Exploring and Categorizing Error Spaces using BMC and SMT

Tim King[1], Clark Barrett[1]
[1]New York University, taking|barrett@cs.nyu.edu

Abstract

We describe an abstract methodology for exploring and categorizing the space of error traces for a system using a procedure based on Satisfiability Modulo Theories and Bounded Model Checking. A key component required by the technique is a way to generalize an error trace into a category of error traces. We describe tools and techniques to support a human expert in this generalization task. Finally, we report on a case study in which the methodology is applied to a simple version of the Traffic Air and Collision Avoidance System.

1 Introduction

Finding traces that represent errors in hardware and software by the means of Bounded Model Checking (BMC) has been one of the great recent success stories in formal methods [4]. Both Boolean Satisfiability (SAT) and Satisfiability Modulo Theories (SMT) solvers have been used as effective engines for BMC [10]. While SAT techniques are more established, SMT solvers have the advantage of being able to reason natively at a higher level of abstraction, easing the modeling process and often leading to efficiency gains as well [5].

In this paper, we suggest a novel use of BMC: instead of searching for a single error trace, we develop a method for exploring and categorizing the space of all errors. Our proposed approach is to repeatedly complete the following steps: first, use SMT-based BMC to find an error trace; second, generalize the trace into a set of traces (we call this set a *category*); third, specify the category formally (using the language of the SMT solver); and finally, use the formal specification to exclude this category from the next iteration of the BMC search. If the system is finite, or the categories can be made sufficiently general, the process terminates with a complete categorization of all error traces together with a sample error trace for each category.

Such a method may be helpful in situations when a system is known to have many error traces (according to some specification) and the system designers believe these error traces to be sufficiently rare or benign (while changing the system is seen as costly) as to warrant not fixing the errors. By exploring and categorizing the error space, this conjecture can be tested and either confirmed (by verifying that all categories are non-problematic) or challenged (by finding a category of serious errors that were previously unknown). Even if no serious errors are found, the procedure can be seen as an aid in developing a more refined specification (e.g. the original specification can be extended with a formal characterization of "error" categories that are deemed non-critical).

A key step in our procedure is the generalization of an error trace into a category. This step is challenging because it must balance generality (which is desirable to ensure the procedure terminates relatively quickly) with meaningfulness (since each category should be limited to a closely related set of error traces). In this paper, we specifically and intentionally consider the case in which categories will be evaluated by a human

(i.e. no formal specification exists for what an "acceptable" error trace might be). This also motivates the need for keeping the number of categories to a minimum.

The paper is organized as follows. We begin with a review of SMT, BMC, and other necessary background information. We then explain the main algorithm for error space exploration and categorization. Next, we introduce a modeling language called `transmit`, which helps bridge the gap between the system model and the SMT back-end. We then describe results on a case study that motivated this work: a simplified version of the Traffic Air and Collision Avoidance System (TCAS). Finally, we conclude with a discussion of related and future work.

2 Preliminaries

We assume the reader is familiar with standard notions from many-sorted first order logic and the Satisfiability Modulo Theories (SMT) problem (see for example [12, 5]). We assume SMT-DECIDE is a model-generating algorithm solving the SMT problem for a theory $\mathfrak{T}$ with signature $\Sigma_{\mathfrak{T}}$. SMT-DECIDE takes a $\Sigma_{\mathfrak{T}}$-formula φ and returns $(\mathbf{sat}\ M)$ if φ is satisfiable (where M is a $\mathfrak{T}$-interpretation that satisfies φ) and $(\mathbf{unsat})$ if the formula is unsatisfiable.

Bounded Model Checking (BMC) is a verification technique for systems that works by considering finite traces of the system up to some maximum size. While limited in its ability to verify properties, it has been very effective at bug-finding. For our purposes, BMC refers to the process of: selecting a bound k on the number of system steps; creating a formula that represents execution of the system from the initial state through k transitions into an error state; and then using an SMT solver to decide whether this formula is satisfiable. If the formula is satisfiable, this indicates that the error state is reachable. Furthermore, if the SMT solver can provide information about the satisfying assignment, this can be used to generate a specific error trace. On the other hand, if the formula is unsatisfiable, this proves that the error cannot be reached in k steps.

Formally, given a background theory $\mathfrak{T}$, we will take as our system model triples of the form $(V, \mathcal{I}, \mathcal{T})$. Here, V is a set of *state* variables over sorts from $\Sigma_{\mathfrak{T}}$ that describe the state of the system. We call a $\Sigma_{\mathfrak{T}}$-formula, all of whose free variables are from V, a state formula. $\mathcal{I}$ is a state formula which is true exactly when the state variables take on values representing a valid initial state of the system. $\mathcal{T}$ is a $\Sigma_{\mathfrak{T}}$-formula whose free variables are from V and V' (a copy of V containing a variable x' for each variable $x \in V$) which is true iff the system can transition from a state represented by V to a state represented by V'. We assume that error states can also be described by state formulas (for simplicity, we consider only safety properties).

