Solving Queries for Boolean Fault Tree Logic via Quantified SAT

Caz Saaltink
c.saaltink@student.utwente.nl
University of Twente
Enschede, the Netherlands

Stefano M. Nicoletti*
s.m.nicoletti@utwente.nl
University of Twente
Enschede, The Netherlands

Matthias Volk*
m.volk@utwente.nl
University of Twente
Enschede, The Netherlands

Ernst Moritz Hahn*
e.m.hahn@utwente.nl
University of Twente
Enschede, The Netherlands

Mariëlle Stoelinga*
m.i.a.stoelinga@utwente.nl
University of Twente
Enschede, The Netherlands
Radboud University
Nijmegen, The Netherlands

Abstract

Fault trees (FTs) are hierarchical diagrams used to model the propagation of faults in a system. Fault tree analysis (FTA) is a widespread technique that allows to identify the key factors that contribute to system failure. In recent work [26] we introduced *BFL*, a Boolean Logic for Fault trees. *BFL* can be used to formally define simple yet expressive properties for FTA, e.g.: 1) we can set evidence to analyse what-if scenarios; 2) check whether two elements are independent or if they share a child that can influence their status; 3) and set upper/lower boundaries for failed elements. Furthermore, we provided algorithms based on binary decision diagrams (BDDs) to check *BFL* properties on FTs. In this work, we evaluate usability of a different approach by employing quantified Boolean formulae (QBFs) instead of BDDs. We present a translation from *BFL* to QBF and provide an implementation—making it the first tool for checking *BFL* properties—that builds on top of the Z3 solver. We further demonstrate its usability on a case study FT and investigate runtime, memory consumption and scalability on a number of benchmark FTs. Lastly, qualitative differences from a BDD-based approach are discussed.

CCS Concepts: • **Theory of computation → Logic and verification**; **Logic and verification**; • **Computer systems organization → Dependable and fault-tolerant systems and networks**.

Keywords: fault trees, QSAT, quantified Boolean formulae

ACM Reference Format:
Caz Saaltink, Stefano M. Nicoletti, Matthias Volk, Ernst Moritz Hahn, and Mariëlle Stoelinga. 2023. Solving Queries for Boolean Fault Tree Logic via Quantified SAT. In *Proceedings of the 9th ACM SIGPLAN International Workshop on Formal Techniques for Safety-Critical Systems (FTSCS '23), October 22, 2023, Cascais, Portugal.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3623503.3623535

1 Introduction

Fault trees are a common model for safety-critical systems, such as nuclear power plants, aviation, and autonomous vehicles [35]. A fault tree (FT) models how a system can fail due to (sub-)component failures [42]. FT analysis (FTA) is utilised to analyse the causes of system failures, identify critical components and calculate the overall reliability of a system. FTA traditionally only considers a limited number of simple measures, such as the overall system reliability or identifying a minimal numbers of components which lead to system failure. As systems grow more complex, the need for more complex analysis arises, capturing dependencies between different system components. Recently, [26] introduced the *Boolean FT Logic (BFL)* which provides a rich formalism to specify such properties for FTs. BFL allows writing first-order logic statements about FTs. The atoms in this logic are the events of the FT. In addition to standard operators such as negation, conjunction, implication, etc., BFL also provides FT-specific operators: minimal cut sets (MCS), minimal path sets (MPS), independence of events, and setting evidence, i.e., only including scenarios where events have a set value.

The paper on BFL [26] provides algorithms to 1) check if a BFL formula holds for a given FT and status vector —providing the state (operational or failed) per basic event; 2) find all status vectors that satisfy a BFL formula for a given

*This work was partially funded by the NWO grant NWA.1160.18.238 (PrimaVera), and the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 101008233, and the ERC Consolidator Grant 864075 (*CAESAR*).

FTSCS '23, October 22, 2023, Cascais, Portugal

ACM ISBN 979-8-4007-0398-0/23/10.
https://doi.org/10.1145/3623503.3623535

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 events cause the different outcomes.

All algorithms described in [26] make use of (reduced ordered) binary decision diagrams (BDDs), a common data structure to represent FTs [15, 34, 38] However, no tool support for the BDD-based analysis of BFL is provided.

Motivation. Performance differences between BDD- and SAT-based approaches have been researched in the past [1, 9, 23, 24, 43]. Literature shows 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. [23, 24, 43] show that some problems are solved more quickly with a SAT solver, and others with a BDD-based approach. In light of these findings, a comparison between a SAT- and BDD-based approach for solving BFL-related queries seems desirable.

Contribution. In this work, we present the first tool to support BFL analysis. Our approach is based on SAT solving. We introduce a translation from BFL formulae to quantified Boolean formulae (QBF) and employ QBF-solvers such as Z3 [16] for the analysis by means of an artifact. The translation is implemented in Python and publicly available at [36]. Our evaluation shows the feasibility of the QBF-based analysis of BFL.

Our approach. We first take a BFL formula and a FT, then translate them both to QBF. Once we have a QBF formula encoding this combination, we use a SAT solver to check it. The result depends on the type of query expressed by the QBF formula: in total, three different kinds of queries can be expressed (see Sect. 2.2.4 and Fig. 2).

Outline. We provide background information in Sect. 2 and present related work in Sect. 3. The translation to QBF is introduced in Sect. 4. Sect. 5 applies it on a case study and Sect. 6 presents an experimental evaluation of the proposed approach. In Sect. 7 we discuss differences between the QBF- and a BDD-based approach to then conclude in Sect. 8.

2 Background

2.1 Fault Trees

A *fault tree (FT)* is a rooted directed acyclic graph [35]. The leaves are called *basic events (BE)* and represent atomic subcomponents which can be operational or failed. The inner nodes are logical *gates* and model the propagation of failures through the FT. For instance, an AND gate is called failed if all its children are failed; an OR gate is failed if at least one child is failed. A third, often used gate is the voting gate (also called: VOT, VT, k/n, or k-out-of-n gate), which fails if at least k out of its n total children have failed. In this paper, the voting gate is extended to use any comparison $\bowtie \in \{<, \leq, =, \geq, >\}$. The original k/n gate is equal to a $\mathtt{VOT}(\geq, k)$ gate. The root node is called the *top-level event (TLE)* and its failure represents a complete system failure.

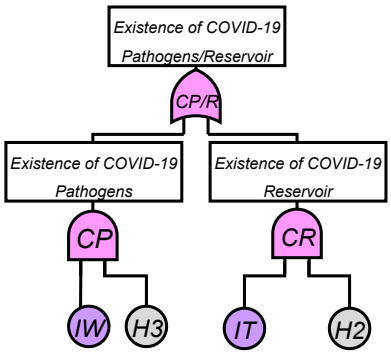

Figure 1. Small example FT (excerpt from Fig. 3).

Definition 2.1. A *Fault Tree* is a tuple $T = (\mathtt{E}, A, t)$ where $(\mathtt{E}, A)$ is a rooted directed acyclic graph ($\mathtt{E}$ are the vertices, called *events*) and t is a map specifying the type $\mathtt{E} \to \{\mathtt{AND}, \mathtt{OR}, \mathtt{VOT}(\bowtie, k), \mathtt{BE}\}$ such that $t(e) = \mathtt{BE}$ iff e is a leaf.

