Finding Locally Smallest Cut Sets using Max-SMT

Daniel Larraz, Cesare Tinelli
The University of Iowa
USA

Abstract

Model-based development (MBD) is increasingly being used for system-level development of safety-critical systems. This approach allows safety engineers to leverage the system model created during the MBD process to assess the system's resilience to component failure. In particular, one fundamental activity is the identification of minimal cut sets (MCSs), i.e, minimal sets of faults that lead to the violation of a safety requirement. Although the construction of a formal system model enables safety engineers to automate the generation of MCSs, this is usually a computationally expensive task for complex enough systems. We present a method that leverages Max-SMT solvers to efficiently obtain a small set of faults based on a local optimization of the cut set cardinality. Initial experimental results show the effectiveness of the method in generating cut sets that are close or equal to globally optimal solutions (smallest cut sets) while providing an answer 5.6 times faster on average than the standard method to find a smallest cut set.

Keywords: Safety Analysis, Minimal Cut Set, SMT-based Model Checking, Max-SMT

1 Introduction

Safety analysis is a crucial and well established activity in the design of critical systems that is often mandated by certification regulations. It aims at proving that a given system operates within some level of safety in the presence of faults. Traditionally, safety analysis has been performed manually based on informal design models, making the analysis highly subjective and dependent on the skill of the practitioner. However, in recent years there has been a growing interest in Model-based Safety Analysis (MBSA) [9]. This is an approach in which the design and safety engineers share a common system model created using a Model-based development (MBD) process. In MBD, the development is centered around a formal specification, or model, of the system. This model can then be subject to various kinds of rigorous analysis and synthesis such as completeness and consistency analysis, model checking, test case generation, etc. MBSA uses the system model to assess the system's resilience to component failure, and construct safety analysis artifacts such as minimal cut sets and fault trees.

In this context, a *minimal cut set (MCS)* is a minimal set of faults, a.k.a *basic events*, that lead to the violation of a safety requirement or some other failure, the so called *top level event (TLE)*. These sets of faults, or *fault configurations*, can be arranged in a *fault tree*, a tree making use of logical gates to depict the logical interrelationships linking such events with the TLE. Finding cut sets is important to assess the fault tolerance level of a system design, and investigate how failures propagate through the system.

In this paper, we focus on the safety analysis of behavioral models of infinite-state reactive systems. In this setting, safety requirements are (LTL) regular safety properties of the intended system model, which can always be recast as invariant properties. A system model consists of a *nominal* model, which specifies the behavior of the system in the absence of faults, and a set of faulty behaviors, which augment the nominal behavior whenever their corresponding faults are present. Thus, we consider the problem of (dis)proving safety properties in the presence of faults, and computing minimal cut sets, if any, for the violation of a safety property.

Typically, system models can be faithfully encoded as logical formulas. For such systems the problem above can be addressed with logic-based model checking techniques, such as k-induction [14] and IC3 [5], that capitalize on the power of solvers for satisfiability modulo theories (SMT). When these model checking techniques disprove a safety property under failure conditions, they also produce a counterexample demonstrating how faults lead to a failure. These counterexamples can be used to reason about the evolution of faults over time, and extract a fault configuration, although a non-necessarily minimal one.

We work under the *monotonicity assumption*, commonly adopted in safety analysis, that additional faults cannot prevent the violation of an already violated safety property. Under this assumption, a minimal cut set for a property is preferable to a super-set of it, since the latter will still cause the property to fail. Moreover, smaller MCSs are preferable over larger ones, since, in practical cases, the smaller a MCS the greater the probability that the safety property can be violated. This leads to the standard practice, in particular for complex systems, of computing only MCSs up to a maximum cardinality. However, that may still be computationally expensive and therefore its application may be pushed only to

HILT'22, October 2022, Oakland, Michigan, USA

late phases of the development process. As a consequence, to help safety engineers with early detection of design issues it is fundamental to be able to resort to more efficient methods for computing small cut sets.

The key observation of this work is that *ensuring* minimality or minimal cardinality with respect to *all* fault configurations and counterexamples that lead to the violation of a safety property is not always required for the early detection of problems in a system design. By definition, a safety property is one that fails to hold if and only if it is violated by a *finite counterexample*, i.e., a finite execution of the system. For this reason, it is often sufficient to compute a cut set with minimal cardinality over all counterexamples of a *given* length n. This often results in a small cut set that is close or equal to a globally optimal solution (a smallest MCS) and, as such, is enough to point to flaws in the system design.

To illustrate this point and the other concepts introduced so far, we will use a simple example of an aircraft controller derived from previous work [11]. The example is introduced in Section 3, but first, in the next section, we give a brief description of the notions and notations that will be used throughout the paper. The rest of the paper is organized as follows. Section 4 describes how to encode faulty behaviors into a nominal system. Section 5 presents the base method to compute a (not necessarily minimal) cut set using a faulty model. Section 6 describes how to compute a cut set with minimal cardinality. Section 7 presents the method we propose to obtain efficiently a small cut set based on a local optimization of the cut set cardinality. Experimental results comparing both approaches are reported in Section 8. Section 9 presents related work, and Section 10 concludes with a discussion of further research.

