CompoSAT: Specification-Guided Coverage for Model Finding

Sorawee Porncharoenwase, Tim Nelson, and Shriram Krishnamurthi

Brown University

Abstract. Model-finding tools like the Alloy Analyzer produce concrete examples of how a declarative specification can be satisfied. These formal tools are useful in a wide range of domains: software design, security, networking, and more. By producing concrete examples, they assist in exploring system behavior and can help find surprising faults.
Specifications usually have many potential candidate solutions, but model-finders tend to leave the choice of which examples to present entirely to the underlying solver. This paper closes that gap by exploring notions of coverage for the model-finding domain, yielding a novel, rigorous metric for output quality. These ideas are realized in the tool CompoSAT, which interposes itself between Alloy's constraint-solving and presentation stages to produce ensembles of examples that maximize coverage.
We show that high-coverage ensembles like CompoSAT produces are useful for, among other things, detecting overconstraint—a particularly insidious form of specification error. We detail the underlying theory and implementation of CompoSAT and evaluate it on numerous specifications.

1 Introduction

Model-finding tools, like the popular Alloy Analyzer [1], find concrete examples of how a set of declarative constraints can be satisfied. These tools have found application in a wide range of domains because of their power and generality. Specifying a network configuration may yield examples of packets traversing a firewall [2]. A UML class diagram may yield corresponding object diagrams [3, 4]. Other applications abound in security [5, 6], protocol design [7], network switch programming [8, 9] and more. Output models can either act as counterexamples to expected properties or more generally improve intuition and aid understanding of a system. However, specifications define a (frequently large) *set* of models, each of which is useful to differing degrees (e.g., some showing a bug, some not). The choice of *which models to present* and in *what order* is usually left entirely to the underlying solvers, which are performance-focused and unconcerned with the quality of each model found.

Some have proposed more rigorous notions of model quality. For instance, *minimal* models [10–12] disgard extraneous information that may clutter the model and hamper comprehension. Although it is appealing in broad strokes, minimality falls short when it comes to showing what is merely possible or contingent—negating one of the chief strengths of model finding. Indeed, recent

studies [13] suggest that purely-minimal output does not suffice. Most current model-quality notions are also defined only in terms of the content of the models themselves—i.e., they are purely *semantic*—rather than what models can reveal about the specification, making them ill-suited to debugging.

In this work, we break with this trend to explore *syntax-guided* notions of which models are best. For any specification, we extract a maximally representative *ensemble* (i.e., set) of models from the considerably larger stream provided by the solver. To do so, we draw inspiration from test-suite coverage [14] but, as we will show, doing so is subtle in this domain. One complication lies in the fact that, with traditional coverage, code and tests are in principle written independently. In contrast, the solver generates models directly from the specification. Thus, using the specification to dictate whether a model is "good" may appear circular. However, the essence of our work consists in filtering the generation process itself. We detect how a specification constrains portions of a model in context, effectively showing the "weight" of individual constraints in the specification. Where the default enumeration may produce bad coverage, ours does far better—with relatively few models. We demonstrate this in Sec. 2.

Our theory and algorithms are realized in CompoSAT, a new extension to the Alloy Analyzer ecosystem. CompoSAT directs users to high-coverage models that exercise contingencies in the specification, rather than ignoring contingent behavior (like a minimal model-finder) or potentially concealing them in a stream of mediocrity (like Alloy's default enumeration). Similarly to classical coverage, this approach can also reveal when portions of the specification are never exercised by any output model. Finally, as we will show, coverage is particularly suited to detecting overconstraint bugs, which the online Alloy tutorial [15] goes so far as to call the "bane of declarative modeling".

2 Motivation and Example

We first show a small running example: an address book for email contacts. (Sec. 8 examines more substantial, real-world specifications.) This example is similar to others in Jackson [1, Chapter 2] but, for brevity, is less complete.

```
1  abstract sig Target {}
2  abstract sig Name extends Target {}
3  sig Alias, Group extends Name {}
4  sig Addr extends Target {}
5  one sig Book {
6    entries: Name -> Target
7  } {
8    all n: Name | n not in n.^entries -- No cycles
9    all a: Name | some a.entries -- Names denote
10 }
11 run {}
```

Lines 1–4 declare types (called `sig`s in Alloy). The `abstract` keyword says that a type is equal to the union of its subtypes. `Addr`s are meant to denote

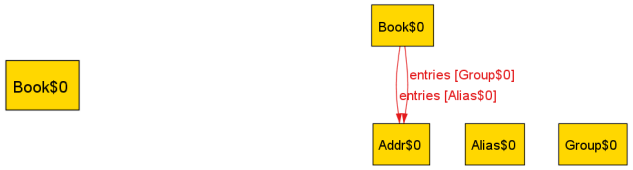

Fig. 1. The two-model ensemble produced for the specification of Sec. 2.

actual email addresses; `Name`s denote `Alias`es or `Group`s that address books will translate. At this stage, only one `Book` (lines 5–10) is allowed in each model. Books have an `entries` field—a relation between `Name`s and `Target`s. Line 8 says that no `Name` atom is self-reachable via `entries` (`^` denotes transitive closure and `->` means product). Line 9 says that the book contains entries for all `Name`s.

The final line tells Alloy to run the specification and produce models up to the default size bound: 3 of each top-level `sig`. At this bound, Alloy produces 21 non-isomorphic models—daunting to page through, even at this small scale. On the other hand, a *minimal* model finder would produce only one: the leftmost model in Fig. 1, which contains nothing but an empty `Book`. This is because no constraint forces `Name`s to exist, in spite of the fact that lines 9 and 10 apply if any do. Neither solution is ideal: one risks overloading the user with a huge example set, but the other hides important insight into how constraints behave.

CompoSAT stakes out a position between these two extreme approaches. The high-coverage ensemble it generates contains *two* models (both shown in Fig. 1). These are not chosen arbitrarily: they together demonstrate—in a way we will make precise in the next sections—*all possible ways* the constraints force truth or falsity in models of the specification.

3 Adapting Coverage to Model Finding

Coverage for *software* [14] measures test-suite quality in terms of the statements or branches of the program it exercises. Obtaining a similar definition for model finding is subtle. We might start by defining the coverage of a model analogously to *line coverage* in software by saying that a top-level constraint (e.g., line 8 in Sec. 2) is covered by a model if it is true in that model. But this is unhelpful since *all* models found must make every such constraint true!

The difficulty is that, even if a constraint is true in a model, it does not *directly determine* the contents of that model in the same way that executing a program statement determines that program's behavior. We would nevertheless like to capture an analogous intuition. Our solution has two components.

