# The Power of "Why" and "Why Not": Enriching Scenario Exploration with Provenance


Tim Nelson
Brown University, USA

Natasha Danas
Brown University, USA

Daniel J. Dougherty
Worcester Polytechnic Institute, USA

Shriram Krishnamurthi
Brown University, USA



## ABSTRACT

Scenario-finding tools like the Alloy Analyzer are widely used in numerous concrete domains like security, network analysis, UML analysis, and so on. They can help to verify properties and, more generally, aid in exploring a system's behavior.

While scenario finders are valuable for their ability to produce concrete examples, individual scenarios only give insight into what is *possible*, leaving the user to make their own conclusions about what might be *necessary*. This paper enriches scenario finding by allowing users to ask "why?" and "why not?" questions about the examples they are given. We show how to distinguish parts of an example that cannot be consistently removed (or changed) from those that merely reflect underconstraint in the specification. In the former case we show how to determine which elements of the specification and which other components of the example together explain the presence of such facts.

This paper formalizes the act of computing provenance in scenario-finding. We present Amalgam, an extension of the popular Alloy scenario-finder, which implements these foundations and provides interactive exploration of examples. We also evaluate Amalgam's algorithmics on a variety of both textbook and real-world examples.



## CCS CONCEPTS

• **Software and its engineering** → *Formal methods*;

## KEYWORDS

Model finding, formal methods, provenance, Alloy analyzer

**ACM Reference format:**
Tim Nelson, Natasha Danas, Daniel J. Dougherty, and Shriram Krishnamurthi. 2017. The Power of "Why" and "Why Not": Enriching Scenario Exploration with Provenance. In *Proceedings of 2017 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Paderborn, Germany, September 4–8, 2017 (ESEC/FSE'17),* 11 pages.
https://doi.org/10.1145/3106237.3106272





*ESEC/FSE'17, September 4–8, 2017, Paderborn, Germany*





ACM ISBN 978-1-4503-5105-8/17/09…$15.00
https://doi.org/10.1145/3106237.3106272


## 1 INTRODUCTION

Scenario-finders produce concrete examples that satisfy formal specifications. They have been popularized by tools like Alloy [17], which has been widely used in many domains to (e.g.) debug and understand UML diagrams [24, 25], analyze firewall configurations, security policies [23, 28], network switches [32], and web security [1], and discover an oversight [40] in the Chord [35] distributed hash-table protocol. Scenarios found may correspond to examples of access requests, class diagrams, faulty protocol executions, network topologies, theorem counterexamples, and so on. Since scenario-finders function even in the absence of formal correctness properties, they are often used to help users understand a system by example, discover new properties to check, or perform *property-free* analyses such as semantic differencing [25, 28] of systems.

It is crucial that tools empower users to *understand* the scenarios[1] they are presented with, rather than merely *show* them examples consistent with the original specification. However, currently there is only limited tool support for helping users understand scenarios. For example, the Alloy Analyzer provides an evaluator that allows users to evaluate expressions in the context of the currently shown scenario; in this way users can answer "*what is true?*" questions. But users should also be able to ask the much more interesting "*Why?*" and "*Why not?*" questions. For instance, one might ask: "Why does my network configuration take that action on this packet?", "Why doesn't this class implement an interface in this example?", or even "What parts of my specification prevent me from adding another node to this binary-search tree?".

Answering such questions is hard enough in the context of a deterministic system where behavior generally has one cause or chain of causes. In scenario-finding, however, "Why is this here?" may have zero answers (i.e., nothing forces that portion of the scenario to be present) or more than one answer (when multiple constraints in the specification make the element necessary). Moreover, explanations will usually be contingent on what else is (and is not) present in the example shown. Giving users answers to such explanatory questions is therefore non-trivial, yet still vital for enabling productive, disciplined use.

Finally, although the canonical use-cases for scenario-finding often involve human-generated specifications, many applications (e.g., [5, 11, 24, 25, 27–29]) compile software artifacts like UML diagrams

[1] Scenario-finding tools are also called "model-finders". However, the word "model" is heavily overloaded, and can mean either a specification (i.e., "model of software") or a scenario found via logical methods. Throughout this paper, we will always use the term in the latter, logical sense—an interpretation of relation symbols over a set of atoms—to formalize our notion of scenario. Although they sometimes share internal strategies, scenario- or model-finders like Alloy are distinct from model-*checkers*, another class of tool that typically focuses on verifying that a system satisfies temporal properties.

or firewall policies to specifications and invoke the scenario-finder as a back-end. Since the scenarios produced are then in terms of machine-generated translations rather than meticulous, human-crafted specifications, Alloy bears an even greater burden to help users understand the scenarios shown.

*Contributions.* In this paper, we establish novel, well-defined notions of "Must this be here?" and "Why is this here?" for scenario-finding. These ideas are realized in **Amalgam**[2], an enhanced version of the widely-used Alloy scenario-finder. We choose to build atop Alloy because it is used by multiple and diverse communities, and also due to its expressive power (all of first-order logic, along with relational operators such as transitive closure). Amalgam's novel features comprise: the ability to say what is and is not necessary in a scenario (i.e., cannot be changed without consequences elsewhere); rigorous, proof-based explanations (*provenance*) for necessity; and disciplined, user-guided scenario alteration that enables users to explain why elements of a scenario *can* be altered.

Amalgam facilitates a richer workflow than what Alloy currently provides. We illustrate this via a worked example in Sec. 2. We then lay out the logical foundations (Sec. 3) and algorithmics (Sec. 4) for provenance generation before discussing Amalgam's implementation (Sec. 5). We evaluate Amalgam (Sec. 6) and contrast it to related work (Sec. 7) before concluding with discussion in Sec. 8.

## 2 WORKED EXAMPLE

To illustrate how "Why?" and "Why not?" questions arise naturally in scenario finding, we first introduce an example adapted from an exercise in Jackson [17]: *undirected trees with node coloring*:

```
1  abstract sig Color {}
2  one sig Red extends Color {}
3  one sig Blue extends Color {}
4  sig Node {
5    neighbors: set Node,
6    color: one Color
7  }
8  fact undirected {
9    neighbors = ~neighbors    -- symmetric
10   no iden & neighbors       -- antireflexive
11 }
12 fact graphIsConnected {
13   all n1: Node | all n2: Node-n1 |
14     n1 in n2.^neighbors }
15 fact treeAcyclic {
16   all n1, n2: Node | n1 in n2.neighbors implies
17     n1 not in n2.^(neighbors-(n2->n1)) }
```

Lines 1–7 declare the basic types in the problem: a notion of color (line 1; `sig` denotes a type declaration), and two concrete colors (lines 2–3). The `abstract` keyword enforces that the `Color` type is the union of its subtypes: `Red` and `Blue`. The `one` keyword constrains the `Red` and `Blue` types to each contain a single, distinct color atom. `Node`s each have a set of `neighbors` and a single `color` (forced by the prior declarations to be either `Red` or `Blue`). Line 9 enforces symmetry, making the graph undirected; line 10 prevents self-loops. Lines 12–14 use transitive-closure (^) to force the graph to be connected. Lines 15–17 enforce acyclicity by saying that removing any edge disconnects its endpoints.

[2]http://cs.brown.edu/research/plt/dl/fse2017/

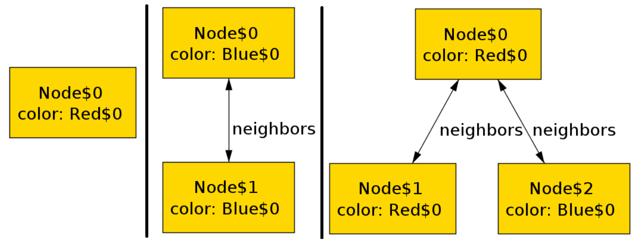


**Figure 1: Three example scenarios produced by Alloy. In Alloy, `A$k` denotes the $k^{th}$ element of the type `A`; here indexes range from 0 to 2. Colors are named in the same way and appear under the name of each node. The edges show the `neighbors` relation.**

Alloy converts this specification to a theory of first-order logic with transitive closure, with types as unary relations and `neighbors` and `color` each assigned a binary relation. Running the specification in Alloy produces a stream of models that satisfy that theory (up to a user-specified size). Fig. 1 contains 3 (of many) example models found up to a bound of 3 `Node`s.

### 2.1 Challenge: Detecting Underconstraint

It is easy to accidentally omit a constraint when writing a specification: e.g., the antireflexivity constraint on line 10. If it is left out, the specification is satisfiable, but *underconstrained*: it is satisfied by some models that contain self-loops. This error is revealed if Alloy produces a cyclic model, but since tool parameters like SAT-solver choice affect the order in which Alloy generates models, there is no guarantee that a cycle will be shown even after several invocations.

This issue illustrates a weakness of purely model-based output: vital information may be withheld well past the point where most human users stop requesting new models. In contrast, by giving additional information in the form of *necessity* and *provenance*, Amalgam can reveal some bugs even if the model shown is "correct". Consider the leftmost model of Fig. 1. This model is a (singleton) tree that satisfies the specification. However, without an antireflexivity constraint, the specification permits adding an edge from `Node$0` to itself (in spite of the acyclicity constraint). By reporting what is and is not *locally necessary* (Sec. 3) in the model, Amalgam alerts the user to underconstraint, without hiding the error in a stream of (potentially complex) models. Users can then instruct Amalgam to *augment* the model with the new edge—providing them with a counterexample to their expectation—and then use Alloy's evaluator to explore why existing constraints do not suffice.

### 2.2 Challenge: Tracing Overconstraint

Implicit assumptions in evolving specifications can lead to errors—as we discover if we decide to allow self-loops in our trees. To do so, we again remove the antireflexivity constraint on line 10. Unfortunately, this edit reveals a subtle *overconstraint*: self-loops remain forbidden in models larger than 1 node.

