Minimizing Sequents to Find Modeling Errors in KeYmaera X

Ben Gafford[⋆] and Myra Dotzel[⋆]

Carnegie Mellon University, Pittsburgh PA 15213, USA
gafford@cmu.edu
mdotzel@andrew.cmu.edu

Abstract. Modeling errors can cause misconceptions of what was believed to have been proven versus what was actually proven, resulting in weak and potentially erroneous models. These modeling flaws can be expensive to resolve when they are not realized until later in the implementation process. Furthermore, these mistakes could result in unsafe behavior. Accurately modeling system behavior has safety-critical implications for a number of applications including autonomous vehicles and medical technology. Therefore, specifying erroneous models that fail to meet necessary safety guarantees for the intended system could also have devastating safety consequences. In this paper, we introduce a static analysis for identifying extraneous or over-specified model components for valid **dL** formulas. To this end, the method we develop leverages strategies from metamorphic testing to concoct stronger, mutant instances of a given **dL** formula that we test for validity. A stronger (i.e. more generalized) mutant **dL** formula proving valid likely indicates over-specification of the original model which we relay back to the user as feedback. Contributions of this work include candidate mutations, implementation of the analysis, and robust formal guarantees, e.g., soundness and termination.

1 Introduction

Theorem provers like KeYmaera X [6] are critical for designing and implementing safe systems, because they allow system designers to formally prove safety properties for a given system. As systems become increasingly complex, reasoning about important safety properties can be difficult, and testing may be incomplete or insufficient. Provided tools to develop a model for a given system, prove safety guarantees for that model, and synthesize a controller implementation from that provably correct model, a programmer has end-to-end verification that enforces much confidence in the correctness of a given model.

However, even with perfect tooling, humans are error-prone. As we understand in traditional software engineering, bugs are an inevitable occurrence in any sufficiently complex software system. For modeling languages like **dL**, bugs can exist in models just as they exist in programs written in traditional programming languages. For code written in traditional programming languages,

[⋆] Equal Contribution. Section: 15-824

developers have well-established tools and techniques to minimize programming errors in practice. However, there exists no such tool for modeling languages like **dL**. Users are encouraged to incrementally build their models and to thoroughly examine a given model to identify errors before attempting to prove it [10]. Sometimes, these errors can later be caught in the process of a manual proof. However, with increasingly powerful blackbox automated theorem proving tools this fallback becomes increasingly insufficient. These errors sometimes culminate in unprovable models, which can be frustrating for developers to reckon with. Alternatively, these errors could result in over-constrained models which are provably safe but useless. Over-constrained models are as dangerous as unsound inferences anywhere else in the proof process, because these are essentially an unsound abstraction of a concrete problem. That is, proving safety guarantees for an incorrect model of a physical system could spawn false confidence in a model's design making it difficult for programmers to catch their modeling errors.

While "modeling errors" in general are a fuzzy idea since models may be correct or incorrect depending on the purpose of the model, we argue that certain properties of models are indicative of modeling errors. For example, consider a model for safe automated cruise control. There may be a complex controller that permits acceleration or braking under certain conditions P_{acc} and P_{break}. In the proof, you would expect these conditions for acceleration and deceleration to be necessary to complete the proof. If this is not the case, then these guards serve no purpose in the proof of the model, and their role in the model become suspect. While it is certainly possible that these conditionals are intentionally superfluous in the model, it is more likely that there is something wrong with the model that is making it easier to prove than expected. For example, it may be the case that the domain constraints are too restrictive and the safety property is provable from the continuous system dynamics alone. More generally, we make the claim that modeling errors can be determined by identifying components of a given model that were not necessary in its proof.

This observation motivates the method we present in this paper. In this work, we develop a mutation-based analysis capable of detecting superfluous or over-specified program components. Key contributions of this work include the following:

- a set of candidate mutations and their relation to suggested mutations
- formal guarantees for the proposed analysis including soundness and termination theorems in addition to their respective proofs
- an implementation of the proposed analysis, consisting of both static and dynamic components

2 Topic Overview

The high-level idea of our approach is to identify from the set of all facts introduced throughout a proof which facts were actually necessary for the proof and which facts were over-specified. To this end, our analysis, outlined in fig. 1, first accumulates all facts introduced over the course of the proof by augmenting KeYmaera X to enable tracking this information over Bellerophon tactics [4]. Next, we distinguish facts that could have been used in the proof from those that were actually used in the proof. In order to identify which facts were used, we look to the terminal proof steps of a proof of a given **dL** formula. The process of identifying usable facts varies according to which tactic was used to close a given branch. This process is addressed through dynamic analysis which we discuss in section 6.2. Once we have identified unused facts, we then determine whether a given fact is an artifact of the model or an artifact of the proof, and relay mutation feedback regarding used facts back to the user. While an automated approach to discriminating between model artifacts and proof artifacts would be nice, easy to perform manually, and would not generate false positives, this automated feature is not necessary for the utility of our tool. Nevertheless, this is an interesting result, so we include a preliminary discussion of this algorithm in the appendix.

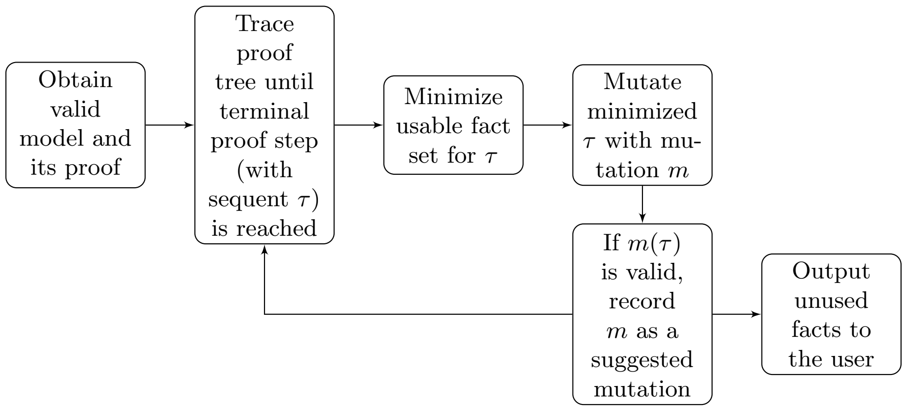
Obtain valid model and its proof
Trace proof tree until terminal proof step (with sequent τ) is reached
Minimize usable fact set for τ
Mutate minimized τ with mutation m
If $m(\tau)$ is valid, record m as a suggested mutation
Output unused facts to the user

Fig. 1. Analysis Flowchart

3 Mutation-Based Analysis

In this section we explore the mutation-based analysis we introduce in this paper. The mutation-based analysis employs a combination of static and dynamic approaches. The static analysis performs a scan of the proof tree starting at the root while working its way up towards the leaves. We call sub-models associated with these leaves *terminal sub-models* and tactics by which these terminal sub-models close *terminal proof tactics*, i.e. auto, QE, id, $\mathbb{R}$. We target our dynamic analysis toward these terminal proof steps where the analysis performs a series of mutation-based tests. After completing these dynamic analyses across all terminal sub-models, we then return the aggregate of successful mutations and their corresponding model components to the user. This feedback informs the user of which parts of their model are over-specified or potentially erroneous.