2 Preliminaries

2.1 SAT, Max-SAT, and Max-SMT

Let $\mathcal{P}$ be a finite set of *propositional variables*. If $p \in \mathcal{P}$ then p and $\neg p$ are *literals*. A *clause* is a disjunction of literals. A *propositional formula (in conjunctive normal form)* is a conjunction of clauses. The problem of *propositional satisfiability (or SAT)* consists in determining whether or not a given formula is *satisfiable*, or has a *model*: an assignment of truth values to its variables that makes it true.

A generalization of SAT is the *satisfiability modulo theories* (SMT) problem [3], which consists in deciding the satisfiability of a given (typically) quantifier-free first-order formula with respect to a background theory $\mathcal{T}$. In this setting, a model (which we may also refer to as a solution) is an assignment of values from the theory to the formula's variables that satisfies the formula and interprets function and predicate symbols consistently with the axioms of $\mathcal{T}$. Here we will consider the theories of *linear real/integer arithmetic (LRA/LIA)*, where literals are linear inequalities over real and integer variables, respectively.

Another generalization of SAT is the *Max-SAT* problem [3] which considers formulas in conjunctive normal form where each conjunct or *clause* is labeled as a *hard* or a *soft* constraints and each soft constraint is assigned a positive weight. The problem consists in finding an assignment of truth values for the formula's variable that satisfies all the hard constraints and maximizes the sum of the weights of the satisfied soft constraints, or dually, that minimizes the sum of the weights of the soft constraints it falsifies.

Max-SMT [13] is the natural extension of the Max-SAT problem to SMT where formulas can contain variables over additional data types other than the Booleans, and so the sought maximizing assignments are over such variables as well.

In general, if F is a formula and $\boldsymbol{x}$ is a tuple of variables, we write $F[\boldsymbol{x}]$ to indicate that the elements of $\boldsymbol{x}$ are free in F. If then $\boldsymbol{t}$ is a tuple of terms of the same type as $\boldsymbol{x}$, we denote by $F[\boldsymbol{t}]$ the formula obtained from F by simultaneously replacing every occurrence of a variable from $\boldsymbol{x}$ by the corresponding term in $\boldsymbol{t}$.

2.2 Transition Systems, Invariants, and LTL specifications

We represent a system model as a state transition systems $S = \langle \boldsymbol{s}, I[\boldsymbol{s}], T[\boldsymbol{s}, \boldsymbol{s'}] \rangle$ where $\boldsymbol{s}$ is a vector of typed state variables, I is the initial state predicate over the variables $\boldsymbol{s}$, and T is a two-state transition predicate over the variables $\boldsymbol{s}$ and $\boldsymbol{s'}$ where $\boldsymbol{s'}$ is a renamed version of $\boldsymbol{s}$ denoting the next state. We will use $\langle I, T \rangle$ to refer to transition system S when the vector of state variables $\boldsymbol{s}$ is clear from the context or not important. We will assume without loss of generality that T has the structure of a top-level conjunction, that is, $T[\boldsymbol{s}, \boldsymbol{s'}] = T_1[\boldsymbol{s}, \boldsymbol{s'}] \wedge \cdots \wedge T_n[\boldsymbol{s}, \boldsymbol{s'}]$ for some $n \geq 1$. Notice that this is the norm in specification languages, like Lustre [8], where the modeled system is expressed as the synchronous product of several subcomponents, each of which is in turn formalized as the conjunction of one or more constraints. The conjunctive formulation is also common in languages that express the transition relation as a set of guarded transitions. By a slight abuse of notation, we will then identify T with the set $\{T_1, \ldots, T_n\}$ of its top-level conjuncts.

A *state property* $P[\boldsymbol{s}]$ for a system $S = \langle \boldsymbol{s}, I[\boldsymbol{s}], T[\boldsymbol{s}, \boldsymbol{s'}] \rangle$, expressed as a predicate over the variables $\boldsymbol{s}$, is *invariant for* S if it holds in every reachable state of S.

We use standard notions and notation from Linear Temporal Logic (LTL) to formalize temporal properties of transition systems.

2.3 Bounded Model Checking

Bounded Model Checking (BMC) [2] is a method involving checking potential executions of a system model in an

incremental fashion against the negation of a state property by encoding them as propositional satisfiability formulas. Although BMC was originally developed for propositional encodings of finite-state systems, the technique has been successfully extended and applied to SMT encodings of (in)finite-state systems. Given a transition system $S = \langle \boldsymbol{s}, I[\boldsymbol{s}], T[\boldsymbol{s}, \boldsymbol{s}'] \rangle$, a state property P, and a bound k, BMC unrolls the system k times to produce a SMT formula φ_k such that φ_k is satisfiable iff P has a counterexample of length k or less:

$$\varphi_k = I[\boldsymbol{s}_0] \wedge \bigvee_{i=0}^{k} \bigwedge_{j=0}^{i-1} (T[\boldsymbol{s}_j, \boldsymbol{s}_{j+1}] \wedge \neg P[\boldsymbol{s}_i])$$

where $\boldsymbol{s}_0, \ldots, \boldsymbol{s}_k$ are each a fresh renaming of $\boldsymbol{s}$. Formula φ_k is given to an SMT solver to be checked for satisfiability. If it is satisfiable, then the SMT solver will provide an assignment that satisfies φ_k. With this assignment, the counterexample is constructed using the values extracted from variables $\boldsymbol{s}_i$. If φ_k is unsatisfiable, that means no state is reachable in k steps or less such that the state violates property P.