The `treeAcyclic` constraint is the culprit. Its formulation implicitly assumes irreflexivity, i.e., that `n1 in n2.neighbors` implies that `n1` and `n2` must be different nodes. The fix is just to make that assumption explicit, adding `n1 != n2` to the antecedent of the constraint. But if the user is only presented with a series of models that look like valid trees (some even with self-loops), how can they discover the fix? How will they *even learn of the error* without iterating until no more models are available, remembering every detail of previous models seen and then mentally synthesizing the fact that some self-loops were never shown?

In contrast, Amalgam can reveal the problem on the first model larger than 1 node. Consider the middle, 2-node model in Fig. 1. This is a valid tree, but neither of the self-loop edges can be added without consequences (i.e., adding or removing additional nodes or edges). Amalgam detects this, reporting these edges' absence as locally necessary. Furthermore, Amalgam can explain this necessity, guiding the user to the appropriate fragment of the constraint.

The four-panel Fig. 2 shows how Amalgam presents provenance as a deductive argument, highlighting four crucial steps:

(1) The first highlight is a top-level constraint that leads to the local necessity of `Node$1` having no self loop.
(2) The next highlight shows the subformula after binding `n1=n2=Node$1` (possible since atom `Node$1` is a `Node`).
(3) The right-hand side of the implication is false under that instantiation whether or not `Node$1` has a self-loop.
(4) Finally, since the right-hand side of the implication is true, Amalgam concludes that the left-hand side must be false: `not Node$1->Node$1 in neighbors`.

## 2.3 Challenge: Multiple Explanations

Finally, suppose we constrain node coloring by saying that leaf nodes ought to be colored blue and internal nodes red:

```
all n: Node | n.color = Blue iff lone n.neighbors
```

That is, the node is blue if and only if it is connected to 0 or 1 other nodes. In Fig. 1, the middle model satisfies this constraint but the other 2 do not. Revisiting the center model, Amalgam indicates that it is necessary for both nodes to be blue. But why? Asking for a provenance of `Node$0`'s blue color actually yields a *pair* of provenances corresponding to the explanations:

- The declaration of `Node` said that every node has exactly `one` color; we cannot remove `Node$0`'s color.
- The new constraint forces `Node$0` to be blue since it has no more than one neighbor.

Each of these encapsulate a way that constraints and current scenario together imply (Sec. 3) that `Node$0`'s blue color cannot be removed without consequences elsewhere in the scenario. Different explanations may be useful in different situations: either reminding the user that every node must have a color or pointing them to the coloring constraint. For this reason, Amalgam generates *sets* of provenances rather than only single explanations.

*Comparison to Unsatisfiable Cores.* If a specification has no models, Alloy can obtain and highlight a minimal *core* of the specification that is itself unsatisfiable. This allows the user to zero in on which constraints are mutually unsatisfiable under the current bounds. (For more information on the uses of unsatisfiable cores in Alloy, see Torlak, et al.'s [36] insightful work.) However, such cores can only be found when a specification is indeed *unsatisfiable*. Amalgam's provenance and necessity information give insight into overconstraint even when the specification can be satisfied, making it useful for debugging subtle errors that may eliminate a handful of models, but not all. Moreover, Amalgam produces a *set* of provenances that can be explored, each of which provides different insight, rather than a single core. (We discuss potential future applications of core-extraction to provenance in Sec. 8.)

# 3 FOUNDATIONS

We now establish foundations for provenance. Although we use Alloy syntax throughout, our results apply to any scenario-finding tool that uses bounded first-order logic. In this section as well as Sec. 4, we will use the term *model* to formalize the notion of scenario as a logical structure over a relational language. We also use *theory* to formalize the specification as a set of logical formulas.

## 3.1 Syntax

Alloy's surface syntax includes the usual predicate-logical operators (quantification is *sorted*). To these, Alloy adds relational operators: join ($\cdot$), product ($\times$), transitive closure ($\hat{}$), and others (see Fig. 3 for a full list). These operators have the usual first-order and relational semantics, which we sketch in Fig. 3.

## 3.2 Model-Finding

The general model-finding problem consists of satisfiability search: finding a model that satisfies some theory. A *bounded model-finding problem* $(\mathcal{L}, \mathcal{T}, \mathcal{U}, \mathcal{LB}, \mathcal{UB})$ comprises:

(1) a language $\mathcal{L}$;
(2) a theory $\mathcal{T}$ over the symbols in $\mathcal{L}$;
(3) a finite domain $\mathcal{U}$ (the *universe*); and
(4) upper and lower-bound functions $\mathcal{LB}$ and $\mathcal{UB}$ defined for each ($n$-ary) relation $R \in \mathcal{L}$ such that $\mathcal{UB}(R) \subseteq \mathcal{U}^n$, $\mathcal{LB}(R) \subseteq \mathcal{U}^n$ and $\mathcal{LB}(R) \subseteq \mathcal{UB}(R)$.

A *solution* to such a problem is a model $\mathbb{M}$ over the language $\mathcal{L}$ such that ($R^{\mathbb{M}}$ denotes the interpretation of $R$ in $\mathbb{M}$):

(1) $\mathbb{M} \models \mathcal{T}$;
(2) $|\mathbb{M}| = \mathcal{U}$; and
(3) for each relation $R \in \mathcal{L}$, $\mathcal{LB}(R) \subseteq R^{\mathbb{M}} \subseteq \mathcal{UB}(R)$.

Since the bounded model-finding problem is restricted to searching for models with a specific finite domain $\mathcal{U}$, satisfiability and testing truth in a model are each decidable. In fact, bounded satisfiability can be checked by reducing the problem to the purely *propositional* domain, with each possible tuple membership $\bar{t} \in R$ (i.e., each member of the problem's Herbrand base) being assigned a single Boolean variable. We embrace this perspective, and will implicitly enrich $\mathcal{L}$ with a distinct constant for every element $E$ of $\mathcal{U}$. For brevity, we abuse notation somewhat and name these constants identically with the elements they represent. Thus, all formulas we consider will be *closed*; i.e., without free variables.

*Example 3.1.* The undirected-tree example of Sec. 2 describes a theory over the language (superscripts denote arity):

$$\mathcal{L} = \{\text{Node}^{(1)}, \text{Color}^{(1)}, \text{Red}^{(1)}, \text{Blue}^{(1)}, \text{color}^{(2)}, \text{neighbors}^{(2)}\}$$

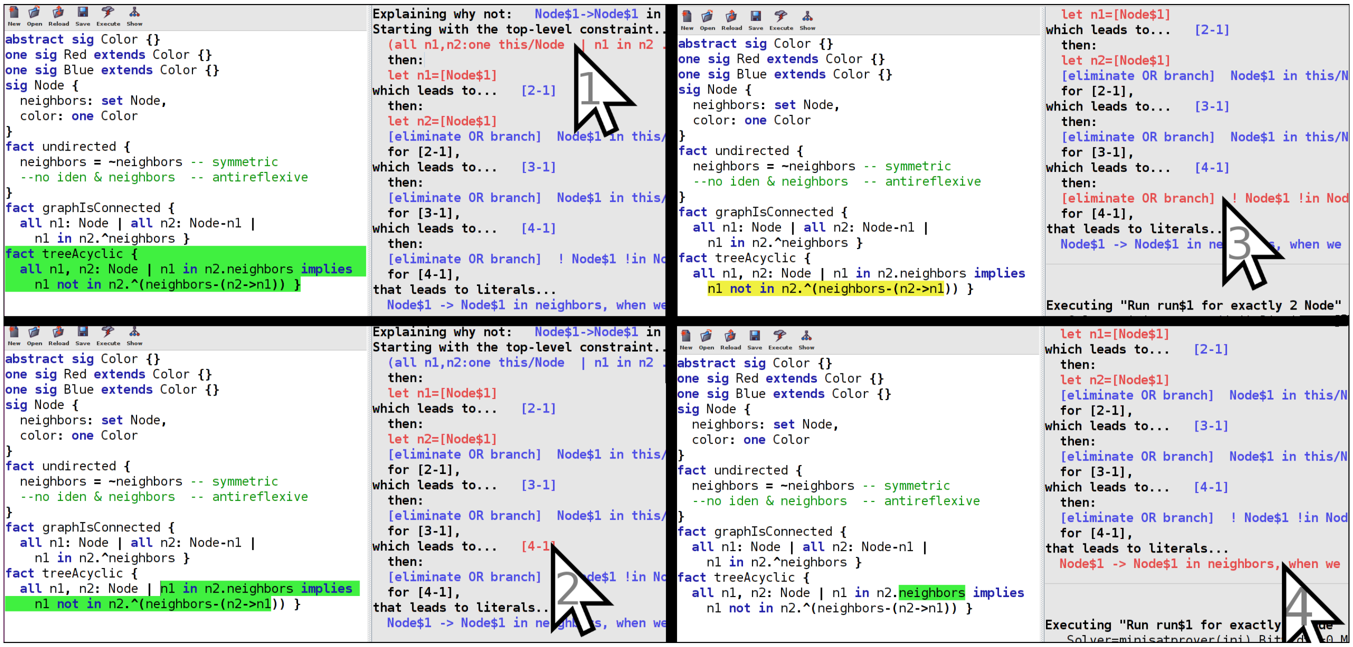


**Figure 2: Sample provenance and interaction for the 2-node model in Fig. 1 asking why `Node$1` cannot have a self-loop edge. The right-hand side of each of the four panels is a step-by-step deductive argument. The left-hand side highlights portions of the specification corresponding to where the user is pointing in the right-hand pane. Yellow highlights (shown in panel 3) comprise the set of subformulas that force local necessity (Sec. 3). Green highlights (shown in panels 1, 2, and 4) correspond to steps of the recursive descent algorithm we present in Sec. 4.**