Besides V', we also define V_i to be a copy of V (containing variables x_i for each $x \in V$) for each $i \geq 0$. Also, for $i \geq 0$, the indexing operator $(\cdot)_i$ takes a state formula ϕ and produces a formula ϕ_i by replacing each occurrence of $x \in V$ by the corresponding variable $x_i \in V_i$. Similarly, the indexing operator applied to $\mathcal{T}$ produces a formula $\mathcal{T}_i$ obtained by replacing each occurrence of $x \in V$ with $x_i \in V_i$ and each occurrence of $x' \in V'$ with $x_{i+1} \in V_{i+1}$.

The unrolling function UNROLL takes as input a system and an unrolling depth k, and produces a formula that represents running the system for k steps from a valid initial state:

$$\text{UNROLL}(V, \mathcal{I}, \mathcal{T}, k) := \mathcal{I}_0 \wedge \bigwedge_{i=0}^{k-1} \mathcal{T}_i.$$

A trace τ of length k is a $\mathfrak{T}$-interpretation satisfying a k-step unrolling of the system:

EXPLORE$(V, \mathcal{I}, \mathcal{T}, \mathcal{E}, k)$

1 $q_0 \leftarrow$ UNROLL$(V, \mathcal{I}, \mathcal{T}, \mathcal{E}, k)$
2 $i \leftarrow 0$
3 **while** SMT-DECIDE$(q_i) = (\textbf{sat}\ \epsilon_i)$
4 **do** $\mathcal{C}_i \leftarrow$ ANALYZE(ϵ_i)
5 $q_{i+1} \leftarrow q_i \wedge \neg \mathcal{C}_i$
6 $i \leftarrow i + 1$
7 **return** $[\mathcal{C}_0, \ldots \mathcal{C}_i], [\epsilon_0, \ldots, \epsilon_i]$

Figure 1: The EXPLORE procedure.

$$\tau \models \text{UNROLL}(V, \mathcal{I}, \mathcal{T}, k).$$

Given a state formula $\mathcal{E}$ describing an error state, an error trace, ϵ, is a system trace that additionally satisfies $\mathcal{E}_i$ for some i. We extend the unrolling function UNROLL to take as an additional input an error formula $\mathcal{E}$ and to produce a formula additionally requiring $\mathcal{E}$ to be satisfied within k transitions:

$$\text{UNROLL}(V, \mathcal{I}, \mathcal{T}, \mathcal{E}, k) := \mathcal{I}_0 \wedge \bigwedge_{i=0}^{k-1} \mathcal{T}_i \wedge \bigvee_{i=0}^{k} \mathcal{E}_i.$$

We refer to the set of all error traces as the *error space* of the system.

3 Exploring the Error Space

Applications that employ BMC typically use it to generate a single error trace which is immediately reported to the user as a bug. The user then analyzes the trace and updates the system accordingly. In a scenario in which some errors may be deemed acceptable and modifying the system is considered to be expensive, this simple bug-finding and patch loop may no longer be appropriate. BMC can still be used to produce error traces, but a more complete picture of the error space is desirable in order to determine whether an update to the system is warranted.

The EXPLORE procedure employs BMC in a more general loop that explores and categorizes multiple error traces. A diagram of the procedure is shown in Fig. 1. EXPLORE takes as input a system, an error state, and an unrolling depth k. The procedure unrolls the system to obtain a formula q_0 that is satisfied if there are any error states reachable within k steps. q_i is sent to an SMT solver to decide if it is satisfiable (i is initially 0.) If q_i is satisfiable, the model-generating feature of the SMT solver (SMT-DECIDE) is used to obtain information about a satisfying interpretation of q_i. The satisfying interpretation ϵ_i is an error trace. The abstract procedure ANALYZE takes ϵ_i and generalizes it to a category formula $\mathcal{C}_i$, such that $\epsilon_i \models \mathcal{C}_i$. (We discuss different possible choices for ANALYZE in more detail at the end of the section and in Sec. 4.) To exclude the traces in $\mathcal{C}_i$ and discover a new satisfying error trace, q_i is conjoined with the negation of $\mathcal{C}_i$. The next iteration of the process then occurs, with the SMT solver being invoked with the new query q_{i+1}. If at some point q_i is unsatisfiable, then the categories include all the error traces possible for this k. In this case, the procedure terminates and outputs the set of categories and satisfying error traces, $[\mathcal{C}_0, \ldots \mathcal{C}_i], [\epsilon_0, \ldots, \epsilon_i]$.

EXPLORE is a straightforward generalization of a well-known solution for extracting multiple solutions of a single formula. Any solution implemented with multiple calls to an SMT solver must guarantee that the model returned in one iteration is excluded from the next. The easy way to do this is simply to assert the negation of some formula ϕ that is satisfied by the model. ϕ is an abstraction of the model. In EXPLORE, this process of abstracting the model and generating such a formula is captured by the ANALYZE procedure.

