Humboldt-Universität zu Berlin
Mathematisch-Naturwissenschaftliche Fakultät
Institut für Informatik

When Does This Line Get Triggered? Explainable Line Reachability Using Semantic Constraints

Bachelorarbeit

zur Erlangung des akademischen Grades
Bachelor of Science (B. Sc.)

eingereicht von: Angelina Teodoridis
geboren am: 24.05.2000
geboren in: Berlin

Gutachter/innen: Prof. Lars Grunske
Marc Carwehl

eingereicht am: ..... verteidigt am: .....

Abstract
Context: Explaining code is an important task that enables software engineers to better navigate their own or other people's code more easily. With a growing amount of code lines, confusion as to what a given program does in the first place may be obscured, leading to more errors in maintaining or expanding the given code.
Objective: Avicenna has been shown to work effectively work on bug-triggering inputs, diagnosing what kinds of inputs cause program failures to occur. We extend this approach to work for any given code line, allowing us to gain line-constraints that explain input structures that trigger the given line under test.
Method: In order to extend Avicenna, we use a new class, AviX, which inherits Avicenna's basic uses. AviX will be used to navigate the different inputs needed to scan program files. Furthermore, we will add a new kind of oracle, a line oracle, to Avicenna's base oracle constructor class.
Results: AviX is effective in diagnosing line-triggering inputs, and is comparable to Avicenna's performance when diagnosing bug-triggering inputs. The biggest limitations are found in the pattern catalog, which is used for Avicenna's ISLearn pattern matcher, which determines all kinds of constraints that may be found. Missing patterns will lead to worse constraints, as specific program circumstances may not be properly described.

Contents

1 INTRODUCTION

Understanding previously written code is a challenge every software engineer is faced with at some point. Because of this, much research has gone into determining and improving code readability [14, 19, 43, 44, 45, 52], explaining bugs [1, 10, 15, 16, 20, 24], and locating bugs[2, 34, 48, 53, 57, 61]. The general idea for most of these practices is, to reduce the amount of direct interactions a software engineer has with the code at hand in order to save time.

We wish to further enable programmers' understanding of code they or someone else has written in the past, as well as reliably automate aspects of testing and debugging program code, by introducing AviX. AviX adds extra functionality on top of Avicenna's [1] feedback loop, allowing it to explain single lines as desired.

Normally, Avicenna would only able to mine constraints for inputs that will trigger a bug in a given program under test. AviX generalizes this approach to investigate whether a line is triggered by an input, explaining the input's properties. Instead of checking for faulty behavior triggered by an input, the constraint miner checks for line-triggering input behavior. This way, a constraint that describes inputs that are likely to trigger the line under test can be found, enabling us to better understand a section of code and program behavior and fortify test suites verifying the code branch our line resides in.

Avicenna is the tool introduced in the paper 'Semantic Debugging' [1], wherein Avicenna is shown to effectively diagnose faulty code behavior by returning constraints that explain the structure of failure-triggering inputs.

What AviX can be used for: AviX can generate constraints defined by the Input Specification Language (ISLa) [51] for almost any line in a given program under test. Out of 30 different lines that we tested for unique program branches across our four subjects, 29 constraints were found that give full or partial categorizations of inputs that could trigger these lines.

The mined constraints can be used for improving code understanding by directly linking them to a line in the code, using them to categorize already existing inputs, or generate new inputs that will trigger the line of interest. Constraints could be used in code documentation, because ISLa-produced constraints are first-order logic formulas. Furthermore, ISLa's semantic fuzzer also allows us to generate new inputs using ISLa constraints and context-free grammars.

The constraints describe semantic properties that inputs must fulfill in order to trigger a given line, which can be used to initially understand a code base, double-check the code's behavior, or to confirm code behavior.

Due to Avicenna's sophisticated feedback loop, and high quality outputs, AviX can be used to lay out the groundwork for highly precise input generation that could be used for further bug-proofing a project. Test-suites could be extended by adding test inputs or even minimized by using semantic constraints to classify current test-inputs and removing inputs with duplicate properties. Oversights in code could be exposed by describing behavior that should not be possible in a given line, but is when testing Avicenna's feedback loop on it. For existing bugs, Avicenna's base use is still fully

accessible, diagnosing faulty program behavior.
Because of the aforementioned points, AviX could help in further automating the testing and debugging processes. Line-constraints may also be used for code documentation, describing input properties that are likely to trigger certain lines.
We will also show that AviX can give us information on dead code or code that is always triggered.
Why extend Avicenna? We chose to extend Avicenna due to its demonstrated efficacy and speed [1], as well as its versatile code-base, which allows us to easily add alternative approaches in classifying constraints. For AviX, this is the line-oracle. Especially when compared to other constraint-driven methodology such as KLEE [7**?**], which uses symbolic execution [27, 59], Avicenna's speed and quality offers a strong alternative for light-weight code diagnosis.
Due to Avicenna's speed, its analysis can actively help in a debugging effort, as it can be run whenever needed, averaging runtimes under 10, and often even 5, minutes for our tests. Furthermore, Avicenna is extendable, which offers a strong foundation on which to build further uses upon.
Our Contributions: First, we showcase Avicenna's adaptability by expanding it. AviX extends Avicenna by adding an extra kind of oracle – a line-oracle – for tracing triggered lines during a given program run. This extension is enabled in big parts by SFLKit [48, 49].
Therefore, this is a feasibility study, in which we aim to show Avicenna's ability to be extended, but also introduce AviX as a code line-analysis extension in particular. Secondly, we test AviX on four different subjects, testing the mined constraints for their input-generation and input-classification abilities.
We show that Avicenna's feedback loop works for single-file programs, which allows us to scan an input's line coverage in the program-under-test. The coverage information lets us assess and categorize the inputs as line-triggering, allowing us to extract their semantic properties.
Last, we discuss our results by evaluating our collected data as well as analyzing potential threats to our research's validity.

Paper Overview: Section 2 contains our motivation for extending Avicenna and shows how AviX works using an example. Section 3 introduces all relevant background information needed to understand AviX's implementation, which is broken down in Section 4. Our experimental setup in Section 5 poses our research questions and gives a detailed rundown of our methodology and experiments. We evaluate our results in the following section, Section 6, discussing and summarizing our data and what we can take from it. Limitations and threats to the validity of our research are discussed in Section 7. Lastly, we discuss other related work that is concerned with adjacent problems to ours in Section 8 and conclude our paper in Section 9.

Introduction Summary

AviX is our tool that extends Avicenna by adding a line-oracle that can trace program lines as the program is run. By using this information, we showcase that Avicenna's analysis approach for bug-triggering inputs can be extended to more general information, such as investigating a line's trigger-conditions.
We believe that AviX can aid in improved code understanding and debugging practices by diagnosing code behavior in a more generalized manner.

2 MOTIVATION

By explaining code behavior through categorizing line-triggering inputs, AviX seeks to reduce confusion when reading code [3, 11, 13, 19], and make it more readable by providing semantic explanations for these code lines. By leveraging Avicenna's ability to reliably and quickly return constraints [1], we seek to use these constraints to add an extra layer of code explainability [10, 16].
Consider `middle`, a program used to return the middle value of three integer inputs greater than 0, `x, y, z`.

2.1 Motivating Example – Middle

In the following example, we will showcase how AviX may assist in understanding a code line's functionality by providing us with a semantic constraint over the line-triggering input space. The input space is defined by a context-free grammar, portrayed in Listing 10 in the appendix.
First off, let us analyze Middle's code – as seen in Figure 1 – by hand. Most lines are made up of if-statements that determine the bigger of two values taken from the inputs. For example, Line 5 is reached and y is returned as 'middle', if and only if `y < z` (Line 3), and `x < y` (Line4).
Now, in order to do this for every line, we could separate the program into its branches and check each condition, documenting it along the way. Alternatively, AviX could do it for us. By providing it with an input grammar for Middle and defining the target line, we can receive an explanation for each line in Middle that will tell us exactly what we need to know.
Without knowing this program, we can just pick a line at random in order to understand its constraints. Two kinds of outputs are possible for a run of a random line using AviX.

* A semantic constraint, describing conditions that must be fulfilled by an input in order to trigger the given line.
* A sort of error that informs us that we are missing a line-triggering or non-triggering input.

```
1  def middle(x, y, z):
2      m = z
3      if y < z:
4          if x < y:
5              m = y
6          elif x < z:
7              m = x
8      else:
9          if x > y:
10             m = y
11         elif x > z:
12             m = x
13     return m
```

```
(forall <<> elem_0 in start:
  exists <y> elem_3 in start:
    (>= (str.to.int elem_0) (str.to.int elem_3)))
AND
(forall <x> elem_1 in start:
   exists <z> elem_2 in start:
     (> (str.to.int elem_1) (str.to.int elem_2)))
```

Figure 1: `middle`'s source code and the constraint mined by AviX for line 12.

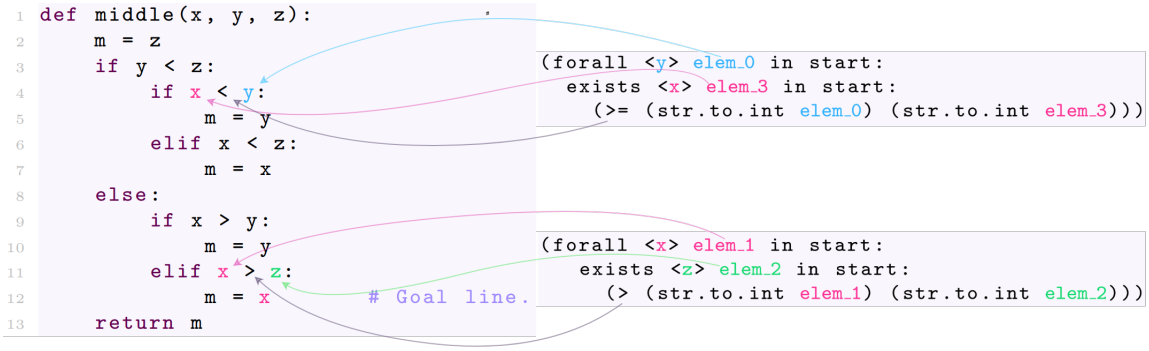

Figure 2: The constraint matched to `middle`'s source code. x and y are compared in the first step, whereas x and z are compared later on. Inputs that fulfill both constraint parts will trigger line 12.