3.1 rec_analyze

Given a valid model and its proof, our analysis performs a bottom-up scan of the proof tree. The analysis skips all non-terminal proof steps in the proof until it reaches a terminal proof step. At this point, the analysis performs a series of candidate test mutations as defined in section 4 and tags facts with their corresponding successful mutations (if applicable) as determined through the dynamic analysis. For non-terminal proof steps that spawn multiple branches, the suggested mutations from each of these branches are unioned together and the conglomerate is returned. Implementation is facilitated by letting *ret* be a set of references to facts such that the union need only reconcile fact differences across branches. For non-terminal proof steps that introduce new facts, e.g., dI, dC, cut, we track dependencies from the newly-introduced facts and their parent facts. This has the effect of ensuring that suggested mutations for facts introduced during the proof can be easily associated with the corresponding components in the original model.

3.2 Helper Functions

mutate_for_auto_id For every fact in $test\_facts$, $mutate\_for\_auto\_id$ tests every candidate mutation and records which mutations are successful. For the sake of implementation, candidate mutations can be thought of as being stored in a global mutation oracle whose contents we define in definition 1.

mutate_for_QE_real This function is similar to $mutate\_for\_auto\_id$ with the added consideration that QE and $\mathbb{R}$ are numeric proof tactics. This means that QE and $\mathbb{R}$ only prove numeric statements, and so we can limit the scope of testing here accordingly - *M:AssumptionR*, *M:AssumptionW*, *M:PostA*.

4 Candidate Mutations

Here we define a set of candidate mutations that are selected to be tested according to pattern matches. Selected mutations are then tested against valid **dL** formulas in a sort-of guess-and-check procedure in that for each test (as specified in definition 1) that matches a given **dL** formula, we test the resulting mutant **dL** formula for validity. Each mutation rule satisfies the invariant that mutant models are harder to prove than their non-mutant counterparts. If valid, a given mutant model would be capable of proving a more specified post-condition via a more generalized model. This observation strongly motivates our notion of soundness which we formally define in section 5.

```
procedure REC_ANALYZE(model, proof)
    if $proof \neq \emptyset$ then
        $(p : \_) \leftarrow proof$
        $acc \leftarrow \emptyset$
        for all $branch \in p$ do
            $ret \leftarrow branch.factset$
            if $branch = Auto$ then
                $tested\_facts \leftarrow$ MUTATE_FOR_AUTO_ID($statement, test\_facts, auto$)
                return $tested\_facts$
            else if $branch = QE$ then
                $tested\_facts \leftarrow$ MUTATE_FOR_QE_REAL($statement, test\_facts, real$)
                return $tested\_facts$
            else if $branch = R$ then
                $tested\_facts \leftarrow$ MUTATE_FOR_QE_REAL($statement, test\_facts, real$)
                return $tested\_facts$
            else if $branch = Id$ then
                $tested\_facts \leftarrow$ MUTATE_FOR_AUTO_ID($statement, test\_facts, auto$)
                return $tested\_facts$
            else
                $ret \leftarrow$ REC_ANALYZE($model, branch$)
            $acc \leftarrow acc \cup ret$
        return $acc$
    return $empty$
procedure ANALYZE(model, proof)
    $orig\_facts \leftarrow model.facts$
    $tested\_facts \leftarrow$ REC_ANALYZE(model, proof)
    $proof\_artifacts \leftarrow \emptyset$
    $model\_artifacts \leftarrow \emptyset$
    $suggested\_mutations \leftarrow \emptyset$
    for $f \in tested_facts$ do
        if $f.mutations = \emptyset \wedge f \in orig\_facts$ then $model\_artifacts.add(f)$
        else if $f \notin orig\_facts \wedge f.mutations = \emptyset$ then $proof\_artifacts.add(f)$
        else if $f \in orig\_facts \wedge f.mutations \neq \emptyset$ then $suggested\_mutations.add(f)$
    return $(model\_artifacts, proof\_artifacts, suggested\_mutations)$
```

Fig. 2. Mutation-Based Analysis

```
procedure MUTATE_FOR_AUTO_ID(statement, test_facts, tactic)
    ▷ global mutation_oracle ◁
    for $f \in test\_facts$ do
        for $m \in mutation\_oracle$ do
            $mutated\_model \leftarrow m(statement, (test\_facts - statement))$
            if $tactic(mutated\_model)$ then
                $f \leftarrow f.add\_mutation(m)$
    return $test\_facts$
procedure MUTATE_FOR_QE_REAL(statement, test_facts, tactic)
    ▷ $AssumptionR, AssumptionW, PostA \in mutation\_oracle$ (global) ◁
    $numeric\_mutation\_oracle \leftarrow \{AssumptionR, AssumptionW, PostA\}$
    for $f \in test\_facts$ do
        for $m \in numeric\_mutation\_oracle$ do
            $mutated\_model \leftarrow m(statement, (test\_facts - statement))$
            if $tactic(mutated\_model)$ then
                $f \leftarrow f.add\_mutation(m)$
    return $test\_facts$
```

Fig. 3. Mutation Functions

4.1 Syntax

Here, we explain the syntax for **dL** formulas we assume for our mutation-based analysis. In the scope of this paper, we consider **dL** formulas with the syntax defined in fig. 4.

The BNF in fig. 4 specifies syntax for the **dL** formulas we consider here. In the scope of this project, we only consider **dL** formulas that use conjunctive relations (i.e. $S \wedge R$), although we believe a variation of the ideas outlined in this paper to be extensible to other modings, e.g., disjunctive relations. Additionally, we constrain negation to be only admissible for equations and inequalities P' as additional reasoning may be necessary to perform this analysis on **dL** formulas with e.g., diamond modalities and disjunctive relations among **dL** formulas.

$$S, R ::= S_0 \mid S \wedge R \mid S \rightarrow R \mid [\alpha]S$$

$$S_0 ::= e = \tilde{e} \mid e \leq \tilde{e} \mid \neg S_0$$

$$e, \tilde{e} ::= x \mid c \mid e + \tilde{e} \mid e \cdot \tilde{e}$$

Fig. 4. dL Formula Syntax

For the mutations in definition 1, we represent **dL** formulas in the form $A \vdash [\alpha]P$ where A, α, and P are specified in fig. 5.

Here, we define A to be a set of assumptions, P to be a non-empty set of post-conditions, and α to be a hybrid program. If α contains differential dynamics with domain constraints, we denote this set of domain constraints by Q. Additionally, we define $?C$ to be a conditional depending on some set of conditions C, however for the remainder of this paper we use $cond(C)$ to refer to $?C$ containing the set interpretation of C as opposed to the literal **dL** statement. Analogously,

$$A, P, Q, C ::= S$$
$$\alpha, \beta ::= x := e \mid x := * \mid x' = f(x) \& Q \mid ?C \mid \alpha; \beta \mid \alpha \cup \beta \mid \alpha^*$$

Fig. 5. Syntax for **dL** Formula Components

to facilitate our reasoning about mutations in section 5, we choose to interpret A, α, Q, and P as sets of facts and reason about the relations among these facts outside of the given mutations.

Additionally, $p(\cdot, *)$ and $p'(\cdot, *)$ are predicate symbols corresponding to $\cdot < *$ and $\cdot \leq *$ (without loss of generality). Our mutations also make extensive use of uniform substitutions. For example, the series of substitutions in *M:ConditionalW* $m(\cdot) = \{cond(C) \mapsto cond(m'(C))\}$, $m'(\cdot) = \{p(c_1, c_2) \mapsto p'(c_1, c_2)\}$ replaces the original conditional with its mutated conditional. The mutation performed here is specified by m' which simply replaces a condition of the form $c_1 < c_2$ with $c_1 \leq c_2$ in C. We exploit uniform substitutions due to its order-preserving benefits. Specifically, uniform substitution allows us to ensure that our mutations are only changing components of a model that are intended to be mutated and not e.g., the order in which statements occur in the mutated model.

