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

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; 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.

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 as
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, 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 algorithms (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   }  
16 fact treeAcyclic {  
17      all n1, n2: Node | n1 in n2.neighbors implies  
18      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. Nodes 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.

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 Nodes.

![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.](http://cs.brown.edu/research/plt/dl/fse2017/)

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 `Node0` 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 Power of “Why” and “Why Not”: Enriching Scenario Exploration with Provenance

The \texttt{treeAcyclic} constraint is the culprit. Its formulation implicitly assumes irreflexivity, i.e., that \( n_1 \) in \( n_2.\text{neighbors} \) implies that \( n_1 \) and \( n_2 \) must be different nodes. The fix is just to make that assumption explicit, adding \( n_1 \neq n_2 \) 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 \( \text{Node}$.\text{color}=\text{Blue} \) having no self-loop.
2. The next highlight shows the subformula after binding \( n_1=n_2=\text{Node}$.\text{color} \) (possible since \( \text{atom \ Node}$.\text{color}=\text{Node}$.\text{color} \)).
3. The right-hand side of the implication is false under that instantiation whether or not \( \text{Node}$.\text{color}=\text{Blue} \) 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 \( \text{Node}$.\text{color}=\text{Blue} \) in \( \text{neighbors} \).

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:

\[
\text{all } n : \text{Node} \mid n.\text{color} = \text{Blue} \text{ iff } \text{one } n.\text{neighbors}
\]

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 \( \text{Node}$.\text{color}=\text{Blue} \)'s blue color actually yields a pair of provenances corresponding to the explanations:

- The declaration of \( \text{Node} \) said that every node has exactly one color; we cannot remove \( \text{Node}$.\text{color} \).’s color.
- The new constraint forces \( \text{Node}$.\text{color}=\text{Blue} \) since it has no more than one neighbor.

Each of these encapsulate a way that constraints and current scenario together imply (Sec. 3) that \( \text{Node}$.\text{color}=\text{Blue} \) 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 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 (\( \times \)), product (\( \times \)), transitive closure (\( \cdot \)), 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 \( (n\text{-ary}) \) 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 \( \mathcal{M} \) over the language \( \mathcal{L} \) such that \( \mathcal{R}^\mathcal{M} \) denotes the interpretation of \( R \) in \( \mathcal{M} \):

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

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 \( \mathcal{T} \) in \( \mathcal{R} \) (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.

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Meaning</th>
<th>Syntax</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\phi \times \psi$</td>
<td>Cartesian product</td>
<td>$\phi \cdot \psi$</td>
<td>relational join</td>
</tr>
<tr>
<td>$\phi \cup \psi$</td>
<td>union</td>
<td>$\phi \cap \psi$</td>
<td>intersection</td>
</tr>
<tr>
<td>$\phi \setminus \psi$</td>
<td>set difference</td>
<td>$\phi + \psi$</td>
<td>overriding union</td>
</tr>
<tr>
<td>$\phi \mathbin{\ll} \psi$</td>
<td>retain rows in $\psi$ with first column in $\phi$</td>
<td>$\phi &gt;: \psi$</td>
<td>retain rows in $\phi$ with last column in $\psi$</td>
</tr>
<tr>
<td>$\phi \bowtie \psi$</td>
<td>transitive closure</td>
<td>$\phi \approx$</td>
<td>reflexive transitive closure</td>
</tr>
<tr>
<td>$#\phi$</td>
<td>cardinality</td>
<td>$\phi \triangledown$</td>
<td>relational transpose</td>
</tr>
<tr>
<td>iden</td>
<td>identity relation (binary)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\alpha \land \beta$</td>
<td>conjunction</td>
<td>$\alpha \lor \beta$</td>
<td>disjunction</td>
</tr>
<tr>
<td>$\alpha \Rightarrow \beta$</td>
<td>implication</td>
<td>$\alpha \iff \beta$</td>
<td>bi-implication</td>
</tr>
<tr>
<td>$\alpha ?: \gamma$</td>
<td>if-then-else</td>
<td>$\phi$ in $\psi$</td>
<td>relational containment</td>
</tr>
<tr>
<td>$\neg \alpha$</td>
<td>negation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\forall x: T[a(x)]$</td>
<td>universal quantification</td>
<td>$\exists x: T[a(x)]$</td>
<td>existential quantification</td>
</tr>
<tr>
<td>one $\phi$</td>
<td>$</td>
<td>\phi</td>
<td>= 1$</td>
</tr>
<tr>
<td>some $\phi$</td>
<td>$</td>
<td>\phi</td>
<td>\geq 1$</td>
</tr>
</tbody>
</table>

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 $UB(Blue) = LB(Blue) = \{Blue$0\}$ (and similarly for red). If the specification is run for up to 3 nodes, then $LB(Node$) = \emptyset$ and $UB(Node$) = $\{Node$0$, Node$1$, Node$2\}$. $U$ is therefore $\{Node$0$, Node$1$, Node$2$, Blue$0$, 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 $U$ and bounding functions $LB$ and $UB$ of the current problem. We reinforce this by writing entailment with a subscript $\Vdash_U$.