For this example, let us pick line 12, because it is relatively deep within Middle, and requires multiple if-statements to be traversed. After a few minutes, AviX returns a constraint that accurately describes an input's required properties in order to trigger line 12 when running Middle with it. The constraint can be seen in Figure 1.
We separated the two parts of the constraint, as each half describes one line. When concatenating those two parts, AviX describes inputs that fulfill all of the restrictions described by the constraint.
The constraint is pretty much a summary of all relevant lines that lead us to line 12. As shown in Figure 2, each half of the constraint describes a relevant if-branch. A compelling example of Avicenna's strong pattern matching can be seen in the first half of this constraint, perfectly inverting the if-clause in line 4 from `x < y` to `x ≥ y`, as equal inputs would not cause the branch to be gone through as well. The same works in line 12, which is hidden behind multiple if-clauses. This shows us that AviX can dive deeply into a program in its process of mining line-constraints.
Though they are helpful for understanding code snippets, constraints can be used for more than just that. By using ISLa's semantic fuzzer, we can create new inputs that fulfill the constraint we have just received. This allows us to fortify our test-suite

by adding more inputs of a specific kind, allowing us to test for more than just one input at a time.
Furthermore, we can use the constraint to parse inputs to check if they fit into the constraint. If they do, they would trigger line 12, were we to run `middle` with that input. By doing this, we can further classify inputs in test suites, automating parts of testing and debugging that can be rather tedious and confusing when testing large code bases.
Even imperfect constraints can be used for fuzzing, as we will show later on.

Motivation Summary

We can use AviX to automatically explain what kinds of inputs trigger certain lines. The explanations come in the shape of semantic constraints, written as first-order logic formulas.
These constraints can be used for fuzzing, predicting input behavior (is it line-triggering or not), or be read to gain a deeper understanding of the line's trigger-requirements.

3 BACKGROUND

In this section, we contextualize AviX and the methodology and tools it employs in order to deepen the reader's understanding. In order of the following list, we will go deeper into Avicenna's inner workings, finishing with an overview of Avicenna overall:

* Automated Debugging
* Code Understanding
  - Code Readability
  - Code Explanation
* Fault Localization – SFLKit
* Invariants and Constraint Solvers – ISLa and ISLearn
* Test Oracles
* Avicenna

3.1 Automated Debugging

Debugging is time-intensive and difficult. Because of this, many procedures have been developed over the years that are focused on reducing the human workload in debugging. Examples for automated debugging approaches include: fault localization, [2, 34, 48, 53, 57, 61] understanding and explaining code [1, 20, 22, 24, 50, 54, 57], input generation [6, 9, 17, 32, 33], and automated program repair [7, 30, 31, 36, 40, 42]. We will focus on fuzzing, fault localization, and explaining code for our background section.

Fuzzing Fuzzing is the process of randomly generating new inputs for a program. Because of its high degree of randomness in its simplest form, fuzzing has been extended over the years. To name a few approaches, input grammars [58], directed greybox fuzzing [6], and evolutionary fuzzing [18] are some that seek to improve the accuracy of fuzzing and make it more worthwhile. The `GrammarFuzzer`, provided by TheFuzzingBook, utilizes a context-free grammar that describes valid inputs for a given program. Grammar fuzzers are generally fast, and depending on the grammar, perform very well. The more general a grammar is written, the worse performing grammar fuzzers become.
Avicenna uses a grammar fuzzer in its feedback loop in order to create new inputs. Furthermore, we will utilize grammar fuzzing for our later experiments, as well as semantic fuzzing, which will combine our semantic constraints with our input grammars.

3.2 Code Comprehension

In simple terms, we define code comprehension, or code understanding, as the level of understanding a developer has when regarding their project. A low level of code understanding may mean that a developer only confidently knows their way around whatever they are currently working on at the moment. In this example, a developer would have to re-interpret their old code, as they might have forgotten what their previous code did in its detail. On the contrary, a high level of code comprehension would be a clearer understanding of each part of code written by the developer. They might have to read old code to remember, but once they do, they can find their way around older code fragments easily.
In general, as an approach towards explainable error behavior, Avicenna aims to increase comprehension of code by offering alternative ways of interpreting failure information. AviX takes this a step further, as we want to not only explain buggy behavior, but any line's behavior if possible. Important for this are the following: Code readability and code explanations.

Code Readability Code readability research relates to code reviews in the sense that reviewing code for releases and ongoing maintenance, the readability of code is a crucial factor in determining how easy and efficient this process is.
Code readability and a level of confusion can be related when reading through old code [5, 12, 14, 19, 37, 43, 44, 45, 52]. The confusion usually stems from a lack of readability practices [13, 19], often described as syntactical and structural features [5, 52]. In-line documentation, such as comments [19], lexical, and semantic properties [43] have also been found to help with a code's level of comprehension, when classified by metrics and people alike. The semantic features mentioned here mostly refer to readable variable names. In combining in-line commentary with our semantic constraints, however, we believe that code readability may be further improved.
This shows that semantic information for code lines can help with the readability and therefore understanding of code – it is easier to read what you understand.

Code Explanations In regard to automated debugging, code explanations are oftentimes logical constraints –or invariants– [1, 25, 50] or summaries in natural language [26, 56] that help deepening a person's code understanding.
The main purpose here is to describe *what* happened, not *where* it happened.
Avicenna returns ISLa invariants [50], which help categorize inputs that are relevant to us. ISLa constraints will be introduced more closely in a following section. These constraints help us categorize our inputs even more precisely and more importantly, they can tell us exactly what semantic features we are looking for in our inputs. Through pairing such a constraint with a line in our code, a lot of heavy lifting is taken off of the person reading through the code, because they will not have to manually track the program path to their desired line that they are trying to understand.
Furthermore, by using a semantic constraint in combination with our input grammar, we can describe the desired inputs quite precisely. Even a constraint that is not 100% precise can assist in understanding a program's inner workings here. These grammars and constraints may also be used for fuzzing, adding another layer of comfort to the debugging process.

3.3 Fault Localization - SFLKit

Fault localization specifically refers to finding faulty or potentially faulty lines in a program [2]. While AviX is not a traditional fault localization tool, as opposed to regular Avicenna maybe, we use different methods when tracking program behavior for finding out if a line has been triggered or not.
With AviX, in order to find out if an input triggered a line, we track all lines that have been triggered by this input. Program instrumentation allows us to do this for every program under test. When instrumenting a program, we denote every code line we can find in our program under test and add an extra function call to it. This function call tracks if the line had been triggered or not. That means that in order to track those lines, we need to run our instrumented program version whenever testing for triggered lines. Triggered lines are saved to event-files. These event-files contain our coverage information, which we can parse later in order to retrieve all lines a given input has triggered.
We instrument our programs under test, create our event-files, and analyze our event-files using SFLKit. Originally, SFLKit [48, 49] is a fault localization tool, employing metrics to determine suspiciousness scores for buggy lines. Here, we can use the foundation for denoting faulty lines and combine them with Avicenna's oracle function in order to keep track of inputs that trigger the target line.
This also shows us the adjacency of AviX to fault localization. AviX could be used for investigating suspicious lines or even find constraints for lines that exhibit behavior that is not intended.

Listing 1: Two constraints that we will present for `expression`, Line 105

```
# Constraint 1 - for middle
forall <y> elem_1 in start:
  exists <z> elem_2 in start:
    (>= (str.to.int elem_1) (str.to.int elem_2))

# Constraint 2
exists <operator> elem_xy in start:
    inside(elem_xy, start)
```

3.4 Invariants and Constraint Solvers - ISLa and ISLearn

The explanations that Avicenna provides are semantic constraints. These constraints describe semantic properties that behavior-triggering inputs must fulfill in the bounds of our pre-defined input grammar. The Input Specification Language (ISLa) [50] provides us with the means to combine these constraints with our grammar.
ISLa is defined as a "declarative specification language for context-sensitive properties [...] based on context-free grammars." [50]
ISLa is based on Satisfiability Modulo Theorem (SMT) formulas, which are written in first-order logic, meaning they employ the use of predicates such as 'for all' or 'exists'. Furthermore, we may define properties such as 'greater than' or 'is inside', which are context-sensitive, semantic properties. Predicate logic cannot be expressed in context-free grammars, as exactly these predicates set a context by comparing numbers or defining scopes. Constraints may look like the ones in the motivating example from the beginning, or like the ones shown in Listing 1.
SMT formulas such as these can be solved using the Z3 SMT-solver, which determines the satisfiability of a given rule. The semantic properties added by predicates can cover blind spots of our otherwise context-free input grammars.
The definitions of invariants and constraints are near-interchangeable for our purposes. Generally, invariants describe conditions that must be met throughout the program-run, whereas constraints must only be valid at a specific point in time. Avicenna returns constraints that can be used for input generation. They are not invariants, because inputs may change from their starting point at any time.
ISLa also provides us with a fuzzer [50]. This semantic fuzzer combines our input grammars with the semantic constraints, allowing us to express semantic properties of our inputs without having to create a context-sensitive grammar. While this fuzzer takes longer to create single inputs, it is highly precise in creating inputs that are relevant to our given constraint.
ISLa constraints can be mined by their own pattern-matching learner, named ISLearn. ISLearn is provided with hand-picked ISLa-formula patterns that it uses in order to match relevant semantic properties of inputs that are fed to it. By classifying inputs as behavior-triggering and not behavior-triggering, the learner can match constraints to the triggering inputs, while maintaining generality by attempting to exclude non-triggering inputs with its matched constraint.
ISLearn has been inspired by Daikon [20]. Daikon is an engine used for finding

invariants when running a program over a set of test cases [20]. These invariants describe general program behavior, such as loop exit conditions, for example. Another similarity to our approach is the use of program instrumentation when determining these constraints.
Avicenna is provided with ISLa's fuzzer, and ISLearn's pattern matching and learning capabilities, and we will use the fuzzer for our input generation experiments in later sections.