4.2 Design Considerations

Since modeling errors tend to result from over-specification, the overall idea here is to strengthen the model by weakening some of its components. We strengthen the model by applying these mutations and then test whether these mutated models prove under the same tactic. We call these *suggested mutations* as they are mutations under which the model is still valid. These suggested mutations are relayed back to the user as feedback. To this end, the mutations we introduce here make extensive use of the function *test* which takes a valid model $\mathcal{M}$, a terminal proof tactic σ (e.g., auto, QE, id, $\mathbb{R}$), and a time limit $T \in \mathbb{Q}_{\geq 0}$, and returns a Boolean, i.e. *true* if the mutated model was provable under the same tactic in T time, and $false$ otherwise. These tests are embodied by statements of the form $test(A \vdash [\alpha]P, \sigma, T)$. This statement should be interpreted as follows: if the judgement $A \vdash [\alpha]P$ could be proved under tactic σ in a specified time limit T, *test* returns *true*. Otherwise, *test* returns *false*. Note that *test* need only determine whether the mutated **dL** formula is still valid, meaning that *test* could evaluate the mutated **dL** formula via any terminal proof tactic and not just the tactic that proved in initial terminal **dL** formula. For sake of efficiency, however, we *test* a mutated **dL** formula using the same tactic used to prove its initial **dL** formula.

4.3 Mutations

Here, we discuss the mutations introduced in definition 1. *M:AssumptionR* removes an assumption and checks whether the mutant **dL** formula is still valid.

M:AssumptionW weakens an assumption by replacing a strict inequality with its non-strict version and performs a similar check for validity. *M:DomConstrR* removes a domain constraint and *M:DomConstrW* weakens a domain constraint by taking an approach analogous to *M:AssumptionW*. For example, this could have the effect of making initially non-overlapping domain constraints overlapping, a feature necessary to ensure smooth and continuous dynamics that realistically model the desired physical system. *M:ConditionalR* removes a condition. If C has only one condition c, *M:ConditionalR* has the effect of unconditionally executing the branch previously constrained by c. Otherwise, *M:ConditionalR* has the effect of removing a condition while executing that branch conditionally under the remaining set of conditions. *M:ConditionalW* weakens a conditional via the method previously described for *M:AssumptionW* Lastly, *M:PostA* adds a post-condition. For simplicity, we define this new postcondition to be a non-constant assumption (i.e. $a \notin const(A)$) that is not already in P, although in reality there could be much liberty in making this choice.

Definition 1. *Define test* : $(\mathcal{M} \to \sigma \to T) \mapsto$ *Boolean*, $p(\cdot, *) \mapsto (\cdot < *)$, *and* $p'(\cdot, *) \mapsto (\cdot \leq *)$.

1. ***M:AssumptionR.*** *Given* $a \in A$, $m(\cdot) = \{\cdot \mapsto \cdot \setminus \{a\}\}$, *and* $test(A \vdash [\alpha]P, \sigma, T)$, $test(m(A) \vdash [\alpha]P, \sigma, T)$.
2. ***M:AssumptionW.*** *Given* $p(a_1, a_2) \in A$, $m(\cdot) = \{p(a_1, a_2) \mapsto p'(a_1, a_2)\}$, *and* $test(A \vdash [\alpha]P, \sigma, T)$, $test(m(A) \vdash [\alpha]P, \sigma, T)$.
3. ***M:DomConstrR.*** *Given* $Q \in \alpha$, $q \in Q$, $m(\cdot) = \{Q \mapsto Q \setminus \{q\}\}$, *and* $test(A \vdash [\alpha]P, \sigma, T)$, $test(A \vdash [m(\alpha)]P, \sigma, T)$.
4. ***M:DomConstrW.*** *Given* $Q \in \alpha$, $p(q_1, q_2) \in Q$, $m(\cdot) = \{Q \mapsto m'(Q)\}$, $m'(\cdot) = \{p(q_1, q_2) \mapsto p'(q_1, q_2)\}$, *and* $test(A \vdash [\alpha]P, \sigma, T)$, $test(A \vdash [m(\alpha)]P, \sigma, T)$.
5. ***M:ConditionalR.*** *Given* $c \in C$, $cond(C) \in \alpha$, $m(\cdot) = \{cond(C) \mapsto cond(C \setminus \{c\})\}$, *and* $test(A \vdash [\alpha]P, \sigma, T)$, $test(A \vdash [m(\alpha)]P, \sigma, T)$.
6. ***M:ConditionalW.*** *Given* $p(c_1, c_2) \in C$, $cond(C) \in \alpha$, $m(\cdot) = \{cond(C) \mapsto cond(m'(C))\}$, $m'(\cdot) = \{p(c_1, c_2) \mapsto p'(c_1, c_2)\}$, *and* $test(A \vdash [\alpha]P, \sigma, T)$, $test(A \vdash [m(\alpha)]P, \sigma, T)$.
7. ***M:PostA.*** *Given* $a \in A$, $a \notin P$, $a \notin const(A)$, $m(\cdot) = \{P \mapsto P \cup \{a\}\}$, *and* $test(A \vdash [\alpha]P, \sigma, T)$, $test(A \vdash [\alpha]m(P), \sigma, T)$.

5 Analysis Properties

Our approach guarantees that the application of a given suggested mutation to a valid **dL** formula would not affect the validity of that **dL** formula while simultaneously generalizing that **dL** formula. This idea motivates our notion of *soundness*. To this end, we observe that if a mutation is suggested, it must be that the mutant **dL** formula passed *test*, and hence the mutant **dL** formula is valid. Therefore, we define the following semantics for suggested mutations which will aid in our proof of soundness.

Here, we again make use of the predicates $p(\cdot, *) \mapsto (\cdot < *)$ and $p'(\cdot, *) \mapsto (\cdot \leq *)$, and mutation functions $m_1(\cdot) \ldots m_7(\cdot)$ which are given by:

1. $m_1(\cdot) = \{\cdot \mapsto \cdot \setminus \{a\}\}$
2. $m_2(\cdot) = \{p(a_1, a_2) \mapsto p'(a_1, a_2)\}$
3. $m_3(\cdot) = \{Q \mapsto Q \setminus \{q\}\}$
4. $m_4(\cdot) = \{Q \mapsto m'_4(Q)\}$, $m'_4(\cdot) = \{p(q_1, q_2) \mapsto p'(q_1, q_2)\}$
5. $m_5(\cdot) = \{cond(C) \mapsto cond(C \setminus \{c\})\}$
6. $m_6(\cdot) = \{cond(C) \mapsto cond(m'_6(C))\}$, $m'_6(\cdot) = \{p(c_1, c_2) \mapsto p'(c_1, c_2)\}$
7. $m_7(\cdot) = \{P \mapsto P \cup \{a\}\}$

Interpretation of the semantics in fig. 6 is analogous to the interpretation of candidate mutations. The key difference here is that the *test* for a given mutant **dL** formula has returned true which motivates our ability to express static semantics for suggested mutations in this manner. From this observation, we can unwrap these *test* statements and deal with sequents directly. In doing so, we preserve the time limit T. That is, it is implicit that a sequent under a suggested mutation proved under time limit T, and for this reason, we elide this fact from the following representation and simply mention it here.