Fix a bounded model-finding problem over $L$, $T$, and $U$. 109
3.3 Necessity and Provenance

We are interested in exploring why a given literal \( L \) must hold in a model \( M \) in order to satisfy theory \( T \). In this section we make this notion of “why” precise. At one extreme \( L \) may be a logical consequence of \( T \). At the other extreme, \( L \) might be a “gratuitous” fact about \( M \), not contributing to making \( M \) a model of \( T \) 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 \( T \), but, in the context of the rest of the model \( M \), cannot be negated without falsifying \( T \). Our main goal is to analyze this latter situation closely.

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

A literal \( L \) true in \( M \) is locally necessary for \( T \) in \( M \) if \( M^L \not\models T \).

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

Theorem 3.4. \( L \) is locally necessary for \( T \) in \( M \) if and only if \( T \cup (\Delta(M) \setminus \{L\}) \models T \) with \( M^L \).

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

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

Definition 3.5 (Provenance). A provenance for \( L \) in \( M \) with respect to \( T \) is a set of sentences \( a_1, \ldots, a_n \) (\( n \geq 0 \)), each true in both \( M \) and \( M^L \), such that \( T \cup (a_1, \ldots, a_n) \models T \) and \( a_i \) entails \( L \) under \( T \), 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 \( a_i \) point back to the specification \( T \). Each \( a_i \) computed by the algorithm presented in Sec. 4 is a substitution instance of a subformula of an axiom in \( T \), providing links to specific places in the specification that are “to blame” for the truth of \( L \). In our implementation, the \( a_i \) formulas are those highlighted yellow (Fig. 4).

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 \( \mathcal{U} = \emptyset \) and for all three relations \( \mathcal{L}B(\cdot) = \emptyset \) and \( \mathcal{U}B(\cdot) = \emptyset \). If \( M = \langle P(0), Q(0), R(0) \rangle \) 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 \( M \) with respect to \( T \) if and only if \( L \) is locally necessary for \( T \) in \( M \).

4 ALGORITHMIC

Fix a bounded model-finding problem over \( \mathcal{L}, T \), and \( \mathcal{U} \) with upper and lower bounds for each \( R \in \mathcal{L} \). Let \( M \models T \) and \( L \) be locally necessary for \( T \) in \( M \). To obtain a set of provenances for \( L \) in \( M \), 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 \( T \) in \( M \) and \( M^L \), desugaring as necessary and recording subformulas that lead to \( T \)’s failure in \( M^L \) (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 exploit 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.

---

Figure 4: Desugaring Alloy operators and instantiation by upper bound via desugaring(). Numeric operators are subject to bounds on bitwidth; a bitwidth of 8 (e.g.) corresponds to the range -2^8 through 7. Cardinality (\#) expressions desugar to formulas that express the failure of the expression in \( M^L \). 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 = \{\text{Node$0$.Blue$0$} \in \text{color}\}$.

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

\[
\begin{align*}
\text{all } n : \text{Node} & \implies n.\text{color} = \text{Blue} \iff \text{lonen.neighbors} \\
\text{all } n : \text{Node} & \implies \text{one } n.\text{color}
\end{align*}
\]

We obtain provenance for $L$ by computing an explanation for why $L$ fails in $\mathbb{M}^{-}$. 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 $\mathbb{M}^{-}$:

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

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 $T$ 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 $T$) we present them separately.

Pointing back to $T$. The remark following Definition 3.5 concerning the tight connection between provenance components and axioms of $T$ 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 of $\phi$ from $T$ such that $\alpha$ is an instance of $\phi$ true in $\mathbb{M}$ and in $\mathbb{M}^{-}$. 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 $Y(\cdot)$, defined on sentences whose interpretation changes from true in $\mathbb{M}$ to false in $\mathbb{M}^{-}$. There is a function $Y_{\lnot}(\cdot)$, for sentences false in $\mathbb{M}$ and true in $\mathbb{M}^{-}$, whose definition is dual to $Y(\cdot)$, but omitted here for lack of space.

If $\alpha$ is a literal, $Y$ is only defined if $\alpha \equiv L$ (since otherwise $\mathbb{M}$ and $\mathbb{M}^{-}$ could not differ on the interpretation of $\alpha$): the provenance here is empty, since clearly $L \equiv L$. When $\alpha$ is a negation, we invoke $Y_{\lnot}(\alpha)$.

