Anvil: An Empirical Study of LLM-Powered Alloy Synthesis, Validation, and Repair

Yang Hong*
University of Texas at Austin
Austin, TX, USA
yang22@utexas.edu

Chenbo Yin*
University of Texas at Austin
Austin, TX, USA
chenboyin@utexas.edu

Shan Jiang
University of Texas at Austin
Austin, TX, USA
shanjiang@utexas.edu

Yulei Fu
University of Texas at Austin
Austin, TX, USA
yuleifu@utexas.edu

Sarfraz Khurshid
University of Texas at Austin
Austin, TX, USA
khurshid@ece.utexas.edu

Abstract

Declarative specifications are critical for building safe software, yet writing them correctly remains a significant challenge — particularly in languages like Alloy, whose relational first-order logic differs fundamentally from the imperative paradigms most developers know. We present Anvil, a systematic empirical study of how well large language models (LLMs) can generate, validate, and repair Alloy formulas through a three-stage pipeline. Using three Gemini models of varying scale (Gemini 3.1 Pro, Gemini 3.0 Flash, Gemini 3.1 Flash Lite) and one open-weight cross-family model (Llama 3.3 70B), we evaluate 22 synthesis tasks derived from 11 well-studied graph and binary-relation properties — chosen as a controlled baseline with unambiguous ground truths enabling rigorous equivalence checking. For each task, we ask LLMs to produce 20 structurally distinct solutions, probing compositional reasoning beyond memorization. Gemini 3.1 Pro produces correct formulas for all 11 properties, averaging 18.2 correct variants per synthesis task across the 22 tasks, while even the smallest Gemini model succeeds on 9 of 11 in English-to-Alloy. Performance varies substantially across Gemini model scales despite shared training data, indicating that capability — not recall alone — drives quality. All three Gemini models complete 11/11 Alloy sketches on the first attempt without the test cases that tools like ASketch require (Llama 3.3 70B was not evaluated on sketching). In our validation stage, including test suites in prompts reduces semantic errors by up to 34% for the smallest Gemini model, increasing overall correct outputs from 56% to 66%. In the repair stage, ARepair fixes 183 of 369 semantically erroneous Gemini outputs (50%) across the three Gemini models, and 47 of 118 for Llama 3.3 70B. Across all four models, ARepair also repairs 6 of 13 original faulty predicates (46%), a shared input set. These results demonstrate that combining LLM generation with test-guided validation and automated repair effectively produces correct Alloy models, lowering the barrier to adopting lightweight formal methods in model-driven engineering.

*Both authors contributed equally to this research.

MODELS 2026, Málaga, Spain

ACM ISBN 979-8-4007-2809-9/2026/10
https://doi.org/10.1145/3822455.3830309

CCS Concepts

• **Software and its engineering → Formal methods**; **Automatic programming**; • **Computing methodologies → Natural language processing**.

Keywords

Alloy, model-driven engineering, declarative specifications, program synthesis, program sketching, large language models, formal methods

ACM Reference Format:

Yang Hong, Chenbo Yin, Shan Jiang, Yulei Fu, and Sarfraz Khurshid. 2026. Anvil: An Empirical Study of LLM-Powered Alloy Synthesis, Validation, and Repair. In *ACM/IEEE 29th International Conference on Model Driven Engineering Languages and Systems (MODELS 2026), October 04–09, 2026, Málaga, Spain.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3822455.3830309

1 Introduction

Declarative languages have long played a key role in automating the design and implementation of safe and reliable software systems [5, 15, 21, 33, 38, 52]. However, their full potential has not yet been realized. A key issue is that their use often requires developers to learn unfamiliar notations with very different semantics from the imperative languages they use daily.

Large language models (LLMs) have substantially changed software development and are now in widespread use. However, it remains unclear how much utility they offer for writing declarative specifications in languages that have a different semantic basis, offer a higher level of abstraction, and have much less training data available than commonly used programming languages.

Our focus in this paper is Alloy [21], a well-known declarative language based on relational first-order logic with transitive closure. The Alloy analyzer provides fully automatic bounded analysis via a SAT-based backend (Section 2).

The Alloy toolset has been used in many applications over the last two decades, including software design [42, 43], test case generation [44], deep analysis of code [23], repair of faulty programs [14], specification-driven execution [45, 51, 58], security [40, 48], and networking [56]. In model-driven engineering (MDE), Alloy has been employed to express and verify metamodel constraints [42, 43] and to generate test cases from design models [44]. Despite this utility, Alloy's use remains largely confined to academia. In our experience of using and teaching Alloy, a key issue that prevents its much wider adoption is the learning curve that it poses to developers who work with imperative languages.

Our insight is that LLMs have the potential to substantially reduce the cost of writing declarative specifications, benefiting the Alloy toolset specifically, and lightweight formal methods more generally. This paper presents Anvil, a framework for creating Alloy specifications through a three-stage pipeline — generation, validation, and repair — described in Section 4.

Alloy's fully automatic SAT-based backend makes it an ideal specification language to be supported by LLMs. Their outputs can immediately be validated by the Alloy backend. For example, an output formula that has no instance (i.e., is unsatisfiable) is (most likely) invalid. As another example, when the input query to the LLM has a reference formula, the LLM's output can directly be checked for equivalence against the reference by looking for a counterexample to the asserted equivalence.

We conduct the experimental evaluation using synthesis and sketching tasks derived from 11 well-studied subject properties defined over graphs and binary relations, and employ 4 models: three from the Gemini family (Gemini 3.1 Pro, Gemini 3.0 Flash, and Gemini 3.1 Flash Lite) and one open-weight cross-family model (Llama 3.3 70B). The three same-family Gemini models isolate the effect of model scale under a shared training methodology; Llama 3.3 70B serves as an external reference that probes whether the observed trends generalize beyond a single vendor and architecture. A key aspect of our evaluation is that for each synthesis problem (from English to Alloy and from Alloy to Alloy), we ask the LLMs to generate multiple equivalent but non-identical Alloy formulas as solutions, thereby performing a deeper study of how well the LLMs handle the semantic and syntactic intricacies of the Alloy language.

The performance of the LLMs is notable for three reasons. One, without fine-tuning and without explaining what a sketch is, Gemini 3.1 Pro produces correct formulas for all 11 properties, averaging 18.2 correct variants per synthesis task across the 22 English-to-Alloy and Alloy-to-Alloy tasks, while even the smallest Gemini model succeeds on 9 of 11 properties. The three Gemini models also complete all 11 Alloy sketches on the first attempt (Llama 3.3 70B was not evaluated on sketching). Two, Alloy's notation, which allows for succinct formulation of complex relational properties, makes the language deceptively hard to master; in our experiments, failures are dominated by semantic errors rather than outright syntax issues. Three, the remaining semantic errors are still largely tractable within our pipeline: adding the test suite increases correct outputs from 56% to 66%, and ARepair repairs up to 62 of 115 faulty LLM semantic variants (54%), while repairing 6 of 13 original faulty predicates.

This paper makes the following contributions:

- **Controlled evaluation methodology.** We present Anvil, a three-stage pipeline (generation, validation, repair) and a systematic evaluation methodology that uses well-studied properties with unambiguous ground truths to enable rigorous equivalence checking — establishing a controlled baseline for future work on LLM-assisted specification.
- **Multi-task Alloy generation study.** We evaluate Anvil on three generation tasks (English to Alloy, Alloy to Alloy, Sketch to Alloy), requesting multiple structurally distinct solutions per task to probe compositional reasoning beyond single-answer recall.
- **Validation and repair analysis.** We study how test suites in prompts reshape error distributions and evaluate ARepair as a downstream engine for fixing semantically erroneous LLM outputs.
- **Model scaling analysis.** By evaluating three models from the same family (Pro, Flash, Flash Lite), we isolate the effect of model scale on declarative specification tasks, finding that capability gaps manifest primarily as semantic errors rather than syntax failures; a cross-family open-weight model (Llama 3.3 70B) further probes whether these trends generalize beyond the Gemini family.

Our results provide evidence that LLMs, combined with validation and automated repair, can assist in producing correct Alloy specifications, suggesting that lightweight formal methods can play a larger role in model-driven engineering toolchains.

2 Background

2.1 Alloy

Alloy [21] is a declarative language based on relational first-order logic with transitive closure. An Alloy model consists of *signatures* (`sig`), which declare sets and their *fields* (relations), and *predicates* (`pred`), which define named constraints. The keyword `set` in a field declaration defines an arbitrary binary relation; `one` and `lone` define a total function and a partial function, respectively.

The Alloy analyzer provides two forms of analysis. The `run` command instructs the analyzer to find an *instance* (a valuation of sets and relations) that satisfies a given predicate. The `check` command searches for a counterexample to a given assertion. Both analyses are bounded: the analyzer translates Alloy formulas to propositional logic with respect to a *scope* (a bound on the universe of discourse, default 3) and employs SAT solvers.

2.2 Alloy sketching

Program sketching [63] provides a partial implementation with placeholder *holes* that must be completed. ASketch [79] adapts sketching to Alloy, using two kinds of holes: *expression holes* (`\E,e\`) and *operator holes* (`\O,o\` or `\CO,co\`). Each hole is annotated with a set of candidate completions defined using regular expressions. Unlike our LLM-based approach, ASketch requires test cases as input to guide sketch completion.