We will use BMC_Encoding(I, T, P, k) to denote a call to a function that returns the formula φ_k given as input a transition system $S = \langle I, T \rangle$, a state property P, and a bound k.

3 Motivating Example

Suppose we want to design a component for an airplane that controls the pitch motion of the aircraft, and suppose one of the system safety requirements is that the aircraft should not ascend beyond a certain altitude. The controller must read the current altitude of the aircraft from a sensor, and modify the next position of the aircraft's nose accordingly. Moreover, we want the system to be fault-tolerant to sensor failures. One way to improve system fault-tolerance is to introduce some redundancy. In particular, we can equip the system with three different altimeters so the controller receives three independent altitude values. Then the controller, with the help of a dedicated component, a *triplex voter*, takes the average of the two altitude values that are closest to each other — as they are more likely to be close to the actual altitude. Following a model-based design, we model an abstraction of the system's environment to which the aircraft's controller will react. We also model the fact that the system relies on a possibly imperfect reading of the current altitude by an altimeter sensor to decide the next pitch value. Finally, we provide a specification for the controller's behavior so that it satisfies the system requirement of interest.

A diagram of our model is shown in Figure 1. The main component, represented by the outermost rectangle, is an *observer* component that represents the full system consisting in this case of just three subcomponents, for simplicity: one component modeling the controller, one modeling a triplex voter, and another one modeling the environment. The observer has three inputs: alt_1, alt_2, and alt_3, representing the altitude values from each altimeter, and an output alt, representing the actual current altitude of the aircraft, which we are modeling as a product of the environment in response to the pitch value generated by the controller.

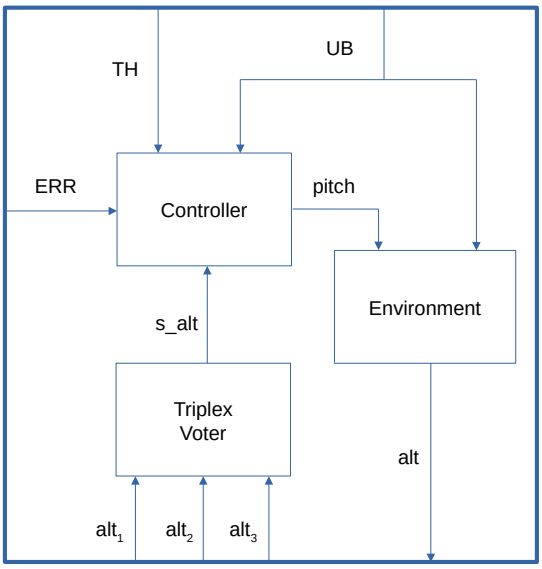

Figure 1. Diagram of the System Model

The system model makes a series of assumptions on the altitude values provided by the sensors and on a number of symbolic numeric constants (TH, UB and ERR) which act in effect as model parameters. Constant TH represents a threshold of the altitude value, constant UB models an upper bound on the change in altitude from one execution step to the next, and constant ERR is a bound on the sensor measurement error (more details below). The first assumption, $C1$, establishes that constants TH and UB are positive, and constant ERR is non-negative. The three next assumptions, $S1$, $S2$, and $S3$ account for the fact that, while the altitude value produced by each altimeter is not 100% accurate in actual settings, its error is bounded by a constant (ERR). That is, the system assumes the satisfaction of LTL formulas $S_i \equiv \Box(\mathsf{alt} - \bigcirc \mathsf{alt}_i \leq \mathsf{ERR})$ for $1 \leq i \leq 3$. Under those assumptions, the system must satisfy the LTL property $R1 \equiv \Box(\mathsf{alt} \leq \mathsf{TH})$, that formalizes the requirement that aircraft maintain its altitude below a certain threshold TH at all times.

Let $M = \min(|\mathsf{alt}_1 - \mathsf{alt}_2|, |\mathsf{alt}_1 - \mathsf{alt}_3|, |\mathsf{alt}_2 - \mathsf{alt}_3|)$. As explained above, triplex voter takes the sensor values and computes an estimated altitude for the controller satisfying the following specification:

$$\mathsf{s\_alt} = \begin{cases} (\mathsf{alt}_1 + \mathsf{alt}_2)/2, & \text{if } M = |\mathsf{alt}_1 - \mathsf{alt}_2| \\ (\mathsf{alt}_1 + \mathsf{alt}_3)/2, & \text{if } M = |\mathsf{alt}_1 - \mathsf{alt}_3| \\ (\mathsf{alt}_2 + \mathsf{alt}_3)/2, & \text{if } M = |\mathsf{alt}_2 - \mathsf{alt}_3| \end{cases}$$

We abstract the dynamics of the Controller and the Environment by omitting details that are not relevant for the satisfaction of the safety requirement $R1$. In the Controller's case, we model the guarantee that the controller will produce a negative pitch value whenever the sensor altitude indicates that the aircraft is getting *too close* to the threshold value TH, by which we mean that the difference between the current altitude and TH is smaller than UB + ERR:

$$L1 \equiv \Box(\mathsf{s\_alt} > \mathsf{LIMIT} \Rightarrow \mathsf{pitch} < 0)$$

with LIMIT = TH − (UB + ERR).