| Syntax | Meaning | Syntax | Meaning |
|---|---|---|---|
| $\phi \times \psi$ | Cartesian product | $\phi \cdot \psi$ | relational join |
| $\phi \cup \psi$ | union | $\phi \cap \psi$ | intersection |
| $\phi \setminus \psi$ | set difference | $\phi{+}{+}\psi$ | overriding union |
| $\phi{<:}\psi$ | retain rows in $\psi$ with first column in $\phi$ | $\phi{:>}\psi$ | retain rows in $\phi$ with last column in $\psi$ |
| $\hat{}\phi$ | transitive closure | $*\phi$ | reflexive transitive closure |
| $\phi[\psi]$ | inverse relational join | $\sim\phi$ | relational transpose |
| $\#\phi$ | cardinality | $\{\bar{t} : T \vert \alpha(\bar{t})\}$ | set comprehension |
| iden | identity relation (binary) | univ | universe (unary) |
| $\alpha \wedge \beta$ | conjunction | $\alpha \vee \beta$ | disjunction |
| $\alpha \Rightarrow \beta$ | implication | $\alpha \iff \beta$ | bi-implication |
| $\alpha ? \beta : \gamma$ | if-then-else | | |
| $\neg\alpha$ | negation | $\phi$ in $\psi$ | relational containment |
| $\forall x : T \vert \alpha(x)$ | universal quantification | $\exists x : T \vert \alpha(x)$ | existential quantification |
| one $\phi$ | $\vert\phi\vert = 1$ | lone $\phi$ | $\vert\phi\vert \leq 1$ |
| some $\phi$ | $\vert\phi\vert \geq 1$ | no $\phi$ | $\vert\phi\vert = 0$ |

**Figure 3: Supported Alloy Surface Syntax. $\alpha$, $\beta$, and $\gamma$ denote formulas which evaluate to true or false. $\phi$ and $\psi$ denote expressions which evaluate to relations. For clarity, we distinguish between relational containment $\phi$ in $\psi$ and tuple membership $\bar{t} \in \phi$, although Alloy uses identical syntax.**

Because there is exactly *one* of each color (`one sig`), Alloy computes that $\mathcal{UB}(\text{Blue}) = \mathcal{LB}(\text{Blue}) = \{\texttt{Blue\$0}\}$ (and similarly for red). If the specification is run for up to 3 nodes, then $\mathcal{LB}(Node) = \emptyset$ and $\mathcal{UB}(Node) = \{\texttt{Node\$0}, \texttt{Node\$1}, \texttt{Node\$2}\}$. $\mathcal{U}$ is therefore $\{\texttt{Node\$0}, \texttt{Node\$1}, \texttt{Node\$2}, \texttt{Blue\$0}, \texttt{Red\$0}\}$. Upper bounds for the binary relations include all well-typed tuples.

Because we will always be interested in a *bounded* model-finding problem, when we speak of entailment it is always restricted to the models that respect the universe $\mathcal{U}$ and bounding functions $\mathcal{LB}$ and $\mathcal{UB}$ of the current problem. We reinforce this by writing entailment with a subscript: $\models_{\mathcal{U}}$.

Fix a bounded model-finding problem over $\mathcal{L}$, $\mathcal{T}$, and $\mathcal{U}$.

*Definition 3.2 (Literal, Diagram).* If $R$ is a n-ary relation in $\mathcal{L}$, $c_1, ..., c_n$ constants for elements of $\mathcal{U}$, and $\bar{t} = [c_1, ..., c_n]$ then the formulas $\bar{t} \in R$ and $\bar{t} \notin R$ are *literals*. The *diagram* of a model $\mathbb{M}$, denoted $\Delta(\mathbb{M})$, is the set of literals true in $\mathbb{M}$.

We use the set-containment idiom for literals rather than writing $R(c_1, ..., c_n)$ because it more closely reflects the syntax of Alloy.

## 3.3 Necessity and Provenance

We are interested in exploring *why* a given literal $L$ must hold in a model $\mathbb{M}$ in order to satisfy theory $\mathcal{T}$. In this section we make this notion of “why” precise. At one extreme $L$ might be a logical consequence of $\mathcal{T}$. At the other extreme, $L$ might be a “gratuitous” fact about $\mathbb{M}$, not contributing to making $\mathbb{M}$ a model of $\mathcal{T}$ at all; here it is reasonable to report that “there is no reason why” $L$ holds. This is already useful information to a user, of course. The interesting case is the one in which $L$ need not hold in *all* models of $\mathcal{T}$, but, in the context of the rest of the model $\mathbb{M}$, cannot be negated without falsifying $\mathcal{T}$. Our main goal is to analyze this latter situation closely.

*Definition 3.3 (L-alternate, Local Necessity).* Fix a literal $L$ true in $\mathbb{M}$. The *L-alternate* $\mathbb{M}^{\mathbb{L}}$ of $\mathbb{M}$ is the model with the same universe as $\mathbb{M}$ whose diagram $\Delta(\mathbb{M}^{\mathbb{L}})$ is $(\Delta(\mathbb{M}) \setminus \{L\}) \cup \{\neg L\}$.

A literal $L$ true in $\mathbb{M}$ is *locally necessary* for $\mathcal{T}$ in $\mathbb{M}$ if $\mathbb{M}^{\mathbb{L}} \not\models \mathcal{T}$.

Note that this is a weaker condition than $\mathcal{T}$-entailment. *Local* necessity captures the fact that other literals in a particular model force $L$ to hold; it might be (and is likely that) $\mathcal{T} \not\models_{\mathcal{U}} L$ in general.

Theorem 3.4. *$L$ is locally necessary for $\mathcal{T}$ in $\mathbb{M}$ if and only if $\mathcal{T} \cup (\Delta(\mathbb{M}) \setminus \{L\}) \models_{\mathcal{U}} L$.*

Proof. If $L$ is not locally necessary for $\mathcal{T}$ in $\mathbb{M}$, then $\mathbb{M}^{\mathbb{L}}$ is a witness for the failure of $\mathcal{T} \cup (\Delta(\mathbb{M}) \setminus \{L\}) \models_{\mathcal{U}} L$. Conversely, to say that $\mathcal{T} \cup (\Delta(\mathbb{M}) \setminus \{L\}) \not\models_{\mathcal{U}} L$ is to say that $\mathcal{T} \cup (\Delta(\mathbb{M}) \setminus \{L\}) \cup \{\neg L\}$ has a model with universe $\mathcal{U}$ that respects $\mathcal{LB}$ and $\mathcal{UB}$. The only such model with diagram $(\Delta(\mathbb{M}) \setminus \{L\}) \cup \{\neg L\}$ is $\mathbb{M}^{\mathbb{L}}$, thus $\mathbb{M}^{\mathbb{L}} \models \mathcal{T}$, and so $L$ is not locally necessary for $\mathcal{T}$ in $\mathbb{M}$. □

We now formalize the notion of “why” via *provenance*.

*Definition 3.5 (Provenance).* A *provenance* for $L$ in $\mathbb{M}$ with respect to $\mathcal{T}$ is a set of sentences $\alpha_1, ..., \alpha_n$ ($n \geq 0$), each true in both $\mathbb{M}$ and $\mathbb{M}^{\mathbb{L}}$, such that $\mathcal{T} \cup \{\alpha_1, ...\alpha_n\} \models_{\mathcal{U}} L$.

The condition that each component $\alpha_i$ holds in $\mathbb{M}$ binds the notion of provenance to $\mathbb{M}$, while requiring that each $\alpha_i$ holds in $\mathbb{M}^{\mathbb{L}}$ ensures that no $\alpha_i$ entails $L$ under $\mathcal{T}$, so that the provenance is non-trivial.

There is an important aspect of provenance that would be tedious to make explicit in Definition 3.5: the way that the $\alpha_i$ point back to the specification $\mathcal{T}$. Each $\alpha_i$ computed by the algorithm presented in Sec. 4 is a substitution instance of a subformula of an axiom in $\mathcal{T}$, providing links to specific places in the specification that are “to blame” for the truth of $L$. In our implementation, the $\alpha$ formulas are those highlighted yellow (Fig. 2).

*Example 3.6.* Consider the theory $\{\forall x.R(x), \forall x.(P(x) \Rightarrow Q(x))\}$ over the language with three unary relations $R$, $P$, and $Q$. Suppose $\mathcal{U} = \{0\}$ and for all three relations $\mathcal{LB}(\cdot) = \emptyset$ and $\mathcal{UB}(\cdot) = \{0\}$. If $\mathbb{M} = \{P(0), Q(0), R(0)\}$ then literals $Q(0)$ and $R(0)$ have provenances: $\{P(0)\}$ and $\emptyset$ respectively. $P(0)$ has no provenance.

The following is an easy consequence of Theorem 3.4.

Lemma 3.7. *$L$ has provenance in $\mathbb{M}$ with respect to $\mathcal{T}$ if and only if $L$ is locally necessary for $\mathcal{T}$ in $\mathbb{M}$.*

# 4 ALGORITHMICS

Fix a bounded model-finding problem over $\mathcal{L}, \mathcal{T}$, and $\mathcal{U}$ with upper and lower bounds for each $R \in \mathcal{L}$. Let $\mathbb{M} \models \mathcal{T}$ and $L$ be locally necessary for $\mathcal{T}$ in $\mathbb{M}$. To obtain a set of provenances for $L$ in $\mathbb{M}$, it is useful to define a *desugaring* function that instantiates and flattens formulas (Sec. 4.1). We then proceed by recursively evaluating the formulas in $\mathcal{T}$ in $\mathbb{M}$ and $\mathbb{M}^{\mathbb{L}}$, desugaring as necessary and recording subformulas that lead to $\mathcal{T}$’s failure in $\mathbb{M}^{\mathbb{L}}$ (Sec. 4.2). Finally, to further focus the provenance on elements of the model, we expand each provenance generated into a *literal* provenance (Sec. 4.4).

