LLMs for Debugging unsatisfiable Forge Specifications

Matthew Prenovitz[1]

Brown University, Providence RI 02912, USA matthew_prenovitz@brown.alumni.edu

1 Introduction

Formal Methods (FM) are a wide class of techniques that use formalism to reason about and verify systems. Forge is a "lightweight" Formal Methods tool [6] designed for educational contexts. Forge focuses on teaching by gradually building on FM concepts through the different language versions, ranging from simple relations to more complex temporal models [8]. It is used in Logic for Systems (CSCI 1710) at Brown University, a course where students model various systems. These range from physical master-key systems to network protocols and more. To illustrate Forge in action, take a simple example: a specification that models a group of friends.

```
1   #lang forge
2   sig Person {
3       bestFriend: one Person
4   }
5   pred notOwnBestFriend {
6       all p: Person |  {
7           p.bestFriend != p
8       }
9   }
10  pred mutualBestFriends {
11      all p: Person |  {
12          p.bestFriend.bestFriend = p
13      }
14  }
15  run {notOwnBestFriend and mutualBestFriends}
16    for exactly 6 Person
```

Fig. 1. Example Forge Model

This example can be broken down into three distinct parts, the `sig`s and `field`s, as seen in lines 2–4, which represent object types and their relationships to other objects in the world respectively. Predicates, as seen in lines 5–14, define sets of constraints over those definitions. The `run` command then invokes a set of

these constraints, calling a solver to find scenarios (often also called *instances*) that *satisfy* those constraints within user-given bounds on scenario size. Here, Forge is instructed to find scenarios in which:

- 5–9 each person has a best friend (who is not themselves); and
- 10–14 each person is the best friend of their best friend (i.e., the best-friends relationship is symmetric).

When Forge solves for these constraints, two results are possible: it either finds an instance (the constraints are *satisfiable*, or sat), or no such instance can be found (the constraints are *unsatisfiable*, or unsat). In this case, the solver successfully finds a 6-person instance satisfying both `mutualBestFriends` and `notOwnBestFriend`, shown in figure 2.

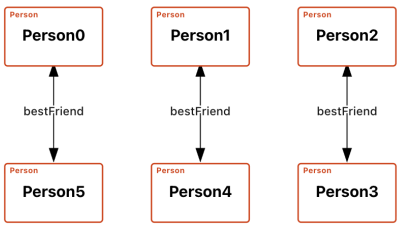

Fig. 2. Satisfiable Instance of three best-friend Pairs

However, what happens when we change the bounds, running the same command but for exactly 5 people instead of 6?

With exactly 5 people in the world, the solver was not able to find an instance that satisfies both the `notOwnBestFriend` and `mutualBestFriends` predicates: this is an unsat result. Crucially, unsat does not always mean that the specification is wrong. Tools like Forge can be used in different ways. Here, we showcase a kind of *exploration*, where the user is interested in seeing instances to help them understand a design space (and a lack of instance is unhelpful). But they

might also be trying to *verify* that a system guarantees some property, in which case failing to find a counterexample is actually desirable.

Regardless, if we want to find scenarios for our example with exactly 5 people, we need to weaken the constraints somehow—perhaps making it such that it is possible for a person to have no best friend, but if they have a best friend they must still be their best friend's best friend. To do this, we first change the field definition of `bestFriend` from `one` to `lone`, which allows the field to take on an empty value. We also change the `mutualBestFriends` predicate to be contingent: *if* a person has a best friend, *then* they must be mutual best friends. Together, these changes allow for odd-numbered friend groups:

```
#lang forge
sig Person {
    bestFriend: lone Person  // CHANGE: one to lone
}
pred notOwnBestFriend {
    all p: Person | {
        p.bestFriend != p
    }
}
pred mutualBestFriends {
    all p: Person | {
        // CHANGE: add guard
        some p.bestFriend
        implies p.bestFriend.bestFriend = p
    }
}
run {notOwnBestFriend and mutualBestFriends}
  for exactly 5 Person
```

While this is just a toy example, Forge has been used to model much more complex systems. These include the "usual" computer-science topics (e.g., self-balancing binary search trees, scheduling algorithms, and blockchains), but also mathematical concepts (e.g., group theory and cellular automata) and even surprisingly complex systems from outside STEM (e.g., baseball games and trading-system exploits in Minecraft, and voting systems). Regardless, as the complexity of the model increases, so too does the complexity of resolving potential unsat bugs.