If alt represents the actual altitude of the aircraft, the Environment satisfies the following specification:

- $E1 \equiv \mathsf{alt} = 0$
- $E2 \equiv \Box(\mathsf{alt} \geq 0)$
- $E3 \equiv \Box(\bigcirc\mathsf{pitch} < 0 \Rightarrow \bigcirc\mathsf{alt} \leq \mathsf{alt})$
- $E4 \equiv \Box(\bigcirc\mathsf{pitch} < 0 \Rightarrow \bigcirc\mathsf{alt} \geq \mathsf{alt} - \mathsf{UB})$
- $E5 \equiv \Box(\bigcirc\mathsf{pitch} > 0 \Rightarrow \bigcirc\mathsf{alt} \geq \mathsf{alt})$
- $E6 \equiv \Box(\bigcirc\mathsf{pitch} > 0 \Rightarrow \bigcirc\mathsf{alt} \leq \mathsf{alt} + \mathsf{UB})$
- $E7 \equiv \Box(\bigcirc\mathsf{pitch} = 0 \Rightarrow \bigcirc\mathsf{alt} = \mathsf{alt})$

The specification captures salient constraints on the physics of our model by specifying that a positive pitch value (which has the effect of raising the nose of the aircraft and lowering its tail) makes the aircraft ascend, a negative value makes it descend, and a zero value keeps it at the same altitude. The specification also states that the actual altitude starts at zero, is always non-negative, and does not change by more than a constant value (UB) in one sampling frame, where a sampling frame is identified with one execution step of the synchronous model (one global clock tick) for simplicity. The latter constraint on the altitude change rate captures physical limitations on the speed of the aircraft.

A model checker can easily prove that safety requirement $R1$ is satisfied by the system model. This provides evidence that the system satisfies the safety requirement in the absence of faults. However, this result is not enough to determine whether the introduced redundancy mechanism makes the system more fault tolerant. To check this, we can consider different faulty behaviors, that is, different ways of injecting faults into the sensors. For this example, we will consider a very general faulty model where any of the sensors can fail and provide a value that does not satisfy assumptions S_i at some step. This way, if one of the altimeters fails, in the sense that it produces an altitude reading with an error greater than the maximum expected error, the other two values should allow the system to compensate for that error. To confirm this, we can compute a smallest cut set and verify the cardinality of the cut set is two, i.e., at least two of the assumptions S_i must fail to hold to trigger the TLE. Perhaps surprisingly though, if we compute a smallest cut set using an algorithm like the one we will present in next section, we see that there exists a smallest cut set of cardinality one that consists of only one of the assumptions S_i. That is, a single sensor failure is enough to lead the system to the violation of safety requirement $R1$. Put differently, property $R1$ requires *all three sensors* to behave according to their specification despite the use of a triplex voter.

As we will see in next section, computing a smallest cut set usually requires finding a series of smaller cut sets and counterexamples associated with it *and* eventually proving there is no counterexample of *any length* with a smaller cut set than the last cut set found so far. However, to spot a flaw like the one described above sometimes it is enough to find a counterexample and a cut set without imposing restrictions on the cardinality of the cut set, and then look for a counterexample of the same length that minimizes the number of cut set elements. Unlike first approach, which performs *global optimization*, the second approach considerably narrows down the search space by considering only counterexamples of a fixed length, determined by the first counterexample found. When applied to our example, this *local optimization* approach can find the *same* cut set of cardinality one as the first approach but it can compute it much more efficiently, as we will see in next section.

After reviewing the model in light of the existence of a cut set of size one, however computed, a designer may conclude that to benefit from the triplex voter it is necessary to decrease the safety limit value LIMIT in the controller's contract. In particular, it is enough to decrease it by doubling the error bound value: LIMIT = TH − (UB + 2 ∗ ERR). After this change, both approaches to compute cut sets return one of cardinality two, consisting of two of the assumptions S_i. In this case though, only the global optimization approach provides the guarantee that no smaller cut set exists.

As shown for our example, however, the local approach we propose, because of its lower computational cost, enables users to check for and discover design issues early in the modeling process. This difference in performance can be increasingly significant as the scale of analyzed systems grows. We present later initial experimental evidence suggesting that the advantages of our approach extend beyond the example given here.

4 Encoding Faulty Behavior

Faulty behavior in a system is specified as an extension of its nominal model. Starting from a nominal model $S = \langle \boldsymbol{s}, I[\boldsymbol{s}], T[\boldsymbol{s}, \boldsymbol{s'}] \rangle$ with $T = \{T_1, \ldots, T_n\}$, the system designer identifies m disjoint non-empty subsets $\boldsymbol{F}_1, \ldots, \boldsymbol{F}_m$ of T corresponding to m possible faults the system can suffer from so that every model component T_j in $\boldsymbol{F}_i$ is affected when fault i occurs. To simplify the exposition, we assume here that each fault affects different parts of the system. Also for simplicity, we assume that faults do not effect the behavior of the system in its initial state(s).[1]

[1] An extension to the case in which a component's behavior may be affected initially and possibly by more than one fault can be done with a slightly more complex formalization.