The choice of category formulas $\mathcal{C}_i$ introduced by ANALYZE must be done with care as this controls which future error traces are generated and also determines how quickly the procedure terminates. In the general presentation above, ANALYZE does not ensure that the different error traces represent *meaningful* distinctions or make progress. For example, ANALYZE could introduce the sequence of categories $x_0 = 1, x_0 = 2, \ldots$ restricting the initial value of a variable x to be a different constant each time (as long as each has a model), resulting in non-termination. On the other hand, suppose ANALYZE introduces the sequence $(x_0 \leq \frac{1}{2})$, $\neg(x_0 \leq \frac{1}{2})$. In this case, we do have termination, but there may be more than two substantially different kinds of bugs in the original system, whereas with this categorization, only 2 representative error traces will be produced. More (and possibly more efficient forms of) exhaustive case splitting can be done, but this may produce too many error traces.[1] While these models may be helpful for assisting other computations, if models are going to be validated by hand, the number presented to the user needs to be quite low. We mention all of this to emphasize that the value of the output of EXPLORE crucially depends on how ANALYZE guides the search through the error space, and also to highlight the difficulty of designing such a procedure.

4 Interactive Exploration of the Error Space

One of the main contributions of this paper is to evaluate an implementation of ANALYZE which uses an interactive approach. In particular, rather than trying to automate this step, we have investigated how best to support a system expert (the *analyst*) in analyzing the error traces and constructing the category formulas. This decision is principally motivated by the observation that the categories need to be meaningful and comprehensible to human evaluators, often taking into account domain-specific concepts, so that there may not be a good general mechanism for producing categories automatically.

Note that the analyst's task is not too different from what must be done by system developers when using BMC as a bug-finding tool. In particular, in order to determine the severity of and appropriate response to an error trace, a developer must thoroughly understand the trace, be able to abstract this understanding into a conceptual idea about what is wrong, and then apply this conceptual understanding to fix the problem. Here, the analyst must similarly understand the error trace and use this to create an abstract idea capturing the cause of the problem. The additional step required is to express this idea as a category formula.

There are several advantages of this approach over a simple use of BMC for bug-finding. First of all, under the assumption that there are many error traces to look at, it may not even be feasible to examine each one individually. Thus the step of categorization is crucial not just to save time but also to have any hope of covering all the traces. A second advantage is that if the analyst is able to capture the abstract concept behind each error trace, this ensures that each new error trace will represent a new conceptual problem. This is much more interesting and instructive than examining many error traces that are just slight variations of the same basic problem. Finally, this process is much more likely to lead the analyst to find rare and

[1] If ANALYZE consistently introduced the negation of the satisfying Boolean assignment to each atom in q_i, EXPLORE would be isomorphic to the SMT equivalent of ALLSAT [16], guaranteeing termination. There are a number of possible variations to this scheme, including predicate abstraction.

unexpected behaviors as they will work quickly to categorize (and thus eliminate) common and understood behaviors.

With a human analyst, the EXPLORE procedure can be summarized as a means of letting the analyst interactively search the error space of a system with the help of SMT-based BMC technology. The computer performs the difficult search for error traces on demand, while the analyst's job is to examine the error traces and produce category formulas for them. This is a paradigm similar to the one followed by users of interactive theorem provers: let the user focus on the big picture, and let the computer deal with the tedious details.

5 Transmit

As part of this work, we developed `transmit`, a language for quickly encoding BMC queries and categories for use with SMT.[2] `transmit` provides the analyst with tools to support the interactive EXPLORE procedure. `transmit` is also an appropriate language for specifying and testing prototypes. A small example of a `transmit` specification is given in Fig. 2 and explained in section 6.

Specifications in `transmit` are written by annotating S-Expressions in an underlying language, such as SMT-LIB v2. This annotated expression is reduced by `transmit` to an S-Expression in the underlying language using simple recursive top-down transformations. The primary constructs of `transmit` are: indexed state variables (`[$v]`), support for setting the index to the value of a constant expression (`[# c (.)]`), bounded repetition of an expression (`[#for start end (.)]`), and binding a parameter to a value (`[bind k c]`).

The specification of state predicates and transition relations for BMC problems can be expressed using `transmit` in a fairly straightforward definitional style. A typical use of `transmit` is to bind some parameters to constants (such as the unroll depth) and then pass the annotated formulas to `transmit` which produces an SMT query. While declarations of the system generally resemble those in more sophisticated high-level modeling languages such as SAL or UCLID, `transmit` specifications are written at a lower level: just above the level of SMT formulas. `transmit` can be thought of as a scripting language for designing SMT-LIB queries, giving the user a large degree of low-level control over the generated query.