3 Illustrative Overview

This section illustrates our approach using the irreflexive property as a running example. Consider the following Alloy model, which introduces a set `S`, a binary relation `r: S×S`, and a predicate `Irreflexive`:

```
sig S { r: set S }

pred Irreflexive {
  -- No element in S is related to itself
  all s, t: S | s->t in r implies s != t
```

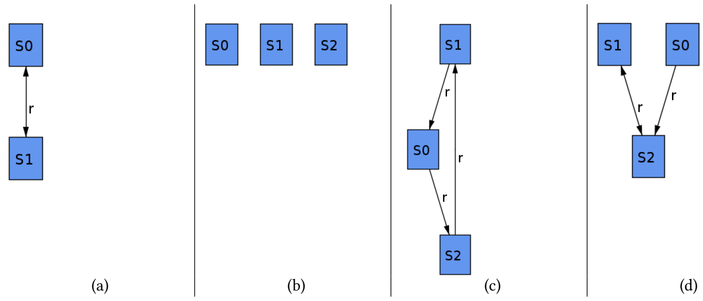

Figure 1: Four example instances generated by the Alloy analyzer. Each instance represents an irreflexive relation. `S0`, `S1`, and `S2` are the atoms in signature `S`. The directed edges represent the tuples in relation `r`.

```
}

run Irreflexive
```

The predicate `Irreflexive` uses a universally quantified (`all`) formula to define irreflexivity as "$\forall s, t \in S \mid (s, t) \in r \Rightarrow s \neq t$". The operator '->' denotes Cartesian product. Fig 1 shows 4 example instances created by the Alloy analyzer for this predicate.

Generation.

English to Alloy.
We query Gemini 3.1 Pro to synthesize the predicate body from only the natural language comment. The prompt provides the signature, the predicate stub with the comment, and requests only the formula. Gemini 3.1 Pro produces:

```
all x: S | x not in x.r
```

The LLM formulates irreflexivity using universal quantification with one variable, where '`x.r`' denotes the relational image of `x` under `r` (i.e., "$\forall x \in S \mid x \notin x.r$"). Note how the formula structure differs from how we wrote irreflexivity in the ground truth predicate.

Validation.
We validate the output using the Alloy analyzer by checking equivalence against the ground truth:

```
check { Irreflexive <=> LLMOutput }
```

The analyzer reports no counterexample, validating correctness with respect to the default scope.

Alloy to Alloy.
When given the reference Alloy formula instead of only the natural language description, Gemini 3.1 Flash Lite produces `all s: S | s->s not in r`, i.e., "$\forall s \in S \mid (s, s) \notin r$" — a structurally different but equivalent encoding.

Sketch to Alloy.
We query Gemini 3.1 Pro to complete a sketch containing three holes (two expression holes and one operator hole) with candidate completions defined using ASketch notation [79]. Despite receiving no explanation of sketch syntax, the LLM correctly fills all holes, producing a formula that matches the ground truth. This is notable because sketch notation is rarely discussed outside of the ASketch literature, making memorization of sketch completions unlikely.

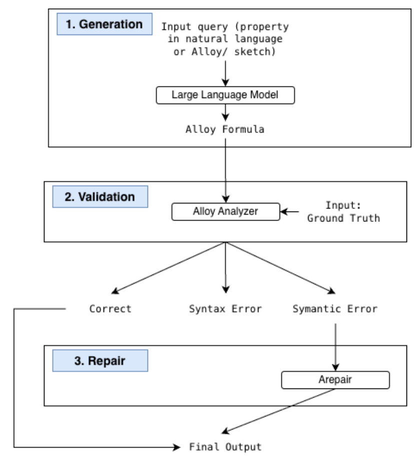

Figure 2: Anvil workflow: generation, validation, and repair of LLM-produced Alloy formulas, using Alloy Analyzer for equivalence checking and ARepair for repairing semantic errors.

Repair.
Semantic errors remaining after validation can be repaired with ARepair, as described in Section 4.

4 Methodology

Anvil is a framework for writing Alloy formulas using LLMs through a three-stage pipeline: generation, validation, and repair.

4.1 LLM-based Alloy formula generation

We investigate three generation methods:

(1) **English to Alloy**. We employ LLMs to write complete Alloy formulas in multiple different ways from given natural language descriptions (in English);
(2) **Alloy to Alloy**. We employ LLMs to create multiple alternative but equivalent formulas in Alloy with respect to given formulas in Alloy; and
(3) **Sketch to Alloy**. We employ LLMs to complete sketches [63, 79] of Alloy formulas and populate the holes in the sketches by synthesizing Alloy expressions and operators so that

the completed formulas accurately represent the desired properties (that are given in natural language).

Figure 2 illustrates the ANVIL workflow. For each task, ANVIL queries the LLM with the target property (in natural language or Alloy) and validates the output using the Alloy analyzer against a ground truth formula. There are three possible outcomes: (1) *correct* — the analyzer finds no counterexample to the equivalence of the LLM's answer and ground truth; (2) *syntax error* — the analyzer fails to compile the output; and (3) *semantic error* — the analyzer finds a counterexample. For "Alloy to Alloy" tasks, the ground truth is the reference formula given as input. For "English to Alloy" and "Sketch to Alloy" tasks, the input does not include the ground truth formula.

We use the LLMs directly via API access. Specifically, we do not fine-tune them. Moreover, the queries we write are minimalistic in their description of the problem domain and do not provide instructions to the LLM on how to approach solving any given task.

4.2 Test-guided validation

Beyond the core generation tasks, we construct English-to-Alloy and Alloy-to-Alloy tasks over the models from the ARepair benchmark to study how test suites affect the correctness of LLM-generated formulas. For each ARepair benchmark predicate, we run each prompt in two configurations: *WithTest*, where the query additionally includes the unmodified ARepair test suite for the corresponding model, and *NoTest*, where we omit this test suite and keep the rest of the prompt unchanged. For each configuration and predicate, we ask the LLM to generate multiple Alloy models and classify each result as correct (Cor), syntax error (Syn), or semantic error (Sem) according to the Alloy analyzer; we further mark a model with a semantic error as repairable (Rep) if a downstream repair tool can automatically produce a correct model from it.

4.3 Automated repair with ARepair

To assess repairability, we use ARepair as an off-the-shelf automated repair engine for Alloy models [75, 77]. For each faulty predicate from the ARepair benchmark, we run ARepair both on the original faulty model and on multiple faulty LLM-generated variants, and record whether ARepair succeeds in producing a repaired model that passes the given tests. This workflow mirrors our intended usage scenario: LLMs generate candidate Alloy models, tests detect semantic faults, and ARepair is then applied to models with semantic errors to see which bugs can actually be fixed.

4.4 Subject tasks

We use 11 well-known properties of graphs and binary relations to create 33 tasks for the LLMs to answer. Three of the properties (DAG, Cycle, and Circular) are regarding edge-labeled graphs, and the remaining eight properties (Connex, Reflexive, Symmetric, Transitive, Antisymmetric, Irreflexive, Functional, and Function) are regarding binary relations. In Alloy, in general, we can use one signature S and one binary relation $r : S \times S$ to represent either an edge-labeled graph or a binary relation. However, in view of the specific domain of graphs, we name the signature '`Node`' and the binary relation '`link`' when creating the tasks relating graph properties. For the tasks relating properties of binary relations, we name the signature '`S`' and the relation '`r`'.

For each property, we create 2 kinds of synthesis tasks: (1) create 20 unique Alloy formulas that represent the given natural language description of the property; and (2) create 20 unique Alloy formulas that are equivalent to the given Alloy formula that captures the property, which is also included as a natural language comment in the prompt. In addition, for each property, we create one sketching task: complete the given sketch of the property with respect to its natural language description that is included as a comment in the prompt. Thus, for each property, we have a total of 3 tasks for the LLM to answer. To obtain these unique solutions, the prompt explicitly asks the LLM for unique encodings; a verification script then filters exact duplicates and variable-renaming variants so that counted solutions are structurally distinct.

Table 1 lists each property, its natural language description, and a reference (ground truth) formula that characterizes it in Alloy. Moreover, Table 2 lists the first 3 properties and their sketches that define the corresponding sketching problems. Together these tables summarize the key elements of our tasks for the LLMs. To illustrate, consider the DAG property. Figure 3 describes the actual prompts we run against each LLM for this property.

In a predicate sketch, certain components of the predicate are placeholder holes [79]. These holes can be of different forms, e.g., comparison operator holes, expression holes, and quantifier holes. For all our sketching tasks, we only use two kinds of holes: comparison operator holes and expression holes. A predicate sketch includes a definition of the sets of possible values that each hole can be completed with. These sets are typically defined using regular expressions [63]. For our DAG sketching task, the comparison operator hole may be completed with one of four possible values from the set { '`=`', '`in`', '`!=`', '`!in`'}, and each expression hole may be completed with one of six possible values from the set { '`Node`', '`n`', '`Node.*link`', '`Node.^link`', '`n.*link`', '`n.^link`' }.

4.5 LLMs under evaluation

We evaluate 4 LLMs: three models from Google's Gemini family, which isolate how model capability affects performance on declarative specification tasks under a shared training methodology, and one open-weight cross-family model as an external reference:

- **Gemini 3.1 Pro**: The largest and most capable Gemini model in our evaluation. We accessed Gemini 3.1 Pro via the Gemini API in March 2026.
- **Gemini 3.0 Flash**: A mid-tier model from the previous generation. Comparing with Gemini 3.1 Flash Lite isolates generational improvements.
- **Gemini 3.1 Flash Lite**: The smallest Gemini model in our evaluation, from the current generation. Comparing with Gemini 3.1 Pro isolates the effect of model scale within the same generation.
- **Llama 3.3 70B Instruct**: An open-weight 70B model from a different vendor and architecture. We accessed it via OpenRouter in July 2026. Including Llama 3.3 70B probes whether Gemini-family trends generalize beyond a single closed-source stack.

Table 1: Subject properties. The table lists for each property, its natural language description that defines the corresponding natural language to Alloy task, and its reference formulation in Alloy that defines the corresponding Alloy to Alloy task.