3.5 Test Oracles

A test oracle is a function that predicts the outcome of a code run. This is known as the test oracle problem [4].
For Avicenna, an oracle is simply the basic source code of our program under test, which works as intended, allowing us to compare the behavior of this program and our current, potentially faulty, program under test. By using inputs that are generated during a run of Avicenna and running the oracle with them, we can determine whether or not these inputs exhibit relevant behavior, namely 'triggers a bug'.
Oracles are crucial for adapting Avicenna to other input classification methods, such as determining if a line as been triggered.

3.6 Avicenna

Now that we have established the baseline for all different kinds of tools and concepts we use, we can take a deeper look at Avicenna's inner workings.
Three key properties are highlighted, as the authors introduce Avicenna in their paper: "Precise, general, and extensible" [1]. Furthermore, Avicenna is described as quick at narrowing down the initial search space, as well as not prone to over-specialization in its constraints as it uses passing *and* failing inputs to create its constraints [1]. As we work with Avicenna, we will call back to these properties in order to determine whether they hold true for our uses.
Avicenna is first and foremost a tool meant for aiding in debugging code. By returning input constraints for faulty program behaviors, the debugging programmer in charge can better understand the issue at hand. The level of explanation offered by these constraints is a sort of semantic trimming of our input class described by our context-free grammar. By working with this reduced input set, we can gain a better overview of what potential causes our bug has, depending on how accurate the constraint is deemed to be by Avicenna.
Now we will describe a general run of Avicenna using Figure 3 as a guide.

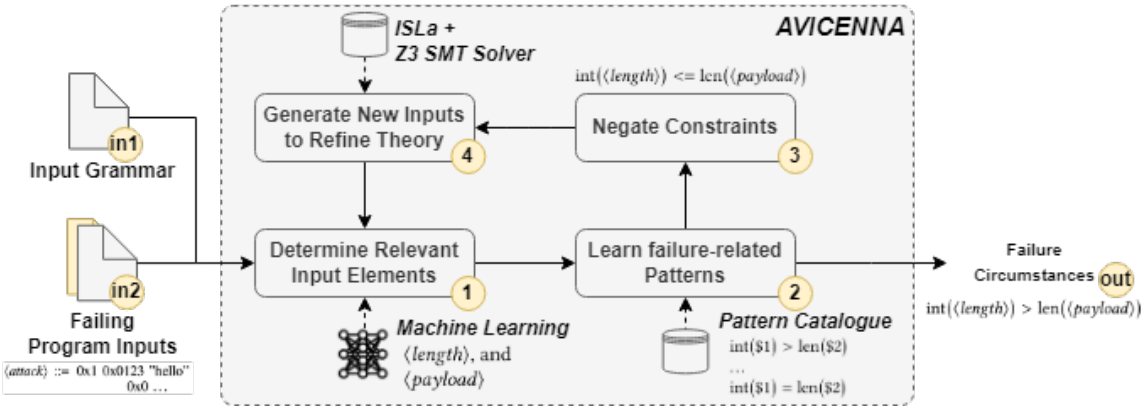

Figure 3: Overview of Avicenna's refinement loop for the heartbleed bug.

1. **Inputs**:
   (a) The **input grammar** is a context-free grammar that describes the broad input space for our program under test.
   (b) Our **initial inputs** are currently mandatory, and must contain at least one passing and one failing input.
   (c) A program **oracle** for our program under test.
2. **Feedback Loop**:
   (a) Inputs are **classified** as failing and passing by the oracle.
   (b) ISLearn tries to **match a constraint to the failing inputs**, checking how many failing inputs are correctly identified by the constraint in order to maximize its recall scoring.
   (c) Then, ISLearn tries to **generalize the constraint** to the passing inputs in order not to falsely classify all inputs as failing, increasing the precision score.
   (d) Constraints are ranked based on their scores.
   (e) Generate new inputs using a fuzzer.
   (f) Start over and repeat until maximum iterations are reached. 10 by default.
3. **Output**:
   (a) **Constraint** describing our failing inputs.
   (b) **Precision and recall** scores, describing our constraint's performance in predicting what inputs will be failure-inducing.

Avicenna's feedback loop works well for faulty programs, achieving around 90% precision and recall for its tests [1].
Due to its structure, we can modify the kinds of inputs we pass into Avicenna's loop. In the upcoming section, we will explain how AviX extends Avicenna in great detail.

Background Summary
In this section, we gave an overview of all relevant background information for AviX.

Importantly, we introduced concepts such as: instrumentation of our programs for line-checking using SFLKit, ISLa's constraints and fuzzing capabilities, ISLearn as a constraint miner, code comprehension (readability and explanations), and lastly, Avicenna as a tool.

4 DETERMINING LINE TRIGGER CONDITIONS

In this section, we give an overview of the differences between Avicenna's base version and AviX, explain our implementation, and give a rundown of how to utilize the additions we implemented.
The bulk of AviX's implementation lay in expanding Avicenna's oracle constructor, since AviX uses the same feedback loop as Avicenna.
Differences to Avicenna: Figure 4 showcases all changes made by AviX, including differences in Avicenna's feedback loop that occur due to the change in oracle. Most importantly, Avicenna's oracle constructor has been expanded to include line-oracles that may trace the line coverage of inputs over the given program under test. The classification of inputs is changed due to the adjustment of the oracle function, leading to a change in behavior learning. The last notable change is that AviX requires the program path to the program under test, including the function that is being tested within the program under test. Currently, AviX only supports single-file programs.
Using AviX: When creating an AviX class object, the same default inputs are required as for baseline Avicenna, meaning a context-free input grammar and a list of initial inputs. Additionally, we require the program path and the line oracle, as mentioned before. Lastly, we can change some fine-tuning in regard to the target precision and recall scores for a mined constraint, as well as the maximum amount of features we allow to be matched for our constraint. Python was used to implement AviX, just like Avicenna.
Program instrumentation takes place during the construction of the AviX object.
AviX can instrument Python programs based on line-events using SFLKit. Because SFLKit has the ability to also instrument programs based on other events, such as triggered branches or functions, these could also likely be added into AviX.
Program instrumentation takes place during the construction of the AviX object. When running AviX, we will only use the instrumented version of the program under test, because it allows us to track any movement.
Later on, this instrumented program will be imported and used in our runs. Currently, AviX only supports single-file projects, due to the limitations of SFLKit's instrumentation at the time of implementation.
Oracle Construction: As mentioned in our background, section Figure 3, the oracle is a function that runs the program under test and determines if the relevant behavior

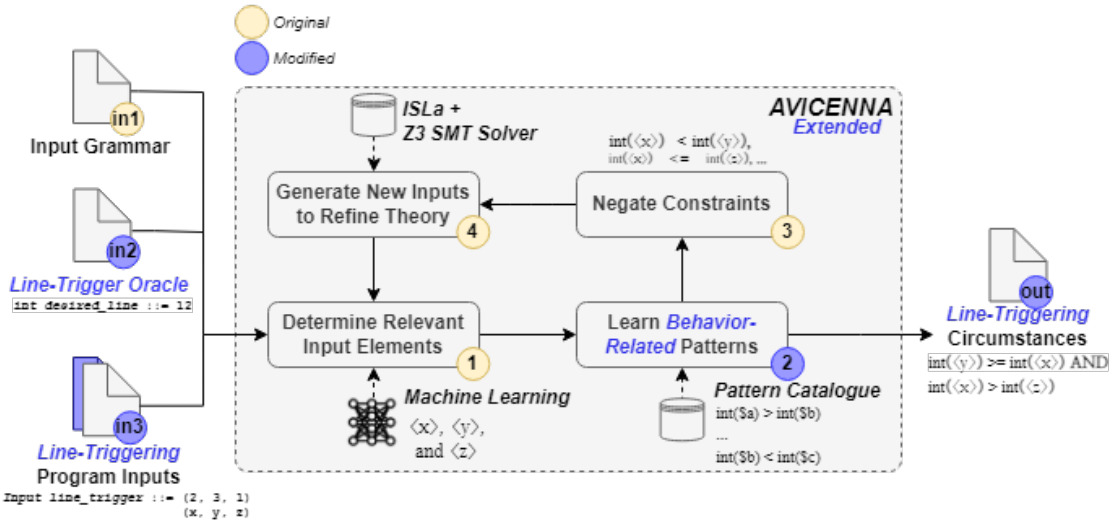

Figure 4: AviX extends Avicenna. Here we see the example for finding a constraint for `middle` at line 12, just like in our motivating example earlier.

was triggered, classifying it as line-triggering or non-triggering.
For constructing the oracle, we need following inputs: The program under test, the desired line, an optional input converter, the resource path, leading to the folder containing the base files, instrumented files, and event files that are gathered during the feedback loop.
The oracle construction method checks what kind of oracle is being constructed. In the case of a line oracle, the above inputs are used to create the oracle function with set parameters. We only need to construct the oracle once before running the Avicenna feedback loop with it.
Using the Oracle: Once the oracle is constructed, it returns a function that takes an input and returns an input classification. The oracle creates a binary event file, which contains all triggered line events. The event file creator is where inputs may be converted and the instrumented function is imported and ran, creating the event file in the process. Afterwards, the coverage of the input is analyzed by SFLKit's Analyzer class, returning a list of integers which represent our triggered lines. We can simply check if our desired line is in the coverage that was returned. If it is, we return a tag that describes that the given input is line-triggering, prompting the learner to focus on these inputs and others like it for its matching process.
For AviX, we used Avicenna version 0.9.1. At the time of writing, a new and improved version is about to be released, however, this should not change many findings in this paper.