First, we focus on the non-determinism in constraints that comes from disjunction, implication, existential quantifiers, etc. Each disjunctive choice is analogous to a branch-point in a program, so we might imagine defining a coverage metric that captures the different ways that models satisfy constraints. Unfortunately,

this approach is computationally infeasible: if a specification has 100 possible branches—a modest number compared to many—there are up to $2^{100} - 1$ paths to cover, and in the worst case just as many models to present! Moreover, the fact that a constraint was satisfied does not imply that it actually had *impact in the model*. For instance, consider the constraint-set `(A or B) and (A and B)`. The first conjunct is satisfied in the model { `A, B` } but without the second conjunct, either literal could be consistently negated.

We therefore restrict our attention to branches that force what we call *local necessity* in the model (we make this precise in Sec. 4). Informally, a portion of a model is locally necessary if altering it would violate the specification in context of the rest of the model. For instance, in the right model of Fig. 1, the address-book entries are locally necessary because (1) a constraint said every `Name` must have an entry and (2) there were no other entries for that `Name` already present. CompoSAT takes the view that a model is "good" to the extent that it provides new demonstrations of local necessity.

Note that a granularity of top level constraints (rather than branches) is too broad for even the simple example in Sec. 2. CompoSAT must be able to distinguish different ways that constraints apply—some, but not all, of which may be unexpected or otherwise useful to show. Because branches often involve expansion and instantiation of high-level constraints, *we cannot define coverage only in terms of source locations* and must instead work with logical formulas.

4 Foundations

Formally, Alloy specifications are theories of relational logic with transitive closure. Their syntax includes the usual Boolean connectives (conjunction, negation, etc.) along with first-order quantification (`all`, `some`) and relational operators (product, join, transpose, transitive closure, etc.). Readers interested in the full grammar of Alloy are encouraged to peruse Jackson [1] or the Alloy documentation [16].

Given a specification $\mathcal{T}$, its *satisfying models*, denoted $Mod(\mathcal{T})$, are the set of finite first-order models[1] $\mathbb{M}$ that satisfy it, i.e., in which it evaluates to true. Truth in a model is defined in the usual, recursive manner [18], e.g. the constraint $\alpha \wedge \beta$ is true in model $\mathbb{M}$ if and only if both α and β are true in $\mathbb{M}$.

4.1 Bounded Model Finding

These definitions mean that the model-finding is just the (finite) satisfiability problem for first-order logic with transitive closure. Unfortunately, this is well-known to be undecidable [19] in general. In order to render satisfiability checking feasible, Alloy performs *bounded* model-finding. In addition to a specification and the name of a predicate to run (essentially just an additional constraint in the specification), users must provide an explicit bound B be given on all

[1] We use the term *model* in its mathematical sense: a relational structure over a set. We follow Milicevic [17] and others in referring to the constraint-set as the *specification*.

top-level `sig`s to check up to. Once given these numeric bounds, Alloy creates concrete atoms (like `Alias$1`, `Book$0`, etc.) that populate the potential universe of models within the bounds given. This yields a finite search space.

We will implicitly enrich the language with a distinct constant for every element in the generated bounds and abuse notation somewhat to write formulas involving these constants, e.g., "`Alias$1 -> Addr$2 in entries`". When a formula only expresses whether a product of atoms is in some `sig` or field name, we will refer to it as *atomic*. A *literal* is either an atomic formula or its negation. The vital intuition here is that each atomic formula is essentially bound to a Boolean variable in each model, since under a bound every specification is a propositional theory—which is what enables Alloy's use of SAT-solving technology. Finally, the *diagram* of a model $\mathbb{M}$, which we denote as $\Delta(\mathbb{M})$, is the set of literals true in $\mathbb{M}$.

4.2 Local Necessity and Provenance

Our goal is to identify when constraints have direct impact upon model contents. To make this intuition precise, we need to define what it means to have "impact". We introduce the following helpful definitions from prior work [20].

Definition 1 (L-alternate model). *Let $\mathcal{T}$ be a specification and $\mathbb{M}$ a model satisfying it. Let L be a literal true in $\mathbb{M}$. The* L-alternate *of $\mathbb{M}$, $\mathbb{M}^{\mathbb{L}}$, is the model with the same universe as $\mathbb{M}$ but with $\Delta(\mathbb{M}^{\mathbb{L}}) = (\Delta(\mathbb{M}) \setminus \{L\}) \cup \{\neg L\}$.*

Definition 2 (Local Necessity). *L is said to be* locally necessary *for $\mathcal{T}$ in $\mathbb{M}$ if and if only if $\mathbb{M}$ makes $\mathcal{T}$ true but $\mathbb{M}^{\mathbb{L}}$ does not.*

That is, a literal is locally necessary if changing its value from positive to negative (or vice versa) would necessitate other changes in the model. This means that whenever a literal is locally necessary, some constraint forces it to hold in the context of the larger model. Local necessity is far weaker than entailment: $\mathcal{T}$ may have models whose diagram contains L's negation, but L cannot be consistently changed *in this particular model* without other concurrent changes.

Constraints may act to force L in different ways. We make this precise by defining *provenances* as particular conditions that satisfy constraint branches and cause L to be locally necessary:

Definition 3 (Provenance). *A* provenance *for L in $\mathbb{M}$ with respect to $\mathcal{T}$ is a set of sentences $\alpha_1, ..., \alpha_n$ where each α_i is true in both $\mathbb{M}$ and $\mathbb{M}^{\mathbb{L}}$ such that $\mathcal{T} \wedge \alpha_1 \wedge ... \wedge \alpha_n$ entails L under the user-provided bounds B.*

For instance, the constraint on line 9 of Sec. 2's example, under the rightmost model in Fig. 1, induces the provenance { `Group$0 in Name` } for the local necessity of the book entry `Group$0->Addr$0`—because the element `Group$0` is a valid instantiation of the quantifier.

We restrict our attention to provenances that are partial instantiations and expansions of original constraints. Although in principle we could consider provinances that are fully expanded into conjunctions of literals, this would result in a plethora of provenances, many of which would imply each other modulo $\mathcal{T}$.

$$\mathbb{E}_+(\beta \vee \gamma) = \begin{cases} \mathbb{E}_+(\beta) \vee \mathbb{E}_+(\gamma) & \text{if } \mathbb{M} \models \beta \text{ and } \mathbb{M} \models \gamma \\ \mathbb{E}_+(\beta) & \text{if } \mathbb{M} \models \beta \text{ and } \mathbb{M} \not\models \gamma \\ \mathbb{E}_+(\gamma) & \text{if } \mathbb{M} \not\models \beta \text{ and } \mathbb{M} \models \gamma \end{cases}$$

$$\mathbb{E}_+(\beta \wedge \gamma) = \mathbb{E}_+(\beta) \wedge \mathbb{E}_+(\gamma)$$

$$\mathbb{E}_+(\neg\beta) = \neg\mathbb{E}_-(\beta)$$

$$\mathbb{E}_+(L) = L \text{ where } L \text{ is a literal}$$

Fig. 2. Expansion function $\mathbb{E}_+$for expanding provenance formulas in a positive context. For brevity, we omit the symmetric negative-context $\mathbb{E}_-$, as well as cases for routine syntactic sugar like bi-implication. Quantifiers are eliminated by instantiation up to the (always finite) pertinent upper bound.