2 Motivation

Unsatisfiable specifications are often challenging to debug for many reasons. Primarily, unsat by itself is a very uninformative. There is little indication of

how the model is being over-constrained or what conflicting lines of code are causing this over-constraint.

Forge can optionally return unsat cores, or the minimal set of conflicting constraints causing the unsat, in exchange for some performance cost. Forge's VSCode extension will highlight lines corresponding to different pieces of the core. However, these are not always immediately helpful because this can be overwhelming for students as they are actively learning and may not have built up the intuition or understanding of formal specifications to properly interpret these cores. This means there is no good formula for a student to debug. Oftentimes the only effective way for diagnosing the issue is going through each line of code until the conflicting restraints are identified. This is inefficient and time consuming, and not always beneficial for student learning. Students do have access to office hours with teaching assistants (TAs) who have solution code; however, for open ended projects such as for midterm and final projects, the TA may not be well equipped to assist in debugging code they have not seen before. This work is about testing a new strategy for debugging unsat to help students overcome some of these challenges. It posits that LLMs can effectively diagnose the cause of unsat instances in students' Forge models by utilizing context from the Forge file and unsat cores provided by the solver.

3 Methodology

This project explored two different ways for the LLM to interact with Forge: a Racket wrapper for Forge, which directly get instances from Racket memory via a CLI agent, and a Model Context Protocol (MCP) server that connects to an MCP compatible desktop LLM interface and retrieves running Forge instances through a mocked Sterling visualizer web socket. The resulting prototype code is available on Github[1].

3.1 MCP with Mock Sterling

The MCP is an open standard framework for connecting LLMs to external data sources. By providing a developer-written set of tools to the LLM, this can expand the functionality of an LLM to make API calls, query databases, run scripts, etc. These tools essentially act as functions for the LLM to call and utilize when responding to a query, allowing it to act outside of default functions. The LLM determines which tool is best for a given task based on descriptions, additional context provided to the LLM as a string at the head of each tool, potentially describing what the tool is, when and how the tool should be used, and any additionally context at the discretion of the the developer. .

For this research, the MCP allows for compatible desktop LLM interfaces to interact directly with Forge. Because the desktop applications do not allow for direct file access, it is necessary to use the MCP as it is the only means of

[1] `https://github.com/mprenovitz/forge-mcp-server.git`

enabling tooling to expand the functionality of the LLM to provide it Forge code and instances to interpret. Four tools are contained within the MCP:

- `getForgeInstance:` Retrieves an instance from running Forge code by sending a request to a mock Sterling visualizer for an instance of Forge. The instance is then passed back to the LLM in JSON format, and is now available within the chat for the user to query about.
- `runForgeFile:` take in Forge code provided within the users query, the tool then writes this code to a temporary file, runs the new file using shell commands, and the newly running Forge instance is retrieved through `getForgeInstance`, which is then passed back to the LLM. The key distinction between the tools is the context in which they are used
- `debugForgeFile:` Logically the same as `runForgeFile`, however the description provides the LLM alternative instructions for debugging so that it knows to edit and run the current Forge file and iterate if necessary if the proposed fix does not achieve the desired result. This distinction is necessary because the LLM will hand back what it believes to be a fixed file without running to verify.
- `killForgeInstance:` A utility tool that is only used within the other tools to ensure that any Forge processes are killed before a new Forge file is run or after an unsat instance is retrieved.

With these tools, Claude desktop is provided with all necessary functionality to interact with Forge specifications and, through its own inference capabilities, interpret and debug code. However, for this architecture, the Sterling visualizer prevents the MCP from intercepting instances as they are run, requiring a mock to take its place.

The mocked Sterling visualizer fulfills the same role as the real version, albeit without any visualization. Typically, Forge would connect to the real Sterling visualizer and send instance data to Sterling, serialized to XML which the visualizer can use. The mocked websocket behaves similarly, retrieving the XML from Forge; however, instead of opening a web browser for visualization, the mock sends the XML data directly to the MCP during a `getForgeInstance` call. This XML data is able to be parsed by the LLM in-chat, so no additional processing is necessary, allowing for a close approximation of a real Sterling visualizer. For the MCP to connect to the mock, certain options or configuration settings, need to be set. Two options are necessary:

```
option run_sterling serve
option sterling_port <some_port_number>
```

The former denotes if Forge should be run with or with or without the sterling visualizer, where passing "serve" indicates that the provider server is open and expecting Sterling-like requests. The latter denotes the port for Racket to use that the Sterling visualizer, or in this case the mocked Sterling, will connect to.