$$\textsc{Sm:AssumptionR}\quad \frac{A \vdash [\alpha]P}{m_1(A) \vdash [\alpha]P}\ (a \in A) \qquad \textsc{Sm:AssumptionW}\quad \frac{A \vdash [\alpha]P}{m_2(A) \vdash [\alpha]P}\ (p(a_1, a_2) \in A)$$

$$\textsc{Sm:DomConstrR}\quad \frac{A \vdash [\alpha]P}{A \vdash [m_3(\alpha)]P}\ (Q \in \alpha, q \in Q)$$

$$\textsc{Sm:DomConstrW}\quad \frac{A \vdash [\alpha]P}{A \vdash [m_4(\alpha)]P}\ (Q \in \alpha, p(q_1, q_2) \in Q)$$

$$\textsc{Sm:ConditionalR}\quad \frac{A \vdash [\alpha]P}{A \vdash [m_5(\alpha)]P}\ (c \in C, cond(C) \in \alpha)$$

$$\textsc{Sm:ConditionalW}\quad \frac{A \vdash [\alpha]P}{A \vdash [m_6(\alpha)]P}\ (p(c_1, c_2) \in C, cond(C) \in \alpha)$$

$$\textsc{Sm:PostA}\quad \frac{A \vdash [\alpha]P}{A \vdash [\alpha]m_7(P)}\ (a \notin const(A), a \in A, a \notin P)$$

Fig. 6. Semantics for Suggested Mutations

5.1 Soundness

Soundness here is motivated by the idea that applying suggested mutations spawn *harder-to-prove*, yet valid, **dL** formulas. A mutated **dL** formula $\mathcal{M}_m$ is harder-to-prove if it satisfies one of the following criteria: 1) $\mathcal{M}_m$ contains a more restrictive post-condition to be proved under the same fact set, or 2) $\mathcal{M}_m$ contains the same post-condition to be proved under a more restrictive fact set. We formalize this notion by saying that mutated **dL** formulas subsume initial **dL** formulas. To this end, we introduce the predicate $A \preccurlyeq B$. If $A \preccurlyeq B$, we say B *subsumes* A. Letting $\mathcal{M}_i$ be the initial **dL** formula and $\mathcal{M}_m$ be the *m-mutated* ***dL*** *formula*, we claim that a suggested mutation is sound if $\mathcal{M}_i \preccurlyeq \mathcal{M}_m$. The m-mutated **dL** formula $\mathcal{M}_m$ is the **dL** formula that results from applying suggested mutation m to $\mathcal{M}_i$. Choices for m are specified in fig. 6.

From this, 1) and 2) can be equivalently restated as $\mathcal{P}_m \preccurlyeq \mathcal{P}_i$ and $\mathcal{F}_i \preccurlyeq \mathcal{F}_m$ where $\mathcal{P}_i$ and $\mathcal{P}_m$ are sets of post-conditions, and $\mathcal{F}_i$ and $\mathcal{F}_m$ are *fact sets*. Fact sets can be intuitively understood as collections of assumptions and hybrid program components in α. When a fact set $\mathcal{F}_m$ subsumes a fact set $\mathcal{F}_i$, we say the α-span of a hybrid program α_m under some assumption set A_m is a superset of the α-span of a hybrid program α_i under some assumption set A_i. Here, $\mathcal{F}_i$ is the fact set associated with initial **dL** formula $\mathcal{M}_i$ and $\mathcal{F}_m$ is the fact set associated with mutated **dL** formula $\mathcal{M}_m$ (and analogously for $\mathcal{P}_i$ and $\mathcal{P}_m$). Additionally, A_i is the set of assumptions associated with $\mathcal{M}_i$ and α_i is the hybrid program associated with $\mathcal{M}_i$, and analogously for A_m and α_m. When we write $[\![\alpha_i]\!]_{A_i}$, we mean the set of all states reachable by α_i under assumptions A_i, and analogously for $[\![\alpha_m]\!]_{A_m}$. For post-conditions, $[\![P_i]\!]$ denotes the set of all states for which formula P_i is true, and analogously for P_m. With this in mind, we define subsumption for fact sets and post-conditions.

Definition 2. *$\mathcal{F}_i \preccurlyeq \mathcal{F}_m$ if and only if $[\![\alpha_i]\!]_{A_i} \subset [\![\alpha_m]\!]_{A_m}$.*

Definition 3. *$\mathcal{P}_m \preccurlyeq \mathcal{P}_i$ if and only if $[\![P_m]\!] \subset [\![P_i]\!]$.*

Lemma 1. *If $\mathcal{F}_i = \mathcal{F}_m$, then $\mathcal{F}_i \preccurlyeq \mathcal{F}_m$ and $\mathcal{F}_m \preccurlyeq \mathcal{F}_i$. Analogously, if $\mathcal{P}_i = \mathcal{P}_m$, then $\mathcal{P}_i \preccurlyeq \mathcal{P}_m$ and $\mathcal{P}_m \preccurlyeq \mathcal{P}_i$.*

Proof. First, $\mathcal{F}_i = \mathcal{F}_m$ implies that $[\![\alpha_i]\!]_{A_i} = [\![\alpha_m]\!]_{A_m}$. Then, by the axiom of extension, $[\![\alpha_i]\!]_{A_i} \subset [\![\alpha_m]\!]_{A_m}$ and $[\![\alpha_m]\!]_{A_m} \subset [\![\alpha_i]\!]_{A_i}$. By definition, $\mathcal{F}_i \preccurlyeq \mathcal{F}_m$ and $\mathcal{F}_m \preccurlyeq \mathcal{F}_i$. Proving $\mathcal{P}_i = \mathcal{P}_m$ implies $\mathcal{P}_i \preccurlyeq \mathcal{P}_m$ and $\mathcal{P}_m \preccurlyeq \mathcal{P}_i$ follows a similar argument. □

Definition 4. *$\mathcal{M}_i \preccurlyeq \mathcal{M}_m$ if and only if $\mathcal{P}_m \preccurlyeq \mathcal{P}_i$ and $\mathcal{F}_i \preccurlyeq \mathcal{F}_m$.*

With these properties, we can now begin formally reasoning about soundness.

Theorem 1. *Let m be a suggested mutation, $\mathcal{M}_i$ be an initial valid* ***dL*** *formula, and $\mathcal{M}_m$ be the m-mutated* ***dL*** *formula. Then, $\mathcal{M}_i \preccurlyeq \mathcal{M}_m$ and $\mathcal{M}_m$ is valid.*

Proof. To prove that $\mathcal{M}_i \preccurlyeq \mathcal{M}_m$, we use rule induction on the suggested mutations in fig. 6.

Case Sm:AssumptionR. Here, we denote $A \vdash [\alpha]P$ as $\mathcal{M}_i$ and $m_1(A) \vdash [\alpha]P$ as $\mathcal{M}_m$. Now assume that $\mathcal{M}_i$ is valid and $a \in A$. Then by Sm:AssumptionR, $\mathcal{M}_m$ is valid. To show that $\mathcal{M}_i \preccurlyeq \mathcal{M}_m$, we endeavor to prove that $\mathcal{P}_m \preccurlyeq \mathcal{P}_i$ and $\mathcal{F}_i \preccurlyeq \mathcal{F}_m$. From m_1, we can define $A' = m_1(A) = A \setminus \{a\}$, $(\cap_i [\![A_i]\!]) \subset (\cap_j [\![A'_j]\!])$ for $A_i \in A$ and $A'_j \in A'$. For the case where A contains only one assumption, $(\cap_i [\![A_i]\!]) \subset (\cap_j [\![A'_j]\!])$ follows from the fact that $[\![\emptyset]\!] = S$ vacuously. Then, $[\![\alpha_i]\!]_A \subset [\![\alpha_m]\!]_{A'}$. Then by definition, $F_i \preccurlyeq F_m$. Since $\mathcal{P}_m = \mathcal{P}_i$ under Sm:AssumptionR, it follows by lemma 1 that $\mathcal{P}_m \preccurlyeq \mathcal{P}_i$. Then by definition, $\mathcal{M}_i \preccurlyeq \mathcal{M}_m$.