We denote the top event by e_{top}, and the set of children of an event e by $ch(e) = \{e' \mid (e, e') \in A\}$. Slightly abusing notation, we denote the set of *basic events*, e with $t(e) = \mathtt{BE}$, as $\mathtt{BE}$, whose elements we enumerate $\mathtt{BE} = \{e_1, \ldots, e_n\}$. We also define the set of intermediate events $\mathtt{IE} = \mathtt{E} \setminus \mathtt{BE}$.

Example 2.2 (Fault Tree)**.** The (sub)tree represented in Fig. 1 is an excerpt from the FT in Fig. 3 representing an infected worker joining on site. The TLE of Fig. 1 — Existence of COVID-19 Pathogens/Reservoir — is refined by an OR-gate (*CP/R*). For *CP/R* to fail, either pathogens must exist on the workplace, i.e., Existence of COVID-19 Pathogens (*CP*), or there must be an infected object of some kind, i.e., Existence of COVID-19 Reservoir (*CR*) has to happen. Both *CP* and *CR* are AND-gates: for them to fail, all their respective children need to fail. For *CP* this means that an Infected worker joining the team (*IW*) and a failure in detecting this, i.e., Detection error (*H3*) must happen. For *CR* this means that an *Infected object used by the team* (*IT*) and a General disinfection error (*H2*) must happen.

MCS and MPS. The most important techniques in Fault tree analysis (FTA) are cut set and path set analysis, to find minimal cut sets (MCSs) or minimal path sets (MPSs). A *cut set* is a set of basic events that, if they occur, cause the TLE to occur. A cut set is *minimal* if no events can be removed

from it without failing to cause the TLE. On the other hand, a *minimal path set (MPS)* is a minimal set of basic events that prevent system failure, i.e., the TLE cannot occur when these basic events do not occur.

Example 2.3 (MCS and MPS)**.** For the FT in Fig. 1, $\{IW, H3, IT\}$ is a cut set, though it is not minimal. The MCSs of the tree are $\{IW, H3\}$ and $\{IT, H2\}$. The MPSs are $\{IT, IW\}$, $\{IT, H3\}$, $\{IW, H2\}$, and $\{H3, H2\}$.

Status vector. A *status vector* $\overline{b} \in \{0,1\}^{|BE|}$ represents the statuses of the basic events. A value of 1 means that the BE does occur (it failed), and 0 means that it does not (it is operational). A status vector can also be represented as a set. In this case, it contains all basic events with a status of 1.

2.2 Boolean Fault tree Logic

Boolean Fault tree Logic (BFL) is a logic to reason about FTs [26]. With *BFL*: 1. We can set evidence to analyse what-if scenarios. E.g., what are the MCSs, given that BEA or subsystem B has failed? What are the MPSs given that A or B have not failed? 2. We can check whether two elements are independent or if they share a children that can influence their status. 3. We can check whether the failure of one (or more) element E always leads to the failure of TLE. 4. We can set upper/lower boundaries for failed elements. E.g., would element E always fail if at most/at least two out of A, B and C were to fail? BFL can be divided into *formulae* and *queries*. We define BFL *queries* as expressions that yield a result. A summary of BFL queries can be found in Sect. 2.2.4.

Example 2.4 (A BFL query)**.** For example, the following BFL query $\exists(CP/R[IT \mapsto 0, H3 \mapsto 1])$ returns *True* for the the FT in Fig. 1, since there is a way for the TLE to fail, when IT remains operational and $H3$ fails.

Similar to Boolean formulae, BFL formulae can be combined with Boolean connectives. The atoms in a BFL formula are the events in the FT.

Definition 2.5 (BFL syntax)**.** The minimal grammar of BFL formulae is given below, where e can be any event in the FT.

Layer 1: $\phi ::= e \mid \neg\phi \mid \phi \wedge \phi \mid \phi[e \mapsto 0] \mid \phi[e \mapsto 1] \mid \text{MCS}(\phi)$

Layer 2: $\psi ::= \exists\phi \mid \forall\phi \mid \text{IDP}(\phi, \phi)$

BFL consists of two layers, denoted by ϕ and ψ.

2.2.1 First BFL layer: ϕ.

The *first layer* (ϕ) is the propositional logic layer. The terminal symbols e are events in the FT. Other than the usual logical connectives, the first layer contains operators for setting evidence and minimal cut sets.

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

MCS. The $\text{MCS}(\phi)$ operator expresses the minimal cut sets of a given formula. While MCSs are usually computed for the TLE, the operator is not restricted and can be used for any BFL formula in the first layer.

2.2.2 Second BFL layer: ψ.

The *second layer* (ψ) adds quantifiers and an operator for independence. All formulae in this layer evaluate to either true or false given a tree T.

Quantifiers. An existentially quantified formula $\exists\phi$ is true iff there exists a status vector, i.e., statuses of BEs, that satisfies ϕ for a given tree.

Independence. The IDP operator can be used to check whether two formulae are independent under a given tree, i.e., the two formulae do not share any dependent variables. A formula's dependent variables are those that can influence the result of the formula. For example, c is an independent variable in $(a \wedge b) \vee (a \wedge b \wedge c)$ because it never influences the result of the formula. The formula is only satisfied iff a and b are true; the value of c is irrelevant.

2.2.3 Syntactic sugar.

Several derived operators including the usual Boolean operators are defined in [26]:

$$\begin{aligned}
\phi_1 \vee \phi_2 &::= \neg(\neg\phi_1 \wedge \neg\phi_2)\\
\phi_1 \Rightarrow \phi_2 &::= \neg(\phi_1 \wedge \neg\phi_2)\\
\phi_1 \equiv \phi_2 &::= (\phi_1 \Rightarrow \phi_2) \wedge (\phi_2 \Rightarrow \phi_1)\\
\phi_1 \not\equiv \phi_2 &::= \neg(\phi_1 \equiv \phi_2)\\
\text{MPS}(\phi) &::= \text{MCS}(\neg\phi)\\
\text{SUP}(e) &::= \text{IDP}(e, e_{top})
\end{aligned}$$

$$\underset{\bowtie k}{\text{Vot}}(\phi_1, \ldots, \phi_N) ::= \bigvee_{\substack{U \subseteq \{1,\ldots,N\} \\ |U| \bowtie k}} \left(\bigwedge_{u \in U} \phi_u \right) \wedge \left(\bigwedge_{u \in \{1,\ldots,N\} \setminus U} \neg\phi_u \right)$$

MPS. The $\text{MPS}(\phi)$ operator expresses the minimal path sets as the MCS of the negated formula.

Superfluous. The $\text{SUP}(e)$ operator is used to check whether an event $e \in \mathsf{E}$ in a tree T is superfluous, i.e., its value can not influence the TLE. In other words, the event e is independent from the TLE e_{top}.

