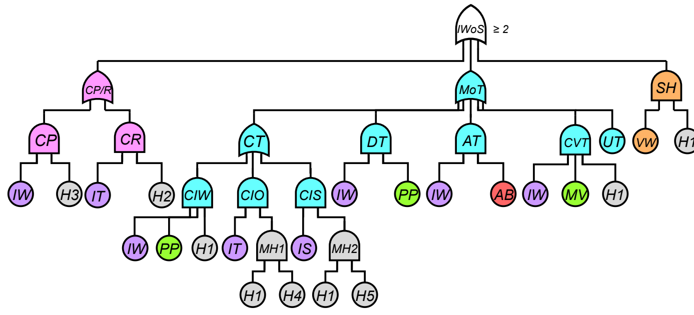
Algorithm 3 Generate counterexample for non-satisfying status vector $\overline{b}$.

Input: Status vector $\overline{b} = \langle b_1, \dots, b_n\rangle$, Boolean formula f
Output: A counterexample $\overline{c} = \langle c_1, \dots, c_n\rangle$
Method:

```
if f is unsatisfiable then
    return
end if

for all b_i ∈ b̄ do
    f_new ← f ∧ Γ(⟨b_i⟩)
    if f_new is satisfiable then
        c_i ← b_i
    else
        c_i ← ¬b_i
        f_new ← f ∧ ¬Γ(⟨b_i⟩)
    end if
    f ← f_new
end for
return c̄
```

Algorithm 3 is used to generate counterexamples. The non-satisfying status vector is modified in such a way that only those basic events that make the status vector non-satisfiable are changed. If the input formula is not satisfiable there cannot be a counterexample, thus nothing is returned in that case. The order in which the for loop is executed can affect the



COVID-19 TLE:	
COVID-19 Infected Worker on Site	IWoS
COVID-19 Pathogens/Reservoir:	
Exist. of COVID-19 Pathogens/Reservoir	CP/R
Exist. of COVID-19 Reservoir	CR
Exist. of COVID-19 Pathogen	CP
COVID-19 Mode of Transmission:	
Exist. of Mode of Transmission	MoT
Contact Transmission	CT
Droplet Transmission	DT
Airborne Transmission	AT
Common Vehicle Transmission	CVT
Unknown Modes of Transmissions	UT
Contact with Infected Worker	CIW
Contact with Infected Object	CIO
Contact with Infected Surface	CIS
COVID-19 Susceptible Hosts:	
Exist. of Susceptible Host	SH
Exist. of Vulnerable Worker	VW

Infected Object/Worker:	
Infected Worker joining the Team	IW
Infected Object used by the Team	IT
COVID-19 Infected Surface	IS
Physical Contact:	
Physical Proximity	PP
Mutual Use of Vehicles	MV
COVID-19 Status:	
COVID-19 Airborne	AB
Human Error:	
Non respect of outbreak procedures	H1
General Disinfection Error	H2
Detection Error	H3
Object Disinfection Error	H4
Surface Disinfection Error	H5
Multiple Human Errors v1	MH1
Multiple Human Errors v2	MH2

Fig. 2. COVID-19 Fault Tree.

counterexample generation; a different order may result in a different counterexample. In the Python implementation [18], the order may be different in separate invocations, so a different counterexample may be returned in a different run.

5 CASE STUDY

In this section, we will go over several BFL queries and examine their translations and results to showcase and validate the implementation. The queries were chosen to maximise the demonstration of the different parts of the implementation. In all queries, we use the COVID-19 fault tree from Figure 2. The tree is adjusted slightly from the one in [12]; the `AND` gate of the TLE is replaced with a `VOT`$(\geq, 2)$ gate to demonstrate the translation of a voting gate. The translation of this voting gate can be found in the second case study example.

The Python implementation of our approach can be found at [18]. All results in the examples in the case study were found using this implementation. In the `examples` directory of the repository, you can find a file named `case-study.bfl`. This file includes the tree from Figure 2, along with all the BFL queries used in the case study.