3.2 LLM CLI agent with a Racket Wrapper

Because Forge is built in Racket, it can be treated as a Racket library, allowing the LLM to directly access instances the racket memory. When Forge is run, it provides a lazy stream of instances in memory that would typically be parsed into XML data to be sent to Sterling to visualize. The the wrapper, however, allows for the bypasses the serialization step. In the Racket memory, the lazy stream of instances is represented as a tree, where each leaf is an instance and all its associated information. The wrapper functions as follows: the Forge file gets executed as a Racket module; this puts all of the aspects of the Forge file: `predicates`, `sigs`, `run` statements, etc., into scope. With everything in scope, the run statements can now be represented as objects which can be used to invoke the solver once an instance is asked for. From here, the wrapper goes into the Racket memory, finds the tree, and pulls the first instance off the tree, which is then parsed and printed for the user to view in the CLI. The wrapper provides an alternative approach towards LLM interaction as it is driven predominantly by Forge, with the LLM only being necessary for parsing and explaining instances. While it can write to files and run code via the CLI, this can also be done by the user and is not explicit functionality that is afforded by the LLM. Rather, the wrapper is allowing the LLM to access Forge, unlike the MCP, which is necessary for allowing the LLM to retrieve the XML data and use bash commands for running files when using the a desktop application with no direct file access.

4 Preliminary Results

Preliminary results on seeded bugs provide an initial gauge of the LLMs performance. These results are broken down into two tasks that focused on different potential solver outputs:

- Explaining sat instances: given a satisfiable specification, obtain a satisfying instance from the solver and ask the LLM to explain it in the context of the specification.
- Explaining unsat instances: given an unsatisfiable specification, explain why the specification is unsatisfiable. In some cases, the LLM was also provided with the unsatisfiable core generated by the solver in terms of row and column locations in the specification.

4.1 Explaining Satisfiable Instances

To test sat explanations, we used a small collection of CSCI 1710 homework assignments. Many of these specifications contained descriptive comments, `predicate` names, `sigs`, etc. For instance, the tic-tac-toe specification contains `sig` names like ‘Player‘, ‘X‘, and ‘O‘, which (due to the well-known nature of the game) might let the LLM generate explanations based on its *domain* knowledge rather than its insight into the specification. Thus, different portions of each file were

Model	Forge Type	Removed Comments	Masked Predicates and Functions	Masked Sigs	Masked Filename	Explained Instance	Identified Model
Tic Tac Toe	Froglet	No	No	No	No	Yes	Yes
Tic Tac Toe	Froglet	Yes	No	No	Yes	Yes	Yes
Tic Tac Toe	Froglet	Yes	Yes	No	Yes	Yes	Yes
Tic Tac Toe	Froglet	Yes	Yes	Yes	Yes	Yes	Yes
River Crossing	Temporal	No	No	No	No	Yes	Yes
River Crossing	Temporal	Yes	No	No	Yes	Yes	Yes
River Crossing	Temporal	Yes	Yes	No	Yes	Yes	Yes
River Crossing	Temporal	Yes	Yes	Yes	Yes	Yes	Yes
Reference Counting	Relational	No	No	No	No	Yes	Yes
Reference Counting	Relational	Yes	No	No	Yes	Yes	Yes
Reference Counting	Relational	Yes	Yes	No	Yes	Yes	Yes
Reference Counting	Relational	Yes	Yes	Yes	Yes	Yes	No

Fig. 3. Results for Satisfiable Explanation Queries

incrementally masked or removed before the LLM was prompted with a (now obfuscated) instance. Figure 3 summarizes the results.

For these small specifications, the LLM correctly explained instances every time. Moreover, when asked whether each specification was similar to a real-life system, we found that it was able to guess the exact nature of each (masked) example except one: a model of reference-counting memory management, which it misidentified as one of a logical framework system, a build system, or dependency manager. It identified a use of transitive closure in the model (to check reachability between memory cells), but attributed it to steps of logical inference. Nevertheless, it was able to explain the instance correctly, in general relational terms. This was encouraging in that it gave us confidence that the LLM could lay out the structure of an instance and explain why it satisfied the specification.