## 4.1 Desugaring Alloy

Alloy’s syntax contains several operators that are effectively syntactic sugar, and bounds enable even more simplification. When generating provenance, it will be useful to instantiate some quantifiers, relational expressions, and derived operators. To do so, we utilize the problem’s upper and lower bounds to convert (e.g.) universally quantified formulas to a conjunction over the upper bound of the quantified variable’s type. Since variable types need not be basic relations, we extend the notion of upper bound to include arbitrary relational expressions. Most of these details are routine, but Fig. 4 shows some of the more interesting cases—like quantification—where we must explicitly depend on the *boundedness* of the model-finding problem to perform instantiation. If a desugaring step produces an *empty* conjunction or disjunction, it means that the bounds themselves are in some way incompatible and might need to be increased—another useful distinction that the stream-of-models paradigm fails to make.

$$
\begin{aligned}
&\mathit{desugar}(\forall x^{\phi}\ \alpha(x)) &&= \textstyle\bigwedge\{\bar{t} \in \phi \Rightarrow \alpha(\bar{t}) \mid \bar{t} \in \mathcal{UB}(\phi)\}\\
&\mathit{desugar}(\exists x^{\phi}\ \alpha(x)) &&= \textstyle\bigvee\{\bar{t} \in \phi \wedge \alpha(\bar{t}) \mid \bar{t} \in \mathcal{UB}(\phi)\}\\
&\mathit{desugar}(\phi \text{ in } \psi) &&= \textstyle\bigwedge\{\bar{t} \notin \phi \vee \bar{t} \in \psi \mid \bar{t} \in \mathcal{UB}(\phi)\}\\
&\mathit{desugar}((a_1, a_n) \in \hat{}\phi) &&= \textstyle\bigvee\{\bigwedge\{(a_i, a_{i+1}) \in \phi \mid 1 \leq i \leq n\} \mid\\
&&&\qquad [a_1, ..., a_n] \text{ is a path in } \mathcal{UB}(\phi)\}\\
&\mathit{desugar}(\bar{t} \in \phi \cdot \psi) &&= \textstyle\bigvee\{\bar{t}_1 \in \phi \wedge \bar{t}_2 \in \psi \mid\\
&&&\qquad \bar{t}_1 \in \mathcal{UB}(\phi), \bar{t}_2 \in \mathcal{UB}(\psi), \bar{t}_1 \cdot \bar{t}_2 = \bar{t}\}\\
&\qquad\text{when } c \text{ is a constant:}\\
&\mathit{desugar}(\phi < c) &&= \textstyle\bigvee\{(\phi = n) \mid \mathit{MIN} \leq n < c\}\\
&\mathit{desugar}(\phi = c) &&= c \in \phi\\
&\mathit{desugar}(\#\phi = c) &&= \textstyle\bigwedge(\{\bar{t} \in \phi | \bar{t} \in \mathcal{UB}(\phi), \bar{t} \in \phi^{\mathbb{M}} \cap \phi^{\mathbb{M}^{\mathbb{L}}}\} \cup\\
&&&\qquad \{\bar{t} \notin \phi | \bar{t} \in \mathcal{UB}(\phi), \bar{t} \notin \phi^{\mathbb{M}} \cup \phi^{\mathbb{M}^{\mathbb{L}}}\}) \Rightarrow\\
&&&\quad \textstyle\bigwedge(\{\bar{t} \in \phi | \bar{t} \in \mathcal{UB}(\phi), \bar{t} \in \phi^{\mathbb{M}}, \bar{t} \notin \phi^{\mathbb{M}^{\mathbb{L}}}\} \cup\\
&&&\qquad \{\bar{t} \notin \phi | \bar{t} \in \mathcal{UB}(\phi), \bar{t} \notin \phi^{\mathbb{M}}, \bar{t} \in \phi^{\mathbb{M}^{\mathbb{L}}}\})
\end{aligned}
$$

**Figure 4: Desugaring Alloy operators and instantiation by upper bounds via *desugar*$(\cdot)$. Numeric operators are subject to bounds on bitwidth; a bitwidth of 4 (e.g.) corresponds to the range −8 through 7. Cardinality (#) expressions desugar to formulas that express the failure of the expression in $\mathbb{M}^{\mathbb{L}}$) Other operators (which we elide for space) are either routine or proceed similarly.**

## 4.2 Computing Provenance

To build intuition, we first step through the concrete node-coloring example from Sec. 2.3, where we constrain each node's color depending on its neighbors. Recall that $\mathbb{M}$ (the middle model of Fig. 1) has two nodes, `Node$0` and `Node$1`. We focus on the fact that `Node$0` is blue: the literal $L$ = [`Node$0`,`Blue$0`] $\in$ *color*.

This literal is locally necessary, since the $L$-alternate model $\mathbb{M}^{\mathbb{L}}$ obtained by removing this tuple from the color relation fails two top-level constraints in $\mathcal{T}$:

```
all n : Node | n.color = Blue iff lone n.neighbors
all n : Node | one n.color
```

We obtain provenance for $L$ by computing an explanation for why $L$ fails in $\mathbb{M}^{\mathbb{L}}$. Since a conjunction fails if *any* of its subformulas fail, we extract independent provenances from the two failing constraints. We start with the former axiom.

Our algorithm instantiates the universal quantifier as a conjunction (maintaining the sort restriction as a guard on each conjunct). We then discover that only the following instantiation fails in $\mathbb{M}^{\mathbb{L}}$:

```
Node$0 in Node implies
  (Node$0.color = Blue iff lone Node$0.neighbors)
```

The implication fails because the guard is true (in either model) but the consequent becomes false in $\mathbb{M}^{\mathbb{L}}$. Because the obligation to make the consequent true is only triggered by the fact that `Node$0` is a node, we add that to the provenance, then continue recursively, explaining why the consequent fails in $\mathbb{M}^{\mathbb{L}}$. The bi-implication desugars to a pair of implications, with the "if" implication failing:

```
lone Node$0.neighbors implies Node$0.color = Blue
```

As before, we add the antecedent `lone n.neighbors` to the provenance. We continue to recur until "hitting bottom" at the input literal $L$. The context collected comprises a provenance $P_1$:

```
Node$0 in Node
lone Node$0.neighbors
```

The latter axiom from $\mathcal{T}$ also produces a provenance. We instantiate as before and desugar `one` according to the upper bounds, eventually producing the following provenance $P_2$:

```
Node$0 in Node
not (Node$0->Red$0 in color and
     Node$0->Blue$0 not in color)
```

Since $P_1$ and $P_2$ arise from separate failing conjuncts (the top-level constraints in $\mathcal{T}$) we present them separately.

*Pointing back to $\mathcal{T}$.* The remark following Definition 3.5 concerning the tight connection between provenance components and axioms of $\mathcal{T}$ can be understood more clearly now that we see how the provenance computation works. For each item $\alpha$ we add to a provenance there is a subformula $\phi$ from $\mathcal{T}$ such that $\alpha$ is an instance of $\phi$ true in $\mathbb{M}$ and in $\mathbb{M}^{\mathbb{L}}$. By collecting these instances verbatim we are able to track failing sub-constraints and present them to the user. On the other hand, Sec. 4.4 explains how to break these formulas down into literals if desired.

*The Algorithm.* Fig. 5 gives the recursive provenance function $\mathcal{Y}(\cdot)$, defined on sentences whose interpretation changes from true in $\mathbb{M}$ to false in $\mathbb{M}^{\mathbb{L}}$. There is a function $\mathcal{Y}_\neg(\cdot)$, for sentences false in $\mathbb{M}$ and true in $\mathbb{M}^{\mathbb{L}}$, whose definition is dual to $\mathcal{Y}(\cdot)$, but omitted here for lack of space.

$$\mathcal{Y}(\bar{t} \in R) \equiv \begin{cases} \{\emptyset\} & \text{if } L = \bar{t} \in R \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\mathcal{Y}(\alpha_1 \wedge \ldots \wedge \alpha_n) \equiv \bigcup\{\mathcal{Y}(\alpha_i) | \mathbb{M} \models \alpha_i, \mathbb{M}^{\mathbb{L}} \not\models \alpha_i\}$$

$$\mathcal{Y}(\rho_1 \vee \ldots \vee \rho_n \vee \gamma_1 \vee \ldots \vee \gamma_m) \equiv$$
$$\{\{\neg\rho_1, \ldots, \neg\rho_n\} \cup p | p \in (\mathcal{Y}(\gamma_1) \overset{\cup}{\times} \ldots \overset{\cup}{\times} \mathcal{Y}(\gamma_m))\}$$
(where $\mathbb{M} \not\models$ each $\rho_i$ and $\mathbb{M} \models$ each $\gamma_j$)

$\mathcal{Y}(\neg\alpha) \equiv \mathcal{Y}_\neg(\alpha)$ $\quad$ $\mathcal{Y}(\alpha) \equiv \mathcal{Y}(desugar(\alpha))$ in all other cases.

**Figure 5: Explanation function $\mathcal{Y}$.**

If $\alpha$ is a literal, $\mathcal{Y}$ is only defined if $\alpha \equiv L$ (since otherwise $\mathbb{M}$ and $\mathbb{M}^{\mathbb{L}}$ could not differ on the interpretation of $\alpha$): the provenance here is empty, since clearly $L \vDash_{\mathcal{U}} L$.

When $\alpha$ is a negation, we invoke $\mathcal{Y}_\neg(\alpha)$.