AviX aims to use Avicenna's fault explanation capabilities and expand them to any line in a given program. The resulting constraints may be used to help explain how to reach a code line using a specification of inputs. Due to the expansion of Avicenna's oracle construction, we can show how lightweight this expansion approach

is, and how modifiable Avicenna truly can be.
The source code for Avicenna and AviX may be accessed in our git repository:
https://github.com/Angeloony/avicenna/tree/dev/src/avicenna

Implementation Summary
AviX's implementation is introduced in the above section.

We give a rundown of the differences to Avicenna, emphasizing the line-oracle and the impact it has on the pattern learner during the feedback loop. We also present what kinds of inputs are given to AviX in order to run it, and explain how the oracle works.

5 EXPERIMENTAL SETUP

This section goes over our research questions and goals, as well as the structure of our experiments.

Brief Rundown: For the experiment, we will collect data on four subjects, which contain our programs-under-test, fitting context-free grammars, inputs for each run, and the lines we want to analyze. Because sometimes Avicenna will find different invariants for the same subject, each subject is run 10 times, so that we can get a better idea of how much variance there is in the analysis results of a given program. Afterwards, we will evaluate AviX as a producer and a predictor and compare this performance to Avicenna's initial precision and recall measurements.

5.1 Research Questions

We ask the following Research Questions and answer them in the upcoming section.

1. **RQ1: Is AviX feasible as an extension of Avicenna?**
   Here, we determine whether AviX is usable for lines and if it is feasibly and sensibly adjustable for more improvements to be implemented in the future.
   We compare our average precision and recall scores to the average precision (88%) and average recall (91%) of Avicenna.
2. **RQ2: How good are AviX's constraints for producing additional inputs?**
   We are going to use the constraint of each subject and line that was found to fuzz more inputs, and see if these inputs also trigger that line.
3. **RQ3: Are the found constraints valuable predictors of input behavior?**
   This will be determined by randomly fuzzing inputs, checking if they trigger a line, and then using a constraint to predict whether a given input triggers it.

5.2 Detailed Setup

We chose a quantitative research approach to evaluating AviX's performance, as we believe that it is statistics that let us determine if a tool is worth using, especially

when debugging code. Avicenna is a light-weight tool that allows for easy installation, but expanding its use to testing line reachability will only matter if it maintains the strong performance of the original Avicenna feedback loop. Therefore, we will evaluate the runtime for determining a constraint, as well as the use of these constraints as a Producer and a Predictor for other inputs. We will go into the exact methodology with more detail in the following paragraphs.
As this paper serves in big parts as a feasibility study, we have decided to pick four test subjects that were familiar to us. Each subject was analyzed, and at least one line per program-branch was analyzed in an AviX run. This allowed us to cover the entire program with our line-trigger analyses. For each line, the explain algorithm containing the feedback loop was run 10 times in order to test for variance in AviX's performance in runtime and constraint quality.
We structured our subject tests by creating a class which contained their belonging grammar, lines under test, input converters – if needed for running the instrumented programs – and initial inputs to kick off the feedback loop.
For each line, we ran the explain algorithm 10 times. This allowed us to test for possible variance in the findings of this algorithm for a given line, determining AviX's variance in runtime and constraint quality as we go. For each run, we gave Avicenna the following values:

* **Minimum precision: 0.6**
  This is the minimum value Avicenna tries to optimize for in its feedback loop.
* **Minimum recall: 0.9**
  Just like minimum precision, but for recall.
* **Feedback loop iterations: 10**
  Each line had 10 refinement runs in the loop in order to maximize the constraint quality.
* **Top relevant features: 3**
  The maximum amount of relevant features used when mining for new constraints. This opens up extra detailed constraints, but may also increase runtime.

We have chosen the above values to find a good middle-ground between quality, and also runtimes that are in a reasonable range.
For the experiments themselves, we checked one line at a time. Starting by constructing a line-oracle, then going into Avicenna's explain algorithm right away. The runtimes, mined constraints, and Avicenna's precision and recall scores were saved in csv-files, which can be accessed in our git repository.
After mining the constraints, we fuzzed our 100,000 inputs for our predictor tests, removed duplicate inputs, and classified them with all unique constraints that were found.
At this point, all relevant parts have been collected, and we proceed to test our constraints as predictors and producers, using the parse and solve functions provided by `ISLaParser`, which were described above.
In total, we analyzed 30 different lines, leading to 300 runs of the Avicenna's `explain()` algorithm. Most lines had only equivalent constraints describing them,

Algorithm	Analyzed	Total Lines	What does it do?
`middle`	8	13	Returns middle value of 3 values, x, y, and z.
`markup`	4	16	Markup tag parser. Returns string without markup tags.
`calculator`	6	32	Evaluates mathematical terms, notably sqrt, sin, cos, tan functions.
`expression`	12	111	Mathematical expression parser. Returns the result of a given mathematical expression.
Total	**30 | 17%**	**172**	

Table 1: Overview of our test subjects and their sizes.

meaning that we would only use one constraint to predict or produce inputs for our experiments. Where applicable, we would use differing constraints in multiple runs of producing and predicting, as well as take note of missing constraints.
For each run, we have collected:

1. Avicenna's outputs, which include:
   - The mined constraint.
   - Precision and Recall scores that are used to determine the returned constraint.
2. The explain algorithm's runtime.

These data points will allow us to determine how reliably AviX can find constraints for lines, and how well they perform in our later Producer and Predictor tests.

5.3 Introducing our Subjects

For each subject, we investigated each line in our early test runs. This allowed us to very easily hone in on the relevant lines that would return interesting results. We proceeded to remove lines that would always get triggered, line that never get triggered, and lines that exist in the same block, which were likely to return duplicate constraints.
Notably, our subjects include rather small programs that are all contained in a single file. Our subjects are summarized in Figure 1.

5.4 Producer Tests

For our producer tests, we will use the mined constraints – if one was found – to fuzz 1,000 inputs per unique mined constraint. Since over 10 runs some constraints are more likely to be found, we will only use one of those for fuzzing.
We will use ISLa's provided semantic fuzzer, which is used by calling the `ISLaSolver's` solve function. The `ISLaSolver` uses our input grammar and our mined constraint in order to solve the given constraint with inputs derived from our grammar.
We decided to generate 1,000 inputs using the semantic fuzzer, because of the following:

1. Semantic fuzzing takes longer and comes with a higher computational cost than other fuzzing strategies.

2. The inputs are fuzzed more precisely than with random fuzzers. Enabling us to showcase the performance of the fuzzed inputs more clearly in smaller numbers.
3. Due to the high grade of similarity in the fuzzed inputs, more than 50% of all inputs are removed when accounting for duplicates. The need for fuzzing thousands more inputs is greatly diminished by the common repetitions.

After fuzzing, we check each input using a line-oracle made for the line that is being tested. Inputs that trigger the line are determined to be true positives, inputs that do not trigger the line are considered false positives. We determine the constraint's input generation accuracy score using these classifications. The accuracy represents the ratio of line-triggering to non-line-triggering inputs that were generated.

5.5 Predictor Tests

Avicenna's constraints can also be used to predict program behavior for the input in question.
To test this, we randomly generated 100,000 inputs, using the context-free grammars that were used for running AviX in the first place. All duplicate inputs are culled, and then the remaining inputs are used to establish the ground truth for what line is triggered by what inputs. We do this for every line and every test subject.
Next, we parse all randomly fuzzed inputs with our constraint and the input grammar. `ISLaParser` provides us with the parse function, which checks if an input is valid under the given constraint and context-free grammar. parse returns an error if the input violates either the grammar (`SyntaxError`) or the constraint (`SemanticError`). This means that inputs who are parsed successfully fulfill the constraint, and inputs who fail parsing violate it, causing a SemanticError. Since the inputs are originally fuzzed using the grammars, they cannot fail the syntax check.

* **True Positives** are inputs that were parsed successfully and if the input in question is also line-triggering, meaning the constraint successfully predicted the program behavior.
* An input that does not trigger a line, but is not caught by the constraint, is a **False Positive**. The prediction was wrong.
* A **True Negative** is an input that fails the parsing step, meaning it violates the given constraint and also does not trigger the line. The prediction was successful in this case.
* A prediction fails in the case of an input violating a constraint, even though it triggers the line in question, causing a **False Negative**.

Using these input classifications, we will determine the tested precision and recall scores of our constraints and compare them directly to the scores given by Avicenna during its feedback loop. This will allow us to not just determine the quality of the mined constraints as predictors of program behavior, but also verify or dispute Avicenna's internal scoring system.

Experiment Summary
In this section, we go over our experimental process in granular detail.

We introduce our test subjects, break down our research questions, explain our producer and predictor tests, disclose our constraint mining runs, and give an overview of the general scale of our collected data.
Importantly, we focus on creating representative data for our chosen subjects in order to review AviX's expected performance effectively moving forward.