4.2 Explaining Unsatisfiable Instances

To test unsat explinations we used several files from the CSCI 1710 and Forge repository seeded with bugs for the unsat experimentation. These bugs took the form of either single line or multiline bugs. Multiline bugs are either:

- two independent unsat causing lines and;
- multiple lines compounded together that cause unsat

. For each bug, the LLM was queried in two independent chats:

- one with unsatisfiable cores generated by the solver and;
- one without unsatisfiable cores generated by the solver.

Tests contained no masking, only comment removal, since this is more accurate to student files the tool would see. Figure 4 summarizes the results of some of the tests.

Figure 4 identifies a few notable behaviors of the LLM. First, most bugs are resolved in one attempt per bug. All single-line bugs that were successfully fixed were resolved in one iteration. The LLM resolved multiline bugs in one

Model	Forge Type	Bug ID	Core Returned	Single or Multiline	Fixed	Iters to Fix	Interesting Behavior
Abstract Algebra Groups	Relational	1	No	Single	Yes	1	None - directly identified the bug
Abstract Algebra Groups	Relational	1	Yes	Single	Yes	1	None - directly identified the bug
LTL	Temporal	2	No	Multiline (Compound)	Yes	1	None - directly identified the bug
LTL	Temporal	2	Yes	Multiline (Compound)	Yes	1	None - directly identified the bug
Tic Tac Toe	Froglet	3	No	Single	No	∼	Completely misidentified the bug
Tic Tac Toe	Froglet	3	Yes	Single	Yes	1	None - no indication core was the reason for fix
Prim's Algorithm	Temporal	4	No	Multiline (Independent)	No	∼	Misidentified bugs in correct lines from the original
Prim's Algorithm	Temporal	4	Yes	Multiline (Independent)	No	∼	Misidentified bugs in correct lines from the original; missed second bug
Ring Leader Election	Temporal	5	No	Multiline (Independent)	No	∼	Treated an unbugged predicate as source of unsat
Ring Leader Election	Temporal	5	Yes	Multiline (Independent)	No	∼	Treated an unbugged predicate as source of unsat
Ring Leader Election	Temporal	5	No	Multiline (Independent)	Yes	2	When instructed to iterate, correctly identified bugs
Ring Leader Election	Temporal	5	Yes	Multiline (Independent)	Yes	2	When instructed to iterate, correctly identified bugs

Fig. 4. Results for unsat Explanation Queries.

or two iterations at the most (it tended to tackle independent bugs one at a time). Even with multiple conflicting constraints, access to the full file from the instance would still give the LLM enough context to debug small specifications without iterating.

Second, performance remained relatively consistent regardless of the core being generated: the cores appeared to not affect the LLM's ability to debug the provided files. While some cases showed a discrepancy (e.g., an instance generating cores had a different fix than an instance not generating cores) between chats, nothing in the LLMs responses indicated this is dependent on the absence or presence of an unsatisfiable core. Moreover, of these discrepancies, some cases showed fixed bugs when cores were not generated and unfixed bugs when cores were. This difference is most likely due to an LLMs non-deterministic nature. Even when provided an identical query and an identical bug, there is still going to be variation in the response. Typically this would mean variations in the responses diction, however this can also result in larger issues like inconsistent bug identification, or hallucinations as seen in bug 4.

Finally, effective prompting is important for ensuring consistent model behavior. While the LLM debugged well, performance could vary based on the specification and bug. Prompting impacted debugging performance, as seen in the bug 5 results. The LLM was able to identify multiline bugs better when instructed to iterate on itself (enabled by the MCP). Informing the LLM it was

possible for multiple bugs to occur prevented it from assuming there was only a single bug within the file.

5 Related Work

Forge: Forge is a *lightweight* FM tool built for education. The term "lightweight" FM was originally presented by Jackson and Wing [6], entailing a focus on ease of use and automatability over expressive power. Forge comprises three sub languages, each with increasing complexity [8]:

- Froglet: a "starter" language that limits the expressive power of the tool to partial and total functions;
- Relational Forge: which removes Froglet's restrictions, giving students arbitrary relations;
- Temporal Forge: which adds temporal-logic operators for reasoning about systems over time

The Forge language is based on Alloy, which is widely used but not made specifically for education [5].