When $\alpha$ is a conjunction, the failure of any conjunct causes the overall formula to fail in $\mathbb{M}^{-}$. 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 disjunction must evaluate to false in $\mathbb{M}^{-}$ and at least one must hold in $\mathbb{M}$. For intuition, view the disjunction as an implication with the disjuncts false in $\mathbb{M}^{-}$ and true in $\mathbb{M}$ negated in the antecedent. It is these subformulas (false in both $\mathbb{M}$ and $\mathbb{M}^{-}$; labeled $p_i$ in Fig. 5) that imply the others (true in $\mathbb{M}$ and false in $\mathbb{M}^{-}$; labeled $\gamma_i$ in Fig. 5) and force the failure of the overall formula. $Y$ recurs for each failing $\gamma_i$, combines their provenances with union product and adds each $p_i$ 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 $A \times B = \{a_i \cup b_j | 1 \leq i \leq n, 1 \leq j \leq m\}$. 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.

\[
Y(\alpha \in R) = \begin{cases} 
\{0\} & \text{if } L = \{T \in R \} \\
\text{undefined} & \text{otherwise}
\end{cases}
\]

\[
Y(\alpha_1 \land \ldots \land \alpha_n) = \bigcup \{Y(\alpha_i) | M \models \alpha_i, \mathbb{M}^{-} \not\models \alpha_i\}
\]

\[
\begin{align*}
Y(p_1 \lor \ldots \lor p_n \lor \gamma_1 \lor \ldots \lor \gamma_m) = \\
\{\{\lnot p_1, \ldots, \lnot p_n\} \cup \{p \in \{\gamma_1 \times \ldots \times Y(\gamma_m)\}\} & \text{ (where } M \not\models \text{ each } p_i \text{ and } M \models \text{ each } \gamma_j\}
\end{align*}
\]

\[
Y(\lnot \alpha) = Y_{\lnot}(\alpha) \quad Y(\alpha) \equiv Y(\text{desugar}(\alpha)) \text{ in all other cases.}
\]
The Power of “Why” and “Why Not”: Enriching Scenario Exploration with Provenance 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 $T$ over $L, M \models T$ and $L \in \Delta(M)$. First note that $\mathcal{Y}(\sigma)$ is defined and returns a non-empty result if $M \models \sigma$ but $M^{\mathcal{L}} \not\models \sigma$. Thus a provenance is always produced if $L$ is locally necessary in $M$ (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 $T$ in $M$ then $T \cup \{\alpha_1 \land \ldots \land \alpha_n\} \models_{\mathcal{U}} L$.

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 $M$ and $M^{\mathcal{L}}$ could not differ on the interpretation of $\sigma$), and clearly $L \models_{\mathcal{U}} L$. 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, $\sigma \land P \models_{\mathcal{U}} L$.

If $\sigma$ is a disjunction, each disjunct must be false in $M^{\mathcal{L}}$ or the formula would not fail. Let those true in $M$ (i.e., that become false) be $y_1, \ldots, y_n$ and the others (which remain false) be $p_1, \ldots, p_m$. Then $\sigma$ is equivalent to $\neg p_1 \land \ldots \land \neg p_m \models y_1 \lor \ldots \lor y_n$. By the inductive hypothesis, each $\mathcal{Y}(y_i)$ produces a set of provenances $P_i$ such that $y_i \land p_i \models_{\mathcal{U}} L$ ($p_i \in P_i$). Thus, $y_1 \lor \ldots \lor y_n \land p_{\text{conseq}} \models_{\mathcal{U}} L$ for each $p_{\text{conseq}} \in P_1 \land \ldots \land P_n$. Therefore $\sigma \land \neg p_1 \land \ldots \land \neg p_m \land p_{\text{conseq}} \models_{\mathcal{U}} L$.

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 $M$. However, it is sometimes useful to see a provenance that focuses blame onto just the parts of $M$ 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 $\alpha_1, \ldots, \alpha_n \models_{\mathcal{U}} L$ 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 $M$ 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 $M$ or $M^{\mathcal{L}}$. Even more, some sentences may require contingent reasoning, with different literals leading to truth in $M$ versus $M^{\mathcal{L}}$.

ESEC/FSE’17, September 4–8, 2017, Paderborn, Germany
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. Dtt for m injects a flaw in dtree’s edge injectivity constraint. The two colored, undirected trees specifications (etrees and etreesb) 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. Cddif1 and cddif2 are the semantic differences of those two models (i.e., cdd1 ⊆ cdd2 and cdd2 ⊆ cdd1) produced by CDiff [25]. We also include the authentic 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, cddif1, and cddif2 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 \( f \) 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 engine 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 2.53 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 \( a \) 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 cddif2’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 cddif2 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 \( a \) 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 cddif2 blow up significantly when converted to literal form. This is because some \( a \) formulas in these provenances depend on large swathes of the scenario. For instance, an \( a \) 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 \( a \) formulas true. For example, this happens in gwh because most of the leaf formulas 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.
Figure 6: Number of provenances, provenance complexity (depth, leaves, highlighting), and runtime for both Provenance (Pr) and scenario (Bd) generation. For each row, Max Bnd denotes the largest bound in the specification. For provenance depth, leaves, highlighting, and count we report median, average (µ), 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. Uzuncuova 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—ie., 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 irrelexivity. 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 $T = \{ p \iff q, r \}$ and the scenario $M = \{ p, q, r \}$. $M$ is not minimal since $[r]$ also satisfies $T$, but each literal in $M$ 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 $T$ in a given scenario $M$ if and only if $T \cup (A(M) \setminus (L)) \vdash q, L$. 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.
The Power of “Why” and “Why Not”:
Enriching Scenario Exploration with Provenance

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


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