Voting. The voting operator $\underset{\geq k}{\text{Vot}}(\phi_1, \ldots, \phi_N)$ with $k \leq N$ and $\bowtie \in \{<, \leq, =, \geq, >\}$ holds if at least k of the formulae $\phi_1, \ldots, \phi_N$ hold. The operator is thus similar to the VOT gate in a FT. As an example, $\underset{<3}{\text{Vot}}(\phi_1, \ldots, \phi_N)$ is true iff at most 2 of the formulae $\phi_1, \ldots, \phi_N$ are true, and $\underset{\geq 4}{\text{Vot}}(\phi_1, \ldots, \phi_N)$ is true iff at least 4 of the formulae are true.

2.2.4 Queries.

The semantics of the first BFL layer is given by the satisfaction relation $\models$. We write $\overline{b}, T \models \phi$ iff status vector $\overline{b} = \langle b_1, \ldots, b_k \rangle$ satisfies formula ϕ with FT T. For example, if we take the FT in Fig. 1 as T, and the status vector is given in the order IW, $H3$, IT, $H2$, we have $\langle 1,1,0,0 \rangle, T \models CP/R$ and $\langle 1,0,0,0 \rangle, T \not\models CP/R$. BFL allows us to write three different kinds of queries. Firstly, it can be checked whether

a formula ϕ in the first layer is satisfied by a status vector and a tree, i.e., whether $\overline{b}, T \models \phi$. Secondly, the *satisfaction set* $[\![\phi]\!]$ of a formula ϕ in the first layer can be computed, i.e., all status vectors satisfying ϕ. Lastly, it can be checked whether a formula ψ in the second layer is satisfied by a tree, i.e., whether $T \models \psi$.

Counterexample. If—for some tree T—a status vector does not satisfy a formula ϕ in the first layer, i.e., $\overline{b}, T \not\models \phi$, a counterexample can be created [26]. The 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 $\langle c_1, \ldots, c_{i-1}, b_i, c_{i+1}, \ldots, c_n \rangle, T \not\models \phi$.

3 Related Work

Because the paper on BFL [26] is quite recent, no work related to translating BFL to QBF exists. However, replacing the use of BDDs with other methods is a solution explored in related literature [10, 14]. These works all aim to solve the *space explosion problem*: the problem that arises due to BDDs potentially taking 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 formulae [10, 14]. 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 formulae instead of finite-state models.

3.1 Quantified Boolean formulae

Quantified Boolean formulae (QBFs) extend propositional formulae with existential and universal quantifiers over the propositional variables [11]. Implementing effective reasoning tools to perform satisfiability checks of QBFs is an important research issue in artificial intelligence and computing, especially formal verification [11]: in fact, QBF solvers have already been proposed for a wide range of tasks in knowledge representation and reasoning, in automated planning, and in formal methods for computer-aided design. In a recent survey, Shukla et al. present different areas in which QBF encodings and technologies are successfully applied: these include reactive synthesis [25, 2, 21], equivalence checking [28, 27, 8], adversarial games [29, 3, 18], non-monotonic reasoning [19, 20, 4] and model checking [10, 12, 17].

3.2 QBF Solvers

As we use a QBF solver in this paper, it is appropriate to discuss some popular QBF solvers and some of their differences. QBFLIB contains a large collection of QBF solvers [22]. 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 [31], which only allows CNF and PNF, and QCIR [30], which allows non-CNF and non-PNF.

QuAbS (Quantified Abstraction Solver) [41] is an interesting and well-performing non-CNF solver. It even accepts non-PNF formulae 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 formulae to PNF before solving them. CAQE [32] is perhaps the best CNF solver, created partially by the same author as [41]. However, it unfortunately only accepts formulae written in CNF. As rewriting a formula to CNF can be a complex task, the choice of a CNF solver seems preferable. Another interesting solver is Z3 [16]. Z3 is an SMT solver with support for quantifiers, making it a QBF solver as well.

Our choice. We favour Z3 for implementing our solution. What makes Z3 a very appealing option is the API it provides to easily write formulae inside different programming languages. With Z3, it is not necessary to write in QCIR format (the format used by [41]), 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.

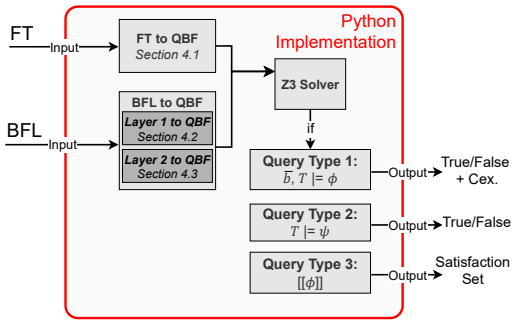

Figure 2. Overview of the QBF translation approach.

4 Algorithms & Implementation

We provide an overview of our approach in Fig. 2. Given an FT and a BFL formula, the FT is encoded into QBF and the two layers of the BFL formula are also translated to QBF. Afterwards, the Z3 solver is used to answer the given query.

4.1 Translating an FT to QBF

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

Definition 4.1 (FT to QBF)**.** The translation function of an FT T is given by the function $\Psi_T \colon \mathsf{E} \to$ (**Q**)**BF** defined recursively as follows:

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

where $\overline{\mathbf{B}}(e)$ is a Boolean formula consisting of only a variable representing event e. Note that we use the short form $\Psi_T(ch(e)) := \Psi_T(ch(e)_1), \ldots, \Psi_T(ch(e)_n)$ for brevity.

$\textsc{AtMost}_k(\phi_1, \ldots, \phi_n)$ and $\textsc{AtLeast}_k(\phi_1, \ldots, \phi_n)$ are functions that return true iff at most k inputs and at least k inputs are true, respectively. Such functions are called Boolean cardinality constraints and can be encoded in different ways [39].

4.2 Translating first layer BFL formulae to QBF

Priming. A prime $'$ can be applied to a set of events, a status vector, or a BFL formula and creates a copy of the given data structure with new unique variables. 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 operators. If multiple MCS 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.

Definition 4.2 (First layer BFL formula to QBF)**.** We let X_1 be the set of layer one formulae: the translation function of a first layer BFL formula ϕ for a tree T is given by the function $B \colon T \times X_1 \to$ (**Q**)**BF** defined recursively as follows:

$$\begin{aligned} B(T, e) &= \Psi_T(e) \\ B(T, \neg\phi) &= \neg(B(T, \phi)) \\ B(T, \phi_1 \wedge \phi_2) &= B(T, \phi_1) \wedge B(T, \phi_2) \\ B(T, \phi[e_i \mapsto 0]) &= \forall e_i . \neg e_i \implies B(T, \phi) \\ B(T, \phi[e_i \mapsto 1]) &= \forall e_i . e_i \implies B(T, \phi) \\ B(T, \mathrm{MCS}(\phi)) &= B(T, \phi) \wedge (\neg\exists \mathsf{BE}'. \ \mathsf{BE}' \subset \mathsf{BE} \wedge B(T, \phi')) \end{aligned}$$

with $\mathsf{BE}' = \{e' \mid e \in \mathsf{BE}\}$ and the atoms of $\phi' \in \mathsf{BE}'$. When translating operators to set evidence in a given formula, we employ universal quantification and implication to ensure the meaning of the corresponding BFL formula is preserved. For example, consider $\phi[e \mapsto 0]$, where $\phi = \neg e$. According to BFL semantics, $\neg e[e \mapsto 0]$ is *true*, because for all status vectors $\overline{b}$ in which event e is fixed to 0, it holds $\overline{b}, T \models \neg e$. The same holds true in our QBF translation: $\forall e . \neg e \implies \neg e = (1 \implies 1) \wedge (0 \implies 0) =$ *true*. This would not hold, however, if one chooses to translate this BFL formula employing $\wedge$: $\forall e . \neg e \wedge \neg e = (1 \wedge 1) \wedge (0 \wedge 0) =$ *false*. Furthermore, even though the $\underset{\bowtie k}{\mathrm{Vot}}(\phi_1, \ldots, \phi_N)$ operator is syntactic sugar, we take advantage of native support for $\textsc{AtMost}_k$ and $\textsc{AtLeast}_k$ in Z3.