When $\alpha$ is a conjunction, the failure of *any* conjunct causes the overall formula to fail in $\mathbb{M}^{\mathbb{L}}$. The resulting provenance-set is therefore the union of all explanations for each conjunct's failure.

When $\alpha$ is a disjunction, local necessity means that *every* disjunct must evaluate to false in $\mathbb{M}^{\mathbb{L}}$ and at least one must hold in $\mathbb{M}$. For intuition, view the disjunction as an implication with the disjuncts false in $\mathbb{M}$ negated in the antecedent. It is these subformulas (false in both $\mathbb{M}$ and $\mathbb{M}^{\mathbb{L}}$; labeled $\rho_i$ in Fig. 5) that imply the others (true in $\mathbb{M}$ and false in $\mathbb{M}^{\mathbb{L}}$; labeled $\gamma_i$ in Fig. 5) and force the failure of the overall formula. $\mathcal{Y}$ recurs for each failing $\gamma_i$, combines their provenances with *union product* and adds each $\rho_i$ to every provenance in the resulting set. Here, the union-product of a pair of sets of sets $A = \{a_1, \ldots, a_n\}$ and $B = \{b_1, \ldots, b_m\}$ is $A \overset{\cup}{\times} B = \{a_i \cup b_j | 1 \leq i \leq n, 1 \leq j \leq m\}$. The union product operator is similar to Cartesian product, but rather than building ordered pairs of element sets, it combines those elements with union.

Otherwise, we perform one desugaring step and recur.

One might initially expect the dual of the conjunctive case (which uses union) to use *intersection*. However, this would mean combining multiple provenances—*all* of which must apply—by discarding unshared components. We therefore use union product to enforce that some full provenance for each subformula is respected.

*Complexity.* Our algorithm amounts to a tree-search of a partially desugared $\mathcal{T}$, testing for truth in $\mathbb{M}$ and $\mathbb{M}^{\mathbb{L}}$ while descending. Both time and space complexity are therefore proportional to the size of the model $\mathbb{M}$ times the size of the theory $\mathcal{T}$ post-desugaring. Expanding a quantifier over $\phi$ (Fig. 4) produces one guarded instantiation per element in the upper bound of $\phi$. Thus, worst-case quantifier-elimination is exponential in the quantifier-nesting depth of $\mathcal{T}$. Expanding transitive closure enumerates possible paths, and so there the worst case is superexponential: the number of potential paths between a fixed source and destination in a complete $n$-graph is $\sum_{0 \leq x \leq n-2} \binom{n-2}{x} x!$. This challenge is shared by Alloy's model-finding engine as it passes to propositional logic and is thus not unique to provenance generation. Moreover, in practice (Sec. 6) these worst cases rarely manifest. This is because bounds tend to be both separated into disjoint types and small (well under 10 atoms) in keeping with Jackson's *small scope hypothesis* [17], which says that small examples usually suffice.

## 4.3 Correctness

Fix a bounded model-finding problem for $\mathcal{T}$ over $\mathcal{L}$, $\mathbb{M} \models \mathcal{T}$ and $L \in \Delta(\mathbb{M})$. First note that $\mathcal{Y}(\sigma)$ is defined and returns a non-empty result if $\mathbb{M} \models \sigma$ but $\mathbb{M}^{\mathbb{L}} \not\models \sigma$. Thus a provenance is always produced if $L$ is locally necessary in $\mathbb{M}$ (i.e., Amalgam is *complete*). It remains to show that the provenances produced are correct.

Theorem 4.1 (Correctness of Provenance-Generation). *If the algorithm of Fig. 5 produces a provenance $\alpha_1, \ldots, \alpha_n$ for $L$ with respect to $\mathcal{T}$ in $\mathbb{M}$ then $\mathcal{T} \cup \{\alpha_1 \wedge \ldots \wedge \alpha_n\} \vDash_{\mathcal{U}} L$.*

Proof. Proceed by induction on ordered pairs consisting of the number of steps to fully desugar $\sigma$ and the size of $\sigma$. If $\sigma$ is a literal, $\mathcal{Y}$ is only defined if $\sigma \equiv L$ (since otherwise $\mathbb{M}$ and $\mathbb{M}^{\mathbb{L}}$ could not differ on the interpretation of $\sigma$), and clearly $L \vDash_{\mathcal{U}} L$. If $\sigma$ is a negation, the theorem holds by direct application of the inductive hypothesis. If $\sigma$ is a conjunction, then the result is the union of all provenance sets obtained by calling $\mathcal{Y}$ on $\sigma$'s subformulas. By the inductive hypothesis we have that for each provenance $P$ in that union, $\sigma \wedge P \vDash_{\mathcal{U}} L$.

If $\sigma$ is a disjunction, each disjunct must be false in $\mathbb{M}^{\mathbb{L}}$ or the formula would not fail. Let those true in $\mathbb{M}$ (i.e., that become false) be $\gamma_1, \ldots, \gamma_n$ and the others (which remain false) be $\rho_1, \ldots, \rho_m$. Then $\sigma$ is equivalent to $\neg\rho_1 \wedge \ldots \wedge \neg\rho_m \Rightarrow \gamma_1 \vee \ldots \vee \gamma_n$. By the inductive hypothesis, each $\mathcal{Y}(\gamma_i)$ produces a set of provenances $P_i$ such that $\gamma_i \wedge p_i \vDash_{\mathcal{U}} L$ ($p_i \in P_i$). Thus, $\gamma_1 \vee \ldots \vee \gamma_n \wedge p_{conseq} \vDash_{\mathcal{U}} L$ for each $p_{conseq} \in P_1 \overset{\cup}{\times} \ldots \overset{\cup}{\times} P_n$. Therefore $\sigma \wedge \neg\rho_1 \wedge \ldots \wedge \neg\rho_m \wedge p_{conseq} \vDash_{\mathcal{U}} L$.

In all other cases, $\sigma$ is desugared before $\mathcal{Y}$ recurs and the inductive hypothesis can be applied directly. □

## 4.4 Obtaining Literal Provenance

The provenances generated in Sec. 4.2 say which *subformulas and instantiations* are responsible for $L$'s local necessity in $\mathbb{M}$. However, it is sometimes useful to see a provenance that focuses blame onto just the parts of $\mathbb{M}$ responsible for local necessity. This reveals a spectrum of provenance complexity: higher-level formulas can be concise, but lower-level formulas are tied more closely to the model being understood. A literal provenance, which contains only literals, stands at the far end of that spectrum:

*Definition 4.2 (Literal Provenance).* A *literal provenance* for $L$ is a provenance $\alpha_1, \wedge \ldots \wedge \alpha_n \vDash_{\mathcal{U}} L$ where each $\alpha_i$ is a literal.

To obtain a literal provenance from an arbitrary provenance $P$, we convert each non-literal formula $\alpha$ in $P$ to a set of literals true in $\mathbb{M}$ that force $\alpha$ to hold. To do this, we traverse the negation normal-form of $\alpha$, seeking conjunctions of literals that entail it, desugaring as needed. This process amounts to evaluating the formula in reverse, extracting pieces of the model responsible for $\alpha$'s truth.

However, this process can potentially return a conjunction of literals that contains $L$ or $\neg L$, which would violate our definition of provenance since either is false in either $\mathbb{M}$ or $\mathbb{M}^{\mathbb{L}}$. Even more, some sentences may require contingent reasoning, with different literals leading to truth in $\mathbb{M}$ versus $\mathbb{M}^{\mathbb{L}}$.

*Example 4.3.* Let $\alpha = ((0 \in P) \wedge (0 \in Q)) \vee ((0 \notin P) \wedge (0 \in R)) \vee s$, $\mathbb{M} = \{(0 \in P), (0 \in Q), (0 \in R)\}$, and $L = (0 \in P)$. While $\alpha$ holds in both $\mathbb{M}$ and $\mathbb{M}^{\mathbb{L}}$, neither branch suffices on its own; *both* $(0 \in Q)$ and $(0 \in R)$ must appear together.

To resolve this problem we search, in parallel, for a pair of conjunctions that satisfy $\alpha$ in $\Delta(\mathbb{M})$ and in $\Delta(\mathbb{M}^{\mathbb{L}})$. If one conjunction does not involve $L$ it explains $\alpha$'s truth in both models since they differ only by $L$. Otherwise, one must contain $L$ and the other $\neg L$; in this case we combine them and remove both $L$ and $\neg L$. This is sound since if $L \wedge \beta \Rightarrow \alpha$ and $\neg L \wedge \gamma \Rightarrow \alpha$ it holds that $\beta \wedge \gamma \Rightarrow \alpha$.

# 5 IMPLEMENTATION

Amalgam is implemented as a drop-in extension to Alloy 4.2, rather than a standalone tool. This means that Alloy users can experiment with new features and incrementally adopt them without any disruption of their workflow. Users can access Amalgam's extensions via Alloy's existing *evaluator*, a prompt that allows them to evaluate expressions in the current scenario. We extend this facility to give insight into local necessity via provenance. Users can either ask for a broad list of what is locally necessary or browse the set of all provenances generated for individual literals. Amalgam provides both basic provenance display (as seen in Fig. 2) and an expert interface that shows the details of every step of the recursive descent described in Sec. 4. In both, mousing over components of a provenance highlights the corresponding portions of the specification.

The tool also allows users to *augment* scenarios by adding or removing literals that are not locally constrained. Much like provenance can be helpful in cases of *over*-constraint, augmentation can be helpful if a specification is *under*-constrained since it allows users to move quickly to surprising scenarios for further investigation. Our approach to augmentation is similar to other tools, such as Aluminum [30], except that users can both add and remove elements of a scenario. (We discuss further differences in Sec. 7.)