Now, for all faults $i = 1, \ldots, m$, the designer provides an alternative, faulty behavior specification $\widehat{T}_j$ of every model component T_j in $\boldsymbol{F}_i$. This faulty behavior is enabled by a Boolean flag f_i that represents the occurrence of fault i. Let $T^\diamond$ collect the components of T affected by none of the modeled faults, that is, $T^\diamond = T \setminus (\boldsymbol{F}_1 \cup \cdots \cup \boldsymbol{F}_m)$. Then, the faulty model is given by transition system $S^\star = \langle \boldsymbol{z}, I[\boldsymbol{z}], T^\star[\boldsymbol{z}, \boldsymbol{z}'] \rangle$, where the vector of typed variables $\boldsymbol{z}$ extends $\boldsymbol{s}$ with the fresh Boolean constants $f_1, \ldots, f_m$ and the transition predicate is defined by

$$T^\star[\boldsymbol{z}, \boldsymbol{z}'] \quad = \quad T^\diamond \cup \{\, T_j \vee (f_i \wedge \widehat{T}_j) \mid 1 \le i \le m,\ T_j \in \boldsymbol{F}_i \,\}$$

For convenience and generality, the system $S^\star$ is defined so that the presence of a fault i in a system execution (corresponding to the flag f_i having value true) may or may not trigger the faulty behavior in the affected components at any particular step of the execution. Note that, in effect, we can obtain the nominal model S from $S^\star$ by conjoining the constraints $\neg f_1, \ldots, \neg f_m$ to the initial state predicate.

5 Computing a cut set and a counterexample

Consider again a transition system $S = \langle \boldsymbol{s}, I[\boldsymbol{s}], T[\boldsymbol{s}, \boldsymbol{s}'] \rangle$ with faulty behavior specifications $\boldsymbol{F}_1, \ldots, \boldsymbol{F}_m \subseteq T$ enabled by faults $\boldsymbol{f} = \langle f_1, \ldots, f_m \rangle$, respectively. Given a state property P expected to be invariant for S, we can look for a cut set and a counterexample that leads the system to the violation of P by checking whether P is also invariant for the extended system $S^\star$ defined as in the previous section. If we can disprove P, the constant values assigned to $\boldsymbol{f}$ in any trace that leads $S^\star$ to the violation of P determines a cut set, namely, the set of all f_i's that are true. Such cut set describes a fault configuration that jeopardizes the invariance of P. On the other hand, if we prove P invariant, we can conclude that the system is robust to all faults as far as P is concerned.

In this work, we assume we have access to a black-box procedure Verify to perform these invariance checks. From a theoretical standpoint, Verify is an oracle since the invariance problem is undecidable in the infinite-state case. In practice, however, SMT-based model checking techniques, such as k-induction and IC3, yields incomplete versions of Verify that often provide a sound answer in reasonable time. In our concrete implementation, we make the verification check terminating by imposing a time limit and extending the type of the returned result with an additional value (unknown) to account for the timeout being reached.

6 Computing a globally smallest cut set

Building on top of the basic ideas described in the previous section, we can compute a smallest cut set and an associated counterexample, or determine that no such cut set exists, using Algorithm 1. The procedure can return Unknown due to the undecidability of the underlying model checking problem. It can also return a cut set that does not necessarily have minimal cardinality, which is indicated by a flag.

Algorithm 1 ComputeGlobalCutSet($\langle \boldsymbol{s}, I, T \rangle$, $\langle f_i, \boldsymbol{F}_i \rangle_{1 \le i \le m}$, P)

```
1:  T° := T \ (F_1 ∪ ··· ∪ F_m)
2:  T* := { T_j ∨ (f_i ∧ T̂_j) | 1 ≤ i ≤ m, T_j ∈ F_i } ∪ T°
3:  t := m; res := unknown; θ := ∅; f := ⟨f_1, ..., f_m⟩
4:  do
5:      I* := I ∧ AtMostK(f, t)
6:      res, θ' := Verify(s, I*, T*, P)
7:      if res = unsafe then
8:          t := t − 1; θ := θ'        ▷ Store last counterexample
9:      end if
10: while t ≥ 0 ∧ res = unsafe
11: if t < m then                      ▷ Unsafe with t + 1 faults
12:     if t < 0 then
13:         return ⟨∅, θ, true⟩        ▷ Nominal system itself unsafe
14:     else
15:         C := ExtractCutSet(θ, f)
16:         is_a_smallest_sol := (res = safe)
17:         return ⟨C, θ, is_a_smallest_sol⟩
18:     end if
19: else
20:     if res = unknown then
21:         return Unknown
22:     else
23:         return NoSolution
24:     end if
25: end if
```

The key idea of the algorithm is to add to the initial state predicate I a cardinality constraint AtMostK($\boldsymbol{f}, k$) over the fault flags $\boldsymbol{f}$ that restricts possible solutions to fault configurations with at most k (present) faults. Lines 5-11 use this reduction to find an MCS of minimal cardinality. If none exists, or the first call to Verify returned unknown, the condition in line 12 is false, and the algorithm returns the corresponding result in each case in lines 21-25. Specifically, if the first call to Verify reaches the time limit without determining the invariance of property P under the faulty conditions, the algorithm returns Unknown at line 22. If the first call to Verify returns safe, i.e., proves that P is invariant under faulty conditions, the algorithm returns there is no cut set solution (at line 24).