5 Algorithmics of Coverage

We might stop with the definitions in Sec. 4 and define the provenance-coverage of a model $\mathbb{M}$, denoted $\mathit{Provs}(\mathbb{M})$, to be the set of provenances that it induces across all literals. However, this definition proves to be unsatisfying. We discuss and address three improvements that make it more practicable.

5.1 Expansion

Consider the constraint `all a: Name | some a.entries` in Sec. 2's example, along with the rightmost model in Fig. 1. Why is it locally necessary that the `Book`'s `entries` contain the tuple `Alias$0 -> Addr$0`? Because: **(1)** `Alias$0` is a `Name` and thus the variable `a` can be bound to it, and **(2)** there are no other extant entries for `Alias$0`. Without some means of telling the two cases apart, the provenance for the `Group$0` entry is identical. That is, a model with (say) only `Group`s would cover the constraint applying to an `Alias`.

To account for this and other (possibly nested) disjunction[2] in provenance formulas, we expand each formula to reflect which disjunctive branches satisfy it in $\mathbb{M}$. In the above example, then, the first formula becomes either `Alias$0 in Alias` or `Group$0 in Group`, depending on context. This difference reflects the different insight these two provenances bring. Fig. 2 gives the expansion function $\mathbb{E}_+$, which maps formulas to expanded formulas, in more detail. The input formula must be an α from some provenance (and thus true in both $\mathbb{M}$ and $\mathbb{M}^{\mathbb{L}}$). The output formula is fully desugared and instantiated, in that it will only contain the operators $\wedge$, $\vee$, $\neg$, and literals.

5.2 Canonicalization

While expansion makes provenance-coverage more fine-grained, there are cases in which we need to do the opposite and gloss over differences. For instance,

[2] The original constraint is equivalent to `all a: Alias+Group | some a.entries`.

two models that are identical except for atom names will have differing provenance sets, yet their provenances give the same information. To eliminate this and other similar issues caused by atom names, we *canonicalize* provenances (post-expansion) to eliminate variation in atom name. This amounts to simple substitution: replacing atom names and integer constants with canonical representatives. A provenance that has undergone both expansion and canonicalization is called a *provenance skeleton*.

5.3 Coverage and Subsumption

The *provenance coverage* of a model $\mathbb{M}$, $Provs(\mathbb{M})$, is a set of provenance skeletons:

$$\{p | p \text{ is a provenance skeleton for some literal } L \text{ in } \mathbb{M} \text{ with respect to } \mathcal{T}\}$$

We lift this to a *set* E of models: $Provs(E) \triangleq \bigcup_{\mathbb{M} \in E} Provs(\mathbb{M})$. It is now reasonable to speak of one set (i.e., ensemble) of models providing more provenance information than another. Naturally, $Mod(\mathcal{T})$ has the largest provenance coverage for $\mathcal{T}$. As user time and attention is precious, the ideal goal is thus to find a *minimal* set of models E with the same coverage.

While attempting to cover a set of skeletons, we observe that some contain strictly more information than others. By not attempting to cover superfluous skeletons, we can reduce the runtime and memory requirements of coverage computation (Sec. 6), and even the eventual ensemble size.

Consider the (propositional) constraint `(q or r) implies s` and three models that satisfy it: $\mathbb{M}_1$, $\mathbb{M}_2$, and $\mathbb{M}_3$ with diagrams $\Delta(\mathbb{M}_1) = \{q, \neg r, s\}$, $\Delta(\mathbb{M}_2) = \{\neg q, r, s\}$, and $\Delta(\mathbb{M}_3) = \{q, r, s\}$. The literal s is locally necessary in all three. Let p_1, p_2, p_3 be provenance skeletons for s from $\mathbb{M}_1$, $\mathbb{M}_2$, and $\mathbb{M}_3$ respectively. In explaining why s holds, p_3 gives q and r as a reason, while p_1 and p_2 give either q or r but not both. Because p_3 blames a strict superset of branches that p_1 and p_2 do, we say that p_3 *subsumes* p_1 and p_2. Formally, the provenance subsumption relation $\leq$ is a preorder on provenance skeletons. Given two provenance skeletons for the same literal $P_1 = \{\alpha_1, ..., \alpha_n\}$ and $P_2 = \{\beta_1, ..., \beta_m\}$, $P_1 \leq P_2$ iff $\forall \alpha_i, \exists \beta_j, \alpha_i \leq \beta_j$, where the subsumption relation on subformulas is given in Fig. 3. If $p_i \leq p_j$ and $p_j \not\leq p_i$, then p_i strictly contains information less than p_j, so we can safely remove p_i from consideration.

$$L_1 \leq L_2 \triangleq L_1 = L_2 \text{ for literals } L_1, L_2$$

$$\bigvee_i^n A_i \leq \bigvee_j^m B_j \triangleq \forall A_i, \exists B_j, A_i \leq B_j$$

$$\neg A \leq \neg B \triangleq A \leq B$$

Fig. 3. Subformula subsumption relation $\leq$. (The conjunction case is defined similarly.)

6 Implementation

CompoSAT is implemented as an extension to Alloy, leveraging our Amalgam [20] provenance-generation toolkit. High-coverage ensembles are enabled via menu options. We made this design choice so that users can seamlessly transition from default Alloy to CompoSAT and back again without disruption—or even exiting the tool. CompoSAT supports the same rich subset of Alloy that Amalgam does, namely relational and boolean operators, transitive closure, set cardinality and numeric inequalities without arithmetic.

Given a (user-defined) time budget, the tool first enumerates as many models as possible, behind the scenes, via the underlying solver. As each model arrives, CompoSAT generates all provenances via Amalgam, then performs expansion and canonicalization to produce a skeleton set for each model. A subsumption check then removes extraneous skeletons. Once the time limit or the supply of models has expired, CompoSAT solves the set-cover problem to produce the optimal ensemble. We use the Z3 [21] solver for this purpose.

In principle, a specification may have far more models than can feasibly be enumerated. In such cases, CompoSAT reports that its enumeration was incomplete: enumeration has produced only a subset of $Mod(\mathcal{T})$. Thus, the provenances obtained form a subset of $Provs(Mod(\mathcal{T}))$. However, even when the provenances obtained are a *strict* subset of $Provs(Mod(\mathcal{T}))$, our evaluation (Sec. 8) shows that the tool still often produces an ensemble that is demonstrably better than what Alloy's enumeration would provide, skipping over dozens or hundreds of models that give no new provenance information. Moreover, as we will discuss in Sec. 7, even an incomplete high-coverage enumeration can be useful in revealing errors and giving modelers new insight.