Case Sm:AssumptionW. Let $\mathcal{M}_i$ be $A \vdash [\alpha]P$ and $\mathcal{M}_m$ be $m_2(A) \vdash [\alpha]P$. Assume that $\mathcal{M}_i$ is valid and $p(a_1, a_2) \in A$. Then by Sm:AssumptionW, $\mathcal{M}_m$ is valid. By definition of m_2, p, and p' we have $p(a_1, a_2) \subset p'(a_1, a_2)$. Then we can define $A' = m_2(A)$, and hence, $(\cap_i [\![A_i]\!]) \subset (\cap_j [\![A'_j]\!])$ for $A_i \in A$ and $A'_j \in A'$. Therefore, $[\![\alpha_i]\!]_A \subset [\![\alpha_m]\!]_{A'}$. Then by definition, $\mathcal{F}_i \preccurlyeq \mathcal{F}_m$. Again, $\mathcal{P}_m = \mathcal{P}_i$ under Sm:AssumptionW, so $\mathcal{P}_m \preccurlyeq \mathcal{P}_i$ by lemma 1. Therefore, by definition, $\mathcal{M}_i \preccurlyeq \mathcal{M}_m$.

Case Sm:DomConstrR. Let $\mathcal{M}_i$ be $A \vdash [\alpha]P$ and $\mathcal{M}_m$ be $A \vdash [m_3(\alpha)]P$. Assume that $\mathcal{M}_i$ is valid, $Q \in \alpha$, and $q \in Q$. Let $Q' = m_3(Q) = Q \setminus \{q\}$. Then by Sm:DomConstrR, $\mathcal{M}_m$ is valid. Now note that $Q' = Q \setminus \{q\}$ implies that $(\cap_i [\![Q_i]\!]) \subset (\cap_j [\![Q'_j]\!])$ for $Q_i \in Q$ and $Q'_j \in Q'$. Note that for the case where Q only contains one domain constraint, we have $(\cap_i [\![Q_i]\!]) \subset (\cap_j [\![Q'_j]\!])$ due to the fact that $[\![\emptyset]\!] = S$ vacuously. Then by definition, $[\![\alpha_i]\!]_A \subset [\![\alpha_m]\!]_{A'}$, and hence, $\mathcal{F}_i \preccurlyeq \mathcal{F}_m$. $\mathcal{P}_m = \mathcal{P}_i$ under Sm:DomConstrR, so $\mathcal{P}_m \preccurlyeq \mathcal{P}_i$ by lemma 1. So, $\mathcal{M}_i \preccurlyeq \mathcal{M}_m$ by definition.

Case Sm:DomConstrW. Let $\mathcal{M}_i$ be $A \vdash [\alpha]P$ and $\mathcal{M}_m$ be $A \vdash [m_4(\alpha)]P$, and assume that $\mathcal{M}_i$ is valid, $Q \in \alpha$, and $p(q_1, q_2) \in Q$. Then define $Q' = m_4(Q)$. Then by Sm:DomConstrW, $\mathcal{M}_m$ is valid. By definition of p' and p, $p(q_1, q_2) \subset p'(q_1, q_2)$, and hence $(\cap_i [\![Q_i]\!]) \subset (\cap_j [\![Q'_j]\!])$. Now let $w \in [\![\alpha_i]\!]_{A_i}$ and note that $(\cap_i [\![Q_i]\!]) \subset (\cap_j [\![Q'_j]\!])$ implies $w \in [\![\alpha_m]\!]_{A_m}$, and hence $\mathcal{F}_i \preccurlyeq \mathcal{F}_m$. Then $\mathcal{M}_i \preccurlyeq \mathcal{M}_m$ follows by analogous reasoning as presented in the previous argument.

Case Sm:ConditionalR. Let $\mathcal{M}_i$ be $A \vdash [\alpha]P$ and $\mathcal{M}_m$ be $A \vdash [m_5(\alpha)]P$. Suppose that $\mathcal{M}_i$ is valid, $c \in C$, and $\mathit{cond}(C) \in \alpha$. Then, define $C' = m_5(C) = C \setminus \{c\}$. Then by Sm:ConditionalR, $\mathcal{M}_m$ is valid. $C' = C \setminus \{c\}$ implies that $C' \subset C$, and hence, $(\cap_i [\![C_i]\!]) \subset (\cap_j [\![C'_j]\!])$ for $C_i \in C$ and $C'_j \in C'$. Note that for the edge case where C consists of only one condition, $(\cap_i [\![C_i]\!]) \subset (\cap_j [\![C'_j]\!])$ follows by the fact that $[\![\emptyset]\!] = S$ vacuously. The conclusion again follows by reasoning analogous to the previous argument.

Case Sm:ConditionalW. Let $\mathcal{M}_i$ be $A \vdash [\alpha]P$ and $\mathcal{M}_m$ be $A \vdash [m_6(\alpha)]P$. Suppose that $\mathcal{M}_i$ is valid, $p(c_1, c_2) \in C$, and $\mathit{cond}(C) \in \alpha$, and let $\alpha' = m_6(\alpha)$. Then Sm:ConditionalW implies that $\mathcal{M}_m$ is valid. By definition of p' and p, $p(c_1, c_2) \subset p'(c_1, c_2)$. Then $(\cap_i [\![C_i]\!]) \subset (\cap_j [\![C'_j]\!])$ for $C_i \in C$ and $C'_j \in C'$, and hence, $\mathcal{M}_i \preccurlyeq \mathcal{M}_m$ follows by the previous argument.

Case Sm:PostA. Define $\mathcal{M}_i$ to be $A \vdash [\alpha]P$ and let $\mathcal{M}_m$ be $A \vdash [\alpha]m_7(P)$. Suppose that $\mathcal{M}_i$ is valid, $a \in A$, and $a \notin P$, and let $P' = m_7(P) = P \cup \{a\}$. Then

Sm:PostA implies that $\mathcal{M}_m$ is valid. $\mathcal{F}_i \preccurlyeq \mathcal{F}_m$ follows from the fact that A, α, and Q are unmutated under Sm:PostA. So $\mathcal{F}_i = \mathcal{F}_m$, and hence $\mathcal{F}_i \preccurlyeq \mathcal{F}_m$ follows by lemma 1. So it remains to be shown that $\mathcal{P}_m \preccurlyeq \mathcal{P}_i$, i.e. $S(\mathcal{P}_m) \subset S(\mathcal{P}_i)$. For this, we observe that $P' = P \cup \{a\}$ implies that $(\cap_i [\![P'_i]\!]) \subset (\cap_j [\![P_j]\!])$ for $P'_i \in P'$ and $P_j \in P$, and so $[\![P']\!] \subset [\![P]\!]$. By definition, $\mathcal{P}_m \preccurlyeq \mathcal{P}_i$. Then $\mathcal{M}_i \preccurlyeq \mathcal{M}_m$ by definition.