If the nominal system S itself does not satisfy P, condition in line 14 is true, and the unique MCS is the empty set. Otherwise, a cut set is extracted at line 16 from θ, which is the last error trace found in the do-while loop. This is done simply by collecting all the flags f_i that are assigned value true by the error trace. Before returning the cut set and the counterexample in line 18, the procedure determines

Algorithm 2 ComputeLocalCutSet($\langle \boldsymbol{s}, I, T \rangle$, $\langle f_i, \boldsymbol{F}_i \rangle_{1 \leq i \leq m}$, P)

```
1:  Let T* be as in Algorithm 1
2:  res, θ := Verify(s, I, T*, P)
3:  if res = unknown then
4:      return Unknown
5:  else
6:      if res = safe then
7:          return NoSolution
8:      else
9:          k := length(θ);
10:         φ_k := BMC_Encoding(I, T*, P, k)
11:         SmtAssertHard(φ_k)
12:         for i := 1 to m do
13:             SmtAssertSoft(¬f_i, 1)
14:         end for
15:         SmtCheckSat()
16:         θ' := GetCounterexample()
17:         C := ExtractCutSet(θ', f)
18:         return ⟨C, θ'⟩
19:     end if
20: end if
```

whether the cut set has minimal cardinality by checking if the last call to Verify returned `safe` (line 17).

7 Computing a locally smallest cut set

Algorithm 1 from Section 6 ensures that the cut set returned at line 18 has minimal cardinality provided that none of the calls to Verify return unknown. But it does that at the cost of having to solve multiple model checking problems, and having to prove the input property invariant for the provided system in the last call to Verify, which is often a harder problem to solve than disproving the satisfaction of a property.

In this section we describe an alternative method which does not ensure global optimality but can nonetheless find a small cut set with cardinality close or equal to a globally optimal solution while needing to call Verify only once.

In addition to Verify, the procedure implementing this method and described in Algorithm 2, relies on an external, off-the-self Max-SMT solver. It starts by checking whether cut sets exist at all by solving the model checking problem described in Section 5. If the call to Verify reaches the time limit without determining the invariance of property P under the faulty conditions, the algorithm returns Unknown at line 4. If the call to Verify returns `safe`, i.e., proves that P is invariant under faulty conditions, the algorithm returns there is no cut set solution (at line 7). Otherwise, we know there exists a cut set. With k being the length of the counterexample θ returned by Verify, the procedure builds a Max-SMT problem

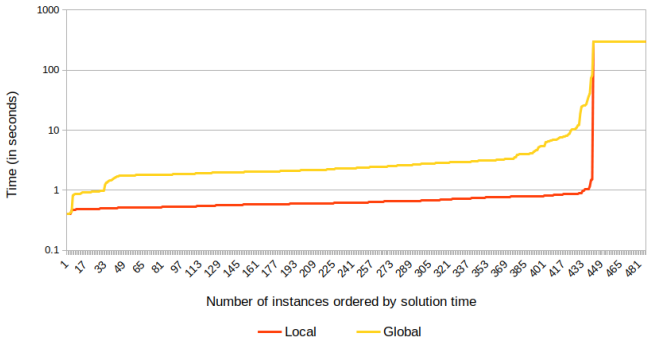

Figure 2. Comparison between local and global optimization

to find the smallest cut set among all cut sets with an associated counterexample of length exactly k (at lines 9-18).[2]

The main idea is to first create a standard Bounded Model Checking encoding (as described in Section 2.3) with the faulty model and invariance property P for a bound k equal to the length of θ. The resulting formula is asserted as a hard constraint to the Max-SMT solver (at line 11). The optimization problem for the Max-SMT solver consists in minimizing the number of active faults needed to violate property P. This is done by asserting to the solver one soft constraint per fault flag f_i, all with weight 1, stating that the flag is false ($\neg f_i$ at lines 12-14). Note that Max-SMT problem so obtained, checked at line 15, is guaranteed to have a solution since the counterexample to P invariance found by the call to Verify is a solution to this problem as well. By the optimality of the solution θ' returned by the Max-SMT solver (at 16), the cut set C computed at line 17 is guaranteed to have smallest cardinality among all cut sets associated to counterexamples of length k, the length of the original counterexample. The procedure returns both the cut set and its associated counterexample, similarly to the global optimization one but without any claims about the cut set's global optimality.

8 Experimental Evaluation

The main premise of this work is that the approach for computing small cut sets based on local optimization, as described in Section 7, is more efficient in practice than the computation providing global guarantees as described in Section 6. Moreover, the solutions computed by the local optimization approach have a cardinality equal or close to the optimal one.

To test our hypothesis, we implemented both approaches in our model checker Kind 2 [11], and compared their performance on a fairly large set of benchmarks. Because of the

[2] Alternatively, we could consider cut sets with associated counterexample of length *at most* k, like in the standard formulation of BMC. However, typical implementations of Verify based on k-induction and IC3 usually provide counterexamples whose length is already minimal or close to it most of the time.

practical difficulty of finding publicly available models readable by Kind 2 and specifically designed with fault tolerance in mind, we retooled a previous set of safety benchmarks to the purposes of our evaluation. Each benchmark in the starting set, largely based on one introduced by Kahsai and Tinelli [10][3] consists of a multi-component model with a single invariance property. We selected benchmarks from the starting set by running Kind 2 on a cluster with ten Intel(R) Xeon(R) E3-1240, 3.4GHz, 4 cores, 16 GB memory machines. We kept the benchmarks for which Kind 2 was able to prove the property valid with a 5 minute timeout, which yielded 486 instances. Then, we considered the problem of proving each property under the possibility of each model component failing to provide outputs consistent with its low-level specification. More specifically, we modeled the faulty behavior simply by allowing the failing component to return any value at all among those allowed by its output type.[4]