Amalgam supports some Alloy features that were unmentioned in Fig. 3, such as multiplicity-constrained type declarations and total ordering. Amalgam's support for numerics includes counting the cardinality of set expressions and inequalities; it does not currently support arithmetic operations.

# 6 EVALUATION

We evaluate Amalgam quantitatively along five dimensions: its performance, the number of leaf formulas (i.e., $\alpha$s gathered by the algorithm in Sec. 4), the depth of the recursive descent (which directly affects the size of the tree displayed), the character-count of the largest highlight shown (which also affects provenance display), and the total number of provenances generated for each literal. For each specification, we take these measurements for each literal permitted by the specification's bounds—asking "Why?" for literals that are present and "Why not?" for those that are absent. We report results for the first *two* scenarios returned by the scenario-finder to mitigate bias in scenario ordering. Fig. 6 reports these results.

Our evaluation suite consists of *22* specifications and comprises a wide mixture of example, educational, and "real-world" specifications. Address book (**addr**), Grandpa (**grand**), and geneology (**gene**) are from Alloy's example set. Grade book (**grade**), bad employee (**bempl**), and other groups (**other**) are Alloy translations of access-control specifications used to benchmark existing scenario-finding work [33]. Directed graph (**digraph**) is a non-empty (but otherwise unconstrained) directed graph, as a baseline for comparison. Directed tree (**dtree**) constrains the graph to be a tree. **Dtbug** injects a flaw in **dtree**'s edge injectivity constraint. The two colored, undirected trees specifications (**ctrees** and **ctreesb**) are the original shown in Sec. 2 and the buggy modification with irreflexivity removed. **Abc** is a logic puzzle that requires hypothetical reasoning. Good Will Hunting (**gwh**) encodes a scenario-finding problem popularized by the cinema: searching for trees where no vertex has degree 2. Transitive-closure and garbage collection lab (**tclab** and **gclab**, respectively) are specifications from labs exercises in an introductory formal methods course. The first gives practice with transitive closure; the second models reference-counting garbage collection and reveals its flaws. The model of propositional resolution (**resfm**) comes from Torlak, et al. [36]. **Flow** reveals a bug in a network program written in Flowlog [29], a language for programming software-defined networks. **Cdd1** and **Cdd2** are Maoz, et al.'s translation [24] of two UML diagrams. **Cddiff1** and **cddiff2** are the semantic differences of those two models (i.e., **cdd1** $\subseteq$ **cdd2** and **cdd2** $\subseteq$ **cdd1**) produced by CDDiff [25]. We also include the authentication model (**web**) from Akhawe, et al. [1]. Together, these cover a wide spectrum of complexity, upper bounds, and Alloy features.

Finally, we note that **flow**, **cdd1**, **cdd2**, **cddiff1**, and **cddiff2** are all machine-translations from software artifacts. The compilers that implement these translations are non-trivial, so the specifications they produce call out for answers to "why?" and "why not?" questions from the compiler developers as well as their end-users.

## 6.1 Performance

We measure performance by calculating the time and peak memory required to generate *all* provenances for each literal by running the $\mathcal{Y}$ function from Sec. 4. To put these figures in context, we compare this to the time Alloy's scenario-finding engine takes to produce the first two scenarios, including the time taken to translate the specification to propositional logic. (Provenance-generation does not impact Alloy's scenario-finding approach in any way, so there is no overhead to generating scenarios in Amalgam.) To stabilize measurement variance, we repeat our experiments 15 times on *each* of our 22 specifications. All results were gathered on an Ubuntu 16.04 / 2.60GHz i5-4278U CPU / 16GB RAM machine. In most cases, it takes less memory to compute provenance than scenarios; however, for larger examples provenance can use slightly more memory. The worst case peak memory usage during provenance generation was 1547 MB (for **flow**), while the maximum during scenario generation was 1201 MB—roughly a 29% difference.

Amalgam usually generates provenances no slower than Alloy generates scenarios (on the order of milliseconds). Indeed, for **web**, scenario-generation is more than two orders of magnitude slower on average than provenance generation: here the complexity is in producing a scenario, not in explaining literals. The only significant outlier is **flow**, which takes on average 2.55 times longer to explain a literal than to produce a scenario. The difference is due to **flow**'s complexity and the unusually large provenance count that some literals in **flow** have; we address this second point further in Sec. 6.3.

## 6.2 Explanation Complexity

For each specification, we report three metrics as a surrogate for comprehensibility, aggregated over all provenances produced: the number of $\alpha$ formulas gathered (i.e., the number of leaves in the tree shown), the depth of recursive descent (i.e., the depth of the tree shown), and the character-count of the largest highlighted region.

*6.2.1 Depth.* In most cases, the average depth does not exceed a dozen, resulting in a fairly succinct derivation. The **tclab** specification has a maximum depth of 17 because it contains a deep tree of predicate calls (the lab is designed to teach students to use helper predicates), each of which contains several relational operators that all take a desugaring step.

*6.2.2 Highlighting.* Since Amalgam highlights concrete source locations in the original Alloy file, highlight size corresponds to the original—not desugared or instantiated—Alloy specification. Amalgam thus produces small highlights in general; most specifications see a maximum well under 100 characters. The largest highlight usually corresponds to the top-level constraint in each provenance (e.g., the largest highlighted region in Fig. 2's provenance is shown in step 1). Large maximum highlights, such as **cddiff2**'s 858, arise when visiting large constraints in the specification and are greatly reduced in future steps (from 858 to 71 in this particular case). We also report the *total* number of characters in each specification, through which we see that even the largest highlight is only roughly 12% of the **cddiff2** specification.

*6.2.3 Leaf Count.* Since new leaf formulas occur whenever branches of a disjunction are eliminated, specifications with large disjunctions, existential quantification with large bounds, or transitive-closure produce high $\alpha$ counts. The largest leaf-counts appear in **gwh** and **gclab**, both of which make heavy use of transitive closure pair with relatively large upper bounds. In this case, our algorithm produces provenances that enumerate all possible paths. However, a conversion to *literal* provenance greatly reduces leaf count (from 20 to 11 on average for **gwh**, and from 66 to 16 in the worst case for **gclab**). Further reduction is likely possible, as we do not currently search for the *smallest* provenances.

In contrast, provenances for **authn**, **flow**, **grand**, and especially **cddiff2** blow up significantly when converted to literal form. This is because some $\alpha$ formulas in these provenances depend on large swathes of the scenario. For instance, an $\alpha$ that contains a universal quantifier implicitly depends on all its potential instantiations. Situations where literal provenances are smaller therefore indicate significant overlap in the parts of the scenario that make $\alpha$ formulas true. For example, this happens in **gwh** because most $\alpha$s there are caused by transitive closure—which desugars in a repetitive way.

**Flow** specifies a state transition function that is defined by a disjunction over logic-program fragments. Each fragment causes a set of literals to be true. Negative literals therefore have provenance encompassing the fact that *none* of these program fragments apply—which is fairly large, as Fig. 6 reports. This pattern of provenances that comprise multiple instantiated specification fragments persists in **gene** and **resfm**.

| | Max | # Prov Trees | | | Tree Depth | | | # Tree Leaves (Prov[LitP]) | | | Largest Highlight (chars) | | | | Runtime (ms) | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Spec | Bnd | Med | $\mu$ | Max | Med | $\mu$ | Max | Med | $\mu$ | Max | Med | $\mu$ | Max | Spec | Pr $\mu$ | $\mathbb{M}$ $\mu$ | Pr Max | $\mathbb{M}$ Max |
| ctrees | 3 | 1 | 2 | 4 | 4 | 4 | 6 | 2[2] | 2[2] | 3[5] | 20 | 21 | 78 | 576 | 1 | 7 | 3 | 23 |
| ctreesb | 3 | 1 | 1 | 3 | 3 | 3 | 5 | 1[2] | 1[2] | 3[4] | 16 | 15 | 48 | 598 | 1 | 6 | 2 | 18 |
| digraph | 4 | 1 | 1 | 1 | 1 | 1 | 4 | 1[1] | 1[1] | 3[3] | 3 | 7 | 22 | 108 | 1 | 5 | 2 | 17 |
| addr | 4 | 1 | 1 | 2 | 4 | 3 | 9 | 2[2] | 2[3] | 7[8] | 20 | 13 | 48 | 1.1k | 1 | 9 | 7 | 31 |
| other | 3 | 1 | 1 | 4 | 4 | 4 | 6 | 2[2] | 2[2] | 3[4] | 15 | 33 | 79 | 1.5k | 1 | 6 | 4 | 15 |
| grade | 3 | 1 | 1 | 3 | 4 | 3 | 8 | 2[2] | 2[2] | 11[5] | 20 | 21 | 85 | 2.0k | 1 | 44 | 54 | 1.1k |
| abc | 3 | 1 | 2 | 5 | 4 | 4 | 5 | 2[3] | 2[3] | 2[3] | 56 | 50 | 56 | 557 | 1 | 6 | 4 | 20 |
| bempl | 3 | 1 | 1 | 5 | 4 | 3 | 7 | 2[3] | 2[2] | 5[8] | 20 | 22 | 56 | 1.4k | 1 | 7 | 4 | 19 |
| dtbug | 4 | 2 | 3 | 7 | 4 | 4 | 5 | 2[3] | 3[2] | 17[7] | 15 | 14 | 23 | 1.7k | 2 | 6 | 16 | 17 |
| grand | 4 | 2 | 2 | 5 | 4 | 4 | 9 | 2[2] | 3[4] | 12[36] | 39 | 32 | 42 | 2.7k | 2 | 11 | 7 | 60 |
| flow | 4 | 1 | 2 | 41 | 7 | 7 | 13 | 3[26] | 4[24] | 20[45] | 242 | 228 | 300 | 12k | 135 | 53 | 967 | 285 |
| tclab | 5 | 1 | 2 | 9 | 5 | 7 | 17 | 2[5] | 5[7] | 17[16] | 21 | 23 | 43 | 1.9k | 8 | 16 | 39 | 35 |
| resfm | 5 | 1 | 2 | 6 | 5 | 5 | 12 | 3[3] | 3[5] | 15[49] | 105 | 76 | 105 | 2.1k | 6 | 72 | 48 | 285 |
| gene | 6 | 3 | 4 | 13 | 5 | 5 | 7 | 7[7] | 6[11] | 10[26] | 50 | 37 | 58 | 2.5k | 15 | 10 | 178 | 47 |
| gwh | 6 | 7 | 6 | 13 | 7 | 7 | 8 | 6[12] | 20[11] | 66[19] | 129 | 93 | 129 | 721 | 37 | 26 | 102 | 97 |
| gclab | 6 | 2 | 2 | 11 | 4 | 4 | 11 | 2[2] | 3[3] | 66[16] | 56 | 50 | 58 | 2.6k | 2 | 10 | 29 | 26 |
| authn | 6 | 1 | 1 | 16 | 4 | 3 | 9 | 2[2] | 2[3] | 19[80] | 25 | 30 | 243 | 19k | 11 | 1.4k | 192 | 31k |
| cddiff1 | 6 | 1 | 1 | 9 | 6 | 6 | 11 | 2[4] | 3[5] | 23[19] | 16 | 25 | 42 | 7.1k | 13 | 41 | 40 | 127 |
| cddiff2 | 6 | 1 | 1 | 4 | 5 | 4 | 14 | 2[3] | 2[4] | 38[202] | 14 | 24 | 858 | 7.1k | 11 | 41 | 45 | 176 |
| dtree | 7 | 1 | 1 | 2 | 1 | 1 | 6 | 1[1] | 1[1] | 3[8] | 2 | 5 | 28 | 649 | 2 | 9 | 185 | 36 |
| cdd 1 | 10 | 1 | 1 | 9 | 5 | 3 | 11 | 2[3] | 2[3] | 24[27] | 16 | 20 | 42 | 4.8k | 28 | 111 | 130 | 1.5k |
| cdd 2 | 10 | 1 | 1 | 4 | 1 | 2 | 12 | 1[1] | 2[4] | 21[34] | 6 | 12 | 42 | 4.1k | 8 | 32 | 61 | 81 |