Unsatisfiable Cores Solver-based tools like Forge and Alloy leverage the unsat cores that the solvers can return to highlight portions of specifications that might be implicated in the unsatisfiability. A core is a subset of the constraints given to the solver that suffices to show unsatisfiability. While there might be multiple conflicts in a constraint set that lead to unsatisfiability (and thus there might be multiple cores), a single core still provides a concise summary of a potential root cause. Cores have been used extensively in solver-based tools, and have a wide variety of potential uses [10], including fault localization.

Fault Localization There is extensive work on fault localization and specification repair for Alloy. Most of these approaches do not leverage generative AI in any way, and they are complementary to the work presented here. For example, AUnit [9], a testing framework for Alloy, has been used to support both automated test generation and mutation testing for Alloy specifications. $AlloyFL_{hy}$ [12], built atop AUnit, uses unsat cores to derive mutations that it then uses to identify potential faults.

There is also growing work in using LLMs to help debug formal specifications. For example, Alhanahnah [1] et al.'s AlloySpecRepair uses a dual LLM "loop", with one model generating repairs and the other judging and giving feedback on the revised specification. Relatedly, LLMs also show promise for generating tests for specifications: Cunha and Macedo [2] find LLMs can be effective at developing test cases for Alloy specifications. Interestingly, they find that negative tests (checking that an instance is not allowed) can be more challenging to generate than positive tests (checking that an instance is allowed).

Hasan et al. [3] combine traditional fault-localization approaches with LLM-based repair techniques. They found that this hybrid approach can outperform

either LLMs or traditional approaches on popular benchmarks like ARepair [11] and Alloy4Fun [7]. More recently, Hasan et al. [4] evaluated the capabilities of LLMs as automated repair tools. Four of their observations are directly relevant for this work:

- LLMs broadly performed worse than traditional approaches, however they were uniquely suited to solving complex bugs that traditional approaches could not.
- Few-shot prompting (providing the model with a few examples and a task description) shows significant improvements over zero-shot prompting (no additional context beyond the bug) by reducing response variation.
- Providing iterative feedback on LLM repairs showed substantial improvements for simpler tasks, but only marginal benefits for more complex specifications.
- Overconstraint errors were common for each LLM during repairs, although it is not clear how many were because the LLM was introducing *new* overconstraints, imperfectly fixing *initially* overconstrained specifications, or both. Regardless, it is clear that LLMs may be more likely to struggle with overconstraint, which influenced our heightened focus on unsat results.

6 Discussion

Limitations The prototype tool primarily used Claude Sonnet 4.5 and 4.6. It is therefore possible that other models would work better (or worse) than we observed.

Because this project began with building an MCP server for Forge, the vast majority of this project's experiments (section 4) used the MCP rather than the alternative Racket wrapper. The tradeoffs between the MCP and the wrapper approach are still relatively unexplored.

Moreover, an agent could potentially be given guided access to the Forge documentation to strengthen its knowledge of the language. The Forge book could likewise provide a useful set of working examples and discussion of bugs for the agent to reference.

The experiments from section 4 predominately used Forge specifications from Brown's CSCI 1710 and Forge's example repository. The majority of these models are quite small, since they were created for educational example purposes, meaning that they may not sufficiently exercise the LLM's reasoning ability.

The LLM also had access to both the original model source and the unsat cores from the solver; it is unclear how much the LLM leveraged the cores versus the raw model text. Moreover, the majority of the bugs that the LLM was asked to diagnose were seeded as part of the experiment.

Finally, while some TAs gave formative feedback on the tool, students did not. The primary goal of this work was to explore the design space and prototype a tool to debug unsat results via LLMs, and the tool itself was made available to students only in the last 2 weeks of the course. Other factors may have contributed to low uptake: students are likely using LLMs for debugging to

an extent. If chat-based methods are effective for their purposes, then perhaps the latest models may already be strong enough to make a separate tool unnecessary for debugging on small homework-scale assignments—especially when those assignments are based on well known systems or logic problems.