7 Qualitative Use Case: Overconstraint

We now discuss two key qualitative advantages of high-coverage ensembles. Both are related to a class of specification bug called overconstraint. A specification is said to be *under*constrained if it is satisfied by unintended models and *over*constrained if some intended models do not satisfy it. One of the advantages of model finding is that underconstraint can be discovered by simply viewing surprising models. Overconstraint, however, is more challenging: *missing* models cannot be discovered without iterating through all of them and then remembering what was never seen. For specifications with many models, this is impractical.

Both of these issues are easy to accidentally introduce when refining constraints. To mitigate this risk, experienced Alloy users often run testing predicates that characterize models that should or should not satisfy the specification. Surprising results then indicate over- and underconstraint respectively. But this technique requires anticipating potential problems in advance; unexpected bugs and failures of intuition can still occur. Moreover, one of the strengths of Alloy is that it facilitates both in-depth, detailed analysis and lightweight experimentation. In the latter case, codifying detailed expectations can be premature.

7.1 Detecting Overconstraint via Local Necessity

Overconstraint can often appear as unexpected local necessity. In such cases, showing the user which portions of their models are locally necessary can lead to surprise and insight. For example, suppose that the constraint on line 9 in Sec. 2 overconstrains: it was meant to say that the book must contain an entry for all `Aliases`, but empty `Groups` should be permitted. Unfortunately, the user has committed a common error and quantified over a too-general type: `Name`. Alloy will never produce an unexpected model due to this error, but it will neglect models that the author expected to see—those with empty `Groups`.

Showing the rightmost model in Fig. 1 along with the information that `Group$0`'s entry is forced by the constraint on line 9 points immediately to the issue and suggests possible sites to implement a fix. Amalgam, which CompoSAT invokes for provenance generation, does exactly this. However, Amalgam's feedback is per-model, and so this useful information will *never appear* if suspect models come late. (We have observed empirically that most users rarely view more than the first few models *unless they have reason to believe they have made an error*. This is especially true for more complex specifications where models may require time and effort to understand.)

In this case, the first suspect model Alloy, and therefore Amalgam, produces is the sixth—i.e., the user must click "Next" five times to have any chance of discovering the bug, even with local-necessity highlighting.[3] But as mentioned in Sec. 2, CompoSAT produces an ensemble of only *two* models, one of which contains the suspect necessity. Coverage thus enables far superior model enumeration in this context.

7.2 Highlighting Uncovered Constraints

High-coverage ensembles have power even without displaying local necessity. If a constraint can never contribute to a provenance, it may indicate either that the uncovered constraint is too weak or that some other constraint overshadows it. Space limitations preclude a full example, but consider again the propositional constraint-set `(A or B) and (A and B)`. The first conjunct can never contribute to a provenance since the second conjunct only admits the model { `A, B` }. This benefit is similar to that observed by Torlak [23], except that we consider highlighting induced by constraint coverage rather than unsatisfiable cores.

8 Evaluation

We evaluate CompoSAT with a variety of metrics in order to answer 3 research questions. Namely: **(RQ1)** Can a relatively small ensemble of models cover a large quantity of total provenances? **(RQ2)** Can model enumeration reasonably be used to discover the provenances of a specification? **(RQ3)** How much coverage does

[3] Model-ordering is dependent on solver engine and other settings. Here, we use Minisat [22] with unsatisfiable cores and symmetry-breaking enabled.

enumerating only *minimal* models achieve? Together, these questions evaluate the practicality of high-coverage ensembles as well as comparing them against the two extremes discussed in Sec. 2: *all* models and *minimal* models.

8.1 Experimental Setup

We evaluate on a wide variety of specifications that exercise different Alloy operations and represent multiple domains. When examples came with only unsatisfiable commands (e.g., properties without counterexamples) we added a command `run {}` to enumerate all models to the default bound.

Paper is the example from Sec. 2. From the Amalgam suite, we include colored, undirected trees (**ctrees**), directed graph (**digraph**), directed tree (**dtree**, **dtbug**), a logic puzzle (**abc**), trees without a vertex of degree 2 (**gwh**), and two labs from an Alloy course: transitive closure and garbage collection (**tclab** and **gclab**). Bad employee (**bempl**), grade book (**grade**), and other groups (**other**) originally come from Saghafi, et al. [11]. Address book (**addr**), geneology (**gene**) and own-grandpa (**grand**) come from the basic examples in the Alloy distribution. Hotel-locking (**hl4**), ring election (**elect**), media asset management (**media**), and memory (**simplem**, **fixedm**, **cachem**) come from the longer case-studies in Jackson [1]. Flowlog [9] (**flow**) specifies a program written in Flowlog, a software-defined network programming language. Model finding reveals a bug in the program. Our UML diagram specifications (**uml1**, **uml2**) come from Maoz, et al. [3], and the semantic differences between those diagrams (**cddiff1**, **cddiff2**) likewise come from Maoz, et al.'s CDDiff [4].

Various tools perform automatic compilation to Alloy from other input languages—the UML, Semantic Differences, and Flowlog specifications were all machine generated in this way. Since specification errors can be introduced by these compilers, improved model output benefits not only end-users but also compiler authors, who must have high confidence in their translation.

All experiments were performed on a Xeon E3-1240 v5 CPU at 3.50 GHz running Debian 9.1. No experiment but **media** consumed more than 4 GB of memory; for uniformity, we terminated **media** when recording its many large provenances exceeded 4 GB. Fig. 4 summarizes the results.

8.2 RQ1: Do small ensembles suffice?

For each experiment, column 5 (*Ensm size*) shows the ensemble sizes needed to achieve 50%, 70%, 90%, and 100% coverage discovered of provenance skeletons. To compute these, we enumerate models and find the optimal 100% ensemble as specified in Sec. 6. We then order each model in the ensemble by number of new skeletons and report how far into the ensemble the pertinent coverage level is reached. This is a conservative metric, as a better ensemble might exist for, say, 50% coverage than a 50% subset of the the optimal 100% ensemble. Low sizes may therefore be especially encouraging.