6 EVALUATION AND DISCUSSION

Now, let us finally dive deep into the results of our research!
First, we will offer a general overview of our collected data for all four subjects. Then, a deeper analysis on each subject's performance will be conducted separately. We will present our data for testing the prediction and input generation capabilities of the mined constraints, as well as other data that we will introduce in each relevant step. During this analysis, outliers will be investigated more closely by showing the outlier constraints and giving brief explanations on why they may have been found whenever possible.
At the end of each subject's section, we give a general performance approximation to prime us for the answering of the research questions in the last step of this section.

General Information on our Data: Precision, Recall, and Accuracy stats were rounded to the second decimal spot. Runtimes were rounded to full seconds. Predictor tables contain a column that denotes how many constraints found in a given line were logically equivalent to other constraints found for that line, denoted in the 'Equivalent' column. Weighted averages and medians are given in the last rows of our data tables. We weighted every found constraint equally for producer data when determining the averages and median. For predictor data, we used weighted means, following this formula:

$$\frac{\sum_{i=1}^{n}(\omega_i \cdot \mu_i)}{n}$$

ω_i is the weight for line i, determined by the amount of equivalent constraints that were found for line i, and μ_i being the mean score that is used in our tables. Maximum weight is 1, minimum weight is 0.1.
Our data includes csv-files containing the constraints, runtimes, and Avicenna scores for all runs of the explain-algorithm for each line, as well as multiple text files, containing our grammar-fuzzed inputs, and our prediction stats for each line. Some lines have multiple predictor and producer results, because we may have found more than one constraint for them within the 10 test runs.

All of our raw data can be found in our git repository under the following link:
https://github.com/Angeloony/avicenna/tree/dev/src/avicenna/experiment

6.1 General Observations

The discussed figures can be found on page 19.

Precision scores are shown in Figure 5 were mostly very high, with only Markup being a clear outlier. Except for Markup, every subject's median precision score was at 1.0, equalling 100%. Markup's median precision was at 66.5%. Outliers in the graph are depicted by stars.
When analyzing each subject separately, we will give an in-depth explanation for Markup's relatively weak performance.
Recall scores looked much the same as the Precision scores, reaching 100% in even more instances than the collected precision scores. We omitted plots, as 28 of 30 investigated lines had at least one constraint with a recall score of 100%, leaving barely any variance to be portrayed at all.
Runtimes were mostly similar, with all subjects' medians being around 250 seconds, which is depicted in Figure 6. Calculator was even faster than that, with a median run time of 29 seconds.
The only outliers to note here are Markup and Expression. While their median runtimes are mostly in line with our other subjects, Markup and Expression had outliers within their runtimes that reached very long runtimes. Especially Expression contained around 10 to 20 outliers which took 2,000 to 4,500 seconds. That is around 40 to 70 minutes *per attempt*.
For runtimes, these outlier runs showed the greatest weakness, although these outliers often occurred within the same line analyses, leading us to believe that they had something to do with the attempted explanations for those lines in particular. Especially runs that did not find any constraints seemed to run for much longer, likely due to Avicenna trying to refine very weak constraints in each step of its loop.

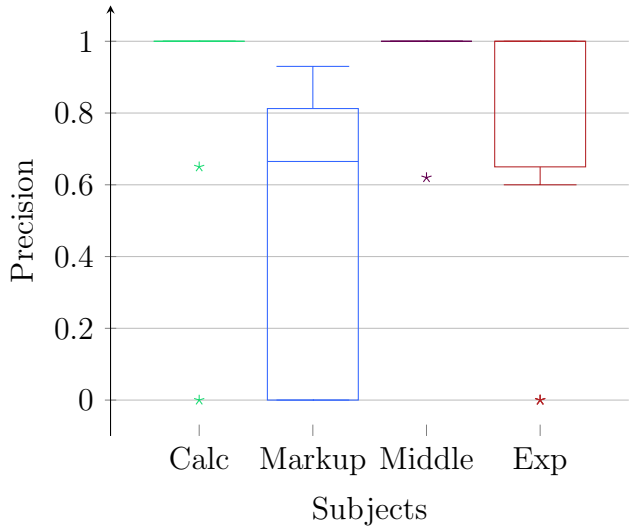

Figure 5: Precision scores for all subjects that were found by Avicenna. Middle and Calculator performed very strongly, meaning they only have few outliers outside of 100% precision rating.

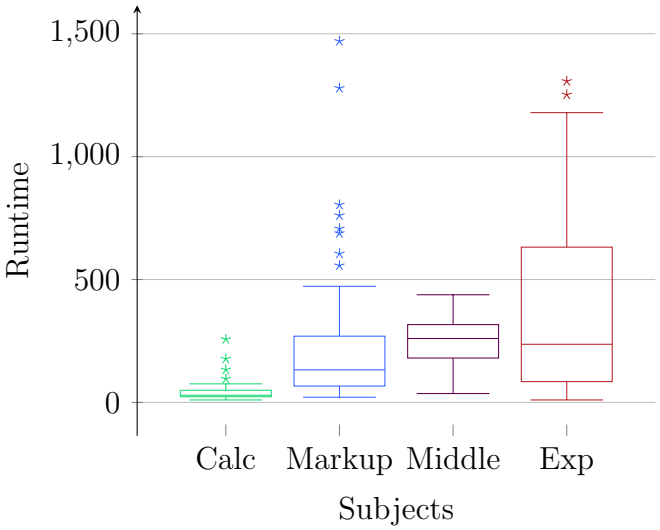

Figure 6: Note: `expression`'s outliers over 1,500 seconds are omitted, as they were too big to portray reasonably within the same graph. We counted 16 runs that were between 1,700 seconds and 4,500 seconds.

6.2 Middle

For Middle, AviX performed very uniformly, and very well at that. In Table 2 and Table 3, we can see that only the second row for line 7 has underperformed. Other than that, a viable constraint was found for each of the eight lines we analyzed.

		Test Medians		Avicenna Median			
Subject	**Line**	**Precision**	**Recall**	**Avi-Precision**	**Avi-Recall**	**Runtime**	**Equivalent**
middle	**4**	**1.00**	**1.00**	**1.00**	**1.00**	110 s	10 / 10
	5	**1.00**	**1.00**	**1.00**	**1.00**	311 s	10 / 10
	6	**1.00**	**1.00**	**1.00**	**1.00**	244 s	10 / 10
	7	**1.00**	**1.00**	**1.00**	**1.00**	256 s	9 / 10
		0.40	0.93	0.62	0.94	65 s	1 / 10
	9	**1.00**	**1.00**	**1.00**	**1.00**	59 s	10 / 10
	10	**1.00**	**1.00**	**1.00**	**1.00**	300 s	10 / 10
	11	**1.00**	**1.00**	**1.00**	**1.00**	322 s	10 / 10
	12	**1.00**	**1.00**	**1.00**	**1.00**	316 s	10 / 10
Avg. Scores		**99.3%**	**100%**	**99.5%**	**99.9%**	**230 s**	
Median Scores		**100%**	**100%**	**100%**	**100%**	**256 s**	

Table 2: Predictor results for middle.

		Producer Stats (Average)			Grammar Fuzzer	
Subject	**Line**	**Accuracy**	**Line-Triggering**	**Unique out of 1,000**	**Accuracy**	**Trigger out of 1,000**
middle	**4**	**1.00**	**567**	567	0.47	470
	5	**1.00**	**34**	34	0.15	152
	6	**1.00**	**49**	49	0.32	318
	7	**1.00**	**44**	44	0.17	171
		0.19	37	195	0.17	171
	9	**1.00**	**585**	585	0.52	522
	10	**1.00**	**53**	53	0.18	181
	11	**1.00**	**90**	90	0.34	341
	12	**1.00**	**34**	34	0.15	154
Avg. Scores		**99%**	**182**	**184**	**28.8%**	**289**
Median Scores		**100%**	**53**	**53**	**25%**	**250**

Table 3: Producer results for middle.

Outliers: Interestingly, only one mined constraint out of 80 total runs performed under par. While this is negligible in the grand scheme of our Middle test runs, we will investigate its performance a little more closely.
The weak constraint is shown in Listing 3. Viewing the source code in Listing 2, we can see that the first half of the constraint correctly describes the branch condition at

line 3, `z > y`. The failure of this constraint lies in its failure to relate the comparison of variables `x` and `z` in line 6. Instead, it inexplicably compares `y` and `int 48`, suggesting a special relation of that integer to the reachability of the line. This is wrong, however.
We assume that this happens due to some local maximum that may have been found by Avicenna's pattern matching approach early on in the run.
The correct constraint was found in the nine other runs of line 7 and should look like Listing 4.

Listing 2: `middle` source code until line 7.

```
def middle(x, y, z):
    m = z
    if y < z:
        if x < y:
            m = y
        elif x < z:
            m = x   # Target line
```

Listing 3: `middle`, Line 7 constraint outlier.

```
# Weak constraint
(forall <z> elem_1 in start:
   exists <y> elem_2 in start:
     (> (str.to.int elem_1) (str.to.int elem_2)) and
forall <y> elem in start:
  (<= (str.to.int elem) (str.to.int '48')))
```

Listing 4: `middle`, Line 7 constraint equivalent to nine out of 10 constraints mined.

```
# Strong constraint
(forall <z> elem_1 in start:
   exists <x> elem_2 in start:
     (> (str.to.int elem_1) (str.to.int elem_2)) and
forall <x> elem_0 in start:
  exists <y> elem_3 in start:
    (>= (str.to.int elem_0) (str.to.int elem_3)))
```

Predictor Test: A total of *81,716 unique inputs* were fuzzed with Middle's grammar. Table 2 shows us the very strong performance of AviX for Middle, with median and average scores for precision and recall scores across the board at 100%.
The predictors performed very strongly in our tests, returning an average precision of 99.3% and an average recall of 100%. The outlier performed slightly worse than was expected by Avicenna's given scores, but since it was only one run out of 80 that returned a faulty constraint, it is negligible.
Average scores are not 100% across the board, solely due to the outlier constraint in line 7.