Future Work This work was formative, in that it built tooling as a proof-of-concept. There are multiple directions for future work:

- While seeding bugs ourselves allowed for quick iteration on the prototype, it cannot replicate actual debugging challenges faced by real students. Design idioms, language features, and other aspects of their (partially-complete) work will likely differ from the ground-truth "staff solution" Providing students with the tool early in the semester would give more insight into both the ergonomics of the tool (i.e., can students use it?) and its practical usefulness.
- Because authentic student data would likely be relatively sparse compared to the vast space of *possible* bugs, seeded bugs still promise to give us insight into the overall problem. This raises the question of how to efficiently and automatically seed complex bugs—even ones tailored to a specific task—into a file. One possibility would be to draw on existing mutation-testing work for Alloy (e.g., [12]). Another would be to guide an AI agent to making these changes in a structured way.
- The specifications tested here were relatively small. It is very likely that, outside the educational context, LLMs would benefit from additional information: not only cores, but perhaps the underlying proofs as well. We anticipate a hierarchy of needs as specifications, and the systems they model, become more complex: in many cases, the LLM will be able to correctly identify root causes without recourse even to the unsat core. In others, the core will be necessary. In other cases, even richer information might be required.
- While Claude Sonnet was useful, there were roadblocks: rate limits, locking advanced models behind subscriptions, and even potential changes in the cost model (e.g., Anthropic's Claude Code subscription) mean that reliance on Sonnet or Opus would be a single point of failure in the classroom. One possible way forward would be to use open-source models that can run on students' laptops, or else on department servers. Such models are often powerful, although less general-purpose, and so further experiments would be needed before an AI debugging tool could be deployed in a real setting.
- Prompt engineering (section 5) has substantial impact on the consistency and quality of results. In this work all prompts were zero-shot, so one possible way forward would be to compare the performance of few-shot prompts: e.g., providing solutions for a given assignment, when available.

References

1. Alhanahnah, M., Hasan, M.R., Xu, L., Bagheri, H.: An empirical evaluation of pre-trained large language models for repairing declarative formal specifications. arXiv preprint arXiv:2404.11050 (2024)

2. Cunha, A., Macedo, N.: Validating Formal Specifications with LLM-generated Test Cases (2026), https://arxiv.org/abs/2510.23350
3. Hasan, M.R., Alhanahnah, M., Stevens, C., Bagheri, H.: Towards More Dependable Specifications: An Empirical Study Exploring the Synergy of Traditional and LLM-Based Repair Approaches. In: 2025 55th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). pp. 88–101 (2025). https://doi.org/10.1109/DSN64029.2025.00023
4. Hasan, M.R., Li, J., Ahmed, I., Bagheri, H.: Automated Repair of Alloy Specifications in the Era of Large Language Models. IEEE Transactions on Software Engineering pp. 1–31 (2026). https://doi.org/10.1109/TSE.2026.3682711
5. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT press (2012)
6. Jackson, D., Wing, J.: Lightweight Formal Methods. Carnegie Mellon University (1996)
7. Macedo, N., Cunha, A., Pereira, J., Carvalho, R., Silva, R., Paiva, A.C., Sozinho Ramalho, M., Silva, D.: Experiences on teaching alloy with an automated assessment platform. Science of Computer Programming **211**, 102690 (2021). https://doi.org/https://doi.org/10.1016/j.scico.2021.102690
8. Nelson, T., Greenman, B., Prasad, S., Dyer, T., Bove, E., Chen, Q., Cutting, C., Del Vecchio, T., LeVine, S., Rudner, J., Ryjikov, B., Varga, A., Wagner, A., West, L., Krishnamurthi, S.: Forge: A Tool and Language for Teaching Formal Methods. Proc. ACM Program. Lang. **8**(OOPSLA1) (Apr 2024). https://doi.org/10.1145/3649833
9. Sullivan, A., Wang, K., Khurshid, S.: AUnit: A Test Automation Tool for Alloy. In: 2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST). pp. 398–403 (2018). https://doi.org/10.1109/ICST.2018.00047
10. Torlak, E., Chang, F.S., Jackson, D.: Finding Minimal Unsatisfiable Cores of Declarative Specifications. In: Cuellar, J., Maibaum, T., Sere, K. (eds.) FM 2008: Formal Methods. pp. 326–341. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
11. Wang, K., Sullivan, A., Khurshid, S.: ARepair: A Repair Framework for Alloy. In: 2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion). pp. 103–106 (2019). https://doi.org/10.1109/ICSE-Companion.2019.00049
12. Wang, K., Sullivan, A., Marinov, D., Khurshid, S.: Fault Localization for Declarative Models in Alloy. In: 2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE). pp. 391–402 (2020). https://doi.org/10.1109/ISSRE5003.2020.00044