The largest 100% ensembles by far belong to incompletely enumerated experiments: **media** (16 and 19) being the highest. It is possible that high-coverage

Spec	Command and Max bound	# skels.	All?	Time (s) [# mdls] to get 50%	70%	90%	100%	Ensm size 50%	70%	90%	100%	Total enum	Total time (s)	Minimal # mdls	Minimal skels. [missed]
paper	`{} 3`	5		<1 [4]	<1 [4]	<1 [6]	<1 [6]	1	1	1	2	21	<1	1	1 [4]
ctrees	`{} 3`	17		<1 [2]	<1 [4]	<1 [4]	<1 [5]	1	1	2	2	12	<1	1	3 [14]
ctreesb	`{} 3`	16		<1 [1]	<1 [6]	<1 [6]	<1 [6]	1	1	2	2	14	<1	1	3 [13]
digraph	`test 4`	5		<1 [1]	<1 [1]	<1 [5]	<1 [5]	1	1	1	2	6343	19	1	3 [2]
dtree	`partialTree 7`	6		<1 [2]	<1 [2]	<1 [2]	<1 [2]	1	1	1	1	2	<1	1	1 [5]
dtbug	`isDTBug 4`	10		<1 [2]	<1 [2]	<1 [2]	<1 [2]	1	1	1	1	15	<1	1	2 [10]
tclab	`connectedK 3`	28		<1 [1]	<1 [1]	<1 [3]	<1 [5]	1	1	2	3	55	2	1	23 [5]
gclab	`completeness 6`	34	✗	<1 [1]	1 [2]	4 [36]	338 [2092]	1	1	1	2	21904	3600	> 3000	34 [2]
abc	`{} 3`	13		<1 [1]	<1 [1]	<1 [1]	<1 [1]	1	1	1	1	2	<1	2	12 [1]
gwh	`{} 6`	10		<1 [1]	<1 [1]	<1 [1]	<1 [1]	1	1	1	1	2	1	2	10 [0]
grade	`noGradeOwn 3`	25		<1 [1]	<1 [1]	<1 [1]	<1 [1]	1	1	1	1	2428	41	1	25 [0]
bempl	`noThief 3`	21		<1 [1]	<1 [1]	<1 [2]	<1 [6]	1	1	1	1	33283	432	3	18 [5]
other	`noThief 3`	10		<1 [1]	<1 [2]	5 [581]	6 [682]	1	1	1	1	1620	10	1	9 [3]
ring	`{} 6`	7		<1 [1]	<1 [1]	<1 [1]	<1 [1]	1	1	1	1	52	4	9	7 [0]
addr	`inAddr 4`	16	✗	<1 [7]	2 [37]	39 [820]	495 [7700]	1	1	1	2	47339	3600	1	8 [11]
grand	`ownGrandpa 4`	36		<1 [1]	<1 [1]	<1 [1]	1 [4]	1	1	1	2	36	5	36	36 [0]
gene	`Show 6`	37		1 [1]	1 [1]	1 [1]	1 [1]	1	1	1	1	768	321	768	37 [0]
hl4	`{} 3`	245		1 [1]	3 [3]	34 [38]	67 [70]	1	2	3	6	168	166	3	182 [66]
elect-1	`AtLeastOne 7`	193	✗	15 [6]	24 [10]	437 [225]	1304 [619]	2	4	7	13	1568	3600	184	193 [0]
elect-2	`looplessPath 12`	179		11 [1]	11 [1]	21 [2]	31 [3]	1	1	2	3	3	31	3	179 [0]
media	`CutPaste 3`	674	M	754 [15]	1652 [33]	1899 [38]	2076 [41]	5	8	10	16	41	2077	6	150 [640]
media	`PasteCut 3`	639	M	626 [11]	1184 [22]	3013 [64]	3582 [75]	5	7	9	19	75	3583	14	221 [535]
simplem	`{} 3`	13		<1 [9]	<1 [18]	<1 [29]	1 [68]	1	1	1	2	5732	139	1	0 [13]
fixedm	`{} 3`	13		<1 [1]	<1 [1]	<1 [6]	<1 [22]	1	1	1	2	769	24	1	0 [13]
cachem	`{} 3`	12	✗	<1 [1]	<1 [1]	1 [13]	1 [13]	1	1	1	1	86035	3600	1	0 [12]
uml1	`cd 10`	79	✗	7 [1]	7 [1]	11 [2]	530 [104]	1	1	1	3	457	3600	2	40 [39]
uml2	`cd 10`	48	✗	2 [2]	19 [18]	19 [18]	786 [659]	1	1	2	3	1829	3600	3	39 [9]
cddiff1	`M1minusM2 6`	80		4 [2]	4 [2]	4 [2]	181 [86]	1	1	1	4	385	827	5	74 [6]
cddiff2	`M2minusM1 6`	63		14 [5]	42 [14]	42 [14]	188 [64]	1	1	1	6	70	211	1	26 [38]
flow	`isTCReallyTC 4`	223	✗	18 [2]	43 [5]	115 [13]	1345 [144]	1	2	5	12	383	3600	12	170 [58]

Fig. 4. Summary of results. Each row describes a separate experiment on a distinct specification, predicate (or assertion) and bounds. *Max bound* denotes the highest bound used. We report the *maximum* because model-generation time tends to be exponential in the bounds. We stop enumeration when either all models have been processed or after *one hour* has passed. Column 3 (**All?**) is blank if all models were enumerated (i.e., the true set of provenances is known). It contains a ✗ if enumeration (for that experiment) terminated after one hour and an **M** if enumeration is incomplete due to the 4 GB memory limit.

models (which would reduce the ensemble size) remain undiscovered, but this will not always be true. These specifications are complex, with many skeletons up to subsumption. We thus conclude that 100%-coverage ensembles are necessarily large for some specifications. However, we temper this with two observations.

When we compare even these relatively high ensemble sizes with the total number of models enumerated in the experiment, we see significant reduction in the number of models shown. Even in **media**, the worst case, we see a 3-fold reduction, and in **flow** a 10-fold reduction. Oftentimes, we see a 100x (**fixedm**), 1000x (**digraph**) or even 10000x (**gclab**) reduction. We also observe that ensemble sizes at the 70% and 90% coverage levels are far more manageable. With the exception of **media**, the 90% coverage ensembles are all under 10 models and 70% coverage ensembles are no larger than 4—for most, only 1 or 2.

Finally, we see in Column 4 that some experiments find relatively high-coverage models within the first few enumerated. However, the lion's share of these occur in specifications with relatively few total models (**gwh** with 2 total or **elect-2** with 3). To achieve 90% coverage almost invariably requires a large number of models be enumerated. This demonstrates the truth of our hypothesis that automatic enumeration can filter valuable models from the chaff. We defer questions of *time to generate* ensembles to the next section, as it separately evaluates the viability of model-enumeration.

8.3 RQ2: Is enumeration effective?

For each experiment, column 4 reports the time taken and number of models enumerated before reaching 50%, 70%, 90%, and 100% coverage. This is subtle since subsumption reduces the number of overall skeletons during enumeration.

If we reported percentages *without subsumption*, the number of skeletons shown would not reflect our actual coverage computation and inaccurately inflate ensemble size. Yet if we reported percentages of the total up to subsumption *by all skeletons found*, it would introduce a pro-CompoSAT bias into these results. To see why, suppose **(A)** the first model contained provenances $\{p_1, p_2\}$ and the thousandth model contained p_3 which subsumed p_1 and p_2. Then there would be only one skeleton up to subsumption, not seen until the thousandth model—but two of the three skeletons in total manifest in the first model. Now consider **(B)** the opposite case: the first model contains p_3 and the thousandth contains $\{p_1, p_2\}$, which are both subsumed by p_3. Our evaluation should make clear that the first model achieves 100% coverage, not merely 33%.