`transmit` is designed to help support the interactive construction of categories during EXPLORE. The category formulas generated during EXPLORE are formulas over the first k states. It is desirable to make the specification of the categories hold for multiple choices of k and support basic temporal reasoning. `transmit`'s approach to this is to have the categories specified using the same underlying language as the system specification. Like system specifications, categories have access to the parameterization of the system, and can be parameterized on k as well. The categories are then given k in the same fashion as the system description, and are compiled by `transmit` into a well-defined first order logic formula C_i (over $\cup_{i=0}^{k} V_i$) which EXPLORE can use to make progress. `transmit` specifications given access to k are powerful enough to express Linear Temporal Logic safety properties [8] with past operators. This shows that basic temporal reasoning is feasible. By giving the user access to such a powerful specification scheme, we maximize their ability to interactively explore the space as they see fit.

To assist the analyst, `transmit` provides tools for parsing models output by a number of SMT solvers[3] into a pair of comma-separated-value files (one for system wide constants, and one for state variables). This format makes it easy to write short scripts that generate visualizations for the traces using tools such as gnuplot or R.

[2] An alpha version of `transmit` is available at `http://cs.nyu.edu/~taking`.

[3] Currently CVC3, CVC4 and Z3 are supported.

```
1  [bind k 2]
2  (set-logic QF_LRA)
3
4  {- System Paramters -}
5  (declare-fun MinFlowRate () Real)
6  (declare-fun Capacity () Real)
7
8  {- State Variables -}
9  [#for 0 k (declare-fun [$tank] () Real)]
10 [#for 0 k (assert (and (<= [$tank] Capacity) (>= [$tank] 0))) ]
11 [#for 0 k (declare-fun [$incoming] () Real)]
12 [#for 0 k (assert (> [$incoming] 0)) ]
13 [#for 0 k (declare-fun [$outgoing] () Real)]
14 [#for 0 k (assert (< [$outgoing] 0)) ]
15
16 {- Initial State -}
17 (assert (>= [# 0 [$tank]] (* (/ 1 2) Capacity)))
18
19 {- Transition relation -}
20 [#for 0 [- k 1] (assert (= [next [$tank]] (+ [$tank] [$incoming] [$outgoing])))]
21 [#for 0 [- k 1] (assert (ite (<= (* 2 [$tank]) Capacity)
22    (> [next [$incoming]] [$incoming]) (< [next [$incoming]] [$incoming])))]
23 [#for 0 k (assert (=> (>= (+ [$tank] MinFlowRate) 0) (<= [$outgoing] MinFlowRate)))]
24
25 {- Eventually the error formula is satsified. -}
26 (assert (or [#for 0 [- k 1] (> [$outgoing] MinFlowRate)]))
27
28 (check-sat)
```

Figure 2: `transmit` specification of a simple hybrid system.

`transmit` provides tools for facilitating the entire process of the interactive EXPLORE procedure. Doing this with a single tool helps the user maintain consistency between the system description, the categories, the query, and the satisfying interpretation.

6 Example using `transmit` and EXPLORE

This section describes an example of a `transmit` specification, and summarizes how a user might employ the EXPLORE procedure on this example. The example file is given in Fig. 2. This example models a simple hybrid system consisting of a water tank with an incoming nozzle whose rate is controllable. The liquid is continually flowing out of the tank, and the flow must be kept above a certain rate, i.e. there is an error if the amount of outgoing liquid is below some threshold.

`transmit` annotations use square brackets and LISP style S-expressions to annotate the formula. Expressions wrapped in `{-` and `-}` are comments in `transmit` (following notation from Haskell). This example is built on top of SMT-LIB v2, and compiles to an SMT-LIB v2 query. Line 1 in the example binds k to the constant 2. Line 2 is the header for the SMT query. Line 5 declares the variable `MinFlowRate`, the amount that should leave the tank every cycle. Line 6 declares the variable `Capacity`. Lines 9-10 declare a state variable `$tank` and assert that its value is always between 0 and `Capacity`. To see how the `#for` construct expands, the `transmit` output for these lines is shown below:

```
(declare-fun |tank_000| () Real)
(declare-fun |tank_001| () Real)
(declare-fun |tank_002| () Real)
```

```
(assert (and (<= |tank_000| Capacity) (>= |tank_000| 0)))
(assert (and (<= |tank_001| Capacity) (>= |tank_001| 0)))
(assert (and (<= |tank_002| Capacity) (>= |tank_002| 0)))
```

Lines 11-14 similarly declare the state variables `[$incoming]` and `[$outgoing]` and assert that the variables are respectively always positive and negative. Line 17 declares that the initial value of `[$tank]` (`[# 0 [$tank]]`) is at least half of `Capacity`. Lines 20-23 specify the transition relation. The first part of this is that the next value of `$tank` (`[next [$tank]]`) is equal to the sum of the current values of `$tank`, `$incoming` and `$outgoing`. The `transmit` output for this is 2 lines, the first of which is:

```
(assert (= |tank_001| (+ |tank_000| |incoming_000| |outgoing_000|)))
```

Lines 21-22 give a basic rule for increasing the incoming rate if the tank is at least half empty and decreasing the rate otherwise. Line 23 specifies that if `[$tank]` has at least the minimum flow rate currently in it, then at least this amount flows out. Line 26 is a statement of the error condition: at some point, less than `MinFlowRate` flows out of the tank. The `transmit` output for line 26 is:

```
(assert (or (> |outgoing_000| MinFlowRate) (> |outgoing_001| MinFlowRate)))
```

Finally, the file is ended by the SMT `(check-sat)` command.