Out of 20 measurements, the fastest running time of `case-study.bfl` was 820ms. In this run, 30ms were spent on parsing the file, and the remaining 790ms were spent on solving the BFL queries. In this experiment, the results of the BFL queries were not printed to reduce overhead.

Universal quantification. Let us take a look at the BFL query $\forall(IS \implies MoT)$ which checks whether an infected surface is sufficient for the transmission of COVID. If we use Algorithm 1, we get the following Boolean formula:

$$\begin{aligned}&\forall IS, IW, PP, H1, IT, H4, H5, AB, MV, UT.\\ &\quad IS \implies (((IW \land PP \land H1) \lor (IT \land (H1 \land H4))\\ &\qquad \lor (IS \land (H1 \land H5)))\\ &\qquad \lor (IW \land PP) \lor (IW \land AB)\\ &\qquad \lor (IW \land MV \land H1) \lor UT)\end{aligned}$$

The Boolean formula is unsatisfiable and thus false.

Existential quantification and Vot operator. We can check whether the TLE can occur with no more than one human error with the following BFL query: $\exists(IWoS \land \mathrm{Vot}_{<2}(H1, \ldots, H5))$. QBF:

$$\begin{aligned}&\exists IW, H3, IT, H2, PP, H1, H4, IS, H5, AB, MV, UT, VW.\\ &\quad \textsc{AtLeast}_2((IW \land H3) \lor (IT \land H2),\\ &\qquad (((IW \land PP \land H1) \lor (IT \land (H1 \land H4))\\ &\qquad\quad \lor (IS \land (H1 \land H5)))\\ &\qquad\quad \lor (IW \land PP) \lor (IW \land AB)\\ &\qquad\quad \lor (IW \land MV \land H1) \lor UT),\\ &\qquad VW \land H1)\\ &\quad \land \textsc{AtMost}_1(H1, H2, H3, H4, H5)\end{aligned}$$

The formula is satisfiable which means the TLE could occur with at most 1 human error. If we input $[\![IWoS \land \mathrm{Vot}_{<2}(H1, \ldots, H5)]\!]$ to see in which cases this can occur, we see that there are 292 different status vectors that satisfy the formula. Such a large number may be overwhelming, so we can instead ask to receive only the minimal examples by using the following BFL query: $[\![IWoS \land \mathrm{Vot}_{<2}(H1, \ldots, H5) \land \mathrm{MCS}(IWoS)]\!]$ (simplified: $[\![\mathrm{MCS}(IWoS) \land \mathrm{Vot}_{<2}(H1, \ldots, H5)]\!]$). The following sets are returned: $\{IT, UT, H2\}, \{H3, IW, PP\}, \{IT, IW, AB, H2\}, \{VW, PP, IW, H1\}, \{H3, IW, UT\}, \{VW, IW, MV, H1\}, \{VW, IW, AB, H1\}, \{H3, IW, AB\}, \{VW, UT, H1\}, \{IT, IW, H2, PP\}$.

Setting evidence and quantification. Let us take a look at the following BFL queries: $\exists(CP[IW \mapsto 0])$ and $\exists(CP[IW \mapsto 0, H3 \mapsto 1])$. The translation of the first query is $\exists H3.\ \forall IW.\ \neg IW \implies IW \land H3$. Here we can see that IW was correctly excluded from the existential quantification because it was already universally quantified by the evidence. Likewise, the translation of the second query is $\forall IW, H4.\ \neg IW \land H3 \implies IW \land H3$, having no existential quantifier at all. Obviously, both BFL queries return false.