	Property	*Natural language desc.*	*Reference Alloy formula*
1	DAG	Directed acyclic graph	`all n: Node | n !in n.^link`
2	Cycle	Graph with directed cycle	`some n: Node | n in n.^link`

3	Circular	The number of nodes is equal to the number of edges and the graph has a directed cycle that visits all nodes	`#Node = #link`
			`all n: Node | one n.link`
			`all m, n: Node | m in n.^link`

4	Connex	For every pair of elements in S, either the first is related to the second or vice versa	`all s, t: S |` `s->t in r or t->s in r`
5	Reflexive	Every element in S is related to itself	`all s: S | s->s in r`

6	Symmetric	If element x in S is related to y, then y is also related to x	`all s, t: S |`
			`s->t in r`
			`implies t->s in r`

7	Transitive	If element x in S is related to y and y is related to z, then x is also related to z	`all s, t, u: S |`
			`s->t in r and t->u in r`
			`implies s->u in r`

8	Antisymmetric	If element x in S is related to y and y is related to x, then x and y are the same element	`all s, t: S |`
			`s->t in r and t->s in r`
			`implies s = t`

9	Irreflexive	No element in S is related to itself	`all s, t: S |` `s->t in r implies s != t`
10	Functional	Every element in S is related to at most one element (making r a partial function)	`all s: S | lone s.r`
11	Function	Every element in S is related to exactly one element (making r a total function)	`all s: S | one s.r`

The three Gemini models control for differences in training data and methodology within one family, allowing us to isolate the effect of model scale; Llama 3.3 70B provides a cross-family reference. Gemini models are accessed via the Gemini API, and Llama 3.3 70B via OpenRouter, all with default temperature and sampling settings. Each query is run in a fresh session.

5 Evaluation

5.1 Experimental setup

We evaluate four LLMs: three Gemini models accessed in March 2026 (Gemini 3.1 Pro, Gemini 3.0 Flash, and Gemini 3.1 Flash Lite) and the open-weight Llama 3.3 70B Instruct accessed via OpenRouter in July 2026. Gemini models use the Gemini API; Llama 3.3 70B uses OpenRouter; all runs use default temperature and sampling settings. Each query is run in a fresh session to avoid context carry-over between tasks. We validate LLM outputs using Alloy Analyzer 6 with the default scope of 3. For RQ1, we request 20 unique solutions per synthesis task; for RQ2 and RQ3, we generate 10 outputs per configuration. We deliberately select well-studied graph and binary-relation properties as subjects because they provide unambiguous ground truths that enable fully automatic equivalence checking. This controlled setting isolates LLM reasoning ability from confounds that would arise with complex, multi-signature models (where equivalence may require manual judgment); we discuss this choice further in Section 5.5.3.

We address the following research questions:

RQ1 (Generation): How effectively do LLMs generate correct Alloy formulas from natural language descriptions, from existing Alloy formulas, and by completing formula sketches? How does performance vary across model scale?

RQ2 (Validation): Do test suites included in LLM prompts improve the correctness of generated Alloy formulas?

RQ3 (Repair): Can automated repair (ARepair) fix semantically erroneous LLM-generated Alloy models, and does generating multiple LLM variants increase repair opportunities compared to the original faulty model?

5.2 RQ1: How effectively do LLMs generate Alloy formulas?

5.2.1 English to Alloy and Alloy to Alloy results. Table 3 presents the synthesis results for each of the 4 LLMs across both English-to-Alloy and Alloy-to-Alloy tasks. For each synthesis task, all three Gemini models find at least one valid solution, i.e., create an Alloy formula that is equivalent to the ground truth; Llama 3.3 70B does so on every Alloy-to-Alloy property and on 9 of 11 English-to-Alloy properties (*Cor*=0 on `Circular` and `Symmetric`).

Table 2: Sketches for Alloy specifications for Properties 1–3.

```
pred DAG {
  // Directed acyclic graph
  all n: Node | \E,e\ \CO,co\ \E,e\
}
co := {| =|in|!=|!in |}
e := {| Node|n|((Node|n).(*|^)link) |}
```

```
pred Cycle {
  // Graph with directed cycle
  some n: Node | \E,e\ \CO,co\ \E,e\
}
co := {| =|in|!=|!in |}
e := {| Node|n|((Node|n).(*|^)link) |}
```

```
pred Circular {
  // The number of nodes is equal to the number
       of edges and the graph has a directed
      cycle that visits all nodes
  #Node = #link
  all n: Node | one n.link
  all m, n: Node | \E,e\ \CO,co\ \E,e\
}
co := {| =|in|!=|!in |}
e := {| (Node|m|n).(*|^)link |}
```

For Gemini 3.1 Pro, the number of correct formulas varies between 10 (for `Circular` property) and 20 (for `Antisymmetric` property); for 11 (out of 11) properties, it creates at least 10 correct formulas. Gemini 3.0 Flash shows a similar pattern in Eng→Alloy synthesis, varying between 10 (for `Circular`) and 20 (for `Connex`); for 11 (out of 11) properties, it again creates at least 10 correct formulas. In contrast, Gemini 3.1 Flash Lite is more uneven in Eng→Alloy synthesis, ranging from 2 (for `Connex`) to 19 (for `Function`), and it reaches at least 10 correct formulas on 9 (out of 11) properties. When switching to Alloy→Alloy, the minimum *Cor* increases from 10/10/2 in Eng→Alloy to 15/15/11 in Alloy→Alloy for Gemini 3.1 Pro/Gemini 3.0 Flash/Gemini 3.1 Flash Lite, respectively, and every property reaches at least 10 correct formulas for all three Gemini models.

5.2.2 Sketch to Alloy results. Table 4 presents the results for each of the three Gemini models (Llama 3.3 70B was not evaluated on sketching). Across all 11 properties, every Gemini model succeeds on the first attempt (all predicates are marked ✓). Since no syntax-error failures occurred, the second-attempt follow-up queries described in the table caption are never triggered in our runs.

5.2.3 Cross-model comparison. Cross-model differences are most visible in English-to-Alloy synthesis, where models must infer an Alloy formulation directly from natural language. The sketch-completion setting is comparatively uniform: all three Gemini models succeed on the first attempt for every property (Llama 3.3 70B was not evaluated on sketching), suggesting that the provided sketch structure largely removes the syntax/formatting burden.

Give me 20 unique solutions to the problem of synthesizing the body of the following Alloy predicate (without markdown or comments) with respect to the property described in the comments:

```
sig Node {
  link: set Node
}
pred DAG{
  // Directed acyclic graph
  // Your code goes here
}
```

(a) "English to Alloy" task

Give me 20 unique solutions to the problem of synthesizing the body of the following Alloy predicate (without markdown or comments) with respect to the property described in the comments:

```
sig Node {
  link: set Node
}
pred DAG{
  // Directed acyclic graph
  all n: Node | n !in n.^link
}
```

(b) "Alloy to Alloy" task

Complete the following sketch of the Alloy predicate (without markdown or comments) by selecting values for the holes with respect to the given constraints such that the predicate is correct with respect to the property described in the comments:

```
sig Node {
  link: set Node
}
pred DAG {
  // Directed acyclic graph
  all n: Node | \E,e\ \CO,co\ \E,e\
}

co := {| =|in|!=|!in |}
e := {| Node|n|((Node|n).(*|^)link) |}
```

(c) "Sketch to Alloy" task

Figure 3: Three tasks for the LLMs with respect to the DAG property.

In English-to-Alloy, all three Gemini models find valid solutions on every predicate, but the *quantity* of correct solutions varies. Gemini 3.1 Pro and Gemini 3.0 Flash are relatively stable, with their minimum *Cor* both at 10 on `Circular`, while they reach *Cor*=20 on multiple predicates (e.g., `Antisymmetric` / `Connex` / `Function` / `Reflexive`). In contrast, Gemini 3.1 Flash Lite is more uneven: it ranges from *Cor*=2 on `Connex` to *Cor*=19 on `Function`, and it produces at least 10 correct formulas on 9 (out of 11) properties. Llama 3.3 70B succeeds on 9 of 11 properties in English-to-Alloy, with *Cor*=0 on `Circular` and `Symmetric`. As a result, we can treat `Antisymmetric`, `Function`, `Reflexive`, and `Symmetric` as "easy" predicates in this setting for the Gemini models, because all three achieve high correctness (all *Cor* counts are at least 18). By contrast, `Connex` and

Table 3: Synthesis results for English-to-Alloy (E2A) and Alloy-to-Alloy (A2A). For each property and each LLM, the table shows the number of correct formulas (*Cor*), syntactically invalid formulas (*Syn*), and semantically erroneous formulas (*Sem*) when asked to create 20 unique solutions.

	English to Alloy												Alloy to Alloy											
	Pro			Flash			Lite			Llama 3.3 70B			Pro			Flash			Lite			Llama 3.3 70B		
Property	Cor	Syn	Sem	Cor	Syn	Sem	Cor	Syn	Sem	Cor	Syn	Sem	Cor	Syn	Sem	Cor	Syn	Sem	Cor	Syn	Sem	Cor	Syn	Sem
Antisymmetric	20	0	0	18	2	0	18	0	2	4	13	3	18	2	0	18	2	0	19	1	0	7	7	6
Circular	10	0	10	10	1	9	6	1	13	0	18	2	20	0	0	15	5	0	11	0	9	20	0	0
Connex	20	0	0	20	0	0	2	0	18	7	0	13	20	0	0	20	0	0	14	1	5	8	5	7
Cycle	18	2	0	20	0	0	12	0	8	5	0	15	20	0	0	17	2	1	18	1	1	5	5	10
DAG	18	2	0	19	1	0	16	1	3	5	6	9	17	3	0	19	1	0	16	1	3	6	3	11
Function	20	0	0	18	0	2	19	1	0	12	0	8	19	1	0	18	1	1	16	2	2	7	2	11
Functional	16	0	4	15	0	5	11	4	5	4	7	9	15	0	5	15	0	5	12	6	2	10	0	10
Irreflexive	18	2	0	17	3	0	18	0	2	18	0	2	18	2	0	18	2	0	15	2	3	12	3	5
Reflexive	20	0	0	19	1	0	18	0	2	15	1	4	19	1	0	19	1	0	16	1	3	5	11	4
Symmetric	18	2	0	19	1	0	18	0	2	0	20	0	18	2	0	20	0	0	18	0	2	9	6	5
Transitive	19	1	0	17	3	0	16	0	4	4	5	11	19	1	0	19	1	0	13	1	6	15	0	5

Table 4: Alloy sketching results. For each property and each LLM, the table shows whether the LLM successfully completes the sketching task on the first attempt (✓) or fails (✗). For any first-attempt failures, we create a follow-up query informing the LLM of the syntax error; in all such cases, the LLM succeeds on the second attempt.