Our measurements therefore take subsumption into account only up to the current model $\mathbb{M}$; skeletons enumerated later will contribute to the denominator (reducing the coverage of $\mathbb{M}$) but no skeleton will be subsumed by one as-yet-unseen. This means that in **(A)**, the first model reaches 66% coverage and in **(B)** the first model achieves 100%—as expected. This approach ensures that early models receive "credit" for skeletons they exhibit even if these are later subsumed.

Columns 6 and 7 report the number of models enumerated and the total time spent. These numbers are often different from the 100% sub-column in column

4 because there we are measuring only how long it takes to reach full coverage. Even if all skeletons are seen early, our experiments still run to completion.

The results here somewhat echo Sec. 8.2 above. For completely enumerated experiments, with one exception (**cddiff** at roughly 3 minutes) enumeration produces 100% coverage ensembles in under 70 seconds. We also see many incomplete experiments (e.g., **gclab**) reaching full coverage (relative to skeletons discovered) far quicker than their duration. A small, truly high-coverage ensemble may be worth the wait. Even if not, the time to achieve 70% and 90% coverage is far more modest across the board, with 27 of 29 experiments reaching 70% in under one minute. **Media** remains an outlier, with new skeletons appearing up until the very end of the enumeration process. This happens because, in **media**, most models only contain a small handful of skeletons. We observe that this case is not common.

In incomplete cases, it is possible that new skeletons (or superior models) could be discovered with further enumeration. This is quite likely for **media**. In others, such as **uml1** (457 enumerated, last skeleton at 104) we see a long trail of enumeration after a relatively early final skeleton discovery—making it more likely, although not certain, that few skeletons remain undiscovered.

Overall, although enumeration has its weaknesses, it appears to be effective for producing high-coverage ensembles in practice. Indeed, as *the only other available option* in Alloy at present is *manual* enumeration, CompoSAT's approach is, at worst, automating that process to produce optimal coverage.

8.4 RQ3: Minimal Model Coverage

Column 8 reports the number of *minimal* models for each experiment, collected via the Aluminum [10] model-finder. It also gives the total provenance skeletons found in these. The bracketed number says how many skeletons (column 2) were *not subsumed* by any skeleton in a minimal model. Because these numbers are computed up to subsumption, the found and unfound skeletons need not always total the value in column 2. Where minimal models do well, it is for one of two reasons. Some specifications (e.g., **gene**) are so constrained that all satisfying models are minimal. Others (e.g., **grade**) contain no implications with unnecessary antecedents; here, minimality covers all possible provenances.

For the remaining 23 experiments, minimal models omit swathes of provenance skeletons. Minimal-model enumeration was complete for all except **gclab**. Thus, the bracketed numbers are a strict *lower* bound on the amount of coverage neglected in all other incomplete cases; minimal models can do no better than we report here. Finally, we note that the number of minimal models is often far larger than the 100% ensemble, and when there are fewer minimal models (as in Sec. 2), their coverage is under 100% in every case.

9 Related Work

Model finders fall into two classes: SEM-style [24], which reasons about a surface logic directly, and MACE-style [25], which compiles to Boolean logic and applies

a SAT solver. Alloy and its internal engine, Kodkod [26], take the latter approach. Our work is not SAT-specific and could apply to either group.

Model Quality Some effort has been applied to improving output-model quality. Aluminum [10] and Razor [11] present only *minimal* models in an attempt to reduce distracting example bloat. The Cryptographic Protocol Shapes Analyzer [27] is a domain-specific model-finder that produces minimal illustrative protocol runs. Minimal models only contain locally-necessary *positive* literals—everything present has a provenance, but negative literals may not be locally necessary. CompoSAT is more general, and can detect when the specification disallows either adding or removing elements. Target-Oriented Model Finding [12, 28] (TOMF) minimizes distance from user-defined targets, enabling, e.g., maximal models. Bordeaux [29] uses relative minimization to find near-miss models that fail to satisfy the specification but nearly do so in terms of edit distance. Our approach differs starkly from all these as it is *syntax-guided* rather than purely semantic.

Alloy and other model-finders endeavor to suppress models that are *isomorphic* to one already presented. Such symmetry-breaking [30, 31] increases the quality of the stream of models shown, but in a way orthogonal to ours.

Coverage in Other Settings Coverage [14] has been a valued metric for test suites since at least the 1960s [32]. While coverage is not without its weaknesses [33], some of which we share (Sec. 10), it provides powerful insight. Our work is the first to explore what it means for *models* to cover *constraints*.

Concolic testing [34–36] is a coverage-driven technique close to ours in spirit. It marries *con*crete test generation and symb*olic* execution [37] to generate high-coverage test suites for programs. CompoSAT operates on declarative constraints rather than code: there may be no "execution" due to often-deliberate underconstraint and the fact that not all specifications are temporal.

Coverage has also been applied (e.g., by Hoskote, et al. [38]) to model-checking to measure how much of a system is exercised by properties. This improves the set of properties to check, not counterexample quality or overconstraint detection. Others [39–42] use declarative specifications to aid in testing *programs*, whereas we are concerned with helping modelers debug the specification itself. As these approaches rely on a correct specification, CompoSAT is not only orthogonal, but also potentially *complementary* to specification-aided test generation.

Coverage for Declarative Specifications Detecting vacuity [43, 44], which can cause constraints to never apply or properties to be unhelpfully true, can be seen as another coverage analysis. Heaven and Russo [45] detect vacuity and other bug-prone patterns in specifications. However, their work is focused on detecting patterns, not optimizing output. Torlak, et al. [23] improve Alloy's unsat-core highlighting, which can be viewed as a coverage metric that applies only to unsatisfiabile specifications. They observe that a suspiciously small core suggests a problem with the property or the bounds given. This insight is useful to debug unsatisfiable results but does not apply to improving models.

AUnit [46, 47] also takes inspiration from code coverage, but differs in foundation and execution. In AUnit, coverage atoms correspond to truth of subformulas

and cardinality of subexpressions. In our analogy, this is a refinement of statement coverage. CompoSAT considers sets of subexpressions that capture unique ways in which models are constrained, analogously to path coverage. Moreover, AUnit enumerates models via SAT invocations until all coverage atoms are seen; CompoSAT post-filters Alloy's default enumeration process. Scenario Tour [48] generates models using a combinatorial test-generation strategy. Combinations comprise a pair of relations having specific cardinalities (empty, singleton, and higher) in models found. This interesting approach is nevertheless more related to pairwise test generation than statement or path coverage.

Provenance Provenance for databases was introduced by Cui and Widom [49] as the set of tuples in a source database that contribute to a tuple's presence in a query answer. Variations exist, e.g., Buneman et al. [50] and others distinguish between tuples that bear responsibility from tuples that provide data in the query answer. Meliou, et al. [51, 52] find provenance for *negative* answers to conjunctive queries. One key difference between this work and CompoSAT is that specifications are strictly more expressive than conjunctive queries.