We used the EXPLORE procedure on this specification, and generated two categories, after which the loop terminated with an unsatisfiable result. Our first category was motivated by the observation that the relationship between `MinFlowRate` and `Capacity` is undefined, so it is possible that `Capacity` is actually smaller than the magnitude of `MinFlowRate` (which is negative), meaning that `(>= (+ [$tank] MinFlowRate) 0)` would always be false. We used the following formula as a generalization of this category of errors:

```
(assert (< (+ Capacity (* 2 MinFlowRate)) 0))
```

Our second category addresses the main problem of the system. At any point, `$outgoing` can be sufficiently negative and `$incoming` sufficiently close to zero that as a result, the value of `$tank` in the next state is less than the magnitude of `MinFlowRate`. We introduced a category for this problem that captures exactly these cases: `(<= (+ [$tank] [$incoming] [$outgoing] MinFlowRate) 0)`. The negation of these two categories together with the original formula is unsatisfiable and EXPLORE terminates.

7 Case Study

The Traffic Air and Collision Avoidance System (TCAS) is a currently deployed collision avoidance system for aircraft [15]. The system provides pilots independent tracking of other aircraft in the local airspace and in emergency situations provides Resolution Advisories (RAs) to the pilots on how to avoid likely collisions. TCAS's RAs are considered a means of defense-in-depth for when normal air traffic management's separation procedures have broken down. Due to TCAS's inherent safety critical nature [1], it has been the subject of a number formal studies in the past [14, 17, 7, 18]. Other aircraft collision avoidance systems have been studied in detail as well [19].

TCAS treats the planes as points and attempts to keep the points sufficiently far apart so as to avoid Near Mid-Air Collisions (NMACs). An NMAC is defined as a situation in which two planes have a horizontal separation (`range`) less than the constant `NMACw` and a vertical separation less than the constant `NMACh`.[4]

[4] We used `NMACw` $= 500$ft and `NMACh` $= 100$ft.

Geometrically, this means that an NMAC occurs if a second plane enters a cylinder centered around the first plane. This over-approximation avoids having to model complex and mostly irrelevant plane and helicopter geometries that more accurate modeling of mid-air collisions would require. TCAS runs a fixed protocol every second during which it both checks its sensors and performs *sense selection* (deciding whether to issue an RA and if so what RA is selected). Non-linear floating point arithmetic calculations are used for estimating position, velocity, and the time of closest horizontal distance (TCA), as well as for determining when to issue an RA and when to stop issuing the RA.

TCAS is an excellent case study for our approach because it is a system that needs to be better understood but which is very difficult and costly to change. Furthermore, if we define any scenario that leads to an NMAC to be an error trace, it is clear that errors cannot always be avoided (a malicious pilot could always cause an NMAC for example). The goal of TCAS is to avoid NMACs in reasonable scenarios. However, it is not clear how to evaluate the success of this goal.

We applied our technique to a simplified version of TCAS called Tiny TCAS. Tiny TCAS was developed at MIT Lincoln Laboratory for the purpose of experimenting with formal techniques.[5] Tiny TCAS restricts its attention to the case when one plane is equipped with Tiny TCAS and a single intruder is equipped only with a transponder (which communicates its position and velocity). Tiny TCAS assumes its variables are real numbers (ignoring approximations and errors introduced by floating point representations). Tiny TCAS contains non-linear real arithmetic constraints for projecting positions in the future and calculating when an RA can be released.

While some existing SMT solvers do have limited support for non-linear real arithmetic[9, 3], there are no currently available solvers able to analyze the Tiny TCAS model without modification.[6] Since Tiny TCAS already makes many simplifying assumptions, we added one additional simplification (holding constant the horizontal rate at which the aircraft are approaching each other), which allowed us to obtain a linear model. Formulas generated from the model then fit within the SMT-LIB logic QF_LRA.

Using a Transmit model and the EXPLORE procedure, we generated five categories of system failure for Tiny TCAS. An automatically generated visualization of an example trace from each category is shown in Fig. 3. An informal description of the categories is given below. The visualizations are automatically generated from error traces using a collection of simple scripts. Each figure shows the altitude of the two planes over time. The blue line is the intruder, and the black line is the plane equipped with Tiny TCAS. The large orange dots represent an NMAC. The first and last vertical green lines are the first and last time the horizontal range is small enough for an NMAC to occur (labeled entry and exit). The middle green vertical line labeled TCA is the time of closest approach or minimum horizontal range. We found automatically generated visualizations like this to be *the key analytical tool* in categorizing error traces. Other formal studies of TCAS have noted the importance of generating visualizations as well [7].