□

The analysis we present here yields valuable feedback for models, however it cannot provide this feedback if it does not terminate. Thus, proving that our analysis terminates is an essential property. This is what the following theorem captures.

5.2 Termination

Theorem 2. *The analysis in fig. 2 terminates.*

Proof. If provided an invalid **dL** formula, the analysis terminates vacuously. Otherwise, the analysis is provided a valid **dL** formula along with its proof P containing finitely-many proof steps. For this, we let p be the total number of proof steps in P, and label each proof step with some $k \in [1..p]$. Then, we define n_1, the number of proof steps remaining in the static analysis, as $n_1 = p - k$.

The dynamic analyses occur at terminal proof steps over terminal sub-models for which there are a finite number of facts. Additionally, there is a finite number of candidate mutations each of which taking at most $T \in \mathbb{Q}_{\geq 0}$ time to test for each of these terminal sub-models. To this end, we define f to be the total number of mutable facts in a terminal sub-model and label each mutable fact with a number $l \in [1..f]$. Analogously, we define m to be the total number of candidate mutations and label each mutation with some number $n \in [1..m]$. Then, we define n_2 to be the number of remaining mutable facts in a sub-model, and n_3 to be the remaining amount of time to test mutations for a given mutable fact of a terminal sub-model. We define n_2 and n_3 as follows:

$$n_2 = \begin{cases} 0 & \text{non-terminal proof step} \\ f - l & \text{terminal proof step} \end{cases}$$

$$n_3 = \begin{cases} 0 & \text{non-terminal proof step} \\ (m - n) * T & \text{terminal proof step} \end{cases}$$

The tuple (n_1, n_2, n_3) lexicographically decreases throughout the analysis, and hence the analysis terminates by the well-ordering principle[1]. □

[1] Unlike other quantities used in the above proof, $T \in \mathbb{Q}_{\geq 0}$ is not a natural number. However this does not threaten our termination via well-ordering argument. That is, the rational numbers are countable (a standard fact that could be verified via e.g., establishing a one-to-one correspondence with $\mathbb{N}$). With this fact, we can show that n_3 terminates by the well-ordering principle. From this, (n_1, n_2, n_3) terminates by the well-ordering principle.

5.3 Discussion

The idea that mutated **dL** formulas subsume original **dL** formulas makes intuitive sense. Mutations strive to strengthen models by generalizing over-specified fact sets and strengthening post-conditions when possible. The behavior exhibited by subsumption in this context closely parallels the relationship specified by subtyping over functions, another example usage of subsumption. To this end, it seems plausible that valid **dL** formulas $\mathcal{M}_i$ and $\mathcal{M}_m$ could be rewritten as functions from post-conditions ($\mathcal{P}_i$, $\mathcal{P}_m$) to their proofs (i.e. fact sets $\mathcal{F}_i$, $\mathcal{F}_m$). The correspondence between programs and proofs is warranted by the Curry-Howard isomorphism for **CdGL** which is specified in prior work [1]. This result establishes the correspondence between **CdGL** winning strategies (for box and diamond modalities) and proofs. Since **CdGL** is strictly more expressive than **dL**, an analogous correspondence would seem to apply here too. From this apparent symbiotic relationship, we observe that $\mathcal{P}_m \preccurlyeq \mathcal{P}_i$ and $\mathcal{F}_i \preccurlyeq \mathcal{F}_m$ imply $(\mathcal{P}_i \rightarrow \mathcal{F}_i) \preccurlyeq (\mathcal{P}_m \rightarrow \mathcal{F}_m)$. The respective contravariant and covariant relations witnessed here parallel similar relations observed by subtyping over function types which has the property that $A_2 <: A_1$ and $B_1 <: B_2$ imply $(A_1 \rightarrow B_1) <: (A_2 \rightarrow B_2)$. From this observation, we offer the following lemma.

Lemma 2. *If $\mathcal{P}_m \preccurlyeq \mathcal{P}_i$ and $\mathcal{F}_i \preccurlyeq \mathcal{F}_m$, then $(\mathcal{P}_i \rightarrow \mathcal{F}_i) \preccurlyeq (\mathcal{P}_m \rightarrow \mathcal{F}_m)$.*

Proof. From $\mathcal{P}_m \preccurlyeq \mathcal{P}_i$ and $\mathcal{F}_i \preccurlyeq \mathcal{F}_m$, we have $[\![\alpha_i]\!]_{A_i} \subset [\![\alpha_m]\!]_{A_m}$ and $[\![P_m]\!] \subset [\![P_i]\!]$ (by definition). Then, $[\![P_i]\!]^c \subset [\![P_m]\!]^c$, and hence, $[\![P_i]\!]^c \cup [\![\alpha_i]\!]_{A_i} \subset [\![P_m]\!]^c \cup [\![\alpha_m]\!]_{A_m}$. By semantics of $\rightarrow$ and by definition of $\preccurlyeq$, $\mathcal{P}_i \rightarrow \mathcal{F}_i \preccurlyeq \mathcal{P}_m \rightarrow \mathcal{F}_m$. $\square$

6 Implementation

We chose to implement our tool into KeYmaera X in order to make use of existing infrastructure and in order to allow for ease of use. Our tool introduces three new Bellerophon [4] proof tactics, **minQE**, **minAuto**, and **minAutoXtreme** which compute the minimum sequent needed to complete the proof using the corresponding tactic. Our tool also introduces an automated analysis for proof trees (as represented by Bellerophon tactics) containing our minimizing tactics. The proof tree analysis identifies facts which were introduced in the proof but left unused to close branches, and provides these unused facts to the user.

6.1 Gathering usable facts

To gather usable facts, we implemented an analysis over the proof tree (as represented by a Bellephoron tactic) which collects all observed sequents at each step of the proof and collects all used sequents, as determined by the minimum sequents emitted by the minimizing tactics described below. We decompose these sequents into sets of atomic formulas which have no subformulas. We then check to see which atomic formulas have been observed but never used to close a branch, and relay this information to the user. Our approach is simple and facts do not carry metadata about their unique identity and provenance, however a more complete approach would capture this information to be more precise about which facts were used and not used. This idea is explored theoretically in the appendix of this paper. We handle modal formulas like box and diamond by scraping the constraints out of the program (i.e., Test and domain constraint formulas), and using these facts in place of the modal formulas themselves.

6.2 Dynamic Analysis

To determine which facts of a sequent were necessary to complete a given proof, we need to take different approaches depending on which tactic is used to close a branch. We refer to such tactics as *terminal tactics* and we consider only **QE** and **auto**, since these can subsume all other terminal tactics. The approaches for identifying which facts have been used are described below.

minQE Because **QE** if offloaded to black box solvers such as Mathematica or Z3, we do not know which components of a sequent were needed for a proof. If this information were available, determining the minimum sequent from a **QE** tactic would be trivial. Since usage information is not readily available, we employ a delta-debugging-style strategy to strategically removing facts from a given provable sequent until it is no longer provabale. Heuristic-style approaches could be used to statically identify facts which could not possibly be used to prove a sequent, thereby reducing the search space, but we did not have time for this optimization and found the naive, brute-force solution sufficient for our purposes. We attempted to use existing infrastructure in KeYmaera X (**smartHide**) to automatically infer these dependencies and infer irrelevant formulas, however upon extensive testing we found that these tactics did not retain validity in practice. Since **QE** doesn't guarantee termination, we applied a timeout of 10 seconds. We also found that textbfautois a far more efficient and powerful proof tactic than **QE**, and internally use textbfautoin place of **QE** for **minQE**.