Satisfaction checking with evidence. Consider the following satisfaction relation query: $\overline{b}, T \models \neg MoT[UT \mapsto 0]$ with $\overline{b} = \{UT\}$. In this query, the given status vector conflicts with the evidence. Setting evidence has priority in this case [12], which means we expect the result to be true. Let us see how the implementation

addresses this. The query translates to[3]:

$$
\begin{aligned}
&(\forall UT.\ \neg UT \implies \neg MoT) \\
&\quad \wedge (UT \wedge \neg IT \wedge \neg IW \wedge \neg IS \wedge \neg H3 \wedge \neg AB \wedge \neg H5 \\
&\qquad \wedge \neg H2 \wedge \neg PP \wedge \neg VW \wedge \neg MV \wedge \neg H1 \wedge \neg H4)
\end{aligned}
$$

As one can see, the event UT from the status vector exists outside the quantification, which means both can happily coexist and the formula is satisfiable as expected.

Minimal cut sets of CP/R.. To find the minimal cut sets of CP/R, we can use the BFL query $[\![\mathrm{MCS}(CP/R)]\!]$, which will be translated into:

$$
\begin{aligned}
&((IW \wedge H3) \vee (IT \wedge H2)) \\
&\wedge (\neg\exists IW', H3', IT', H2', UT', PP', H1', \\
&\quad H4', IS', H5', AB', MV', VW'. \\
&\quad ((IW' \implies IW) \wedge (H3' \implies H3) \wedge (IT' \implies IT) \\
&\qquad \wedge (H2' \implies H2) \wedge (UT' \implies UT) \wedge (PP' \implies PP) \\
&\qquad \wedge (H1' \implies H1) \wedge (H4' \implies H4) \wedge (IS' \implies IS) \\
&\qquad \wedge (H5' \implies H5) \wedge (AB' \implies AB) \wedge (MV' \implies MV) \\
&\qquad \wedge (VW' \implies VW)) \\
&\quad \wedge ((IW' \not\equiv IW) \vee (H3' \not\equiv H3) \vee (IT' \not\equiv IT) \\
&\qquad \vee (H2' \not\equiv H2) \vee (UT' \not\equiv UT) \vee (PP' \not\equiv PP) \\
&\qquad \vee (H1' \not\equiv H1) \vee (H4' \not\equiv H4) \vee (IS' \not\equiv IS) \\
&\qquad \vee (H5' \not\equiv H5) \vee (AB' \not\equiv AB) \vee (MV' \not\equiv MV) \\
&\qquad \vee (VW' \not\equiv VW)) \\
&\quad \wedge ((IW' \wedge H3') \vee (IT' \wedge H2')))
\end{aligned}
$$

This results in the following MCSs: $\{IT, H2\}$ and $\{H3, IW\}$.

Unlike the quantifiers in the second layer (ψ), the MCS and MPS operators do not limit the quantified atoms to just the ones that are used inside the formula. The reason for that is the fact that the quantified atoms in the MCS and MPS translations cannot conflict with already quantified atoms because they are unique to the quantifier for which they were generated. For example, the translation of $\exists(SH[VW \mapsto 1])$ would have a conflicting quantification of VW: $\exists H1, VW.\ \forall VW.\ VW \implies VW \wedge H1$. In this wrongly translated example, VW is universally and existentially quantified at the same time. The correct translation is: $\exists H1.\ \forall VW.\ VW \implies VW \wedge H1$

Minimal path sets of CP/R.. Similarly to the previous example, we can get the MPSs with the formula $[\![\mathrm{MPS}(CP/R)]\!]$, which results in the following translation[4]:

$$
\begin{aligned}
&\neg((\neg IW \wedge \neg H3) \vee (\neg IT \wedge \neg H2)) \\
&\wedge (\neg\exists IW', H3', IT', H2'. \\
&\quad ((IW' \implies IW) \wedge (H3' \implies H3) \wedge (IT' \implies IT) \\
&\qquad \wedge (H2' \implies H2)) \\
&\quad \wedge ((IW' \not\equiv IW) \vee (H3' \not\equiv H3) \vee (IT' \not\equiv IT) \\
&\qquad \vee (H2' \not\equiv H2)) \\
&\quad \wedge \neg((\neg IW' \wedge \neg H3') \vee (\neg IT' \wedge \neg H2')))
\end{aligned}
$$

It gives the following MPSs as result: $\{IT, IW\}$, $\{IT, H3\}$, $\{IW, H2\}$, and $\{H3, H2\}$.