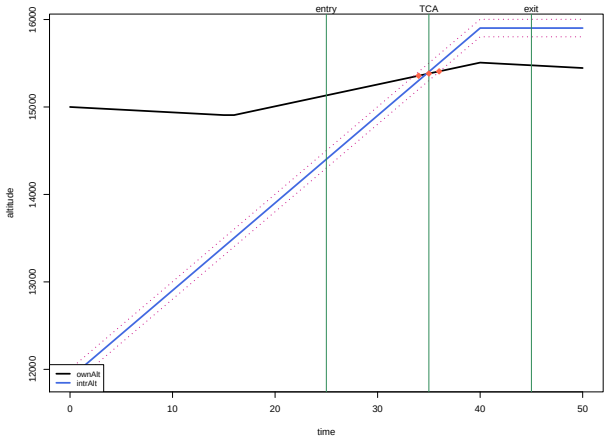
Before explaining the categories, we need the additional concept of a *projected NMAC*. By extrapolating based on the current position and velocities of the planes, the future paths of both planes can be estimated. A projected NMAC exists if an NMAC will occur based on these extrapolated paths.

The intuition behind the categories is as follows:

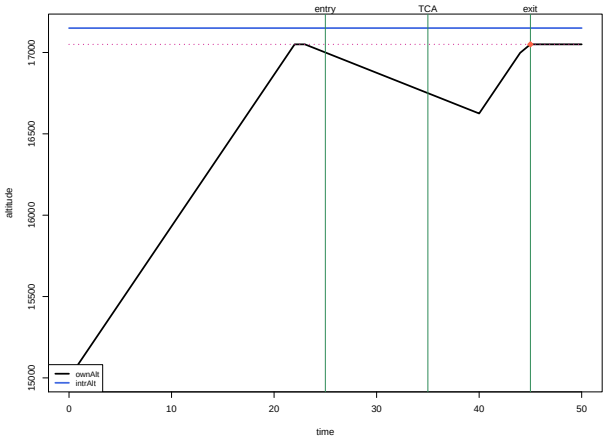
Doomed There is a projected NMAC from the beginning. Furthermore, even if an RA is issued, the plane cannot climb or descend fast enough to avoid an NMAC. This is the only category that was anticipated by the designers. (See Fig. 3a.)

[5] We are working with MIT Lincoln Laboratory to make the description of Tiny TCAS available, but it is not publicly available yet.

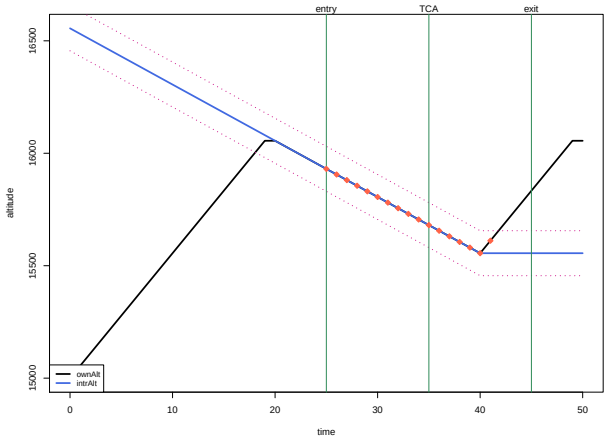
[6] An integration of interval constraint propagation and Simplex has been done within OpenSMT [13, 6]. Unfortunately at the time of this writing, this tool is unavailable.



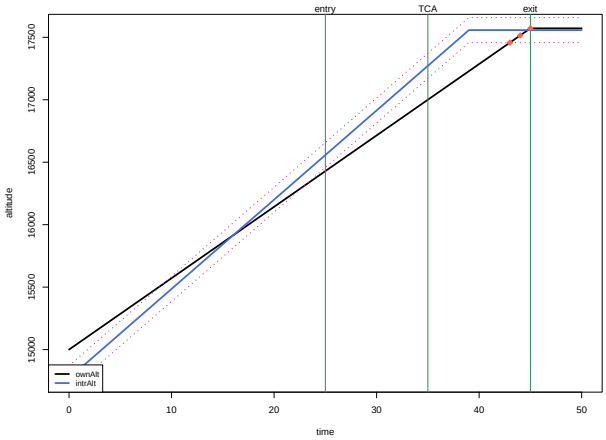
(a) Doomed



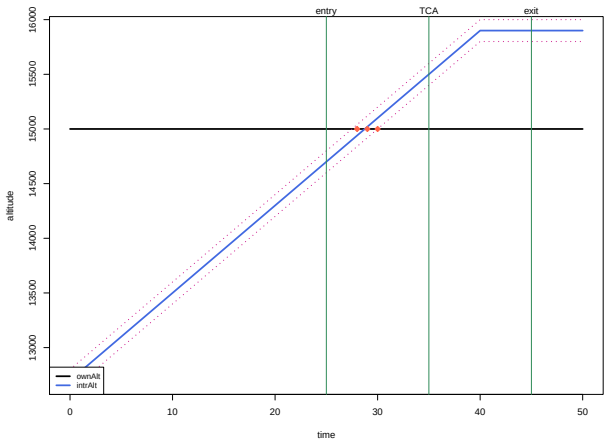
(b) Release



(c) Locked



(d) LockThenLevel



(e) Lazy

Figure 3: Automatically generated visualizations of categories for Tiny TCAS.