Property	*Gemini 3.1 Pro*	*Gemini 3.0 Flash*	*Gemini 3.1 Flash Lite*
Antisymmetric	✓	✓	✓
Circular	✓	✓	✓
Connex	✓	✓	✓
Cycle	✓	✓	✓
DAG	✓	✓	✓
Function	✓	✓	✓
Functional	✓	✓	✓
Irreflexive	✓	✓	✓
Reflexive	✓	✓	✓
Symmetric	✓	✓	✓
Transitive	✓	✓	✓

`Circular` are "hard" for the smallest Gemini model, where Lite's *Cor* drops to 2 and 6, respectively.

Model scale is not strictly monotonic. For example, in English-to-Alloy synthesis, Gemini 3.0 Flash exceeds Gemini 3.1 Pro on `Cycle` (20 vs 18) and `DAG` (19 vs 18), while Gemini 3.1 Flash Lite slightly outperforms Flash on `Function` (19 vs 18). Once the task is converted into Alloy-to-Alloy rewriting, the distributions tighten for all three Gemini models: the minimum *Cor* becomes 15 for Gemini 3.1 Pro (on `Functional`), 15 for Gemini 3.0 Flash, and 11 for Gemini 3.1 Flash Lite (on `Circular`), and every predicate reaches at least 10 correct formulas for all Gemini models. Overall, providing an existing Alloy structure reduces the search difficulty most strongly for smaller models, even though semantic errors remain the dominant failure mode compared to syntax failures.

5.3 RQ2: Do test suites improve LLM-generated formulas?

Tables 5–6 present per-predicate results; we summarize cross-model findings below.

5.3.1 English to Alloy: WithTest vs NoTest. Across all three Gemini models, WithTest increases *Cor* and decreases *Sem* (Table 5). Tests primarily bias the output distribution toward the "fully correct" region and away from semantic faults. Llama 3.3 70B exhibits the opposite pattern in E2A: WithTest yields lower *Cor* than NoTest (26 vs 48), indicating that test-in-prompt guidance does not transfer uniformly across model families.

5.3.2 Alloy to Alloy: WithTest vs NoTest. In Alloy-to-Alloy rewriting (Table 6), both configurations achieve high *Cor* counts with small differences, indicating that starting from an existing Alloy model dominates the benefit of explicit tests.

5.3.3 Cross-model comparison. Table 7 summarizes WithTest vs NoTest totals across models. In E2A, WithTest increases *Cor* from **123→151** (Pro), **116→133** (Flash), and **95→113** (Lite), while *Sem* drops correspondingly — most strongly for Lite (**91→60**, a 34% reduction). This suggests that test suites are particularly effective at steering smaller Gemini models away from semantic errors. WithTest may introduce more syntax errors (Lite *Syn*: **14→27**), but the net effect on correctness is positive because the *Sem* drop outweighs the *Syn* increase. By contrast, Llama 3.3 70B moves in the opposite direction in E2A (*Cor* **48→26** under WithTest), so the benefit of tests is family-dependent.

In A2A, the task is already highly constrained and WithTest provides only marginal benefit for Gemini (e.g., Pro *Cor*: **187** vs **188**), confirming that structural context dominates once a reference Alloy model is given.

5.4 RQ3: Can automated repair fix remaining errors?

Table 8 summarizes the ARepair outcomes on the original faulty (Sem) predicates and on the 10 LLM-generated Sem rewrites per predicate, for each Gemini model and for Llama 3.3 70B (see caption for **Orig** vs **LLM** columns; the Original columns are shared across all 4 models). Repair success varies widely by predicate and by model; aggregate totals are shown in the bottom row. Aggregating across the faulty predicate subset, ARepair repairs **6/13** (**46%**) of the original semantic errors for each of the 4 models (shared original inputs). For LLM-generated semantic rewrites, repair success reaches **62/129** (**48%**) for Gemini 3.1 Pro, **59/125** (**47%**) for Gemini 3.0 Flash, **62/115** (**54%**) for Gemini 3.1 Flash Lite, and **47/118** (**40%**) for Llama 3.3 70B. Summing these LLM-rewrite repair counts across the three

Table 5: WithTest vs NoTest (English-to-Alloy) by predicate with repairability. For each of the 4 LLMs, we generate 10 unique outputs per configuration.

		Gemini 3.1 Pro								Gemini 3.0 Flash								Gemini 3.1 Flash Lite								Llama 3.3 70B							
		Cor		Syn		Sem		Rep		Cor		Syn		Sem		Rep		Cor		Syn		Sem		Rep		Cor		Syn		Sem		Rep	
Predicate	**N**	**WT**	**NT**	**WT**	**NT**	**WT**	**NT**	**WT**	**NT**	**WT**	**NT**	**WT**	**NT**	**WT**	**NT**	**WT**	**NT**	**WT**	**NT**	**WT**	**NT**	**WT**	**NT**	**WT**	**NT**	**WT**	**NT**	**WT**	**NT**	**WT**	**NT**	**WT**	**NT**
arr-NoConflict	10	10	9	0	0	0	1	0	0	10	10	0	0	0	0	0	0	6	7	3	1	1	2	0	2	0	3	10	1	0	6	0	4
balancedBST-Balanced	10	5	8	0	0	5	2	0	0	0	0	8	0	2	10	0	0	2	2	5	3	3	5	0	0	0	0	1	0	9	10	1	1
balancedBST-HasAtMostOneChild	10	10	9	0	0	0	1	0	1	9	10	0	0	1	0	1	0	10	8	0	0	0	2	0	2	4	4	0	0	6	6	5	2
balancedBST-Sorted	10	10	10	0	0	0	0	0	0	10	7	0	3	0	0	0	0	10	9	0	1	0	0	0	0	2	2	0	0	8	8	0	3
bempl-CanEnter	10	0	0	0	0	10	10	0	0	0	0	0	0	10	10	0	0	0	0	2	2	8	8	0	0	0	0	0	1	10	9	0	0
cd-Acyclic	10	8	10	2	0	0	0	0	0	9	9	1	1	0	0	0	0	9	7	1	1	0	2	0	1	2	4	8	0	0	6	0	6
cd-AllExtObject	10	10	10	0	0	0	0	0	0	9	10	1	0	0	0	0	0	0	0	1	0	9	10	9	5	2	2	7	0	1	8	1	3
cd-ClassHierarchy	10	8	5	2	5	0	0	0	0	10	10	0	0	0	0	0	0	10	10	0	0	0	0	0	0	5	7	2	0	3	3	0	0
cd-ObjectNoExt	10	10	10	0	0	0	0	0	0	10	10	0	0	0	0	0	0	8	8	0	0	2	2	2	2	2	7	5	0	3	3	2	3
dll-ConsistentPreAndNxt	10	10	10	0	0	0	0	0	0	10	10	0	0	0	0	0	0	6	4	1	0	3	6	1	4	0	1	0	1	10	8	5	6
dll-Sorted	10	10	10	0	0	0	0	0	0	10	10	0	0	0	0	0	0	10	9	0	0	0	1	0	1	0	2	0	1	10	7	9	6
dll-UniqueElem	10	10	10	0	0	0	0	0	0	10	10	0	0	0	0	0	0	9	9	0	0	1	1	0	0	4	6	0	0	6	4	3	4
farmer-crossRiver	10	0	0	2	6	8	4	0	0	0	0	0	0	10	10	0	0	0	0	10	0	0	10	0	0	0	0	10	10	0	0	0	0
farmer-solvePuzzle	10	10	10	0	0	0	0	0	0	9	10	1	0	0	0	0	0	10	10	0	0	0	0	0	0	1	6	8	0	1	4	1	4
grade-PolicyAllowsGrading	10	10	2	0	0	0	8	0	1	10	2	0	0	0	8	0	0	10	0	0	0	0	10	0	10	1	0	0	0	9	10	3	3
other-CanEnter	10	10	0	0	0	0	10	0	1	0	0	0	0	10	10	0	1	3	3	0	0	7	7	0	1	2	2	0	0	8	8	4	3
student-Contains	10	0	0	0	0	10	10	0	0	0	0	0	0	10	10	0	0	0	0	3	2	7	8	0	0	0	0	0	2	10	8	0	0
student-Count	10	0	0	0	0	10	10	0	0	0	0	0	0	10	10	0	0	0	0	0	4	10	6	0	0	0	0	0	2	10	8	0	0
student-Loop	10	10	0	0	0	0	10	0	2	9	0	0	0	1	10	1	0	0	0	1	0	9	10	3	0	0	0	0	0	10	10	0	0
student-Sorted	10	10	10	0	0	0	0	0	0	8	8	2	2	0	0	0	0	10	9	0	0	0	1	0	1	1	2	0	0	9	8	4	6
Totals	**200**	**151**	**123**	**6**	**11**	**43**	**66**	**0**	**5**	**133**	**116**	**13**	**6**	**54**	**78**	**2**	**1**	**113**	**95**	**27**	**14**	**60**	**91**	**15**	**29**	**26**	**48**	**51**	**18**	**123**	**134**	**38**	**54**

Cor = Correct, Syn = Syntax error, Sem = Semantic error, Rep = Repairable. WT = WithTest, NT = NoTest. N = number of unique outputs per configuration.

Table 6: WithTest vs NoTest (Alloy-to-Alloy) by predicate with repairability. For each of the 4 LLMs, we generate 10 unique outputs per configuration.