We ran Kind 2 on the selected problems using each approach with 15 minute timeout. Figure 2 shows that the local optimization approach significantly out-performs the global optimization one, providing an answer 5.6 times faster on average. Analyzing the individual responses to each problem, we found that both approaches were able to detect the absence of cut sets for 5 of the problems, and the local approach was able to find a cut set with optimal cardinality in 434 out of the 436 cases where both approaches found a cut set. In the two cases where the local approach returned a suboptimal solution, it was off just by one with respect to the size of the optimal solution.

These results should be taken with a grain a salt due to the synthetic nature of the benchmarks, a majority of which had highly fault-intolerant systems. In the majority of cases, the globally optimal cut sets had cardinality one and in the rest it had cardinality two. More seriously perhaps, in most of the benchmarks almost any one of the considered faults would lead, by itself, to the violation of the property, making it easy for our local optimization method to stumble into a globally optimal solution. Although more experiments are needed, we find that our evaluation provides nevertheless encouraging preliminary evidence of the usefulness of the local optimization method.

9 Related Work

The use of Bounded Model Checking to find a cut set and an associated counterexample was first proposed by Abdulla et al. [1]. The focus of that work, however, is not on computing a single cut set but *all* MCSs. To prevent the generation of non-minimal solutions, the paper proposes the computation of cuts sets of increasing cardinality. This approach has also the advantage of generating smaller MCSs before larger MCSs. Thus, one can generate a single smallest MCS just stopping after the first optimal solution is generated. The search for a single smallest MCS is very similar to the one performed by the method presented in Section 6, the main difference being the direction of the search. The technique of Abdulla et al. progressively tries all numbers of faults from 0 to m (forward enumeration), whereas our algorithm tries values from m to 0 (backward enumeration). The rationale of our strategy is to minimize the number of solved problems for which the property is valid, which is often a harder problem to solve than disproving the satisfaction of a property, especially when the length of the generated counterexamples is not very long. As we describe in other work [11, 12], Kind 2 implements the approach above using backward enumeration to compute all MCSs, (and a single globally smallest MCS). Unlike the encoding of faulty models presented in this paper, the actual implementation in Kind 2 only supports faulty behavioral specifications equivalent to true. More specifically, Kind 2 allows the user to choose a set of model elements (assumptions and guarantees, node calls, equations in node bodies, ...) of its input language, an extension of the dataflow Lustre language [8], and Kind 2 will compute minimal sets of those elements whose violation leads the system to an unsafe state. However, this is not really a limitation since the more general case presented in this work can reduced to the more specific case supported by Kind 2: once a faulty model is built following the encoding described in Section 4, the user just have to specify assumptions stating that f_i should be false, and choose those assumptions as the model elements.

Another algorithm for computing all MCSs is described by Bozzano et al. [4]. It too forces the algorithm to proceed by layers of increasing cardinality. Thus, it may also be used to compute a globally smallest cut set. The method relies on a IC3-based routine for parameter synthesis to compute all the solutions in each layer. Therefore, instead of relying on a black-box Verify procedure to solve multiple ordinary model checking queries, they use a specialized algorithm. The main advantage in that case is that the information learnt to block a particular counterexample can be reused when considering new ones.

A different approach to computing all MCSs is the method presented by Stewart et al. [15]. It exploits the duality between the set of *Minimal Inductive Validity Cores (MIVCs)* [6], minimal sets of model elements that are sufficient to prove a property, and the set of Minimal Cut Sets for the same property and computes the latter from the former. This is convenient when the goal is to compute *all* MCSs since one can use an offline algorithm for enumerating all MIVCs [7], which may offer better overall performance than computing one MIVC at a time. The downside of this solution is that, unlike the techniques described earlier, it cannot be used to compute a single MCS for a property without paying the cost of computing all of them.

[3] Available at https://github.com/kind2-mc/kind2-benchmarks.
[4] That is, we did not consider the possibility of an ill-typed result.

10 Conclusion

We presented a method that leverages behavioral modeling and Max-SMT solvers to obtain efficiently a small set of faults that lead to the violation of safety requirements. The method computes a cut set with minimal cardinality over all counterexamples of a *given* length by reducing the problem to an optimization problem over an SMT formula. Initial experimental results are very encouraging in terms of the effectiveness of the method in generating cut sets that are close or equal to globally optimal solutions, and the speed up achieved compared to the standard method for computing a (globally) smallest cut set.

As future work, we want to investigate further the effectiveness of the proposed technique by applying the method to a broader set of benchmarks, and evaluating its performance in determining not only the tolerance of a system against faults, but also its resilience to cyber-attacks. We also want to explore the possibilities of setting different weights for the soft constraints in the Max-SMT problem, which leads to a natural way of establishing preference between difference solutions beyond the cardinality of the cut sets. Specifically, in Algorithm 2, the soft constraints stating that a particular fault should not occur all have weight 1. One can imagine assigning a higher weight to a subset of the soft constraints to give preference to solutions that do not include faults in the subset.