	Time (in s)			Memory (in MB)		
k	CVC3	Z3	CVC4	CVC3	Z3	CVC4
20	3.82	3.77	2.65	306	60	73
40	MO	598.48	12.51	MO	486	114
60	MO	47.85	20.52	MO	707	185
80	MO	43.45	21.50	MO	899	327
100	MO	121.91	98.10	MO	1720	331

Table 1: Wall clock time and maximum memory for CVC3, Z3, and CVC4 generating a model for Tiny TCAS with the constraint ¬Doomed ∧ ¬Release ∧ ¬Locked with `rangeRateMag` $= 0.016$. MO = Memory Out ($> 6GB$)

Locked Because of a projected NMAC, Tiny TCAS issues an RA. However, the trajectory projected by following this RA still leads to an NMAC. (See Fig. 3c.)

LockThenLevel Because of a projected NMAC, Tiny TCAS issues an RA. The new projected trajectory does not contain an NMAC. However, the intruder then changes its altitude rate, resulting in an NMAC. (See Fig. 3d.)

Lazy Tiny TCAS does not issue an RA despite an NMAC being projected. This is caused because the criteria for detecting collision threats are unsound. There are two important sub-cases depending on whether the TCA is being estimated correctly or not. (See Fig. 3e.)

Release Because of a projected NMAC, Tiny TCAS issues an RA, resulting in a new path on which an NMAC is not projected. When it appears safe, Tiny TCAS releases the RA. The pilot then changes the altitude rate in response to the RA being released. This then either directly results in an NMAC, or an additional change in altitude by the intruder results in an NMAC. (See Fig. 3b.)

All of the previously mentioned categories are expressible as safety properties. They are not, however, *easily* expressible as state properties, as they require a significant amount of information about the past. The system is augmented with witness variables that capture a sufficient amount of history to express the category. An example of a witness variable is the time an intruder levels off, which is used as a part of projecting the paths. To avoid introducing non-linearity, case splits are done by `transmit`. This also shows the advantage of using a low level tool like `transmit`.

While challenging for current SMT solvers, we have been able to use the EXPLORE procedure effectively to analyze Tiny TCAS. Table 1 shows the running time and memory consumption for a particular formula from our analysis using three SMT solvers and using different values for k.[7] The query is immediately after the categories Doomed, Release and Locked have been discovered, and these categories are being excluded. The SMT solvers represented are CVC3, Z3, and CVC4.[8] To the best of our knowledge, these are the only 3 SMT solvers that can handle the rewriting of the quasi non-linearity in the constraints correctly. CVC3 runs out of memory on every $k > 20$. This is due to the high memory consumption of the Fourier-Motzkin decision procedure for QF_LRA[20]. Z3 and CVC4 both use variants of the simplex method [11], and have significantly better memory performance.

[7] These experiments were run on an a 2.66GHz Intel Core2 Quad with 8GB memory.

[8] We used Z3 version 2.3 (for Linux), CVC4 version "svn co -r1780 `https://subversive.cims.nyu.edu/cvc4/cvc4/branches/arithmetic/preprocess`" with "–rewrite-arithmetic-equalities –enable-arithmetic-propagation" enabled, and CVC3 2.2 with the flag "+model".

8 Related Work

Our use of BMC is quite similar to that found in [2]. Both approaches focus on hybrid systems, reduce the problem to a single SMT query, and find violations of safety properties. Closest to our work on Tiny TCAS is the work presented in [7, 18]. This work focuses on techniques for proving that an alert is always issued to the pilot on all error traces. The scenario considered there is parallel runway approaches where an intruder deviates from a normal approach by banking. Their work proves the existence of conditions under which an aspect of TCAS is guaranteed to issue an alert. Tiny TCAS and our work focuses on both the case where alerts are issued and not issued, as well as errors that come about as part of conflict resolution. Other well known formal analysis on TCAS has focused on guaranteeing conditional safety in the case where both planes are equipped with TCAS [17].

9 Conclusion and Future Work

We have proposed a novel technique for efficient interactive exploration of the error space of a system. The procedure uses SMT-based BMC to generate error traces, and then relies on the user to guide the generation of additional error traces by generalizing and then excluding the current trace. This interactive approach allows the user to quickly find and understand multiple sources of system failure for complex fixed systems. We have used this approach to analyze Tiny TCAS, and have succeeded in identifying four additional error categories beyond those anticipated by the system designer. This shows the potential use and reasonableness of this approach.