		Gemini 3.1 Pro								Gemini 3.0 Flash								Gemini 3.1 Flash Lite								Llama 3.3 70B							
		Cor		Syn		Sem		Rep		Cor		Syn		Sem		Rep		Cor		Syn		Sem		Rep		Cor		Syn		Sem		Rep	
Predicate	**N**	**WT**	**NT**	**WT**	**NT**	**WT**	**NT**	**WT**	**NT**	**WT**	**NT**	**WT**	**NT**	**WT**	**NT**	**WT**	**NT**	**WT**	**NT**	**WT**	**NT**	**WT**	**NT**	**WT**	**NT**	**WT**	**NT**	**WT**	**NT**	**WT**	**NT**	**WT**	**NT**
arr-NoConflict	10	10	9	0	0	0	1	0	0	8	10	2	0	0	0	0	0	7	6	3	2	0	2	0	0	6	3	1	4	3	3	1	2
balancedBST-Balanced	10	10	10	0	0	0	0	0	0	7	4	0	0	3	6	0	0	2	2	4	5	4	3	0	0	0	1	4	2	6	7	1	0
balancedBST-HasAtMostOneChild	10	9	10	0	0	1	0	1	0	10	9	0	0	0	1	0	1	10	8	0	1	0	1	0	0	8	7	2	2	0	1	0	0
balancedBST-Sorted	10	10	10	0	0	0	0	0	0	7	4	3	3	0	3	0	0	6	8	0	0	4	2	0	0	2	1	1	0	7	9	0	1
bempl-CanEnter	10	0	0	0	0	10	10	0	0	0	0	0	0	10	10	0	0	0	0	0	1	10	9	0	0	0	0	3	1	7	9	0	0
cd-Acyclic	10	8	10	2	0	0	0	0	0	9	9	1	1	0	0	0	0	8	7	1	0	1	3	0	2	5	10	0	0	5	0	5	0
cd-AllExtObject	10	10	10	0	0	0	0	0	0	10	10	0	0	0	0	0	0	8	7	1	0	1	3	1	3	4	6	2	1	4	3	4	3
cd-ClassHierarchy	10	10	10	0	0	0	0	0	0	10	9	0	1	0	0	0	0	8	6	1	3	1	1	0	0	10	7	0	2	0	1	0	1
cd-ObjectNoExt	10	10	10	0	0	0	0	0	0	9	9	1	1	0	0	0	0	8	6	1	2	1	2	1	2	8	5	0	2	2	3	2	3
dll-ConsistentPreAndNxt	10	10	9	0	1	0	0	0	0	10	10	0	0	0	0	0	0	7	3	0	0	3	7	2	3	3	2	1	0	6	8	4	5
dll-Sorted	10	10	10	0	0	0	0	0	0	10	9	0	1	0	0	0	0	8	5	0	0	2	5	2	5	4	6	0	2	6	2	2	2
dll-UniqueElem	10	10	10	0	0	0	0	0	0	10	10	0	0	0	0	0	0	6	9	0	0	4	1	2	1	6	9	0	1	4	0	1	0
farmer-crossRiver	10	10	10	0	0	0	0	0	0	7	7	3	2	0	1	0	1	7	6	1	3	2	1	0	0	1	7	0	3	9	0	0	0
farmer-solvePuzzle	10	10	10	0	0	0	0	0	0	10	10	0	0	0	0	0	0	10	10	0	0	0	0	0	0	9	6	0	3	1	1	1	1
grade-PolicyAllowsGrading	10	10	10	0	0	0	0	0	0	10	10	0	0	0	0	0	0	9	10	0	0	1	0	0	0	4	5	0	0	6	5	2	0
other-CanEnter	10	10	10	0	0	0	0	0	0	10	10	0	0	0	0	0	0	7	7	3	3	0	0	0	0	3	3	0	2	7	5	2	2
student-Contains	10	10	10	0	0	0	0	0	0	8	8	1	0	1	2	0	0	0	0	3	6	7	4	0	0	0	0	0	10	10	0	2	0
student-Count	10	10	10	0	0	0	0	0	0	10	10	0	0	0	0	0	0	0	0	3	1	7	9	0	0	2	0	0	10	8	0	2	0
student-Loop	10	10	10	0	0	0	0	0	0	10	10	0	0	0	0	0	0	0	8	0	0	10	2	3	2	2	0	0	10	8	0	3	0
student-Sorted	10	10	10	0	0	0	0	0	0	10	9	0	1	0	0	0	0	8	8	1	0	1	2	1	2	5	0	0	10	5	0	3	0
Totals	**200**	**187**	**188**	**2**	**1**	**11**	**11**	**1**	**0**	**175**	**167**	**11**	**10**	**14**	**23**	**0**	**2**	**119**	**116**	**22**	**27**	**59**	**57**	**12**	**20**	**82**	**78**	**14**	**65**	**104**	**57**	**35**	**20**

Cor = Correct, Syn = Syntax error, Sem = Semantic error, Rep = Repairable. WT = WithTest, NT = NoTest. N = number of unique outputs per configuration.

Table 7: Aggregate *Cor* counts for WithTest (WT) vs NoTest (NT) in RQ2 (totals over all predicates, N=10 per configuration).

Model	E2A WT	E2A NT	A2A WT	A2A NT
Gemini 3.1 Pro	151	123	187	188
Gemini 3.0 Flash	133	116	175	167
Gemini 3.1 Flash Lite	113	95	119	116
Llama 3.3 70B	26	48	82	78

Gemini models yields **183/369** (50%): a cross-model Gemini aggregate, not the outcome of a single unified experiment. These results suggest that bugs that are already repairable in the original models often admit repairable LLM variants as well, whereas predicates that remain challenging for the original faulty models often stay difficult across rewrites, with exceptions at the per-predicate level.

5.5 Discussion

5.5.1 Solution diversity analysis. As the experimental results show, all three Gemini models successfully create the desired Alloy formulas on every property; Llama 3.3 70B succeeds on 9 of 11 properties in English-to-Alloy (*Cor*=0 on `Circular` and `Symmetric`). The quality of solutions is high for the Gemini models; an expert Alloy user could be expected to create solutions of comparable quality. To illustrate, consider representative formulas created by Gemini 3.1 Pro for the `DAG` property in English-to-Alloy synthesis (18 correct variants out of 20 attempts):

```
1. no iden & ^link           (relational algebra)
2. all n: Node | n not in n.^link  (pointwise)
3. no n: Node | n in ^link.n      (reverse image)
4. all n: Node | no n.^link & n    (set intersection)
```

These span four distinct encoding families: relational algebra over identity, pointwise universal quantification, reverse relational image, and set intersection. Across all 11 properties, the LLMs consistently produce solutions spanning multiple encoding families, not merely syntactic rewrites of a single formulation. Understanding why these encodings are equivalent can require non-trivial Alloy intuition, especially when negations and relation direction are expressed through nested set operations.

5.5.2 Error analysis. We examine the errors produced by the LLMs across our synthesis tasks. Errors fall into two categories: *syntax errors*, where the Alloy analyzer fails to parse the output, and *semantic errors*, where the output parses but is not equivalent to the ground truth.

Common patterns in semantic errors include: (1) incorrect quantifier structure (e.g., using existential instead of universal quantification), (2) confusion between reflexive closure (*) and transitive

Table 8: Pre-repair classification and ARepair results for faulty Alloy-to-Alloy (original vs 10 LLM rewrites) across all 4 LLMs.

	Gemini 3.1 Pro								Gemini 3.0 Flash					Gemini 3.1 Flash Lite					Llama 3.3 70B				
	Original			LLM (10x)			ARepair		LLM (10x)			ARepair		LLM (10x)			ARepair		LLM (10x)			ARepair	
Predicate	**Cor**	**Syn**	**Sem**	**Cor**	**Syn**	**Sem**	**Orig**	**LLM**	**Cor**	**Syn**	**Sem**	**Orig**	**LLM**	**Cor**	**Syn**	**Sem**	**Orig**	**LLM**	**Cor**	**Syn**	**Sem**	**Orig**	**LLM**
arr-NoConflict		NA			NA		NA	NA		NA		NA	NA		NA		NA	NA		NA		NA	NA
balancedBST-Balanced	0	0	1	0	0	10	0/1	0/10	0	0	10	0/1	0/10	0	5	5	0/1	0/5	0	5	5	0/1	0/5
balancedBST-HasAtMostOneChild		NA			NA		NA	NA		NA		NA	NA		NA		NA	NA		NA		NA	NA
balancedBST-Sorted	0	0	1	0	0	10	0/1	0/10	0	0	10	0/1	0/10	0	0	10	0/1	0/10	0	0	10	0/1	0/10
bempl-CanEnter		NA			NA		NA	NA		NA		NA	NA		NA		NA	NA		NA		NA	NA
cd-Acyclic	0	0	1	0	0	10	0/1	7/10	0	1	9	0/1	3/9	0	1	9	0/1	3/9	0	0	10	0/1	0/10
cd-AllExtObject	0	0	1	0	0	10	1/1	10/10	0	0	10	1/1	9/10	0	1	9	1/1	8/9	0	1	9	1/1	6/9
cd-ClassHierarchy		NA			NA		NA	NA		NA		NA	NA		NA		NA	NA		NA		NA	NA
cd-ObjectNoExt	0	0	1	0	0	10	1/1	10/10	0	0	10	1/1	9/10	0	1	9	1/1	9/9	0	1	9	1/1	9/9
dll-ConsistentPreAndNxt	0	0	1	0	0	10	1/1	10/10	0	0	10	1/1	10/10	0	0	10	1/1	8/10	0	0	10	1/1	6/10
dll-Sorted	0	0	1	0	0	10	1/1	7/10	0	0	10	1/1	8/10	0	0	10	1/1	8/10	0	2	8	1/1	7/8
dll-UniqueElem	0	0	1	0	0	10	1/1	9/10	0	0	10	1/1	8/10	0	1	9	1/1	9/9	0	0	10	1/1	8/10
farmer-crossRiver	0	0	1	0	0	10	0/1	0/10	0	3	7	0/1	2/7	2	3	5	0/1	0/5	0	0	10	0/1	0/10
grade-PolicyAllowsGrading	0	0	1	0	0	10	0/1	0/10	0	0	10	0/1	0/10	0	0	10	0/1	0/10	0	0	10	0/1	0/10
farmer-solvePuzzle		NA			NA		NA	NA		NA		NA	NA		NA		NA	NA		NA		NA	NA
other-CanEnter		NA			NA		NA	NA		NA		NA	NA		NA		NA	NA		NA		NA	NA
student-Contains	0	0	1	0	0	10	0/1	0/10	0	1	9	0/1	0/9	0	1	9	0/1	7/9	0	3	7	0/1	2/7
student-Count		NA			NA		NA	NA		NA		NA	NA		NA		NA	NA		NA		NA	NA
student-Loop	0	0	1	0	0	10	0/1	0/10	0	0	10	0/1	0/10	0	0	10	0/1	0/10	0	0	10	0/1	0/10
student-Sorted	0	0	1	0	1	9	1/1	9/9	0	0	10	1/1	10/10	0	0	10	1/1	10/10	0	0	10	1/1	9/10
Total	**0**	**0**	**13**	**0**	**1**	**129**	**6/13**	**62/129**	**0**	**5**	**125**	**6/13**	**59/125**	**2**	**13**	**115**	**6/13**	**62/115**	**0**	**12**	**118**	**6/13**	**47/118**

Orig = ARepair on original faulty model (repaired/total Sem). LLM = ARepair on LLM-generated Sem rewrites (repaired/total Sem). NA = predicate has no faulty version in the original dataset (nothing to repair). Occasional non-zero Cor under LLM (10x) are rare anomalies (2 cases) where rewriting incidentally fixed the bug. The "Original" columns are shared across all models (same faulty inputs).

closure (ˆ), and (3) reversed relation direction. Syntax errors are less frequent and typically involve malformed Alloy expressions, such as incorrect use of set comprehension notation or missing parentheses in relational compositions.