4.3 Translating second layer BFL formulae to QBF

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

Definition 4.3 (Second layer BFL formula to QBF)**.** We let X_2 be the set of all layer two formulae: the translation function of a second layer BFL formula ψ for a tree T is given by the function $B \colon T \times X_2 \to$ (**Q**)**BF** defined recursively as follows:

$$\begin{aligned} B(T, \exists\phi) &= \exists \textsc{FreeVars}(B(T, \phi)) . B(T, \phi) \\ B(T, \forall\phi) &= \forall \textsc{FreeVars}(B(T, \phi)) . B(T, \phi) \\ B(T, \mathrm{IDP}(\phi_1, \phi_2)) &= \bigwedge_{e \in \mathsf{BE}} \big(B(T, \forall\, (\phi_1[e \mapsto 0] \equiv \phi_1[e \mapsto 1]) \\ &\qquad \vee B(T, \forall\, (\phi_2[e \mapsto 0] \equiv \phi_2[e \mapsto 1])) \big) \end{aligned}$$

Quantifiers. The BFL formulae 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 a QBF solver. The result of this query will be $\top$ iff the Boolean formula is satisfiable.

IDP operator. To determine whether two BFL formulae are independent, we first provide a lemma showing how to encode the IDP operator into QBF.

Lemma 4.4 (Encoding of IDP in QBF)**.** *The IDP operator can be translated into QBF as follows:*

$$B(T, \mathrm{IDP}(\phi_1, \phi_2)) := \bigwedge_{e \in \mathsf{BE}} \big(B(T, \forall\, (\phi_1[e \mapsto 0] \equiv \phi_1[e \mapsto 1]) \vee B(T, \forall\, (\phi_2[e \mapsto 0] \equiv \phi_2[e \mapsto 1]))\big)$$

Proof. We first introduce the set of *influencing BEs* [26] defined as follows:

$$\begin{aligned}
\mathrm{IBE}(\phi_1) &:= \{e \in \mathsf{BE} \mid \exists \overline{b}\,.\, \overline{b}, T \models \phi_1[e \mapsto 0] \text{ and } \overline{b}, T \not\models \phi_1[e \mapsto 1] \text{ or vice versa}\} \\
&= \{e \in \mathsf{BE} \mid \exists \overline{b}\,.\, \overline{b}, T \models \big((\phi_1[e \mapsto 0] \wedge \neg\phi_1[e \mapsto 1]) \vee (\neg\phi_1[e \mapsto 0] \wedge \phi_1[e \mapsto 1])\big)\} \\
&= \{e \in \mathsf{BE} \mid \exists \overline{b}\,.\, \overline{b}, T \models (\phi_1[e \mapsto 0] \not\equiv \phi_1[e \mapsto 1])\} \\
&= \{e \in \mathsf{BE} \mid T \models \exists\, (\phi_1[e \mapsto 0] \not\equiv \phi_1[e \mapsto 1])\}
\end{aligned}$$

We use the definition of independence from [26]:

$$\begin{aligned}
\mathrm{IDP}(\phi_1, \phi_2) &:= \mathrm{IBE}(\phi_1) \cap \mathrm{IBE}(\phi_2) = \emptyset \\
&= \forall e \in \mathsf{BE}.\, (e \notin \mathrm{IBE}(\phi_1) \vee e \notin \mathrm{IBE}(\phi_2)) \\
&= \bigwedge_{e \in \mathsf{BE}} (e \notin \mathrm{IBE}(\phi_1) \vee e \notin \mathrm{IBE}(\phi_2))
\end{aligned}$$

The last step follows from having finitely many BEs. Combining this definition with the definition for influencing BEs, we can show the lemma:

$$\begin{aligned}
&B(T, \mathrm{IDP}(\phi_1, \phi_2)) := \\
&\quad \bigwedge_{e \in \mathsf{BE}} \big(B(T, e \notin \mathrm{IBE}(\phi_1)) \vee B(T, e \notin \mathrm{IBE}(\phi_2))\big) \\
&\quad = \bigwedge_{e \in \mathsf{BE}} \big(B(T, \forall\, (\phi_1[e \mapsto 0] \equiv \phi_1[e \mapsto 1]) \\
&\qquad \vee B(T, \forall\, (\phi_2[e \mapsto 0] \equiv \phi_2[e \mapsto 1]))\big) \qquad \square
\end{aligned}$$

Note that in the conjunction, not all BEs have to be considered. Instead it suffices to only consider the BEs occurring in both translated formulae $B(T, \phi_1)$ and $B(T, \phi_2)$ as only these BEs could influence both formulae.

Example 4.5 (A simple translation)**.** According to the proposed translations of layer one and layer two formulae, the following BFL query $\exists(CP/R[IT \mapsto 0, H3 \mapsto 1])$ from Example 2.4 would be translated as follows. Firstly, the FT with root CP/R is translated: $\Psi_T(CP/R) = (IW \wedge H3) \vee (IT \wedge H2)$. Secondly, operators for setting evidence are translated to QBF, resulting in:

$$\forall(IT, H3.(\neg IT \wedge H3 \implies ((IW \wedge H3) \vee (IT \wedge H2)))).$$

Lastly, existential quantification over the free variables is performed, giving the final QBF formula:

$$\exists\big(IW, H2.\forall(IT, H3.(\neg IT \wedge H3 \implies ((IW \wedge H3) \vee (IT \wedge H2))))\big).$$

4.4 Checking BFL queries with QBF solvers

With the knowledge of how to translate BFL formulae to QBF, we can investigate how different queries can be solved. A second layer QBF formula ψ can be checked on tree T using the relationship $T \models \psi$ iff $B(T, \psi)$ is satisfiable.

Satisfaction relation query. To check whether a status vector $\overline{b}$ satisfies a first layer BFL formula ϕ, first, 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 $\Gamma(\overline{s}) = a \wedge \neg b \wedge c$. The satisfaction relation query can then be checked with a QBF solver using the relationship $\overline{b}, T \models \phi$ iff $B(T, \phi) \wedge \Gamma(\overline{b})$ is satisfiable.

Satisfaction set query. Computing all satisfying status vectors can be accomplished by finding all satisfying status vectors. One approach is to block all previously found satisfying assignments, for instance supported by Z3 [13].

Counterexamples. Counterexamples for non-satisfying status vectors can be generated using Algorithm 1. 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 counterexample generation; a different order may result in a different counterexample. In our implementation the order may be different in separate invocations, so a different counterexample may be returned in different runs.