Different counterexamples. In the example before the previous one, we saw that the MCSs of CP/R are $\{IT, H2\}$ and $\{H3, IW\}$. Let us see what happens if we try to check a slightly wrong status vector and see what counterexamples it generates. If we take $\overline{b} = \langle 1, 1, 1, 0\rangle$ for $IW, H3, IT, H2$ and run $\overline{b}, T \models \mathrm{MCS}(CP/R)$, we find that it will either return $\{IW, H3\}$ or $\{IT, H2\}$. The result differs based on the order of the `for` loop in Algorithm 3.

Let us consider the two orders: (1) $IW, H3, IT, H2$; and (2) $IT, H3, IW, H2$. First, consider order (1). When we reach IT, we find that the formula is unsatisfiable with $IT = 1$, so we set $IT = 0$. $H2$ was already set to 0 so no change is needed there. We end up with the counterexample $\{IW, H3\}$. Now consider order (2). We have $IT = 1$ in $\overline{b}$, so when we reach $H3$ in the loop, we find that an MCS with $IT = 1$ and $H3 = 1$ does not exist. Therefore, we set $H3 = 0$ in the counterexample and ultimately end up with $\{IT, H2\}$ as the result.

6 DISCUSSION AND FUTURE WORK

Unfortunately, there is no implementation of BFL that uses BDDs yet, so we cannot make a direct comparison. However, we can discuss some qualitative differences.

The main drawback of using non-reduced BDDs is the space explosion problem, which is the problem of the number of vertices growing exponentially in the number of Boolean variables. While the introduction of reduced ordered BDDs (hereafter, just BDDs) by Bryant revolutionised the field of symbolic representation of Boolean functions, it does not completely solve this problem [6]. In some cases, BDDs can provide a very compact representation of a Boolean function with many Boolean variables. However, in the worst case, BDDs still need $2^n + 1$ vertices to represent a function with n variables.

The size of a BDD is very sensitive to the ordering of the variables [6]. Depending on the ordering, a BDD could have a low (such as $2n$) or very high (such as 2^n) number of vertices. Finding the most optimal ordering is an NP-hard problem [5]. In practical use cases where Boolean formulas have similar structures, there is often a set of heuristics for choosing a good ordering [16]. However, when Boolean formulas can have greatly varying structures—such as in BFL—it is difficult to find good orderings. Because of that, building a performant BFL implementation with BDDs is not a trivial task.

[3] Translation of MoT omitted for brevity.

[4] Irrelevant events inside the MPS translation omitted for brevity.

Many BFL queries are able to use properties that are inherent to BDDs to return a result in constant time. For example, a quantified query with a universal quantifier only needs to check whether the BDD is equal to the terminal 1 node, and for an existential quantifier, it only needs to check that the BDD is not equal to the terminal 0 node. Furthermore, independence between two formulas can easily be checked by testing if the BDDs for those formulas share any variables (this is not possible for Boolean formulas because a formula may contain independent variables[5] whereas a BDD may not). While this sounds great at first glance, a bad variable ordering might cause the construction of the BDD to take up to exponential time, diminishing the mentioned benefits.

Performance differences between BDD- and SAT-based approaches have been researched in the past [1, 2, 10, 11, 21]. The papers show mixed results regarding which is the better approach, however, all agree that BDD- and SAT-based approaches complement each other. Different types of problems are better solved by one approach than the other; a problem that is infeasible to solve with one approach may be trivial for the other. Goldberg et al., Jöbstl et al. and Wille et al. [10, 11, 21] show that some problems in their papers are solved more quickly with a SAT solver, and others with a BDD-based approach.

As of now, it is unclear what properties contribute to easier problem-solving for one approach, and how this can be applied to problems in BFL. It would be interesting to research which problems in BFL are more easily solved by the SAT-based approach versus the BDD-based approach.