**Figure 6: Number of provenances, provenance complexity (depth, leaves, highlighting), and runtime for both Provenance (Pr) and scenario ($\mathbb{M}$) generation. For each row, Max Bnd denotes the largest bound in the specification. For provenance depth, leaves, highlighting, and count we report median, average($\mu$), and maximum; we give median rather than standard deviation because we do not believe the non-performance data are normally distributed. For leaves, we report a value for standard provenance trees, and those expanded to a full literal provenance (in [brackets]). For highlighting, we also report the total specification size (in characters) for comparison. Where numbers exceed 1000, we divide by a thousand and add a "k" suffix.**

### 6.3 Number of Explanations

We measure the number of provenances generated because—much like a stream of scenarios—a large number of provenances may conceal the one or two that will uniquely inform the user. For most specifications, the numbers are promising, with most literals having only *one or two* provenances even for **flow**, **web**, and the **cdd** group.

Some specifications have literals with many provenances. This occurs when literals can affect the truth of many instantiations of top-level constraints at once. Like the colored-trees example in Sec. 2, **gwh** has symmetry, connectivity and acyclicity constraints. Removing an edge violates symmetry, connectivity and possibly the added requirement that no nodes have degree 2. Breaking (e.g.) connectivity generates one provenance for each *pair* of newly disconnected nodes (up to 9 pairs at an upper-bound of 6 nodes). In the case of **flow**, the literal with 41 provenances is that a specific network packet exists. Much of the specification depends on that packet, there are many reasons why it must exist (41 in fact). The other high provenance counts in Fig. 6 occur for similar reasons.

## 7 RELATED WORK

Scenario finding is an active research area with a rich history. While satisfiability is undecidable for first-order logic in general, bounded (or "finite") scenario-finders achieve termination by searching only up to a bounded scenario size. MACE [26]-style scenario-finders like Kodkod [37], Alloy's internal engine, translate bounded problems into propositional logic and then leverage SAT-solving technology.

*Minimal and Targeted Model Finding.* Aluminum [30] is a version of Alloy that produces only minimal scenarios. These minimal scenarios show only locally-necessary positive literals (i.e., positive literals that have provenance). However, Aluminum provides no provenance information at all, and thus explains neither why the scenarios shown are minimal nor how individual literals interact with the rest of the scenario. Such explanations are Amalgam's primary focus. Aluminum also allows users to augment scenarios by making currently-false literals true, then showing the consistent minimal scenarios that contain the original plus the added literal. While this allows users to explore the consequences of the addition, again it focuses solely on scenario-generation and not on the proofs intrinsic to necessity in a scenario. Amalgam incorporates both augmentation and explanation. Moreover, Amalgam supports reasoning about arbitrary scenarios: it can find provenances for *negative* information in the scenario and find justifications that involve *positive* literals, neither of which would be possible if it enforced minimality.

The Razor [33] scenario-finder likewise produces minimal scenarios. By incorporating a notion of provenance into scenario-generation, Razor is able to justify every positive literal in the scenarios it produces. Amalgam does not limit itself only to minimal scenarios, and so is able to detect and explain local necessity of *negative* as well as positive literals. Razor also lacks support for transitive closure.

The Cryptographic Protocol Shapes Analyzer [12] (CPSA) produces examples that show when cryptographic protocol specifications violate desired properties. In contrast, Amalgam is built atop a *domain-independent* scenario finder and answers "Why?" questions—which CPSA does not consider.

Target-Oriented Model Finding [8] adds optimization targets to bounded scenario-finding problems. The tool then minimizes graph edit distance from targets, enabling (e.g.) maximization as well as minimization. While powerful, this approach is still limited to finding streams of scenarios, rather than explaining them.

*Provenance for Software and Systems.* There has been some prior work on provenance for software. WhyLine [19, 20] answers a limited set of "Why did..." and "Why didn't..." questions about Java program behavior. It records and then replays execution history to reconstruct provenance for events. The Y! tool [6, 39] likewise traces both positive and negative provenance for events in network logs. Vermeer [34] constructs reduced causal traces that explain assertion violations in C programs. These tools extract provenance from runtime logs—which are not available to a scenario-finder and have temporal structure that Alloy's scenarios need not possess.

Fault-localization techniques based on test spectra [31], such as Tarantula [18], use test suites to produce causal information. SAT-TAR [14] uses Alloy specifications to synthesize test inputs to aid localization (further illustrating the flexibility of scenario-finding). Such tools focus on using many tests to provide insight about a program, whereas Amalgam helps users understand how different parts of a single scenario interact. Moreover, tools like Tarantula, SAT-TAR, and Vermeer help explain *program* behavior; Amalgam helps users understand their logical *specifications* and overcome specification-specific issues like under- and over-constraint.

*Sanity Checking.* The need for *sanity checking* arises when a system may satisfy properties for uninteresting or erroneous reasons. Antecedent failure, or *vacuity*, was first investigated by Beatty and Bryant [2] for model-checking. Vacuity can point to subtle issues in either system or property specification, as Beer, et al. [4] discuss.

Hoskote, et al. [16] introduce the notion of *coverage* in model-checking to detect when properties fail to fully exercise the system. Kupferman [21] unifies vacuity and coverage, noting that both can be found by mutation of the property and system respectively. Since in scenario-finding both system and property are combined in the specification, our perspective is similar. Beer, et al. [3] and Chockler [7] mutate counterexample traces to find *causality*. Their explanations are with respect to the property, not the system; Amalgam provides causality information with respect to both. These works also focus on counterexample *traces*, but scenarios in Amalgam need not be (and often are not) temporal.

We are not the first to apply static-analysis techniques to Alloy specifications. Heaven and Russo [15] detect vacuity for a rich subset of Alloy. While we likewise draw inspiration from sanity checking, Amalgam explains why literals are present in arbitrary scenarios, regardless of vacuity. Uzuncaova and Khurshid [38] use slicing techniques to prioritize constraints in Alloy and thereby improve performance. The goal of their work is, however, orthogonal to ours.

Ghassabani, et al. [13] explain why properties hold in a model-checker. This is analogous to Alloy's unsat-core highlighting feature. Amalgam focuses on the opposite situation: explaining why portions of counterexamples are locally necessary.

One related classical technique for generating explanations is abduction [10]. Crucially, Amalgam is based in understanding observation in *a particular model*, as opposed to explaining deductions.

## 8 DISCUSSION

Amalgam takes a first step toward enriching scenario-finding by answering "why?" and "why not?" questions. We conclude with discussion, qualitative experiences, and future work.

*Weaknesses of Local Necessity.* Amalgam's provenances can sometimes be excessively local. For example, when working with undirected trees (Sec. 2) it is easy to mistakenly use constraints that work only in the *directed* case. In a directed graph, acyclicity can be captured by `no iden & ^edge`—i.e., that there are no identity tuples in the transitive closure of the edge relation. However, this rules out graphs larger than a single node when combined with axioms for symmetry and irreflexivity. Upon seeing the one-node example, we can ask Amalgam "why can't another node exist?". However, we are then only told that the graph must be connected, and there is no edge connecting this fresh node to the rest of the tree. Instead, we would like a provenance for the *combination* of a new node and new connecting edges—which would direct us to the buggy constraint.