Algorithm 1 Generate counterexample for non-satisfying status vector $\overline{b}$.

Input: Status vector $\overline{b} = \langle b_1, \ldots, b_n\rangle$, Boolean formula f
Output: A counterexample $\overline{c} = \langle c_1, \ldots, c_n\rangle$
Method:
if f is unsatisfiable **then return**
end if
for all $b_i \in \overline{b}$ **do**
  $f_{new} \leftarrow f \wedge \Gamma(\langle b_i\rangle)$
  if f_{new} is satisfiable **then**
    $c_i \leftarrow b_i$
  else
    $c_i \leftarrow \neg b_i$; $f_{new} \leftarrow f \wedge \neg\Gamma(\langle b_i\rangle)$
  end if
  $f \leftarrow f_{new}$
end for
return $\overline{c}$

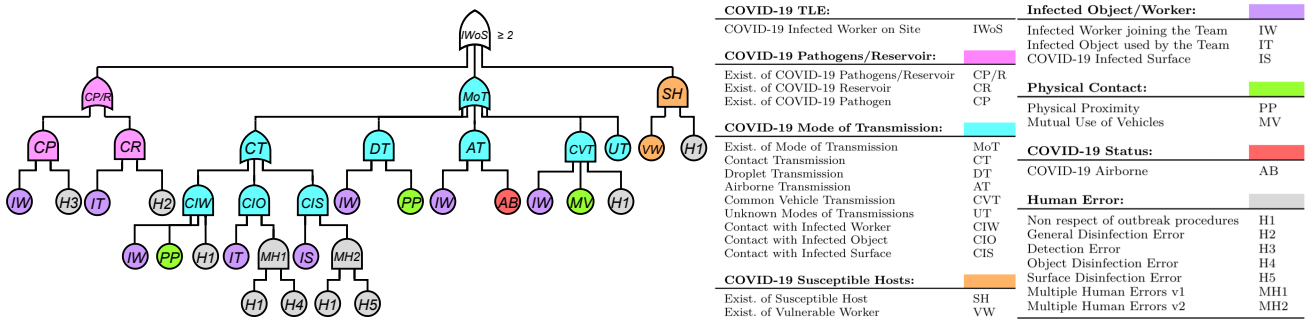

Figure 3. COVID-19 FT from [26].

4.5 Implementation

Our implementation is written in Python and is available online [36]. The tool uses Z3 [16] as QBF solver. The entry point to check BFL queries is the `run_bfl.py` file, that takes a .bfl file as an input. In this file, a FT in the GALILEO format [40] should be provided, alongside BFL queries to run. The given FT and formulae are parsed and then translated following Def. 4.1 to 4.3. Z3's Python API is then used to solve the specified queries. As mentioned, Z3 offers dedicated support for $\textsc{AtMost}_k$ and $\textsc{AtLeast}_k$ and the implementation of the translation proposed in Sect. 4.1 uses a recursive approach. Each function call is stored in a cache to follow dynamic programming standards.

5 Case Study Analysis

We show our QBF translation on the example of the COVID-19 FT from Fig. 3 modelling how a COVID-19 infected worker can manage to be on site. The FT is slightly adjusted from the one in [26]; the AND gate of the TLE is replaced with a $\texttt{VOT}(\geq, 2)$ gate. We consider several BFL queries on this FT, listed in Table 1.

Fault tree. We start by translating the given FT into QBF via the translation in Def. 4.1. The sub-FT with event *MoT* as root can be encoded as follows:

$$\begin{aligned}\Psi_T(MoT) = &\big((IW \wedge PP \wedge H1) \vee (IT \wedge (H1 \wedge H4)) \\ &\vee (IS \wedge (H1 \wedge H5))\big) \\ &\vee (IW \wedge PP) \vee (IW \wedge AB) \\ &\vee (IW \wedge MV \wedge H1) \vee UT\end{aligned}$$

The QBF formula for the complete FT then looks as follows:

$$\begin{aligned}\Psi_T(IWoS) = \textsc{AtLeast}_2\big(&((IW \wedge H3) \vee (IT \wedge H2)), \\ &\Psi_T(MoT), (VW \wedge H1)\big)\end{aligned}$$

Nr.	*BFL* Queries
Q1	$\forall(IS \implies MoT)$
Q2	$\exists(IWoS \wedge \underset{<2}{\text{Vot}}(H1, \ldots, H5))$
Q3	$[\![IWoS \wedge \underset{<2}{\text{Vot}}(H1, \ldots, H5)]\!]$
Q4	$[\![IWoS \wedge \underset{<2}{\text{Vot}}(H1, \ldots, H5) \wedge \text{MCS}(IWoS)]\!]$
Q5	$\exists(CP[IW \mapsto 0])$
Q6	$\exists(CP[IW \mapsto 0, H3 \mapsto 1])$
Q7	$\overline{b}, T \models \neg MoT[UT \mapsto 0]$ with $\overline{b} = \{UT\}$
Q8	$[\![\text{MCS}(CP/R)]\!]$
Q9	$[\![\text{MPS}(CP/R)]\!]$
Q10	$\overline{b}, T \models \text{MCS}(CP/R)$ with $\overline{b} = \{IW, H3, IT\}$

Table 1. Queries on COVID-19 FT.

Q1 - Universal quantification. Let us start with the BFL query $\forall(IS \implies MoT)$ which checks whether an infected surface is sufficient for the transmission of COVID. The translation via Def. 4.3 yields the following QBF formula:

$$\forall IS, IW, PP, H1, IT, H4, H5, AB, MV, UT.\ IS \implies \Psi_T(IWoS)$$

This formula is unsatisfiable and thus evaluates to false.

Q2 - 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: $\exists(IWoS \wedge \underset{<2}{\text{Vot}}(H1, \ldots, H5))$. The QBF encoding is as follows:

$$\begin{aligned}&\exists IW, H3, IT, H2, PP, H1, H4, IS, H5, AB, MV, UT, VW.\\ &\quad \Psi_T(IWoS) \wedge \textsc{AtMost}_1(H1, H2, H3, H4, H5)\end{aligned}$$

The formula is satisfiable which means the TLE could occur with at most 1 human error.

Q3 & Q4 - Satisfaction sets. If we consider the following query $[\![IWoS \wedge \underset{<2}{\text{Vot}}(H1, \ldots, H5)]\!]$ 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: $[\![IWoS \wedge \underset{<2}{\text{Vot}}(H1, \ldots, H5) \wedge \text{MCS}(IWoS)]\!]$ (simplified: $[\![\text{MCS}(IWoS) \wedge \underset{<2}{\text{Vot}}(H1, \ldots, H5)]\!]$). The following minimal cut sets are returned: $\{IT, UT, H2\}$, $\{H3, IW, PP\}$, $\{IT, IW, AB, H2\}$, $\{VW, PP, IW, H1\}$, $\{H3, IW, UT\}$,

$\{VW, IW, MV, H1\}, \{VW, IW, AB, H1\}, \{H3, IW, AB\}$, $\{VW, UT, H1\}, \{IT, IW, H2, PP\}$.