Acknowledgments

This work was partially funded by DARPA grant #N66001-18-C-4006 and by GE Global Research.

References

[1] Parosh Aziz Abdulla, Johann Deneux, Gunnar Stålmarck, Herman Ågren, and Ove Åkerlund. 2004. Designing Safe, Reliable Systems Using Scade. In *Leveraging Applications of Formal Methods, First International Symposium, ISoLA 2004, Paphos, Cyprus, October 30 - November 2, 2004, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 4313)*, Tiziana Margaria and Bernhard Steffen (Eds.). Springer, 115–129. https://doi.org/10.1007/11925040_8

[2] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. 1999. Symbolic Model Checking without BDDs. In *Tools and Algorithms for Construction and Analysis of Systems, 5th International Conference, TACAS '99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99, Amsterdam, The Netherlands, March 22-28, 1999, Proceedings (Lecture Notes in Computer Science, Vol. 1579)*, Rance Cleaveland (Ed.). Springer, 193–207. https://doi.org/10.1007/3-540-49059-0_14

[3] 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. https://doi.org/10.3233/FAIA336

[4] Marco Bozzano, Alessandro Cimatti, Alberto Griggio, and Cristian Mattarei. 2015. Efficient Anytime Techniques for Model-Based Safety Analysis. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 9206)*, Daniel Kroening and Corina S. Pasareanu (Eds.). Springer, 603–621. https://doi.org/10.1007/978-3-319-21690-4_41

[5] Aaron R. Bradley. 2011. SAT-Based Model Checking without Unrolling. In *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6538)*, Ranjit Jhala and David A. Schmidt (Eds.). Springer, 70–87. https://doi.org/10.1007/978-3-642-18275-4_7

[6] Elaheh Ghassabani, Andrew Gacek, and Michael W. Whalen. 2016. Efficient generation of inductive validity cores for safety properties. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, Thomas Zimmermann, Jane Cleland-Huang, and Zhendong Su (Eds.). ACM, 314–325. https://doi.org/10.1145/2950290.2950346

[7] Elaheh Ghassabani, Michael W. Whalen, and Andrew Gacek. 2017. Efficient generation of all minimal inductive validity cores. In *2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, October 2-6, 2017*, Daryl Stewart and Georg Weissenbacher (Eds.). IEEE, 31–38. https://doi.org/10.23919/FMCAD.2017.8102238

[8] Nicolas Halbwachs, Fabienne Lagnier, and Christophe Ratel. 1992. Programming and Verifying Real-Time Systems by Means of the Synchronous Data-Flow Language LUSTRE. *IEEE Trans. Software Eng.* 18, 9 (1992), 785–793. https://doi.org/10.1109/32.159839

[9] A. Joshi, S.P. Miller, M. Whalen, and M.P.E. Heimdahl. 2005. A proposal for model-based safety analysis. In *24th Digital Avionics Systems Conference*, Vol. 2. 13 pp. Vol. 2–. https://doi.org/10.1109/DASC.2005.1563469

[10] Temesghen Kahsai and Cesare Tinelli. 2011. PKind: A parallel k-induction based model checker. In *Proceedings 10th Int'l Workshop on Parallel and Distributed Methods in verifiCation, PDMC 2011 (EPTCS, Vol. 72)*. 55–62. https://doi.org/10.4204/EPTCS.72.6

[11] Daniel Larraz, Mickaël Laurent, and Cesare Tinelli. 2021. Merit and Blame Assignment with Kind 2. In *Formal Methods for Industrial Critical Systems - 26th International Conference, FMICS 2021, Paris, France, August 24-26, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12863)*, Alberto Lluch-Lafuente and Anastasia Mavridou (Eds.). Springer, 212–220. https://doi.org/10.1007/978-3-030-85248-1_14

[12] Daniel Larraz, Mickaël Laurent, and Cesare Tinelli. 2021. Merit and Blame Assignment with Kind 2. *CoRR* abs/2105.06575 (2021). arXiv:2105.06575 https://arxiv.org/abs/2105.06575

[13] Robert Nieuwenhuis and Albert Oliveras. 2006. On SAT Modulo Theories and Optimization Problems. In *Theory and Applications of Satisfiability Testing - SAT 2006, 9th International Conference, Seattle, WA, USA, August 12-15, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 4121)*, Armin Biere and Carla P. Gomes (Eds.). Springer, 156–169. https://doi.org/10.1007/11814948_18

[14] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. 2000. Checking Safety Properties Using Induction and a SAT-Solver. In *Formal Methods in Computer-Aided Design, Third International Conference, FMCAD 2000, Austin, Texas, USA, November 1-3, 2000, Proceedings (Lecture Notes in Computer Science, Vol. 1954)*, Warren A. Hunt Jr. and Steven D. Johnson (Eds.). Springer, 108–125. https://doi.org/10.1007/3-540-40922-X_8

[15] Danielle Stewart, Michael W. Whalen, Mats Per Erik Heimdahl, Jing Liu, and Darren D. Cofer. 2021. Composition of Fault Forests. In *Computer Safety, Reliability, and Security - 40th International Conference, SAFECOMP 2021, York, UK, September 8-10, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12852)*, Ibrahim Habli, Mark Sujan, and Friedemann Bitsch (Eds.). Springer, 258–275. https://doi.org/10.1007/978-3-030-83903-1_17