*Contrasting Local Necessity and Minimality.* In Aluminum [30] and Razor [33], *positive* literals are present if they cannot be consistently removed without *adding* other positive literals. Every positive literal in a minimal scenario is thus locally necessary, but the converse does not hold. Consider the (propositional) theory $\mathcal{T} = \{p \iff q, r\}$ and the scenario $\mathbb{M} = \{p, q, r\}$. $\mathbb{M}$ is not minimal since $\{r\}$ also satisfies $\mathcal{T}$, but each literal in $\mathbb{M}$ is locally necessary: $r$ because it is an axiom and $p$ and $q$ because of each other's presence.

*Future Work: User Studies.* Concurrent work [9] suggests that provenance can indeed be helpful to Alloy users; naturally, we would like to further evaluate Amalgam's effectiveness. To do so, we might manufacture a satisfiable but overconstrained specification (as in Sec. 2.2). We could then divide participants into a control group using Alloy and an experimental group using Amalgam, and ask them to correct the error. We might compare the time taken before effecting a fix, but it would potentially be more interesting to also evaluate the *quality* of fixes made. That is, would either group be more prone to fixing the overconstraint while introducing new problems? It is of course difficult to obtain large pools of Alloy users who also possess the time and inclination to participate in user evaluations.

*Future Work: Other Implementation Strategies.* One promising alternative to the approach in Sec. 4 leverages unsat-core extraction. By Theorem 3.4, a literal $L$ is locally necessary for for a specification $\mathcal{T}$ in a given scenario $\mathbb{M}$ if and only if $\mathcal{T} \cup (\Delta(\mathbb{M}) \setminus \{L\}) \vDash_{\mathcal{U}} L$. If this entailment holds, an unsat core for its negation contains provenance information. While cores are generally not iterable (most solvers would in effect return only a single provenance) tools such as CAMUS [22] escape this limitation.

We opted for recursive descent rather than an unsat-core based approach for several reasons: it avoids potential interference with other features of Alloy, such as symmetry-breaking; it eliminates confounding factors in evaluation (Sec. 6) that could be caused by altering Alloy's scenario-finding; it allows our approach to potentially apply for other tools not based on SAT-solving; and it allowed us to record why each portion of a provenance was generated—improving output quality and easing debugging. Nevertheless, a core-based approach would likely be faster and thus appropriate for applications that make heavy use of provenance.

## ACKNOWLEDGMENTS

We are grateful to Daniel Jackson, Emina Torlak, and the Alloy team. This work was partially supported by the US NSF.

## REFERENCES

[1] D. Akhawe, A. Barth, P.E. Lam, J. Mitchell, and D. Song. 2010. Towards a Formal Foundation of Web Security. In *IEEE Computer Security Foundations Symposium*.
[2] Derek L. Beatty and Randal E. Bryant. 1994. Formally Verifying a Microprocessor Using a Simulation Methodology. In *Design Automation Conference*.
[3] Ilan Beer, Shoham Ben-David, Hana Chockler, Avigail Orni, and Richard J. Trefler. 2012. Explaining counterexamples using causality. *Formal Methods in System Design* 40, 1 (2012), 20–40.
[4] Ilan Beer, Shoham Ben-David, Cindy Eisner, and Yoav Rodeh. 1997. Efficient Detection of Vacuity in ACTL Formulas. In *International Conference on Computer Aided Verification*. 279–290.
[5] Jasmin Christian Blanchette and Tobias Nipkow. 2010. Nitpick: A Counterexample Generator for Higher-Order Logic Based on a Relational Model Finder. In *Interactive Theorem Proving*.
[6] Ang Chen, Yang Wu, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. 2015. Differential provenance: Better network diagnostics with reference events. In *Workshop on Hot Topics in Networks*. ACM, 25.
[7] Hana Chockler. 2016. Causality and Responsibility for Formal Verification and Beyond. In *Workshop on Causal Reasoning for Embedded and safety-critical Systems Technologies*.
[8] Alcino Cunha, Nuno Macedo, and Tiago Guimaraes. 2014. Target oriented relational model finding. In *International Conference on Fundamental Approaches to Software Engineering*. Springer, 17–31.
[9] Natasha Danas, Tim Nelson, Lane Harrison, Shriram Krishnamurthi, and Daniel J. Dougherty. 2017. User Studies of Principled Model Finder Output. In *Software Engineering and Formal Methods*.
[10] Marc Denecker and Antonis C. Kakas. 2002. Abduction in Logic Programming. In *Computational Logic: Logic Programming and Beyond*.
[11] Greg Dennis, Felix Sheng-Ho Chang, and Daniel Jackson. 2006. Modular Verification of Code with SAT. In *International Symposium on Software Testing and Analysis*.
[12] Shaddin F. Doghmi, Joshua D. Guttman, and F. Javier Thayer. 2007. Searching for Shapes in Cryptographic Protocols. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*.
[13] Elaheh Ghassabani, Andrew Gacek, and Michael W. Whalen. 2016. Efficient Generation of Inductive Validity Cores for Safety Properties. In *Foundations of Software Engineering*.
[14] Divya Gopinath, Razieh Nokhbeh Zaeem, and Sarfraz Khurshid. 2012. Improving the effectiveness of spectra-based fault localization using specifications. In *Automated Software Engineering*.
[15] Will Heaven and Alessandra Russo. 2005. Enhancing the Alloy Analyzer with Patterns of Analysis. In *Workshop on Logic-based Methods in Programming Environments*.
[16] Yatin Hoskote, Timothy Kam, Pei-Hsin Ho, and Xudong Zhao. 1999. Coverage Estimation for Symbolic Model Checking. In *Design Automation Conference*.
[17] Daniel Jackson. 2012. *Software Abstractions: Logic, Language, and Analysis* (second ed.). MIT Press.
[18] James A. Jones. 2008. *Semi-Automatic Fault Localization*. Ph.D. Dissertation. Georgia Institute of Technology.
[19] Andrew J. Ko and Brad A. Myers. 2004. Designing the WhyLine: a debugging interface for asking questions about program behavior. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM, 151–158.
[20] Andrew J. Ko and Brad A. Myers. 2009. Finding causes of program output with the Java Whyline. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 1569–1578.
[21] Orna Kupferman. 2006. Sanity Checks in Formal Verification. In *International Conference on Concurrency Theory*.
[22] Mark H. Liffiton and Karem A. Sakallah. 2008. Algorithms for Computing Minimal Unsatisfiable Subsets of Constraints. *Journal of Automated Reasoning* 40, 1 (Jan. 2008), 33.
[23] Ferney A. Maldonado-Lopez, Jaime Chavarriaga, and Yezid Donoso. 2014. Detecting Network Policy Conflicts Using Alloy. In *International Conference on Abstract State Machines, Alloy, B, and Z*.
[24] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. 2011. CD2Alloy: Class Diagrams Analysis Using Alloy Revisited. In *Model Driven Engineering Languages and Systems*.
[25] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. 2011. CDDiff: Semantic Differencing for Class Diagrams. In *European Conference on Object Oriented Programming*.
[26] William McCune. 2003. Mace4 Reference Manual and Guide. *CoRR* cs.SC/0310055 (2003). http://arxiv.org/abs/cs.SC/0310055
[27] Aleksandar Milicevic, Derek Rayside, Kuat Yessenov, and Daniel Jackson. 2011. Unifying Execution of Imperative and Declarative Code. In *International Conference on Software Engineering*.
[28] Timothy Nelson, Christopher Barratt, Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi. 2010. The Margrave Tool for Firewall Analysis. In *USENIX Large Installation System Administration Conference*.
[29] Tim Nelson, Andrew D. Ferguson, Michael J. G. Scheer, and Shriram Krishnamurthi. 2014. Tierless Programming and Reasoning for Software-Defined Networks. In *Networked Systems Design and Implementation*.
[30] Tim Nelson, Salman Saghafi, Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi. 2013. Aluminum: Principled Scenario Exploration Through Minimality. In *International Conference on Software Engineering*.
[31] M. Renieres and S. P. Reiss. 2003. Fault localization with nearest neighbor queries. In *Automated Software Engineering*.
[32] Natali Ruchansky and Davide Proserpio. 2013. A (Not) NICE Way to Verify the OpenFlow Switch Specification: Formal Modelling of the OpenFlow Switch Using Alloy. *ACM Computer Communication Review* 43, 4 (Aug. 2013), 527–528.
[33] Salman Saghafi, Ryan Danas, and Daniel J Dougherty. 2015. Exploring Theories with a Model-Finding Assistant. In *International Conference on Automated Deduction*. Springer, 434–449.
[34] Daniel Schwartz-Narbonne, Chanseok Oh, Martin Schäf, and Thomas Wies. 2015. VERMEER: A Tool for Tracing and Explaining Faulty C Programs. In *International Conference on Software Engineering*. 737–740.
[35] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. 2003. Chord: a scalable peer-to-peer lookup protocol for Internet applications. *IEEE/ACM Transactions on Networking* 11, 1 (2003), 17–32.
[36] Emina Torlak, Felix Sheng-Ho Chang, and Daniel Jackson. 2008. Finding Minimal Unsatisfiable Cores of Declarative Specifications. In *International Symposium on Formal Methods (FM)*.
[37] Emina Torlak and Daniel Jackson. 2007. Kodkod: A Relational Model Finder. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. 632–647.
[38] Engin Uzuncaova and Sarfraz Khurshid. 2008. Constraint Prioritization for Efficient Analysis of Declarative Models. In *International Symposium on Formal Methods (FM)*.
[39] Yang Wu, Mingchen Zhao, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. 2014. Diagnosing missing events in distributed systems with negative provenance. In *Conference on Communications Architectures, Protocols and Applications (SIGCOMM)*. ACM, 383–394.
[40] Pamela Zave. 2012. Using Lightweight Modeling to Understand Chord. *ACM Computer Communication Review* 42, 2 (March 2012), 49–57.