The two approaches' performance on quantified queries makes a fair comparison, as satisfiability testing is the primary function of a SAT solver. On the other hand, obtaining the satisfaction set may be a less optimal task for SAT solvers because their designs often prioritise finding a single satisfying model over finding all possible models. Moreover, the current implementation of checking for independence requires solving two formulas for each shared variable, whereas the BDD approach only needs to construct one BDD for each formula. In this case, the performance of the SAT implementation is heavily influenced by the number of shared variables between the two formulas whereas the BDD approach is not.

Next to the type of BFL query, the structure of the fault tree may also influence the performance of the two approaches in different ways. Ideally, the performance of the approaches should also be compared for different tree shapes and sizes.

7 CONCLUSION

This paper presented the first-ever implementation of BFL, making it possible to use BFL in practice. The implementation uses a QBF solver, a complementary approach to a previously designed BDD-based approach. At the core of the implementation lies the translation algorithm that translates BFL to QBF. The resulting QBF formula is then combined with a QBF solver in different ways depending on the BFL query that must be solved. An algorithm to generate counterexamples is also presented. Different parts of the implementation were demonstrated in a case study. Finally, we discussed the differences from a BDD-based approach and examined various aspects that should be taken into account if this implementation is compared to a BDD-based approach in the future.

ACKNOWLEDGEMENTS

I am deeply grateful for the guidance and encouragement provided by my supervisors, Stefano Nicoletti and Moritz Hahn, throughout the course of my thesis. Their support and extensive feedback have greatly contributed to the quality of my work. Their willingness to assist whenever needed, and their sincere interest in my progress throughout the duration of this thesis have provided a significant source of motivation. Additionally, I would like to thank everyone else who proofread or gave feedback on this work in any way.

REFERENCES

[1] Parosh Aziz Abdulla, Per Bjesse, and Niklas Eén. 2000. Symbolic Reachability Analysis Based on SAT-Solvers. en. In *Tools and Algorithms for the Construction and Analysis of Systems* (Lecture Notes in Computer Science). Susanne Graf and Michael Schwartzbach, (Eds.) Springer, Berlin, Heidelberg, 411–425. ISBN: 978-3-540-46419-8. DOI: 10.1007/3-540-46419-0_28.

[2] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. 1999. Symbolic model checking using SAT procedures instead of BDDs. en. *Proceedings of the 36th annual ACM/IEEE Design Automation Conference*, (June 1999), 317–320. ISBN: 9781581131093. DOI: 10.1145/309847.309942.

[3] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. 1999. Symbolic Model Checking without BDDs. en. In *Tools and Algorithms for the Construction and Analysis of Systems* (Lecture Notes in Computer Science). W. Rance Cleaveland, (Ed.) Springer, Berlin, Heidelberg, 193–207. ISBN: 978-3-540-49059-3. DOI: 10.1007/3-540-49059-0_14.

[4] Nikolaj Bjørner, Leonardo de Moura, Lev Nachmanson, and Christoph Wintersteiger. [n. d.] Programming Z3. (). Retrieved June 20, 2023 from https://theory.stanford.edu/~nikolaj/programmingz3.html.

[5] B. Bollig and I. Wegener. 1996. Improving the variable ordering of OBDDs is NP-complete. *IEEE Transactions on Computers*, 45, 9, (Sept. 1996), 993–1002. DOI: 10.1109/12.537122.

[6] Randal Bryant. 1986. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35, 8, (Aug. 1986), 677–691. DOI: 10.1109/TC.1986.1676819.

[7] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. 2001. Bounded Model Checking Using Satisfiability Solving. en. *Formal Methods in System Design*, 19, 1, (July 1, 2001), 7–34. DOI: 10.1023/A:1011276507260.

[8] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. en. In *Tools and Algorithms for the Construction and Analysis of Systems* (Lecture Notes in Computer Science). C. R. Ramakrishnan and Jakob Rehof, (Eds.) Springer, Berlin, Heidelberg, 337–340. ISBN: 978-3-540-78800-3. DOI: 10.1007/978-3-540-78800-3_24.