Q5 & Q6 - Setting evidence and quantification. Let us take a look at the fifth BFL query $\exists(CP[IW \mapsto 0])$, where we set the constraint of IW remaining operational. The resulting QBF formula is $\exists H3.\ \forall IW.\ \neg IW \implies IW \wedge H3$. 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 sixth query $\exists(CP[IW \mapsto 0, H3 \mapsto 1])$ is

$$\forall IW, H4.\ \neg IW \wedge H3 \implies IW \wedge H3,$$

having no existential quantifier at all. Obviously, both BFL queries return false.

Q7 - Satisfaction checking with evidence. Consider the 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 [26] as we do prioritize the representation of a what-if scenario and because multiple evidences can be set at different depth of formulae nesting. This means we expect the result to be true. The query translates to

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

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.

Q8 - Minimal cut sets. To find the minimal cut sets of CP/R, we use the BFL query $[\![\mathrm{MCS}(CP/R)]\!]$, which translates to:

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

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: $\exists H1.\ \forall VW.\ VW \implies VW \wedge H1$

Q9 - Minimal path sets. Similarly to the previous example, we can get the MPSs with the formula $[\![\mathrm{MPS}(CP/R)]\!]$. The MPS operator is syntactic sugar for the MCS operator. Thus we can obtain the desired result via $[\![\mathrm{MPS}(CP/R)]\!] = [\![\mathrm{MCS}(\neg CP/R)]\!]$. Translating the latter formula and checking it gives the following MPSs as result: $\{IT, IW\}$, $\{IT, H3\}$, $\{IW, H2\}$, and $\{H3, H2\}$.

Q10 - Counterexamples. In the eight query, we saw that the MCSs of CP/R are $\{IT, H2\}$ and $\{H3, IW\}$. We now try to check a slightly wrong status vector and see what counterexamples it generates. We take $\overline{b} = \langle 1, 1, 1, 0\rangle$ for $IW, H3, IT, H2$ and compute a counterexample for $\overline{b}, T \models \mathrm{MCS}(CP/R)$. Algorithm 1 will either return $\{IW, H3\}$ or $\{IT, H2\}$, based on the order of the `for` loop.

Let us consider the order (1) $IW, H3, IT, H2$. 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) $IT, H3, IW, H2$. 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 $\{IT, H2\}$.

6 Experiments

This section discusses our implementation w.r.t. *runtime*, *memory consumption* and *scalability*.

Artifact. We provide our Python tools — including the ones described in Sect. 4.5 — as well as all example FTs and BFL queries from Sects. 5 and 6 in an artifact [36]. In particular, in the `examples` directory of the artifact, the file `case-study-1.bfl` provides the FT from Fig. 3, along with all the BFL queries used in the case study in Sect. 5. The `run_experiments.sh` file provides an entry point to run the experiments of this section, building on top of two other Python scripts, `experiments_bfl.py` and `plots.py`.

All experiments are run on a PC with Intel® Core™ i7-9750H CPU @ 2.60GHz × 12 and 16GB of RAM and Python version 3.10. We devise two experiments:

1. the *single-FT* experiment, that investigates the runtime and memory consumption for ten different properties on the same FT, and

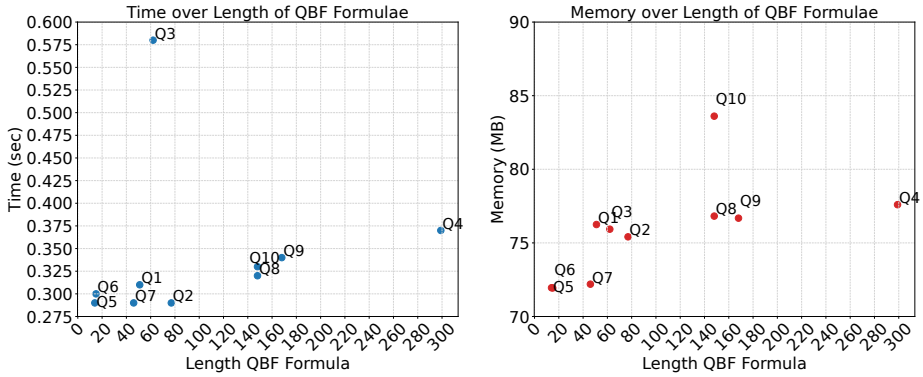

Figure 4. *Single-FT* experiment: runtime and memory for queries Q1, ... , Q10 from Table 1 for FT in Fig. 3.

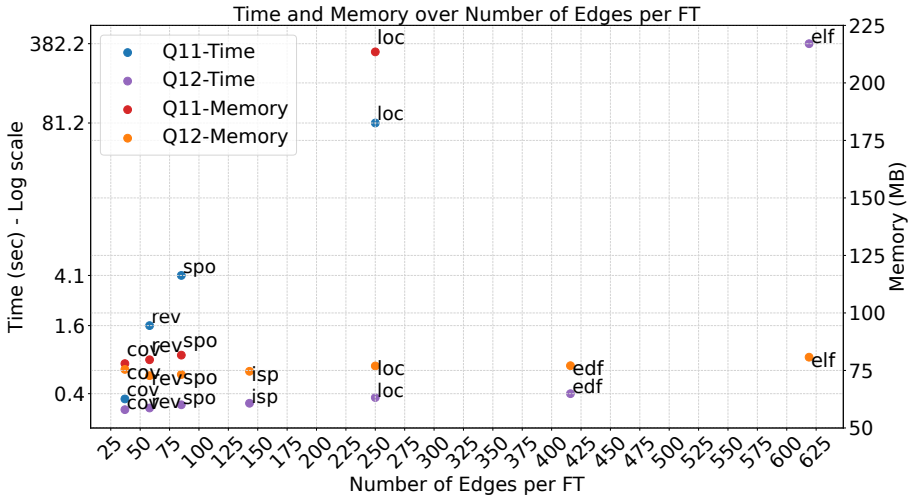

Figure 5. *Multi-FT* experiment: runtime and memory for queries Q11 & Q12 for FTs in Table 2.

2. the *multi-FT* experiment, that investigates the runtime and memory consumption for two different properties on a range of different-sized FTs.

Single-FT. We measure runtime and memory consumption for the case study of Sect. 5. We use the COVID-19 FT from Fig. 3 and check the ten different *BFL* queries Q1 - Q10 from Table 1. Fig. 4 shows two plots for the runtime (left) and memory consumption (right) over the length of each resulting QBF formula (number of operators + number of atoms).

Multi-FT. We utilize seven different FTs: the COVID FT from Sect. 5 and six exemplary industrial FTs from the benchmark set of [6], given in an artifact [7]. The latter FTs come from industrial models for components of a lock used in water navigation (`rev`, `spo`, `loc`) and from the the Aralia benchmark set[1]. Table 2 gives an overview of the considered FTs including their number of nodes and edges. For each FT, we check the same two *BFL* queries:

[1] https://github.com/rakhimov/scram/tree/develop/input/Aralia

Name	Source	FT Nodes	FT Edges
`cov`	[5, 26]	28	37
`rev`	[6]	57	58
`spo`	[6]	86	85
`loc`	[6]	251	250
`isp`	[33]	130	143
`edf`	[33]	198	416
`elf`	[33]	387	619

Table 2. Benchmarked FTs.