Producer Tests: When using ISLa's semantic fuzzer with all Middle constraints, we have a perfect line-trigger rate, other than the outlier in line 7.

The most interesting find here is that random grammar fuzzing can generate more inputs in quick succession, leading to more hits for the given lines. The main benefit from semantically fuzzing comes from the certainty of the constraint structure, as we can be sure that the fuzzed constraints belong to our triggering input class.
The more complex constraints, the ones that determine the relationship of all three variables, have far fewer fuzzed inputs than the more simple constraints with fewer variables. This may be a weakness of the ISLa fuzzer, but is important to note nonetheless.

6.3 Markup

AviX's performance is a lot weaker when regarding Markup's results. Only two out of four lines reliably found constraints, and unlike Middle's constraints, they did not perform perfectly.
As Table 4 shows, lines 12 and 14 consistently failed at determining a fitting constraint, with only one result existing for line 14, and none for line 12.
Still, for 3 out of 4 investigated lines, a viable result was found by AviX.

		Test Medians		Avicenna Median			
Subject	**Line**	**Precision**	**Recall**	**Avi-Precision**	**Avi-Recall**	**Runtime**	**Equivalent**
markup	**8**	0.99	**1.00**	0.87	**1.00**	59 s	10 / 10
	10	0.98	**1.00**	0.77	**1.00**	60 s	10 / 10
	12	/	/	/	/	180 s	10 / 10
	14	0.90	**1.00**	**0.64**	**1.00**	286 s	1 / 10
		/	/	/	/	707 s	9 / 10
Avg. no error		**95.7%**	**100%**	76%	**100%**	**135 s**	
Median Scores		**98%**	**100%**	**77%**	**100%**	**180 s**	

Table 4: Predictor results for markup.

		Producer Stats (Average)			Grammar Fuzzer	
Subject	**Line**	**Accuracy**	**Line-Triggering**	**Unique out of 1,000**	**Accuracy**	**Trigger out of 1,000**
markup	**8**	**1.00**	**200**	200	0.39	389
	10	0.99	197	200	0.39	387
	12	/	/	/	0.02	19
	14	**1.00**	**96**	96	0.45	453
		/	/	/	0.45	453
Avg. no error		**99.7%**	**164**	**165**	**31.3%**	**312**
Median Scores		**100%**	**197**	**200**	**39%**	**388**

Table 5: Producer results for markup. Note that missing constraints have no input generation results.

Outliers:
When investigating the missing constraints, we looked at the source code of lines 12 and 14. We noticed that it relied on the understanding that at a previous point in time, an HTML tag was opened with a `'<'` character. The context of an opened HTML tag could not be found out by Avicenna, which ultimately comes down to a missing pattern in the pre-defined pattern catalog that could be added later.
Keep in mind that adding more patterns to the catalog may always result in increased runtime for all other runs of AviX.
Line 14 is triggered when the parser is currently not parsing the contents of an HTML tag. Due to missing patterns describing a tag context, Avicenna could only describe this by determining that character strings without any sort of HTML tag are valid. While only half correct, this is the best Avicenna could perform under its given circumstances. This result in particular presents Avicenna's ability to find constraints even for circumstances that are difficult for it to properly capture with its given pattern catalog.

Predictor Tests: A total of *33,327 unique inputs* were fuzzed using Markup's grammar.
Weirdly, the constraints performed much better for our predictor assessments than they did during Avicenna's run. The reason for the increase in precision for all of these constraints is likely that the vast majority of inputs trigger the line in question. The generality of the constraint therefore extends to the generality of the code line, making it mostly precise again. This can be seen more clearly when looking at line 14's misclassification of around 3,400 inputs, even though it is still 90% precise.
With the tools given to AviX, it performed as well as it could have, leaving us with numerically strong results for input behavior predictions. Markup is still the worst performing subject overall, as it found the least constraints overall and none of its constraints received perfect scores for prediction.

Producer Tests: For semantic fuzzing, these constraints produce even better scores, but that does not necessarily mean that these constraints are simply better for producing new inputs once again. Because the existence of tags is not currently acknowledged by Avicenna, the produced inputs only produce a subsection of all inputs relevant to triggering this line.
This is fatal when trying to use AviX for reinforcing test suites, as entire input subclasses are left out, which could potentially lead to a false sense of security when testing specific lines in a given test suite.
Even constraints deemed imprecise by Avicenna maintain high accuracy when used for producing new inputs in the bounds of their constraints. For this reason, they may still prove useful for fuzzing precisely.
Again, random grammar fuzzing found more inputs to trigger the given lines, but again, they are uncategorized, and the outcome is not known before running a program with these inputs.

6.4 Calculator

Calculator's results are very good for the most part, but showcase some variance in quality for lines 8 and 15. AviX found a constraint for all six lines, with five of the six lines offering perfect constraints.

		Test Medians		Avicenna Median			
Subject	**Line**	**Precision**	**Recall**	**Avi-Precision**	**Avi-Recall**	**Runtime**	**Equivalent**
calculator	8	0.40	0.67	0.65	**1.00**	12 s	1 / 10
		0.50	**1.00**	**1.00**	**1.00**	17 s	9 / 10
	9	**1.00**	**1.00**	**1.00**	**1.00**	51 s	10 / 10
	15	**1.00**	**1.00**	**1.00**	**1.00**	164 s	2 / 10
		0.50	**1.00**	**1.00**	**1.00**	134 s	7 / 10
		/	/	/	/	25 s	1 / 10
	20	**1.00**	**1.00**	**1.00**	**1.00**	27 s	10 / 10
	24	**1.00**	**1.00**	**1.00**	**1.00**	28 s	10 / 10
	28	**1.00**	**1.00**	**1.00**	**1.00**	27 s	10 / 10
Avg. w/ error		**84%**	**97.8%**	**97.8%**	**100%**	**46 s**	
Median Scores		**100%**	**100%**	**100%**	**100%**	**27 s**	

Table 6: Predictor results for calculator.

		Producer Stats (Average)			Grammar Fuzzer	
Subject	**Line**	**Accuracy**	**Line-Triggering**	**Unique out of 1,000**	**Accuracy**	**Trigger out of 1,000**
calculator	8	0.25	1	4	0.07	66
		0.51	48	94	0.07	66
	9	**1.00**	**1**	1	/	1
	15	**1.00**	**9**	9	0.07	65
		0.56	89	157	0.07	65
		/	/	/	0.07	65
	20	**1.00**	**399**	399	0.10	104
	24	**1.00**	**247**	247	0.10	106
	28	**1.00**	**96**	96	0.12	122
Weighted Avg.		**79%**	**111**	**126**	**9%**	**85**
Median Scores		**100%**	**69**	**95**	**10%**	**77**

Table 7: Producer Results calculator.

Outliers: The only outliers are found in lines 8 and 15.

* Line 8, constraints found in Listing 5, along with the belonging code snippet Listing 6.
  1. Similarly to Middle's outlier for line 7, this constraint compares set integer values. It is practically useless.

2. This constraint describes the general existence of `sqrt` in the input string, without any relation given to the value that the function would assess. This should be the right constraint for this line, as negative values and 0 are treated after line 8. Therefore, we are not sure as to why only the positive inputs are deemed triggering by our oracle.

* Line 15
  1. The first part of this constraint describes that the value of the number in the expression must be greater than or equal to 1. The second part requires `'sqrt'` to be fulfilled. That is the exact requirement for the line.
  2. Only requires `'sqrt'`, falsely allowing for negative numbers to be entered.
  3. No constraint was found.

Listing 5: `calculator`, Line 8 Constraints

```
# Constraint 1
(forall <digits> elem in start:
    (<= (str.to.int elem) (str.to.int '89')) and
forall <number> elem_0 in start:
    (>= (str.to.int elem_0) (str.to.int '0')))

# Constraint 2
exists <function> elem in start:
    (= elem 'sqrt')
```

Listing 6: `calculator` source code snippet for line 8.

```
def sqrt(x):

    if x == 0:       # Allows sqrt(x) with x < 0.
        return 0
    assert x > 0     # Throws error for sqrt(x) with x < 0.
```

Predictor Test: A total of *24,550 unique inputs* were fuzzed with Calculator's grammar.
Table 6 shows that except for the outliers in line 8 and 15, every input predicted the trigger-conditions for its lines perfectly.
The second row for line 8 should have a higher precision, but in our tests, functions of the shape `sqrt( X )`, with `X` being a negative number, did not count as line-triggering for line 8, even though that should have been the case. While running the explain algorithm for these lines, we presume that these inputs were counted as line-triggering, as intended. We are not sure what the cause of this is, but will treat this constraint as though it performed perfectly.

Producer Tests: Producer results are displayed in Table 7. Once again, out of six investigated lines, the two outliers are the only ones that do not perform flawlessly throughout the code generation tests.
As mentioned before, line 8 should be counted as a perfect constraint for row number

two.
Line 9 only has one produced input, because it is triggered by exactly one kind of input, `'sqrt(0)'`. This can be seen in our the calculator code snippet we have provided for the `sqrt(x)` function. This line can be seen in the previously introduced code snippet, Listing 6.
Lastly, line 15's first constraint ends up working poorly for fuzzing, as it only fuzzes 9 different inputs using the ISLa fuzzer, this appears to be a weakness belonging to ISLa, however, as the predictions of this same constraint work perfectly. Its second constraint is the same as the previously mentioned line 8 constraint, except negative values may not trigger this code line, hence its performance of roughly 50% triggering inputs that contain positive values.
Once again, these constraints work rather well for predicting and fuzzing. Some confusion surrounding lines 8 and 15 for predicting and fuzzing would require further research to clearly pin down. Calculator is also the only subject that very consistently produced better inputs than the random grammar. For this reason, we believe that the overall structure and complexity of the program code and the provided context-free grammar greatly influence the quality of the fuzzing. When dealing with a weaker grammar, using semantic constraints could likely improve the quality of fuzzed inputs greatly.
We deem Calculator an overall success.