Across Gemini models, semantic failures dominate the error budget in English-to-Alloy (E2A), especially for the smaller Lite model, while syntax failures remain relatively sparse. Summing across all 11 properties, semantic errors (*Sem*) account for **14**/**16**/**59** cases for Gemini 3.1 Pro/Gemini 3.0 Flash/Gemini 3.1 Flash Lite, respectively (vs syntax errors **9**/**12**/**7**). After switching to Alloy-to-Alloy (A2A), semantic error totals drop to **5**/**7**/**36**, while syntax errors rise slightly to **12**/**15**/**16**. The largest semantic-error contributions in E2A come from `Circular` (Sem = **10** / **9** for Pro / Flash) and from `Connex` for Gemini 3.1 Flash Lite (Sem = **18**). In A2A, the remaining semantic errors concentrate mainly on `Functional` for Gemini 3.1 Pro / Gemini 3.0 Flash (Sem = **5**) and on `Circular` for Gemini 3.1 Flash Lite (Sem = **9**).

The `Circular` property is consistently the most difficult across models because it combines a cardinality constraint, a functionality constraint, and a reachability property in a single predicate; it is also the main source of semantic errors in E2A for both Gemini 3.1 Pro and Gemini 3.0 Flash.

5.5.3 Model scaling observations. Overall, model scaling affects Alloy synthesis mainly through the prevalence of semantic errors rather than through outright syntax failures. In English-to-Alloy (E2A), the two larger models are close in total correctness, with *Cor*=**197** vs **192** and *Sem*=**14** vs **16** for Gemini 3.1 Pro and Gemini 3.0 Flash, respectively, whereas Gemini 3.1 Flash Lite shows a sharp drop in *Cor*=**154** and a large increase in *Sem*=**59**. The same qualitative trend holds in Alloy-to-Alloy (A2A): semantics are far more frequent for Lite (*Sem*=**36**) than for Pro/Flash (*Sem*=**5**/**7**), while syntax remains comparatively sparse. This indicates that smaller models are mainly missing semantic reasoning precision, not merely producing malformed Alloy syntax.

Adding external validation also changes the error balance in a model-dependent way. In E2A validation (RQ2), WithTest substantially reduces *Sem* for all three Gemini models, with the largest absolute gain for Lite (from **91** to **60**), even though it can come with a syntax trade-off (Lite *Syn* increases from **14** to **27**). When the task is already constrained by an existing Alloy model (A2A), the WithTest–NoTest gap becomes small, suggesting that structural context dominates the remaining uncertainty.

Finally, repairability is not strictly monotonic with model size. On the original faulty semantic subset (RQ3), all 4 models have the same repair success (**46%**, i.e., **6/13**) because they share the same original faulty inputs. For LLM-generated semantic rewrites, the smallest Gemini model achieves the highest repair rate (**54%** for Lite vs **48%** for Pro, **47%** for Gemini 3.0 Flash, and **40%** for Llama 3.3 70B), indicating that while Lite generates more semantic errors in E2A, a substantial fraction of its remaining errors still fall within the validator/repairable region.

Together, these findings support a practical implication: cheaper models can be viable for Alloy tasks when combined with stronger scaffolding (A2A inputs, test suites, and downstream repair), but they are less reliable for fully unconstrained English-to-Alloy generation where semantic errors dominate.

5.5.4 Implications for MDE practice. Our three generation modes serve different roles in model-driven engineering workflows. English-to-Alloy lowers the entry barrier for developers unfamiliar with relational logic, enabling them to produce initial specifications from natural language requirements. Alloy-to-Alloy supports design-space exploration, where engineers iterate on existing constraints to discover equivalent but more perspicuous formulations. Sketch-to-Alloy supports expert-guided refinement, where a specifier provides the overall structure and delegates hole completion to the LLM. The finding that test suites most benefit smaller models implies that lightweight validation infrastructure — readily available through Alloy's SAT-based backend — matters more than model size for practical deployment, making the approach accessible without requiring the most expensive LLMs.

5.5.5 Memorization considerations. All 11 subject properties are textbook examples whose ground-truth formulas appear in the Alloy documentation and Jackson's textbook [21]. It is therefore plausible that the LLMs memorized some formulas. We deliberately chose these properties precisely *because* they have unambiguous ground truths that enable fully automatic equivalence checking via the Alloy Analyzer. Complex, multi-signature models would introduce confounds: equivalence becomes undecidable in general, manual inspection introduces subjective bias, and domain-specific knowledge conflates LLM capability with prompt engineering quality. Our controlled setting isolates the core question — can LLMs reason about relational logic? — from these confounds, analogous to how benchmarks such as HumanEval [7] use self-contained functions before tackling full-project synthesis.

Three observations suggest the LLMs do more than recall. First, the LLMs produce solutions spanning multiple encoding families (relational algebra, pointwise quantification, reverse image, set intersection) — generating structurally diverse correct formulas requires compositional reasoning, not retrieval of a single memorized answer. Second, all three Gemini models complete Alloy sketches using ASketch notation [79] without any explanation of sketch syntax (Llama 3.3 70B was not evaluated on sketching); this notation is rarely discussed outside of the ASketch literature, making memorization of sketch completions unlikely. Third, if models were simply recalling memorized formulas, we would expect near-uniform performance across Gemini model scales, since all three Gemini models share the same training data. Instead, we observe a 43-point *Cor* gap between Pro (197) and Lite (154) in E2A, concentrated in properties requiring composed relational reasoning (`Circular`, `Connex`). This pattern is consistent with capability-driven performance, not uniform recall.

We acknowledge this as a limitation (Section 6). Future work should evaluate on non-textbook properties with novel signatures to further disentangle capability from memorization.

6 Threats to Validity

Internal validity. Alloy's equivalence checking is bounded: we use the default scope of 3, so two formulas may appear equivalent at this scope but diverge at higher scopes. We did not verify equivalence at larger scopes. LLM outputs may vary across sessions due to non-deterministic sampling. We use the default temperature and sampling settings provided by the Gemini API and did not control for these parameters.

External validity. All 11 subject properties are well-known graph and binary-relation properties that likely appear in LLM training data. We deliberately chose these properties to enable rigorous automatic equivalence checking; results may not generalize to novel or domain-specific specifications, and extending this methodology to multi-signature models with richer constraint interactions is a necessary next step. Three of the four models are from the same family (Gemini); we partially mitigate this threat by including a cross-family open-weight model (Llama 3.3 70B), but results may still not generalize to other architectures or training regimes.

Construct validity. Requesting "20 unique solutions" is an arbitrary threshold. The number of correct solutions may depend on this parameter and on how each LLM interprets "unique." The memorization risk is significant: all subject properties are textbook examples that likely appear in public training corpora. While some generated solutions appear structurally novel (e.g., DAG formula #9 in Section 5), we cannot rule out that the LLMs have memorized these or similar formulations. At the same time, on the sketch-completion tasks all three Gemini models succeed on the first attempt even though the prompts provide no explanation of ASketch notation (Llama 3.3 70B was not evaluated on sketching); this pattern is more consistent with in-context inference over the given hole grammar than with retrieval of a memorized solution.

7 Related work

Large language models (LLMs) have enabled the automation of many software development and verification themes, including writing code [3, 18, 81, 83, 84], clarifying requirements [46], software maintenance [8, 31, 32, 47], software testing [30, 39, 57], debugging [34, 74], analyzing code [26], and human-centric studies [20, 80]. Our work shares the spirit of previous work on using LLMs to formalize specifications, e.g., postconditions [9], loop invariants [6], or Javadocs [29]. In the specific context of Alloy, LLMs were previously employed to repair faulty Alloy models [1, 17]. Our own earlier study [18] conducted a controlled experiment on the same three generation tasks — English to Alloy, Alloy to Alloy, and Sketch to Alloy — over the same 11 subject properties, using ChatGPT and DeepSeek. Independent of LLMs, ARepair provides an automated framework for repairing faulty Alloy models using test-based oracles and search-based techniques [75, 77]. Indeed, repair and synthesis are closely related; repair can be viewed as a restricted form of synthesis where the faulty part of code that has been localized is replaced with new code. The topic of connecting LLMs and Alloy was also briefly discussed on the Alloy community discourse forum[1]. To our knowledge, Anvil is the first study to combine LLM-based Alloy formula generation with test-guided validation and downstream automated repair in a single pipeline.