Provenance is also useful in other settings. Vermeer [53], for instance, explains assertion violations in C programs via causal traces. WhyLine [54, 55] "Why..." and "Why not..." queries about Java program behavior. Y! [56, 57] finds and presents both positive and negative provenance for network events. These tools all extract provenance from deterministic runtime logs, which possess temporal structure that models need not possess—and are not available to a model finder.

In addition to minimal model output, Razor [11] is also able to give provenance for every piece of a model. To do so, it draws on constructive model-finding ideas (the Chase [58] algorithm) while still leveraging SAT. Amalgam [20] gives provenance in arbitrary, rather than just minimal, models. Although CompoSAT uses Amalgam as an engine to generate provenances, the core topic of this work—syntax-guided model-quality criteria—is separate from provenance generation.

10 Conclusion and Discussion

We have introduced specification-guided coverage as a new metric for producing high-quality model output. We now conclude with discussion.

Coverage vs. Increased Bounds Alloy searches for models of size *up to* to given bounds. E.g, we write "`for 4 Name`" to search for models with *up to* `4 Name`s. Because of this, increasing bounds will never lose provenances. Moreover, as bounds increase, models can contain more skeletons; a higher bound often means that a smaller ensemble is possible (at the cost of more models to enumerate). In Sec. 2's specification, for instance, raising the bound to 4 reveals a single-model optimal ensemble. However, if we permit `exact` rather than upper bounds, this property fails since exact bounds omit models of smaller size.

Weaknesses of Coverage High-coverage ensembles have one significant weakness: they are entirely syntax-guided. CompoSAT may thus do poorly at revealing *under*constraint bugs: e.g., if a relation is left completely unconstrained, CompoSAT

may not demonstrate this. This is analogous to program coverage's blindness to missing complexity [59] in code, and is thus not unique to this work. We see CompoSAT as a new and powerful option in what must become a more diverse toolbox of output strategies, each focusing on a particular set of user needs.

Alternatives to Post-Processing One might wonder why CompoSAT filters Alloy's default output, rather than directly interfacing with SAT. For instance, one could add SAT clauses to find as-yet-unseen locally-necessary literals. Suppose that $\mathcal{T}$ is our specification, and we are interested in enumerating models wherein some literal L is locally necessary. We could reflect this goal by temporarily adding the constraint $L \wedge \neg\mathcal{T}[L \mapsto \bot]$ to the specification. That is, requiring L to be true in any model found, and moreover that if L were false, the specification would not be satisfied.

Example 1. Consider the propositional formula $\mathcal{T} \equiv (1 \vee 2) \wedge (\neg 1 \vee 3) \wedge (4 \vee 5)$. Suppose $L = \neg 1$. Then $\neg\mathcal{T}[L \mapsto \bot] \equiv \neg((1 \vee 2) \wedge (\bot \vee 3) \wedge (4 \vee 5)$ which is equivalent to $(\neg 1 \wedge \neg 2) \vee (\top \wedge \neg 3) \vee (\neg 4 \wedge \neg 5))$. The left and right disjuncts can be discarded since they contradict the original specification when L holds. The resulting addition forces the solver to find models where $\neg 3$ holds, which is enough to render L locally necessary.

Via this technique, CompoSAT could proceed to query SAT for locally-necessary literals in round-robin fashion, ensuring at least one provenance for each local necessity early in enumeration. Unfortunately, one literal may be locally necessary for many different reasons, so this approach, alone, would greatly reduce the granularity of coverage. CompoSAT distinguishes between different *causes* of necessity, which would be challenging to encode in SAT up to subsumption. Post-processing also allows CompoSAT to act as a modular extension to other enumeration strategies, such as TOMF [12].

Acknowlegements

We are grateful to the developers of Alloy and Kodkod, as well as Natasha Danas and Daniel J. Dougherty for useful discussions and their work on the Amalgam tool. We also thank the anonymous reviewers for their helpful remarks. This work is partially supported by the U.S. National Science Foundation.

References

1. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. Second edn. MIT Press (2012)
2. Nelson, T., Barratt, C., Dougherty, D.J., Fisler, K., Krishnamurthi, S.: The Margrave tool for firewall analysis. In: USENIX Large Installation System Administration Conference. (2010)
3. Maoz, S., Ringert, J.O., Rumpe, B.: CD2Alloy: Class diagrams analysis using Alloy revisited. In: Model Driven Engineering Languages and Systems. (2011)

4. Maoz, S., Ringert, J.O., Rumpe, B.: CDDiff: Semantic differencing for class diagrams. In: European Conference on Object Oriented Programming. (2011)
5. Akhawe, D., Barth, A., Lam, P., Mitchell, J., Song, D.: Towards a formal foundation of web security. In: IEEE Computer Security Foundations Symposium. (2010)
6. Maldonado-Lopez, F.A., Chavarriaga, J., Donoso, Y.: Detecting network policy conflicts using Alloy. In: International Conference on Abstract State Machines, Alloy, B, and Z. (2014)
7. Zave, P.: Using lightweight modeling to understand Chord. ACM Computer Communication Review **42**(2) (March 2012) 49–57
8. Ruchansky, N., Proserpio, D.: A (not) NICE way to verify the OpenFlow switch specification: Formal modelling of the OpenFlow switch using Alloy. ACM Computer Communication Review **43**(4) (August 2013) 527–528
9. Nelson, T., Ferguson, A.D., Scheer, M.J.G., Krishnamurthi, S.: Tierless programming and reasoning for software-defined networks. In: Networked Systems Design and Implementation. (2014)
10. Nelson, T., Saghafi, S., Dougherty, D.J., Fisler, K., Krishnamurthi, S.: Aluminum: Principled scenario exploration through minimality. In: International Conference on Software Engineering. (2013)
11. Saghafi, S., Danas, R., Dougherty, D.J.: Exploring theories with a model-finding assistant. In: International Conference on Automated Deduction, Springer (2015) 434–449
12. Cunha, A., Macedo, N., Guimarães, T.: Target oriented relational model finding. In: International Conference on Fundamental Approaches to Software Engineering, Springer (2014) 17–31
13. Danas, N., Nelson, T., Harrison, L., Krishnamurthi, S., Dougherty, D.J.: User studies of principled model finder output. In: Software Engineering and Formal Methods. (2017)
14. Zhu, H., Hall, P.A.V., May, J.H.R.: Software unit test coverage and adequacy. ACM Comput. Surv. **29**(4) (December 1997) 366–427
15. Alloy Team: Overconstraint—the bane of declarative modeling. http://alloy.mit.edu/alloy/tutorials/online/sidenote-overconstraint.html Accessed August 14, 2017.
16. Daniel Jackson: Alloy: a language & tool for relational models. http://alloy.mit.edu/alloy/ (2016) Accessed November 1, 2016.
17. Milicevic, A., Near, J.P., Kang, E., Jackson, D.: Alloy*: A general-purpose higher-order relational constraint solver. In: International Conference on Software Engineering. (2015)
18. Enderton, H.B.: A Mathematical Introduction to Logic. Academic Press (1972)
19. Libkin, L.: Elements of Finite Model Theory. Texts in Theoretical Computer Science. An EATCS Series. Springer (2004)
20. Nelson, T., Danas, N., Dougherty, D.J., Krishnamurthi, S.: The power of "why" and "why not": Enriching scenario exploration with provenance. In: Foundations of Software Engineering. (2017)
21. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Volume 4963 of Lecture Notes in Computer Science., Springer (2008) 337
22. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Theory and Applications of Satisfiability Testing. (2003)
23. Torlak, E., Chang, F.S.H., Jackson, D.: Finding minimal unsatisfiable cores of declarative specifications. In: International Symposium on Formal Methods (FM). (2008)