[9] E. Giunchiglia, M. Narizzano, L. Pulina, and A. Tacchella. 2005. Quantified Boolean Formulas satisfiability library (QBFLIB). Retrieved May 3, 2023 from https://www.qbflib.org.

[10] E.I. Goldberg, M.R. Prasad, and R.K. Brayton. 2001. Using SAT for combinational equivalence checking. In *Proceedings Design, Automation and Test in Europe. Conference and Exhibition 2001.* Proceedings Design, Automation and Test in Europe. Conference and Exhibition 2001. (Mar. 2001), 114–121. DOI: 10.1109/DATE.2001.915010.

[11] Elisabeth Jöbstl, Martin Weiglhofer, Bernhard K. Aichernig, and Franz Wotawa. 2010. When BDDs Fail: Conformance Testing with Symbolic Execution and SMT Solving. In *Verification and Validation 2010 Third International Conference on Software Testing*. Verification and Validation 2010 Third International Conference on Software Testing. (Apr. 2010), 479–488. DOI: 10.1109/ICST.2010.48.

[12] Stefano M. Nicoletti, E. Moritz Hahn, and Mariëlle Stoelinga. 2022. BFL: a Logic to Reason about Fault Trees. In *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN).* 2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). (June 2022), 441–452. DOI: 10.1109/DSN53405.2022.00051.

[13] 2014. QCIR-G14: A Non-Prenex Non-CNF Format for Quantified Boolean Formulas. en. (Apr. 8, 2014). Retrieved Mar. 5, 2023 from https://www.qbflib.org/qcir.pdf.

[14] 2005. QDIMACS standard. en. (Dec. 21, 2005). Retrieved May 3, 2023 from https://www.qbflib.org/qdimacs.html.

[15] Markus N. Rabe and Leander Tentrup. 2015. CAQE: A Certifying QBF Solver. In *2015 Formal Methods in Computer-Aided Design (FMCAD).* 2015 Formal Methods

[5] Recall the explanation of independent variables in Section 2.2.

in Computer-Aided Design (FMCAD). (Sept. 2015), 136–143. DOI: 10.1109/FMCAD.2015.7542263.

[16] R. Rudell. 1993. Dynamic variable ordering for ordered binary decision diagrams. In *Proceedings of 1993 International Conference on Computer Aided Design (ICCAD)*. Proceedings of 1993 International Conference on Computer Aided Design (ICCAD). (Nov. 1993), 42–47. DOI: 10.1109/ICCAD.1993.580029.

[17] Enno Ruijters and Mariëlle Stoelinga. 2015. Fault tree analysis: A survey of the state-of-the-art in modeling, analysis and tools. en. *Computer Science Review*, 15–16, (Feb. 1, 2015), 29–62. DOI: 10.1016/j.cosrev.2015.03.001.

[18] [SW] Caz Saaltink, BFL QSAT Implementation version 1.0.1, June 29, 2023. DOI: 10.5281/zenodo.8093373, URL: https://github.com/CazSaa/BFL-QSAT-implementation.

[19] Leander Tentrup. 2016. Non-prenex QBF Solving Using Abstraction. en. In *Theory and Applications of Satisfiability Testing – SAT 2016* (Lecture Notes in Computer Science). Nadia Creignou and Daniel Le Berre, (Eds.) Springer International Publishing, Cham, 393–401. ISBN: 978-3-319-40970-2. DOI: 10.1007/978-3-319-40970-2_24.

[20] W. E. Vesely, F. F. Goldberg, N. H. Roberts, and D. F. Haasl. 1987. Fault Tree Handbook. (Dec. 17, 1987).

[21] Robert Wille, Daniel Große, Finn Haedicke, and Rolf Drechsler. 2009. SMT-based stimuli generation in the SystemC Verification library. In *2009 Forum on Specification & Design Languages (FDL)*. 2009 Forum on Specification & Design Languages (FDL). (Sept. 2009), 1–6.