Program synthesis is a heavily studied area [55]. A popular form of synthesis is test-driven where user-provided tests are used to validate the synthesized code. SyPet [10] introduced the use of Petri nets in creating sequences of method invocations for complex APIs with respect to given tests. EdSketch [19] and EdSynth [82] defined an optimized backtracking search for completing Java sketches where test executions guided search pruning. Test-Driven Synthesis is defined as an iterative method to build a C# program such that it satisfies all the given tests [54]. Component-based synthesis constructs programs by putting together components from given libraries [25].

Program sketching [2, 30, 61–66], pioneered by the Sketch system [63], offered a significant advance in scaling program synthesis, where a partial implementation is given and the goal is to complete it [4, 10, 11, 13, 16, 27, 28, 36, 37, 41, 53, 61]. The Sketch system provided Java-like Sketch language for writing partial programs, and deployed SAT and inductive synthesis in a counterexample-guided loop to complete them. JSketch enabled sketching Java programs [24] by translating Java to Sketch. PSketch focused on concurrent data structures and enabled sketching them [65].

[1] https://alloytools.discourse.group/t/the-use-of-alloy-and-llms/449

Researchers developed several approaches to assist Alloy users build their models correctly, e.g., by highlighting UNSAT cores [60, 72], reducing symmetries [35, 59, 73], improving scenario exploration [49, 50, 67], supporting state modeling [12, 22, 23, 44, 69], and supporting testing a la traditional unit testing [68, 70, 71, 76].

ASketch [79] introduced sketching of Alloy models in the spirit of the Sketch system. The heart of ASketch is an efficient enumerator for non-equivalent relational expressions [78]. We use the basic ASketch approach for creating the sketching problems for the LLMs. A key difference is that ASketch, just like the Sketch system, requires test cases to be given as input in order to complete the sketch and validate the resulting code. In contrast, our use of LLMs does not require test cases; in addition to the sketch, we simply provide a natural language description of the desired property to the LLMs.

8 Conclusion

This paper presented Anvil, a framework that leverages large language models (LLMs) to produce correct Alloy specifications through a three-stage pipeline: (1) generation — writing complete Alloy formulas from natural language descriptions, creating equivalent alternative formulas from given Alloy formulas, and completing sketches of Alloy formulas; (2) validation — studying how test suites in prompts affect the correctness of LLM-generated formulas; and (3) repair — applying ARepair as a downstream tool for fixing semantically erroneous outputs. We evaluated Anvil on 11 well-studied subject specifications and ARepair benchmark predicates using three Gemini models of varying capability (Gemini 3.1 Pro, Gemini 3.0 Flash, and Gemini 3.1 Flash Lite) and the open-weight Llama 3.3 70B. The results showed that Anvil's pipeline effectively synthesizes correct Alloy formulas across all task types and that test suites and automated repair further improve outcomes.

In particular, scaling mainly reduces semantic errors in unconstrained English-to-Alloy generation (e.g., *Sem* is much higher for Gemini 3.1 Flash Lite than for Gemini 3.1 Pro/Gemini 3.0 Flash), but the gap narrows substantially once the task is constrained by an existing Alloy model (A2A) and guided validation, with many remaining issues still addressable by ARepair.

Anvil provides evidence that LLMs, combined with validation and repair, can assist in producing correct specifications. Our controlled methodology, built on properties with unambiguous ground truths, establishes a reusable baseline for evaluating LLM-assisted specification. Future work will extend this methodology to multi-signature models and domain-specific constraints (e.g., metamodel well-formedness rules), strengthening the connection to industrial MDE practice and further disentangling LLM capability from memorization. Our replication package, including benchmark dataset, prompts, and evaluation scripts, is available at https://doi.org/10.5281/zenodo.20838660.

Acknowledgments

We would like to acknowledge Department of Energy award DE-SC0024467 for their support.

References

[1] Mohannad Alhanahnah, Md Rashedul Hasan, and Hamid Bagheri. 2024. An Empirical Evaluation of Pre-trained Large Language Models for Repairing Declarative Formal Specifications. arXiv:2404.11050 arXiv 2404.11050.
[2] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *FMCAD*.
[3] Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, Vageesh D. C., Arun Iyer, Suresh Parthasarathy, Sriram Rajamani, B. Ashok, and Shashank Shet. 2024. CodePlan: Repository-Level Coding using LLMs and Planning. *Proc. ACM Softw. Eng.* 1, FSE, Article 31 (July 2024), 24 pages.
[4] Rastislav Bodík and Barbara Jobstmann. 2013. Algorithmic program synthesis: Introduction. *STTT* (2013).
[5] Stefano Ceri, Georg Gottlob, Letizia Tanca, et al. 1989. What you always wanted to know about Datalog (and never dared to ask). *IEEE transactions on knowledge and data engineering* 1, 1 (1989), 146–166.
[6] Saikat Chakraborty, Shuvendu Lahiri, Sarah Fakhoury, Akash Lal, Madanlal Musuvathi, Aseem Rastogi, Aditya Senthilnathan, Rahul Sharma, and Nikhil Swamy. 2023. Ranking LLM-Generated Loop Invariants for Program Verification. In *Findings of the Association for Computational Linguistics: EMNLP 2023*.
[7] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, et al. 2021. Evaluating Large Language Models Trained on Code. *arXiv preprint arXiv:2107.03374* (2021).
[8] Malinda Dilhara, Abhiram Bellur, Timofey Bryksin, and Danny Dig. 2024. Unprecedented Code Change Automation: The Fusion of LLMs and Transformation by Example. *Proc. ACM Softw. Eng.* 1, FSE (2024), 631–653.
[9] Madeline Endres, Sarah Fakhoury, Saikat Chakraborty, and Shuvendu K. Lahiri. 2024. Can Large Language Models Transform Natural Language Intent into Formal Method Postconditions? *Proc. ACM Softw. Eng.* 1, FSE, Article 84 (July 2024), 24 pages.
[10] Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W. Reps. 2017. Component-based Synthesis for Complex APIs. In *POPL*.
[11] John K. Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing Data Structure Transformations from Input-output Examples. In *PLDI*.
[12] Marcelo F. Frias, Juan P. Galeotti, Carlos G. López Pombo, and Nazareno M. Aguirre. 2005. DynAlloy: Upgrading Alloy with Actions. In *ICSE*.
[13] Joel Galenson, Philip Reames, Rastislav Bodik, Björn Hartmann, and Koushik Sen. 2014. CodeHint: Dynamic and Interactive Synthesis of Code Snippets. In *ICSE*.
[14] Divya Gopinath, Muhammad Zubair Malik, and Sarfraz Khurshid. 2011. Specification-Based Program Repair Using SAT. In *TACAS*. 173–188.
[15] John V Guttag and James J Horning. 2012. *Larch: Languages and tools for formal specification*. Springer Science & Business Media.
[16] Tihomir Gvero, Viktor Kuncak, and Ruzica Piskac. 2011. Interactive Synthesis of Code Snippets. In *CAV*.
[17] Md Rashedul Hasan, Jiawei Li, Iftekhar Ahmed, and Hamid Bagheri. 2023. Automated Repair of Declarative Software Specifications in the Era of Large Language Models. arXiv:2310.12425 arXiv 2310.12425.
[18] Yang Hong, Shan Jiang, Yulei Fu, and Sarfraz Khurshid. 2025. On the effectiveness of large language models in writing alloy formulas. *arXiv preprint arXiv:2502.15441* (2025).
[19] Jinru Hua and Sarfraz Khurshid. 2017. EdSketch: Execution-Driven Sketching for Java. In *SPIN*.
[20] Mia Mohammad Imran, Preetha Chatterjee, and Kostadin Damevski. 2024. Uncovering the Causes of Emotions in Software Developer Communication Using Zero-shot LLMs. In *ICSE*. Article 182, 13 pages.
[21] Daniel Jackson. 2002. Alloy: A Lightweight Object Modelling Notation. *TSE* (2002).
[22] Daniel Jackson and Alan Fekete. 2001. Lightweight Analysis of Object Interactions. In *TACS*.
[23] Daniel Jackson and Mandana Vaziri. 2000. Finding bugs with a constraint solver. In *ISSTA*. 14–25.
[24] Jinseong Jeon, Xiaokang Qiu, Jeffrey S. Foster, and Armando Solar-Lezama. 2015. JSketch: Sketching for Java. In *FSE*.
[25] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. 2010. Oracle-guided Component-based Program Synthesis. In *ICSE*.
[26] Shan Jiang, Pranoy Kovuri, David Tao, and Zhixun Tan. 2026. CASCADE: LLM-Powered JavaScript Deobfuscator at Google. In *International Conference on Software Engineering, Software Engineering in Practice*. 11 pages.
[27] Shan Jiang, Zijian Yi, and Chenguang Zhu. 2026. Semantic Voting: Execution-Grounded Consensus for LLM Code Generation. *arXiv preprint arXiv:2605.08680* (2026).
[28] Shan Jiang, Zijian Yi, and Chenguang Zhu. 2026. Sketch-and-Verify: Structured Inference-Time Scaling via Program Sketching. *arXiv preprint arXiv:2605.08658* (2026).
[29] Shan Jiang, Chenguang Zhu, and Sarfraz Khurshid. 2024. Generating executable oracles to check conformance of client code to requirements of JDK Javadocs