24. Zhang, J., Zhang, H.: SEM: a system for enumerating models. In: International Joint Conference On Artificial Intelligence. (1995)
25. McCune, W.: Mace4 reference manual and guide. CoRR **cs.SC/0310055** (2003)
26. Torlak, E., Jackson, D.: Kodkod: A relational model finder. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. (2007) 632–647
27. Doghmi, S.F., Guttman, J.D., Thayer, F.J.: Searching for shapes in cryptographic protocols. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. (2007)
28. Macedo, N., Cunha, A., Guimarães, T.: Exploring scenario exploration. In: International Conference on Fundamental Approaches to Software Engineering. (2015)
29. Montaghami, V., Rayside, D.: Bordeaux: A tool for thinking outside the box. In: International Conference on Fundamental Approaches to Software Engineering. (2017) 22–39
30. Crawford, J.M., Ginsberg, M.L., Luks, E.M., Roy, A.: Symmetry-breaking predicates for search problems. In: Principles of Knowledge Representation and Reasoning. (1996)
31. Shlyakhter, I.: Generating effective symmetry-breaking predicates for search problems. Discrete Applied Mathematics (2007)
32. Miller, J.C., Maloney, C.J.: Systematic mistake analysis of digital computer programs. Communications of the ACM **6**(2) (February 1963) 58–63
33. Inozemtseva, L., Holmes, R.: Coverage is not strongly correlated with test suite effectiveness. In: International Conference on Software Engineering. (2014)
34. Godefroid, P., Klarlund, N., Sen, K.: DART: directed automated random testing. In: Programming Language Design and Implementation (PLDI). (2005)
35. Larson, E., Austin, T.: High coverage detection of input-related security faults. In: USENIX Security Symposium. (2003)
36. Sen, K., Marinov, D., Agha, G.: CUTE: a concolic unit testing engine for C. In: Foundations of Software Engineering. (2005)
37. King, J.C.: Symbolic execution and program testing. Communications of the ACM **19**(7) (July 1976) 385–394
38. Hoskote, Y., Kam, T., Ho, P.H., Zhao, X.: Coverage estimation for symbolic model checking. In: Design Automation Conference. (1999)
39. Gopinath, D., Zaeem, R.N., Khurshid, S.: Improving the effectiveness of spectra-based fault localization using specifications. In: Automated Software Engineering. (2012)
40. Marinov, D., Khurshid, S.: TestEra: A novel framework for automated testing of Java programs. In: Automated Software Engineering. (2001)
41. Milicevic, A., Misailovic, S., Marinov, D., Khurshid, S.: Korat: A tool for generating structurally complex test inputs. In: International Conference on Software Engineering. (2007)
42. Shao, D., Khurshid, S., Perry, D.E.: Whispec: White-box testing of libraries using declarative specifications. In: Symposium on Library-Centric Software Design. (2007)
43. Beatty, D.L., Bryant, R.E.: Formally verifying a microprocessor using a simulation methodology. In: Design Automation Conference. (1994)
44. Beer, I., Ben-David, S., Eisner, C., Rodeh, Y.: Efficient detection of vacuity in actl formulas. In: International Conference on Computer Aided Verification. (1997) 279–290
45. Heaven, W., Russo, A.: Enhancing the Alloy analyzer with patterns of analysis. In: Workshop on Logic-based Methods in Programming Environments. (2005)

46. Sullivan, A., Zaeem, R.N., Khurshid, S., Marinov, D.: Towards a test automation framework for Alloy. In: Symposium on Model Checking of Software (SPIN). (2014) 113–116
47. Sullivan, A., Wang, K., Zaeem, R.N., Khurshid, S.: Automated test generation and mutation testing for Alloy. In: Software Testing, Verification and Validation (ICST). (2017)
48. Saeki, T., Ishikawa, F., Honiden, S.: Automatic generation of potentially pathological instances for validating Alloy models. In: International Conference on Formal Engineering Methods (ICFEM). (2016) 41–56
49. Cui, Y., Widom, J.: Practical lineage tracing in data warehouses. In: International Conference on Data Engineering. (2000)
50. Buneman, P., Khanna, S., Wang-Chiew, T.: Why and where: A characterization of data provenance. In: International Conference on Database Theory. (2001)
51. Meliou, A., Gatterbauer, W., Moore, K.F., Suciu, D.: The complexity of causality and responsibility for query answers and non-answers. Proceedings of the VLDB Endowment **4**(1) (2010) 34–45
52. Meliou, A., Gatterbauer, W., Moore, K.F., Suciu, D.: WHY SO? or WHY NO? functional causality for explaining query answers. In: VLDB workshop on Management of Uncertain Data (MUD). (2010) 3–17
53. Schwartz-Narbonne, D., Oh, C., Schäf, M., Wies, T.: VERMEER: A tool for tracing and explaining faulty C programs. In: International Conference on Software Engineering. (2015) 737–740
54. Ko, A.J., Myers, B.A.: 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 (2004) 151–158
55. Ko, A.J., Myers, B.A.: Finding causes of program output with the Java Whyline. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, ACM (2009) 1569–1578
56. Wu, Y., Zhao, M., Haeberlen, A., Zhou, W., Loo, B.T.: Diagnosing missing events in distributed systems with negative provenance. In: Conference on Communications Architectures, Protocols and Applications (SIGCOMM), ACM (2014) 383–394
57. Chen, A., Wu, Y., Haeberlen, A., Zhou, W., Loo, B.T.: Differential provenance: Better network diagnostics with reference events. In: Workshop on Hot Topics in Networks, ACM (2015) 25
58. Maier, D., Mendelzon, A.O., Sagiv, Y.: Testing implications of data dependencies. ACM Trans. Database Syst. **4**(4) (December 1979) 455–469
59. Glass, R.L.: Persistent software errors. IEEE Transactions on Software Engineering **7**(2) (1981) 162–168