6.5 Expression

Expression's results are mostly stable, but showcase some variance in line 105. AviX found a constraint for each of the 12 tested lines. Eight lines have been described with flawlessly performing constraints. The other four lines have been provided well-performing constraints, as will be shown.

Outliers and other Peculiarities:

* Line 59 is triggered when a double whitespace is found in the input string. These are contained in the string after introducing an extra whitespace around parentheses a couple lines prior. The code can be seen in Listing 7.
  The cause of this line trigger definitely come from the existence of parentheses, but their positioning is crucial as well. They need to come either before or after an operator or another parenthesis.
  The pattern catalogs do not contain patterns describing a context like this, similarly to the issue faced when trying to determine the tag context for markup.
  This constraint is as good as Avicenna could do.
* Line 87:
  1. Listing 8 shows the constraint, which describes the requirement for each derived digit to be greater than at least one entire number. Mostly inputs that contain a standalone '0' are deemed valid by this constraint, because digits may not be derived to '0' in this grammar.

2. No constraint was found in most tries.

* Line 93: Line 93 parses our token string if either a `Mul *` or `Div /` operator is contained. The constraint – shown in Listing 9 – recognizes that the operator derivation is key, but it does not match the exact multiplication and division operators in line-triggering inputs.

* Line 105 is analogous to line 93 in that it checks for `Add +` or `Sub -` operators in the given string. The first constraint is the same as in 93, describing the existence of the operator derivation. The other constraints for line 105 add more restrictions that only make them less precise.

Listing 7: `expression` source code snippet for line 59.

```
55 def parse(s: str):
56     s = s.replace('(', ' ( ')
57     s = s.replace(')', ' ) ')
58     while '  ' in s:
59         s = s.replace('  ', ' ')    # Target line
```

Listing 8: `expression`, Line 87

```
1 forall <digit> elem_1 in start:
2   exists <number> elem_2 in start:
3     (>= (str.to.int elem_1) (str.to.int elem_2))
```

Listing 9: `expression`, Line 93

```
1 exists <operator> elem_xy in start:
2     inside(elem_xy, start)
```

Predictor Tests: A total of *51,150 unique inputs* were fuzzed with Expression's grammar.
Shown in Table 8, when predicting input behavior, AviX's performance is on par with or better than the expected precision scores measured by Avicenna's loop. Just one outlier constraint exists for line 105.
This strong performance may be caused due to these very general constraints that only roughly describe a commonly grammar-fuzzed input class. We believe that the strong performance by the outlier constraints named in the previous section is due to a preference in fuzzing a specific type of input displayed by the grammar-fuzzer.
For line 87, its high performance seems to be mostly chance, since its constraint mostly exploits the frequent occurrence of single digit zeroes in this grammar. About half of all non-triggering inputs are predicted to be line-triggering by this constraint, owing its strong performance only to the amount of line-triggering inputs vastly outnumbering non-triggering inputs.
The weaker constraints found for line 105 seem to be caused by local maxima that Avicenna's learning loop may get stuck in. Evidence that speaks for this theory are the exorbitantly long runtimes that can be found in weaker constraints, as well as runs that returned no constraints at all. Due to these local maxima – that perform

Predictor Stats for constraints mined by AviX.

Subject	Line	Test Medians		Avicenna Medians		Runtime	Equivalent
		Precision	Recall	AviX-Precision	AviX-Recall		
expression	10	**1.00**	**1.00**	**1.00**	**1.00**	516 s	10 / 10
	40	**1.00**	**1.00**	**1.00**	**1.00**	40 s	10 / 10
	59	0.93	**1.00**	0.67	**1.00**	385 s	10 / 10
	73	**1.00**	**1.00**	**1.00**	**1.00**	155 s	10 / 10
	85	**1.00**	**1.00**	**1.00**	**1.00**	123 s	10 / 10
	87	0.99	0.86	0.67	0.91	78 s	1 / 10
		/	/	/	/	2,661 s	9 / 10
	93	0.84	**1.00**	0.63	**1.00**	146 s	10 / 10
	95	**1.00**	**1.00**	**1.00**	**1.00**	266 s	10 / 10
	97	**1.00**	**1.00**	**1.00**	**1.00**	183 s	10 / 10
	105	0.84	**1.00**	0.61	**1.00**	26 s	4 / 10
		0.85	0.96	0.62	0.96	815 s	1 / 10
		0.80	0.71	0.63	0.93	1,430 s	2 / 10
		/	/	/	/	349 s	3 / 10
	107	**1.00**	**1.00**	**1.00**	**1.00**	209 s	10 / 10
	109	**1.00**	**1.00**	**1.00**	**1.00**	255 s	10 / 10
Avg. Scores		**87.1%**	**89.4%**	**76.4%**	**89.8%**	**430 s**	
Median Scores		**100%**	**100%**	**100%**	**100%**	**209 s**	

Table 8: Predictor results for `expression`.

Subject	Line	Producer Stats (Average)			Grammar Fuzzer	
		Accuracy	Line-Triggering	Unique out of 1,000	Accuracy	Trigger out of 1,000
expression	10	**1.00**	**86**	86	0.48	481
	40	**1.00**	**88**	88	0.30	300
	59	0.84	118	140	0.43	428
	73	**1.00**	**144**	144	0.49	487
	85	**1.00**	**86**	86	0.30	300
	87	**1.00**	**86**	86	0.61	609
	93	0.89	177	199	0.40	404
	95	**1.00**	**193**	193	0.29	288
	97	**1.00**	**196**	196	0.29	289
	105	0.88	174	198	0.39	388
		/	/	/	0.39	388
		0.62	122	197	0.39	388
	107	**1.00**	**194**	194	0.29	285
	109	**1.00**	**196**	196	0.29	284
Avg. Scores		**94.1%**	**143**	**147**	**38%**	**379**
Median Scores		**100%**	**122**	**144**	**35%**	**344**

Table 9: Producer Results `expression`.

on par with the pre-defined precision and recall scores – Avicenna's learner likely tries to optimize the performance within this local maximum, without attempting to expand the constraint or restart from scratch. This is a clear weakness, but one that could be ameliorated with adjustments to the learner as well as outside the learner, by defining timeouts that determine restarting times for constraints or entire runs.

Producer Tests: The producer results approximately mimic the predictor results' precision scores. Table 9 tells us that the general performance for input generation is very good, but random grammar fuzzing provides more triggering inputs on average, though with a much lower accuracy. For Expression, more line-triggering inputs are found consistently by the grammar fuzzer. We believe this is due to the very intertwined nature of the program code and a grammar that is not very specialized. Note that the second constraint class for line 105 failed in producing new inputs, because of a syntax error occurring in the `solve` function provided by the ISLaParser.

6.6 Summarizing our Findings

Important realizations we have made throughout this evaluation process are the following:

1. The pattern catalog is powerful even with a limited number of patterns as long as they are chosen carefully.
2. Some patterns are currently missing from the catalog, and adding the ability to negate certain patterns might help even further. However, this may come at huge runtime costs that will have to be investigated further.
3. Avicenna often finds the best constraints with what patterns it has available, although sometimes it appears to get stuck in local maxima, leading to outliers with a weaker performance.
4. AviX's constraints perform on par or better than Avicenna's measured scores for the same inputs when tested using our own test inputs. These inputs are grammar fuzzed using the grammars we have used for the Avicenna `explain()` runs.
5. While the ISLa fuzzer allows for the production of inputs that always fulfill the constraint, it only produces a very limited amount of unique inputs in its current state. When grammar fuzzing, the sheer volume of produced inputs may be more useful for most cases, though more work must be put into categorizing whether they are line-triggering or not in the first place.
6. Avicenna, and therefore AviX, lack a control mechanism for ending runs that are taking too long (i.e. longer than 10 minutes) in its current iteration in version 0.9.1.

Most of these points should be easily ameliorated, with only the second point somewhat being a cause for concern, due to potential runtime costs and worsening overall performance of Avicenna because of it.
Now, we will answer the research questions asked in the previous section.

RQ1 Does AviX perform as well as Avicenna as an extension of it? Clearly, as shown with our subjects, AviX works in its essence. The overall efficacy of AviX's diagnoses, powered by Avicenna's feedback loop, are outlined in Figure 5 and Figure 6, as well as fleshed out in our Tables above.
AviX maintains Avicenna's high ratings of around 90% for precision and recall across these four test subjects.
Finally, further extending AviX by allowing for different event-types to be investigated, more flexibility could be offered in these kinds of program analyses. Possible events include function calls or taking different program branches.

RQ2 Predicting input behavior using AviX constraints: As for predictions made by our mined constraints, we can confidently say that high scores by Avicenna are oftentimes accurately describing the precision and recall for predictions made on randomly grammar-fuzzed inputs. We only had two outliers that underperformed when compared to Avicenna's initial scoring, one of which was Calculator line 8, which should have maintained a 100% precision and recall rating to the best of our knowledge.
Constraints with lower initial scores are still often on par or even better than expected at predicting input behavior.
Overall, AviX's predictions are 100% at their medians across the board, and average 91.5% precision, and 97.5% recall.
In summary, the use of AviX constraints for predicting input behavior should prove highly effective, because of the high precision and recall ratings used as a baseline for the refinement loop.