minAuto **auto** is a powerful proof tactic introduced by KeYmaera X to perform automated theorem proving. While **auto** is very powerful, for performance reasons it is also stateless and introspection is very difficult and unstable. Therefore, as with **QE**, we chose to treat **auto** as a black box in order to determine which parts of a sequent were used in a proof. Unlike **QE**, however, the structure of formulas can be more complex, particularly with respect to the Box (always) and Diamond (eventually) modalities as they contain hybrid programs. Because of this, we define mutation rules over **dL** formulas of the typical form and apply these until we reach a weakened **dL** formula that is no longer provable. It is possible to define such mutation rules over arbitrary **dL** formulas, but this is outside of the scope of our project. Since textbfautosimilarly does not guarantee termination, we applied a timeout of 10 seconds as well.

minAutoXtreme minAuto provides individual weakenings that are provable, however in the case of over-constrained models, it is often the case that a single term in a formula (e.g., an always-false program conditional) results in many provable weakenings. As shown in Figure 7, a simple error can over-constrain execution paths in your model, and leave you with many unused facts. In these cases in particular, it is helpful to see which compositions of mutations are valid. We implement this recursively, and continue to weaken the model while we are still able to do so.

Limitations Our implementation is limited in scope for reasons discussed previously. As noted above, our analysis of used facts can lead to redundant facts being mislabeled as used, when that was not the case. Our implementation is also very poorly optimized for performance. Currently, all candidate mutations are evaluated individually and there is a fair amount of avoidable repeated computation. In practice, these mutations could be combined metamorphically to concoct more complicated tests from these simple unit mutations.

Our approach is also implemented in the backend and we have not taken the time to pass unused facts to the web frontend through the REST API. While minimum sequents produced by closed branches can be viewed through the console, we have not found a way to produce proof tree output to the console through the web UI. Currently, this functionality is only visible in tests of imported files.

Our implementation of timeouts is relatively invasive and will sometimes lead to strange errors in KeYmaera X's execution. However, we find that the timeouts are necessary for hard-to-close branches.

Loop invariants also pose issues because they oftentimes make assumptions used even if they weren't otherwise necessary. For future work, we could extend our approach to cover loop invariants in order to weaken them as well. Loop invariants are more difficult to mutate, however, because they are not actually expressions in **dL**.

7 Related Work

7.1 Identifying modeling errors

Another approach to identifying modeling errors is Mitsch & Platzer [9]. Here, the authors present an analysis capable of synthesizing monitors which are able to identify at runtime whether or not a model conforms to the real-world (or simulated) environment. However, this process is far more expensive as it could call for a full system implementation and simulator. This process could also result in difficult-to-interpret false positives which could result in confusing feedback. An example of other model development tools would be that of minimum unsatisfiable cores [11], which can help to make modeling mistakes that result in unprovable formulas obvious.

Fig. 7. Code

```
1    (x<=H & v=0 & x>=0) &
2    (g>0 & 1>=c&c>=0 & r>=0)
3   ->
4    [
5      {
6        {?x=0; v:=-c*v; ?x!=0;}
7        {{x'=v,v'=-g-r*v^2&x>=0&v>=0} ++ {x'=v,v'=-g+r*v^2&x
       >=0&v<=0}}
8      }* @invariant(2*g*x<=2*g*H-v^2&x>=0)
9    ] (0<=x&x<=H)
10
11   // Unused: (x<=H & v=0 & x>=0) & (g>0 & 1>=c&c>=0 & r>=0),
       x>=0&v>0
```

Listing 1.1. A model with a single character transposition error that renders the formula trivially valid. The only difference between this model and a useful model is a missing Choice operator on line 6.

7.2 Testing in other domains

Our work is derived primarily from metamorphic testing [2], which relies on the use of metamorphic relations to define test oracles over multiple executions of a given program. In our case, we are mutating dL formulas rather than inputs, and checking these metamorphic relations using the automated theorem prover. Our work is also strongly inspired by test case minimization techniques. In particular, delta-debugging [12] has been used in test case reduction to isolate the part of the input that causes a given fault. The strategy in delta-debugging is to remove parts of a failing input while the input remains to be failing. Similarly, our approach strategically removes parts of our sequent while that sequent remains provable in order to minimize that sequent. There also exist static analysis tools for bug detection in other languages [3]. For example, cppcheck [8] generates bug reports for C++ programs at compile-time, which is similar to the dynamic analysis approach we plan to take when integrating our analysis with KeYmaera X. Alloy [7] is an object modeling language that can be used to model designs and prove properties about those designs. The Alloy Analyzer includes a tool [11] to find an unsatisfiable core of a larger, unsatisfiable model. Fulton & Platzer [5] describes the use of mutations to hybrid models in order to synthesize candidate models which are then validated against the real-world-environment at runtime.

8 Conclusion and Future Work

Models are by their very essence simplifying abstractions of a problem domain, but even such simplifications can still grow to be intractably complex and error prone. Keeping vigilant while modeling remains good advice, however further tooling is warranted to facilitate programmers' interactions with and understanding of their models. In this paper, we introduce a novel framework for automated testing of models in KeYmaera X. We contemplate the notion of modeling errors, which can be difficult to address in general, but easy to address in a principled way in many cases. Hence, we formalize these observations heuristically through as set of mutation rules and provide a static analysis for our novel technique. For this static analysis, we provide both soundness and termination guarantees in addition to a discussion of other theory surrounding the topic. Additionally, we provide a robust implementation of the analysis to show its usability. Areas of future work extending the scope of this analysis to **dL** formulas encompassing other syntax (e.g., disjunctions, negations, diamond modalities) and to **dGL**. Additionally, improving the efficiency of certain aspects of our analysis is another area of future exploration. Overall, developing tools that facilitate writing correct and realistic models is crucial to offsetting the burden of formal approaches to verifying software.

References

1. Bohrer, B., Platzer, A.: Constructive Hybrid Games. In: Automated Reasoning, 10th International Joint Conference. vol. 12166, pp. 454–473. Springer, Berlin, Heidelberg (2020)
2. Chen, T.Y., Cheung, S.C., & Yiu, S.M.: Metamorphic testing: A new approach for generating next test cases. Tech. rep., The Hong Kong University of Science and Technology (1998)
3. Engler, D., Musuvathi, M.: Static analysis versus software model checking for bug finding. Verification, Model Checking, and Abstract Interpretation **2937** (2004)
4. Fulton, N., Mitsch, S., Bohrer, B., Platzer, A.: Bellerophon: Tactical theorem proving for hybrid systems. In: Interactive Theorem Proving, International Conference. vol. 10499, pp. 207–224. Springer, Berlin, Heidelberg (2017)
5. Fulton, N., Platzer, A.: Verifiably Safe Off-Model Reinforcement Learning. In: Tools and Algorithms for the Construction and Analysis of Systems. vol. 11427, pp. 413–430. Springer, Berlin, Heidelberg (2019)
6. Fulton, N., Mitsch, S., Quesel, J.D., Völp, M., Platzer, A.: Keymaera x: An axiomatic tactical theorem prover for hybrid systems. In: International Conference on Automated Deduction. pp. 527–538. Springer (2015)
7. Jackson, D.: Alloy: A lightweight object modelling notation. Tech. rep., MIT Laboratory for Computer Science (2000)
8. Kaur, A., Nayyar, R.: A comparative study of static code analysis tools for vulnerability detection in c/c++ and java source code. Procedia Computer Science (2020)
9. Mitsch, S., Platzer, A.: ModelPlex: Verified Runtime Validation of Verified Cyber-Physical System Models. In: Formal Methods in System Design. vol. 49.1, pp. 33–74 (2016)
10. Quesel, J.D., Mitsch, S., Loos, S., Arechiga, N., Platzer, A.: How to model and prove hybrid systems with KeYmaera: A tutorial on safety. In: STTT. pp. 67–91 (2016)
11. Torlak, E., C., F.S., Jackson, D.: Finding Minimal Unsatisfiable Cores of Declarative Specifications. In: International Symposium on Formal Methods. Springer, Berlin, Heidelberg (2008)