Q11: $[\![\mathrm{MCS}(TLE) \wedge \mathrm{Vot}_{<2}(BE_1, \ldots, BE_5)]\!]$, and
Q12: $\exists(TLE[BE_1 \mapsto 0, BE_2 \mapsto 1])$.

The BEs are selected such that they are not part of the same subtree, whenever possible. Fig. 5 presents results from this experiment. On the x axis we have the number of edges for each FT, while on the y axis we have the runtime (left, in log scale) and the memory consumption (right).

Discussion. The *Single-FT* experiment suggests a linear correlation between the length of the QBF formula and the time to check it: in Fig. 4 we see that nearly all the points agree with this interpretation. A clear outlier is represented by property number 3. This is not surprising, since we require to compute the satisfaction set for the TLE failure, then filtered by the failure of at most one other BE. This is equivalent to computing all the cut sets for the given FT without requiring minimality, which is an operation know to be computationally challenging. In fact, checking the same property for the slightly larger tree `rev` did not return any result within two hours of computation. Fig. 4 also suggests a linear increase in memory consumption w.r.t. the length of the given QBF formulae. The increase seems to be manageable in the present case, as all of the values lie within a 15MB window.

The *Multi-FT* experiment presents two main findings w.r.t. scalability: 1. the type of property that one specifies has a great impact on runtime and memory consumption; 2. the structure of the FT has more impact on runtime and memory then the number of FT elements. The first point is quite evident when comparing runtime and memory consumption between the chosen two queries: while checking property one results in resource consumption quickly becoming exponential, checking property two on increasingly larger FTs remains manageable time- and memory-wise. In fact, runtime for query two remains below or at 0.4 seconds for 6 out of 7 FTs, with a peak at 382.2 seconds for the FT with most edges in our benchmark. The same is true for memory consumption, that remains comfortably around 75MB for all the tested FTs. Checking query one, however, quickly becomes more resource intensive: both runtime and memory consumption seem to grow exponentially, although more data points would be needed to validate this impression, especially on memory consumption. Furthermore, some data points for memory consumption and runtime of query one are missing: in fact, for `isp`, `edf`, and `elf` we could not perform a check within the two hours timeframe. This suggests that the structure of the FT has more impact on memory and runtime than the number of elements composing it, as highlighted in our second main finding. Even the more taxing query one could be checked successfully on the `loc` FT, despite it having more nodes than `isp`. Upon closer inspection, `isp`, `edf`, and `elf` reveal a more complex structure than other FTs in the benchmark: elements are highly connected, as revealed by the increasingly larger number of edges over the number of elements, and present a higher number of AND-gates when compared with `rev`, `spo` and `loc`, that are mainly composed by OR-gates. This would probably yeld a higher number of MCSs with an increased number of elements in them.

7 Discussion and Future Work

Discussion. When comparing our current approach with an hypothetical implementation based on BDDs, we can at least propose a qualitative reflection. For example, solving many BFL queries can be done leveraging properties that are inherent to BDDs to return a result in constant time. In fact, 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 formulae can easily be checked by testing if the BDDs for those formulae share any variables. The two approaches' performance on quantified queries might then compare fairly, 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.

Future work. As of now, it is impossible to further evaluate what BFL properties contribute to easier problem-solving for both approaches. The very next contribution would then be to develop an implementation based on BDDs to perform a concrete evaluation. Using randomized automatic generation of FTs and BFL properties would be useful to cross-compare both the BDD- and QBF-based approaches on their performance. We could further investigate a restricted translation that only takes a fragment of BFL without quantifiers and evaluate the complexity of the decision problem for such a fragment. Next to the type of BFL query, the structure of the FT may also influence the performance of the two mentioned approaches in different ways, as we preliminarly showed for the QBF approach in Sect. 6. We set out to further test these findings on the BDD implementation as well. A further point of interest would investigate the use of multithreading in both implementations. Finally, useful insights might be provided by conducting BFL user studies.

8 Conclusion

This paper presented the first-ever implementation of BFL, allowing 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 and tested through experiments. Finally, we briefly discussed the qualitative differences between 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.

References

[1] Parosh Aziz Abdulla, Per Bjesse, and Niklas Eén. 2000. Symbolic reachability analysis based on SAT-solvers. In *TACAS* (Lecture Notes in Computer Science). Vol. 1785. Springer, 411–425. DOI: 10.1007/3-540-46419-0_28.

[2] Moayad Fahim Ali, Sean Safarpour, Andreas G. Veneris, Magdy S. Abadir, and Rolf Drechsler. 2005. Post-verification debugging of hierarchical designs. In *ICCAD*. IEEE Computer Society, 871–876. DOI: 10.1109/ICCAD.2005.1560184.

[3] Carlos Ansótegui, Carla P. Gomes, and Bart Selman. 2005. The achilles' heel of QBF. In *AAAI*. AAAI Press / The MIT Press, 275–281.

[4] Ofer Arieli and Martin W. A. Caminada. 2013. A qbf-based formalization of abstract argumentation semantics. *J. Appl. Log.*, 11, 2, 229–252. DOI: 10.1016/j.jal.2013.03.009.

[5] Tarik Bakeli, Adil Alaoui Hafidi, et al. 2020. Covid-19 infection risk management during construction activities: an approach based on fault tree analysis (fta). *Journal of Emergency Management*, 18, 7, 161–176.

[6] Daniel Basgöze, Matthias Volk, Joost-Pieter Katoen, Shahid Khan, and Mariëlle Stoelinga. 2022. BDDs strike back - efficient analysis of static and dynamic fault trees. In *NFM* (Lecture Notes in Computer Science). Vol. 13260. Springer, 713–732. DOI: 10.1007/978-3-031-06773-0_38.

[7] Daniel Basgöze, Matthias Volk, Joost-Pieter Katoen, Shahid Khan, and Marielle Stoelinga. Artifact for "BDDs Strike Back - Efficient Analysis of Static and Dynamic Fault Trees". Version 1.0.1. Zenodo, (Mar. 2022). DOI: 10.5281/zenodo.6390998.

[8] Padmalochan Bera, Pallab Dasgupta, and SK Ghosh. 2009. A verification framework for analyzing security implementations in an enterprise LAN. In *2009 IEEE International Advance Computing Conference*. IEEE, 1008–1015.

[9] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Masahiro Fujita, and Yunshan Zhu. 1999. Symbolic model checking using SAT procedures instead of BDDs. In *DAC*. ACM Press, 317–320. DOI: 10.1145/309847.309942.

[10] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. 1999. Symbolic model checking without BDDs. In *TACAS* (Lecture Notes in Computer Science). Vol. 1579. Springer, 193–207. DOI: 10.1007/3-540-49059-0_14.

[11] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, eds. 2021. *Handbook of Satisfiability - Second Edition. Frontiers in Artificial Intelligence and Applications*. Vol. 336. IOS Press. DOI: 10.3233/FAIA336.

[12] Armin Biere and Daniel Kröning. 2018. Sat-based model checking. In *Handbook of Model Checking*. Springer, 277–303. DOI: 10.1007/978-3-319-10575-8_10.