The most serious limitation of our work is in the handling of non-linearity. We had to resort to simplification of the model in order to obtain linear (or nearly-linear) formulas. We plan on using Tiny TCAS as a motivating example for developing better techniques for solving quantifier free non-linear real arithmetic. Another interesting possibility is mixing in automated techniques to heuristically help the ANALYZE component of the system. Abstract interpretation techniques for trace abstraction seem the mostly likely candidate for success.

References

[1] Investigation report AX001-1-2/02, May 2004.

[2] AUDEMARD, G., BOZZANO, M., CIMATTI, A., AND SEBASTIANI, R. Verifying industrial hybrid systems with MathSAT. *Electronic Notes in Theoretical Computer Science 119*, 2 (Mar. 2005), 17–32.

[3] BARRETT, C., AND TINELLI, C. CVC3. In *Computer Aided Verification*, W. Damm and H. Hermanns, Eds., vol. 4590 of *Lecture Notes in Computer Science*. Springer, Berlin, 2007, ch. 34, pp. 298–302.

[4] BIERE, A., CIMATTI, A., CLARKE, E., AND ZHU, Y. Symbolic model checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems*, W. Cleaveland, Ed., vol. 1579 of *Lecture Notes in Computer Science*. Springer, Berlin, Mar. 1999, ch. 14, pp. 193–207.

[5] BIERE, A., HEULE, M., VAN MAAREN, H., AND WALSH, T. *Handbook of Satisfiability*. Frontiers in Artificial Intelligence and Applications. IOS Press, Feb. 2009.

[6] BRUTTOMESSO, R., PEK, E., SHARYGINA, N., AND TSITOVICH, A. The OpenSMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, J. Esparza and R. Majumdar, Eds.,

vol. 6015 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2010, ch. 12, pp. 150–153.

[7] CARREÑO, V., AND MUÑOZ, C. Aircraft trajectory modeling and alerting algorithm verification. In *Theorem Proving in Higher Order Logics*, M. Aagaard and J. Harrison, Eds., vol. 1869 of *Lecture Notes in Computer Science*. Springer, Berlin, 2000, ch. 6, pp. 90–105.

[8] CLARKE, E. M., GRUMBERG, O., AND PELED, D. A. *Model Checking*. The MIT Press, Jan. 1999.

[9] DE MOURA, L., AND BJØRNER, N. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and J. Rehof, Eds., vol. 4963 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, ch. 24, pp. 337–340.

[10] DE MOURA, L., AND BJØRNER, N. Bugs, moles and skeletons: Symbolic reasoning for software development. In *Automated Reasoning*, J. Giesl and R. Hähnle, Eds., vol. 6173 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2010, ch. 34, pp. 400–411.

[11] DUTERTRE, B., AND DE MOURA, L. A fast Linear-Arithmetic solver for DPLL(t). In *Computer Aided Verification*, T. Ball and R. Jones, Eds., vol. 4144 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2006, pp. 81–94.

[12] ENDERTON, H., AND ENDERTON, H. B. *A Mathematical Introduction to Logic, Second Edition*, 2 ed. Academic Press, Jan. 2001.

[13] GAO, S., GANAI, M., IVANCIC, F., GUPTA, A., SANKARANARAYANAN, S., AND CLARKE, E. Integrating ICP and LRA solvers for deciding nonlinear real arithmetic. *FMCAD* (2010).

[14] IEEE. *On the formal verification of the TCAS conflict resolution algorithms* (1997), vol. 2.

[15] KUCHAR, J. K., AND DRUMM, A. C. The traffic alert and collision avoidance system. *Lincoln Laboratory Journal 16*, 2 (2007), 277–296.

[16] LAHIRI, S., NIEUWENHUIS, R., AND OLIVERAS, A. SMT techniques for fast predicate abstraction. In *Computer Aided Verification*, T. Ball and R. Jones, Eds., vol. 4144 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2006, ch. 39, pp. 424–437.

[17] LIVADAS, C., LYGEROS, J., AND LYNCH, N. A. High-level modeling and analysis of the traffic alert and collision avoidance system (TCAS). *Proceedings of the IEEE 88*, 7 (July 2000), 926–948.

[18] MUÑOZ, C., CARREÑO, V., DOWEK, G., AND BUTLER, R. Formal verification of conflict detection algorithms. *International Journal on Software Tools for Technology Transfer (STTT) 4*, 3 (May 2003), 371–380.

[19] PLATZER, A., AND CLARKE, E. Formal verification of curved flight collision avoidance maneuvers: A case study. In *FM 2009: Formal Methods*, A. Cavalcanti and D. Dams, Eds., vol. 5850 of *Lecture Notes in Computer Science*. Springer, Berlin, 2009, ch. 35, pp. 547–562.

[20] SCHRIJVER, A. *Theory of Linear and Integer Programming*. Wiley, June 1998.