using LLMs. *CoRR* abs/2411.01789 (2024). arXiv:2411.01789
[30] Shan Jiang, Chenguang Zhu, and Sarfraz Khurshid. 2026. OBsmith: LLM-powered JavaScript Obfuscator Testing. In *OOPSLA*. 30 pages.
[31] Zhihan Jiang, Jinyang Liu, Zhuangbin Chen, Yichen Li, Junjie Huang, Yintong Huo, Pinjia He, Jiazhen Gu, and Michael R. Lyu. 2024. LILAC: Log Parsing using LLMs with Adaptive Parsing Cache. *Proc. ACM Softw. Eng.* 1, FSE, Article 7 (July 2024), 24 pages.
[32] Xin Jin and Zhiqiang Lin. 2024. SimLLM: Calculating Semantic Similarity in Code Summaries using a Large Language Model-Based Approach. *Proc. ACM Softw. Eng.* 1, FSE, Article 62 (July 2024), 24 pages.
[33] Cliff B Jones. 1990. *Systematic software development using VDM*. Vol. 2. Prentice Hall Englewood Cliffs.
[34] Sungmin Kang, Gabin An, and Shin Yoo. 2024. A Quantitative and Qualitative Evaluation of LLM-Based Explainable Fault Localization. *Proc. ACM Softw. Eng.* 1, FSE, Article 64 (July 2024), 23 pages.
[35] Sarfraz Khurshid, Darko Marinov, Ilya Shlyakhter, and Daniel Jackson. 2003. A Case for Efficient Solution Enumeration. In *SAT*. 272–286.
[36] Etienne Kneuss, Ivan Kuraj, Viktor Kuncak, and Philippe Suter. 2013. Synthesis Modulo Recursive Functions. In *OOPSLA*.
[37] Viktor Kuncak, Mikaël Mayer, Ruzica Piskac, and Philippe Suter. 2010. Complete Functional Synthesis. In *PLDI*.
[38] Gary T Leavens, Albert L Baker, and Clyde Ruby. 1998. JML: A Java modeling language. In *Formal Underpinnings of Java Workshop (at OOPSLA'98)*. Citeseer, 404–420.
[39] Zhe Liu, Chunyang Chen, Junjie Wang, Mengzhuo Chen, Boyu Wu, Xing Che, Dandan Wang, and Qing Wang. 2024. Make LLM a Testing Expert: Bringing Human-like Interaction to Mobile GUI Testing via Functionality-aware Decisions. In *ICSE*. Article 100, 13 pages.
[40] Ferney A. Maldonado-Lopez, Jaime Chavarriaga, and Yezid Donoso. 2014. Detecting Network Policy Conflicts Using Alloy. In *Abstract State Machines, Alloy, B, TLA, VDM, and Z*. Springer Berlin Heidelberg.
[41] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. 2005. Jungloid mining: helping to navigate the API jungle. In *PLDI*. 48–61.
[42] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. 2011. CD2Alloy: Class Diagrams Analysis Using Alloy Revisited. In *MODELS*.
[43] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. 2011. CDDiff: Semantic Differencing for Class Diagrams. In *ECOOP*.
[44] Darko Marinov and Sarfraz Khurshid. 2001. TestEra: A Novel Framework for Automated Testing of Java Programs. In *ASE*. 22–31.
[45] A. Milicevic, D. Rayside, K. Yessenov, and D. Jackson. 2011. Unifying execution of imperative and declarative code. In *ICSE*.
[46] Fangwen Mu, Lin Shi, Song Wang, Zhuohao Yu, Binquan Zhang, ChenXue Wang, Shichao Liu, and Qing Wang. 2024. ClarifyGPT: A Framework for Enhancing LLM-Based Code Generation via Requirements Clarification. *Proc. ACM Softw. Eng.* 1, FSE, Article 103 (July 2024), 23 pages.
[47] Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. 2024. Using an LLM to Help With Code Understanding. In *ICSE*. Article 97, 13 pages.
[48] Timothy Nelson, Christopher Barratt, Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi. 2010. The Margrave Tool for Firewall Analysis. In *LISA*.
[49] Tim Nelson, Natasha Danas, Daniel J. Dougherty, and Shriram Krishnamurthi. 2017. The Power of "Why" and "Why Not": Enriching Scenario Exploration with Provenance. In *FSE*.
[50] Tim Nelson, Salman Saghafi, Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi. 2013. Aluminum: Principled Scenario Exploration Through Minimality. In *ICSE*.
[51] Razieh Nokhbeh Zaeem and Sarfraz Khurshid. 2010. Contract-Based Data Structure Repair Using Alloy. In *ECOOP*.
[52] Pierre M Nugues. 2006. *An introduction to Prolog*. Springer.
[53] Peter-Michael Osera and Steve Zdancewic. 2015. Type-and-example-directed Program Synthesis. In *PLDI*.
[54] Daniel Perelman, Sumit Gulwani, Dan Grossman, and Peter Provost. 2014. Test-driven Synthesis. *PLDI* (2014).
[55] Amir Pnueli and Roni Rosner. 1989. On the Synthesis of an Asynchronous Reactive Module. In *Proceedings of the 16th International Colloquium on Automata, Languages and Programming*. Springer-Verlag, 652–671.
[56] Natali Ruchansky and Davide Proserpio. 2013. A (Not) NICE Way to Verify the Openflow Switch Specification: Formal Modelling of the Openflow Switch Using Alloy. *SIGCOMM* (2013).
[57] Gabriel Ryan, Siddhartha Jain, Mingyue Shang, Shiqi Wang, Xiaofei Ma, Murali Krishna Ramanathan, and Baishakhi Ray. 2024. Code-Aware Prompting: A Study of Coverage-Guided Test Generation in Regression Setting using LLM. *Proc. ACM Softw. Eng.* 1, FSE, Article 43 (July 2024), 21 pages.
[58] Hesam Samimi, Ei Darli Aung, and Todd Millstein. 2010. Falling Back on Executable Specifications. In *ECOOP*.
[59] Ilya Shlyakhter. 2001. Generating effective symmetry-breaking predicates for search problems. *Electron. Notes Discret. Math.* 9 (2001), 19–35.
[60] I. Shlyakhter, R. Seater, D. Jackson, M. Sridharan, and M. Taghdiri. 2003. Debugging overconstrained declarative models using unsatisfiable cores. In *ASE*.
[61] Rishabh Singh and Sumit Gulwani. 2015. Predicting a Correct Program in Programming by Example. In *CAV*.
[62] Rishabh Singh and Armando Solar-Lezama. 2011. Synthesizing Data Structure Manipulations from Storyboards. In *FSE*.
[63] Armando Solar-Lezama. 2008. *Program Synthesis by Sketching*. Ph. D. Dissertation. University of California, Berkeley.
[64] Armando Solar-Lezama, Gilad Arnold, Liviu Tancau, Rastislav Bodik, Vijay Saraswat, and Sanjit Seshia. 2007. Sketching Stencils. *PLDI* (2007).
[65] Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodik. 2008. Sketching Concurrent Data Structures. In *PLDI*.
[66] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial Sketching for Finite Programs. In *ASPLOS*.
[67] Allison Sullivan, Darko Marinov, and Sarfraz Khurshid. 2019. Solution Enumeration Abstraction: A Modeling Idiom to Enhance a Lightweight Formal Method. In *ICFEM*. 336–352.
[68] Allison Sullivan, Kaiyuan Wang, and Sarfraz Khurshid. 2018. AUnit: A Test Automation Tool for Alloy. In *ICST*.
[69] Allison Sullivan, Kaiyuan Wang, Sarfraz Khurshid, and Darko Marinov. 2017. Evaluating State Modeling Techniques in Alloy. In *SQAMIA*.
[70] Allison Sullivan, Kaiyuan Wang, Razieh Nokhbeh Zaeem, and Sarfraz Khurshid. 2017. Automated Test Generation and Mutation Testing for Alloy. In *ICST*.
[71] Allison Sullivan, Razieh Nokhbeh Zaeem, Sarfraz Khurshid, and Darko Marinov. 2014. Towards a Test Automation Framework for Alloy. In *SPIN*.
[72] Emina Torlak, Felix Sheng-Ho Chang, and Daniel Jackson. 2008. Finding Minimal Unsatisfiable Cores of Declarative Specifications. In *FM*.
[73] Emina Torlak and Daniel Jackson. 2007. Kodkod: A Relational Model Finder. In *TACAS*, Vol. 4424. Springer, 632–647.
[74] Nalin Wadhwa, Jui Pradhan, Atharv Sonwane, Surya Prakash Sahu, Nagarajan Natarajan, Aditya Kanade, Suresh Parthasarathy, and Sriram Rajamani. 2024. CORE: Resolving Code Quality Issues using LLMs. *Proc. ACM Softw. Eng.* 1, FSE, Article 36 (July 2024), 23 pages.
[75] Kaiyuan Wang, Allison Sullivan, and Sarfraz Khurshid. 2018. Automated Model Repair for Alloy. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 577–588. doi:10.1145/3238147.3238162
[76] Kaiyuan Wang, Allison Sullivan, and Sarfraz Khurshid. 2018. MuAlloy: A Mutation Testing Framework for Alloy. In *ICSE*.
[77] Kaiyuan Wang, Allison Sullivan, and Sarfraz Khurshid. 2019. ARepair: A Repair Framework for Alloy. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 103–106. doi:10.1109/ICSE-Companion.2019.00049
[78] Kaiyuan Wang, Allison Sullivan, Manos Koukoutos, Darko Marinov, and Sarfraz Khurshid. 2018. Systematic Generation of Non-equivalent Expressions for Relational Algebra. In *ABZ*.
[79] Kaiyuan Wang, Allison Sullivan, Darko Marinov, and Sarfraz Khurshid. 2018. Solver-based Sketching of Alloy Models using Test Valuations. In *ABZ*.
[80] Wei Wang, Huilong Ning, Gaowei Zhang, Libo Liu, and Yi Wang. 2024. Rocks Coding, Not Development: A Human-Centric, Experimental Evaluation of LLM-Supported SE Tasks. *Proc. ACM Softw. Eng.* 1, FSE, Article 32 (July 2024), 23 pages.
[81] Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2025. Agentless: Demystifying LLM-based Software Engineering Agents. In *Proceedings of the 33rd ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2025*. ACM, 24 pages. To appear.
[82] Zijiang Yang, Jinru Hua, Kaiyuan Wang, and Sarfraz Khurshid. 2018. Test Execution Driven Synthesis of API Sequences With Conditionals and Loops. In *ICST*.
[83] Hua Zhong, Shan Jiang, and Sarfraz Khurshid. 2025. An approach for API synthesis using large language models. *arXiv preprint arXiv:2502.15246* (2025).
[84] Hua Zhong, Shan Jiang, and Sarfraz Khurshid. 2025. APRIL: API Synthesis with Automatic Prompt Optimization and Reinforcement Learning. *arXiv preprint arXiv:2509.25196* (2025).