[13] Nikolaj S. Bjørner, Leonardo de Moura, Lev Nachmanson, and Christoph M. Wintersteiger. 2018. Programming Z3. In *SETSS* (Lecture Notes in Computer Science). Vol. 11430. Springer, 148–201. DOI: 10.1007/978-3-030-17601-3_4.

[14] Edmund M. Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. 2001. Bounded model checking using satisfiability solving. *Formal Methods Syst. Des.*, 19, 1, 7–34. DOI: 10.1023/A:1011276507260.

[15] Olivier Coudert and Jean Christophe Madre. 1993. Fault tree analysis: 10^{20} prime implicants and beyond. In *Annual Reliability and Maintainability Symposium*. IEEE, 240–245.

[16] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. 2008. Z3: an efficient SMT solver. In *TACAS* (Lecture Notes in Computer Science). Vol. 4963. Springer, 337–340. DOI: 10.1007/978-3-540-78800-3_24.

[17] Nachum Dershowitz, Ziyad Hanna, and Jacob Katz. 2005. Bounded model checking with QBF. In *SAT* (Lecture Notes in Computer Science). Vol. 3569. Springer, 408–414. DOI: 10.1007/11499107_32.

[18] Diptarama, Ryo Yoshinaka, and Ayumi Shinohara. 2016. QBF encoding of generalized tic-tac-toe. In *QBF@SAT* (CEUR Workshop Proceedings). Vol. 1719. CEUR-WS.org, 14–26.

[19] Uwe Egly, Thomas Eiter, Hans Tompits, and Stefan Woltran. 2000. Solving advanced reasoning tasks using quantified Boolean formulas. In *AAAI/IAAI*. AAAI Press / The MIT Press, 417–422.

[20] Uwe Egly and Stefan Woltran. 2006. Reasoning in argumentation frameworks using quantified Boolean formulas. In *COMMA* (Frontiers in Artificial Intelligence and Applications). Vol. 144. IOS Press, 133–144.

[21] Peter Faymonville, Bernd Finkbeiner, Markus N. Rabe, and Leander Tentrup. 2017. Encodings of bounded synthesis. In *TACAS (1)* (Lecture Notes in Computer Science). Vol. 10205, 354–370. DOI: 10.1007/978-3-662-54577-5_20.

[22] 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.

[23] Evguenii I. Goldberg, Mukul R. Prasad, and Robert K. Brayton. 2001. Using SAT for combinational equivalence checking. In *DATE*. IEEE Computer Society, 114–121. DOI: 10.1109/DATE.2001.915010.

[24] Elisabeth Jöbstl, Martin Weiglhofer, Bernhard K. Aichernig, and Franz Wotawa. 2010. When BDDs fail: conformance testing with symbolic execution and SMT solving. In *ICST*. IEEE Computer Society, 479–488. DOI: 10.1109/ICST.2010.48.

[25] Hratch Mangassarian, Andreas G. Veneris, and Marco Benedetti. 2010. Robust QBF encodings for sequential circuits with applications to verification, debug, and test. *IEEE Trans. Computers*, 59, 7, 981–994. DOI: 10.1109/TC.2010.74.

[26] Stefano M. Nicoletti, Ernst Moritz Hahn, and Mariëlle Stoelinga. 2022. BFL: a logic to reason about fault trees. In *DSN*. IEEE, 441–452. DOI: 10.1109/DSN53405.2022.00051.

[27] Johannes Oetsch, Martina Seidl, Hans Tompits, and Stefan Woltran. 2009. ccT on stage: generalised uniform equivalence testing for verifying student assignment solutions. In *LPNMR* (Lecture Notes in Computer Science). Vol. 5753. Springer, 382–395. DOI: 10.1007/978-3-642-04238-6_32.

[28] Johannes Oetsch, Martina Seidl, Hans Tompits, and Stefan Woltran. 2006. ccT: A correspondence-checking tool for logic programs under the answer-set semantics. In *JELIA* (Lecture Notes in Computer Science). Vol. 4160. Springer, 502–505. DOI: 10.1007/11853886_47.

[29] Charles Otwell, Anja Remshagen, and Klaus Truemper. 2004. An effective QBF solver for planning problems. In *MSV/AMCS*. CSREA Press, 311–316.

[30] 2014. QCIR-G14: A Non-Prenex Non-CNF Format for Quantified Boolean Formulas. (2014). Retrieved Mar. 5, 2023 from https://www.qbflib.org/qcir.pdf.

[31] 2005. QDIMACS standard. (2005). Retrieved May 3, 2023 from https://www.qbflib.org/qdimacs.html.

[32] Markus N. Rabe and Leander Tentrup. 2015. CAQE: A certifying QBF solver. In *FMCAD*. IEEE, 136–143. DOI: 10.1109/FMCAD.2015.7542263.

[33] Olzhas Rakhimov. Aralia benchmark set. https://github.com/rakhimov/scram/tree/develop/input/Aralia.

[34] Antoine Rauzy. 1993. New algorithms for fault trees analysis. *Reliab. Eng. Syst. Saf.*, 40, 3, 203–211.

[35] Enno Ruijters and Mariëlle Stoelinga. 2015. Fault tree analysis: A survey of the state-of-the-art in modeling, analysis and tools. *Comput. Sci. Rev.*, 15, 29–62. DOI: 10.1016/j.cosrev.2015.03.001.

[36] [SW] Caz Saaltink, Stefano M. Nicoletti, Matthias Volk, E. Moritz Hahn, and Mariëlle Stoelinga, Artifact for the paper Solving Queries for Boolean Fault Tree Logic via Quantified SAT July 2023. DOI: 10.5281/zenodo.8172549, URL: https://doi.org/10.5281/zenodo.8172548.

[37] Ankit Shukla, Armin Biere, Luca Pulina, and Martina Seidl. 2019. A survey on applications of quantified Boolean formulas. In *ICTAI*. IEEE, 78–84. DOI: 10.1109/ICTAI.2019.00020.

[38] Roslyn M Sinnamon and John D Andrews. 1996. Fault tree analysis and binary decision diagrams. In *Annual Reliability and Maintainability Symposium*. IEEE, 215–222.

[39] Carsten Sinz. 2005. Towards an optimal CNF encoding of Boolean cardinality constraints. In *CP* (Lecture Notes in Computer Science). Vol. 3709. Springer, 827–831. DOI: 10.1007/11564751_73.

[40] Kevin J Sullivan and Joanne B Dugan. 1998. Galileo user's manual & design overview. (1998).

[41] Leander Tentrup. 2016. Non-prenex QBF solving using abstraction. In *SAT* (Lecture Notes in Computer Science). Vol. 9710. Springer, 393–401. DOI: 10.1007/978-3-319-40970-2_24.

[42] W. E. Vesely, F. F. Goldberg, N. H. Roberts, and D. F. Haasl. 1987. Fault Tree Handbook.

[43] Robert Wille, Daniel Große, Finn Haedicke, and Rolf Drechsler. 2009. SMT-based stimuli generation in the SystemC verification library. In *FDL*. IEEE, 1–6.

Received 2023-07-21; accepted 2023-08-27