12. Zeller, A., Hildebrandt, R.: Simplifying and isolating failure-inducing input. IEEE Transactions on Software Engineering **28**(2), 1–62 (2002)

9 Appendix

9.1 Analysis for Identifying Extraneous Model Components

The analysis we define in this section performs a bottom-up scan of the proof tree, collecting facts that were not used in the proof. In this scan, we collect parts of the model that were not used during the proof by comparing proof steps to patterns in a *modification oracle* which we specify. Starting at the root of the proof tree, the analysis examines the current proof step, marking facts used during that step as *used* and facts introduced during that step as *usable.* The analysis continues up through the leaves of the proof tree. The purpose of this analysis is two-fold: 1) to identify unused components of the model, and 2) extraneous proof artifacts. This aggregate of unused model facts and extraneous proof artifacts will be presented to the user as feedback for how to simplify their model and their proof. Our approach to this analysis closely parallels the same algorithm employed in mark-and-sweep garbage collection algorithms. Now we discuss the methods employed in this analysis.

analyze. The analysis assumes the given model is valid. Hence, the algorithm begins in `analyze` which takes a model and its proof. It begins by marking all facts in the model as *usable.* The set of these usable facts is an initial pool of facts that we anticipate are necessary to prove the model. With this set of facts, we can begin the recursive analysis.

recanalyze. The analysis performs a recursive scan of the proof tree examining each node at a time. Each node is *matched* with the proof rule it corresponds to, and an oracle uses pattern matching to determine which facts are used and which facts are introduced in this proof step. From this, the analysis updates the set of facts, marking those used as *used* and adding new facts as children of the appropriate dependency graphs. If p produces multiple branches, the *match* returns a set of fact sets each containing facts relevant to their respective proof branches. Note that facts marked *used* may continue to be used later in the analysis and that their status will remain as *used.* In other words, *used* reflects that a given fact is used at least once during the proof. New facts are dependency graphs which are inserted into our list of facts as children of dependency graphs. These represent facts that may be used later in the analysis. If one such new fact is *used*, facts associated with the dependency graphs that spawned that fact are also marked *used*, but not conversely. For example, suppose that somewhere in the analysis facts $x \geq 0$ and $x \leq 0$ generated fact $x = 0$, and that $x \geq 0$ was marked *used* in another branch but not this one. Then only $x \geq 0$ and its predecessors would be marked *used* and $x = 0$ would remain an unused artifact of the proof unless used in some other way. Alternatively, if $x = 0$ were marked *used* given the same dependencies, then its predecessors $x \geq 0$ and $x \leq 0$ would also be marked *used*. Dependency graphs in this context therefore exhibit the behavior consistent with that of *parent pointer tree* implementations.

Since each proof step may disseminate into multiple branches each of which could return a different set of facts, we resolve such discrepancies by computing the difference among these sets. The difference among these fact sets may be broadly understood as the union among these sets. The union here is defined in the usual way, additionally merging dependency graphs when a given fact occurs in multiple fact sets. That is, when a given fact occurs multiple times across different fact sets, we would want to retain any information introduced by all of branches. So the children of this fact across all instances would now point to the same parent fact, and if any instance of that fact was marked *used*, we mark this parent fact *used*.

helper functions. Here, we discuss the helper functions employed in this analysis.

- *intialize.* Given a model *M*, initialize constructs a set of usable facts and returns this set of usable facts to the calling function.
- *spawn_fact.* Given a list of fact dependency graphs *fdg* and new fact *f*, we insert *f* as a child of these dependency graphs *fdg*. The resulting dependency graph is returned to the callee. Here, we consider a list of fact dependency graphs as a newly-spawned fact could result from multiple facts. For example, if we are introducing the fact $x = 0$ given facts $x \geq 0$ and $x \leq 0$, we would want $x = 0$ to reference both these facts as its parents.
- *mark_used.* Given a dependency graph *F*, mark all predecessors of f as *used* and return the result to the callee.
- *match.* Given a proof step p and a list of facts U, we look up the rule p corresponds to. To look up the rule p corresponds to, we define an *oracle,* O that maps **dL** rules to modification functions that modify the given list of facts U. When applied to U, a given modification function would use functions *spawn_fact* and *mark_used* to introduce new facts or mark existing facts in U as *used* accordingly. The *modify* function along with the oracle are discussed in further detail in the following section. The result here is a set of fact sets with each fact set corresponding to a branch generated by p.
- *compare.* Given two lists of facts U and U', compute the diff between U and U'. That is, if a fact that occurs in the new fact set U' also occurs in the original fact set U and is not used in the proof, add it to the set MA. MA keeps track of model artifacts, facts that were introduced in the model that were not used in the proof. Analogously, if a given fact that occurs in U' does not occur in U and is not used in the proof, then it is an artifact of the proof. We record such facts in the set PA.
- *used.* Given a fact u, *used* simply returns the status of its usage flag. If the fact was *used*, it returns true. Otherwise it returns false.

```
procedure INITIALIZE($M$ model)
    $U \leftarrow \emptyset$
    for $f \in M$ do
        $f$.set_flag_usable()
        $U$.append($f$.make_dependency_graph())
    return $U$
procedure SPAWN_FACT($fdg$ list(dependency graph), $f$ dependency graph)
    $f$.set_flag_usable()
    for $i \in [1 \dots fdg.length]$ do
        $fdg[i]$.insert($f$)
    return $fdg$
procedure MARK_USED($F$ dependency graph)
    for $f \in F$ do
        $f$.set_flag_used()
    return $F$
procedure MATCH($p$, $U$ list)
    $S \leftarrow$ MODIFY($O$, $p$, $U$, spawn_fact, mark_used)
    return $S$
procedure IS_USED($u'$)
    return u.is_used
procedure COMPARE($U$ list, $U'$ list)
    $MA \leftarrow \emptyset$
    $PA \leftarrow \emptyset$
    for $u' \in U$ do
        $used \leftarrow$ IS_USED($u'$)
        if $u' \in U$ and $used$ then
            $MA \leftarrow MA$.append($u'$)
        else if $u' \notin U$ and $used$ then
            $PA \leftarrow PA$.append($u'$)
    return ($MA$, $PA$)
procedure ANALYZE($M$ model, $P$ tree)
    $U \leftarrow$ INITIALIZE($M$)
    $U' \leftarrow$ RECANALYZE($P$, $U$)
    $(MA, PA) \leftarrow$ COMPARE($U$, $U'$)
procedure RECANALYZE($P$ tree, $U$ list)
    if $P \neq \emptyset$ then
        $\{U_1, \dots, U_n\} \leftarrow$ MATCH($P$.root(), $U$)
        $U_{acc} \leftarrow \emptyset$
        for $i \in [1 \dots n]$ do
            $U'_i \leftarrow$ RECANALYZE($U_i$)
            $U_{acc} \leftarrow U_{acc} \sqcup U'_i$
        return $U_{acc}$
    else
        return $U$
```