RQ3 Producing inputs with AviX's constraints: AviX constraints work well for semantic fuzzing. Offering a precise fuzzing approach by combining semantic constraints with context-free grammars.
The greatest weakness in semantic fuzzing currently seems to be the semantic fuzzer we used. The ISLa-fuzzer often repeats fuzzed inputs, putting out a lot more duplicates than is desirable. **Overall accuracy for semantic fuzzing was greater than 90%**. In comparison, the random grammar fuzzer we used produced more unique line-triggering inputs on average, but did so with a low accuracy of around 28%.
Still, semantic fuzzing produced around 50 to 100 unique inputs that perfectly align with the exact inputs needed to trigger the given lines. Highly specific line-trigger requirements may also be reproduced, as seen for Calculator in line 9.
AviX is therefore an effective, albeit not perfectly efficient, producer.

Discussion Summary

AviX is a viable extension of Avicenna. AviX has relatively low runtimes, with medians ranging around less than 5 minutes, as seen in Figure 6. All the while, it maintains Avicenna's strong performance of around 90% precision and recall for found constraints.

The line-constraints it provides can also effectively be used for **predicting input-behavior**, meaning it can predict if an input will trigger a line. The precision of these predictions average a **precision rating of more than 85%, and a recall rating of 90%**, with many constraints performing even better than that as seen in Figure 5.

Producing inputs is made possible by using ISLa's semantic fuzzer, which combines the context-free grammar given for each subject and the constraints that were mined for a given line. This allows for highly precise fuzzing, averaging around **90% accuracy in triggering the target code line** with these inputs. The downside of semantic input generation in its current state is the relatively small number of unique inputs that are generated.

All relevant tables and figures can be found in the section above.

7 THREATS TO VALIDITY

Some issues we have encountered during our experiments still remain.
Despite its generally strong performance on our subjects, AviX does not have a 100% success rate. The general idea would be to run AviX once per line in order to mine decent constraints in order to find the best one. As we have shown, some constraints may have vast differences in their quality, but we currently do not offer a solution to reliably maximizing the quality in *every* run. Possible solutions could include re-running if certain thresholds are not met by the resulting constraint, or allowing for extra iterations of the feedback-loop if the runtime is not too long yet.
We have currently only implemented checks for line-events. Due to this limitation, line-bound analysis may push users into more manual code analysis, which we wanted to avoid in the first place. By adding other events for analysis, this could be ameliorated.

7.1 Internal Threats to Validity:

ISLa's provided parser and fuzzer did have some limitations that we came across.
The fuzzer sometimes had much fewer unique inputs than would be desirable, suggesting a tendency of not exploring many different derivation paths within the grammar, depending on the constraint. While in its essence, the fuzzer is functional – as we have shown in our experiments as well – it could work *much* better, and therefore strengthen the goal of using Avicenna-constraints for input-generation.
The parser provided by ISLa that allowed us to test the constraint's predictor qualities worked quite well overall, but did get stuck on a few inputs. We have only removed these inputs instead of looking into what about them made the parsing process crash. Our assumption is that this happened due to the structure of the Expression grammar that made the parsers parse-tree traversal very costly, but we cannot make a conclusive statement on that.

7.2 External Threats to Validity

We used very few subjects for our research. It is hard to generalize the results of four subjects to a wider group of programs, but for our feasibility study, this was an important first step to take.
The 100,000 fuzzed inputs used for our predictor experiments were not balanced to contain the same number of line-triggering and non-triggering inputs. This has skewed the predictions of some constraints to appear far more positive than they may actually be. To gain more insights, more tests could be run using balanced input sets, that are split 50/50 between triggering lines and not triggering lines, as well as hand-selecting around 20 to 100 inputs that are hand-classified and derived from the grammars with the intention of triggering certain lines. These tests would allow us to more clearly assess the effectiveness of our mined constraints' abilities to categorize input classes successfully.
Furthermore, we have not yet included the testing of multiple-file projects, as this was not yet implemented in the version of SFLKit we used. The extension of SFLKit's

instrumentation was happening at the same time as we already ran our research, pushing us towards trying other instrumentation processes. These adaptations would have required us to re-write and re-test big parts of our current oracle construction and event-file analysis, which pushed us toward not extending AviX for the time being.
Lastly, we have not given an overview of all different types of patterns that ISLearn can use to create its constraints, meaning that we have not deeply analyzed all current limits of Avicenna's pattern learner. Crucially, this means that while AviX worked really well on our selection of subjects, these results might not be transferrable due to reasons we have not covered, such as different types of branch conditions, for example.

Limitations Summary

Low test subject count, non-optimized ISLa fuzzer, only allowing for line-analysis, and single-file projects are some of our biggest limitations and challenges this paper faces.
Most of these issues can be solved by larger-scale research and further improvements made to our current code-base.

8 RELATED WORK

Work that is related to AviX and Avicenna will be introduced in this section. For the most part, this section will contain different approaches to solving similar issues. Because of the time intensive nature of debugging code, we have gathered information on different automated debugging approaches, such as listed below. The goal of these approaches is to increase efficiency in debugging by making code easier to read, detecting faulty code, repairing it right away and many other uses.
The following sections will treat these topics:

1. **Code Readability** determines the overall ease with which code may be read during review or maintenance.
2. The **Test Oracle Problem**, which treats the classification of desired and undesired problem behaviors.
3. **Fault Localization**, focusing on detecting faulty code, but also explaining it in more recent approaches
4. **Symbolic Execution** is a code analysis approach that can determine general input classes that trigger interesting behaviors.
5. **Automated Program Repair**, which focuses on automatically detecting and subsequently fixing erroneous code.

8.1 Code Readability

As discussed in our Background section, code readability describes the ease of reading previously written code. Code is more error-prone when little consistency is found in

software components and their documentation, as well as their variable names, for example [43, 44].
Different features [45, 52] and metrics [5] based on these features have been introduced, with the aim to measure readable code based on its aesthetic or syntactical values.
The goal of increasing code readability is to prevent bugs from being introduced in the first place. Documentation, clear variable names, and different formatting practices are seen as improving code readability and therefore support programmers in writing code with little error margins [14, 19, 52].

8.2 Test Oracle Problem

The Test Oracle Problem describes the difficulty in determining the correctness of behavior for a program run [4].
To the best of our knowledge, current research has not found a conclusive method to automate oracle generation, however.
Machine learning approaches face large issues when solving the automation of test oracles, because of the highly complex nature of the behaviors that are exhibited versus the behaviors that are desired of machine learning processes [35].
The oracle automation approach SEER [23], which was introduced in 2022, shows that semantic relations between inputs and outputs may be used to predict program behaviors. The semantic analysis of inputs and outputs are clearly adjacent to Avicenna's approach in finding semantic properties of inputs to determine program behavior, showing that these approaches may benefit from each other in the long run.

8.3 Fault Localization

fault localization, [2, 34, 48, 53, 57, 61] When searching for faulty code locations, approaches oftentimes focus on spectra-based analyses [2, 48, 49], which AviX is inspired by. As mentioned earlier, SFLKit has spectra-based fault localization capabilities, including the instrumentation of programs and evaluation of code based on different suspiciousness scoring systems, as surveyed by Wong et al. [55].
Newer approaches have used large language models (LLM) for finding and explaining faulty code lines [26, 56]. In their paper, Kang et al. [26] present a fault localization approach using LLMs that can also offer explanations on what caused the faulty code in the first place. The explanations are given in natural language.
Yang et al. [56] also use LLMs to find bugs without any test coverage information required by expanding LLMs with their own approach. They leverage LLMs to find suspiciousness scores for buggy programs without any test coverage or need to run the code in itself. Explanations are not present for their approach, however.
Other approaches also seek to diagnose program behavior when finding an issue [29].

8.4 Symbolic Execution

Other important work includes symbolic execution [28]. Symbolic Execution focuses on code coverage, but also uses invariants to depict the reachability of a code branch

[60]. KLEE [8] is a symbolic execution tool that is used in conjunction with program repair [21], for example.
Different problems plague symbolic execution, including the path explosion problem. Due to its impossibly large runtime requirements when it comes to searching deep into a given software [38], symbolic execution approaches are often forced to combine its analyses with coverage mechanisms of other approaches.
Improvements for symbolic execution focus on somehow reducing the path explosion problem, be it through directed symbolic execution [38] or other approaches like concolic testing [46, 47], which combines fuzzing and symbolic execution functionalities.

8.5 Automated Program Repair

Automated Program Repair focuses on fixing found bugs without any further human input.
Recent approaches employed semantics-based detection of code-errors, such as by using symbolic execution tools, such as KLEE [9, 21] or other SMT-based approaches [39], to detect erroneous code.
The greatest challenges in repairing code automatically is generally including a working patch that maintains the original functionality of the program, while removing the faulty behavior. This is often achieved through a fix synthesis, that focuses on maintaining the semantic properties that were found to be the intended function, while removing the semantic features that display faulty behavior [21, 40, 41].

9 CONCLUSION

We have extended Avicenna with our approach called AviX. AviX can successfully utilize Avicenna's feedback loop to mine constraints that define semantic properties of inputs in order to trigger certain lines.
We have determined that AviX works for four different subjects by running Avicenna's `explain()` algorithm on all lines that are in unique branches, leading us to 30 total lines tested, which is equivalent to 17% of total code lines in all of our subjects. Constraints have been found for 29 lines, with 27 of them reliably occurring over the course of our 10 test runs per line.
AviX's constraints work well as predictors of input behavior, maintaining Avicenna's original precision and recall of around 90% for determining faulty program behavior, all the while extending it to general line-trigger diagnoses. For input generation, the constraints have an overall average accuracy of about 90% as well, showing their strength in precisely fuzzing inputs for lines under test.
Some limitations include missing program events to be analyzed, such as function calls instead of lines, a weak semantic fuzzer, and allowing for multiple-file programs. These will be tackled in future work.
We conclude that AviX's line constraints are useful for input generation, program behavior predictions based on inputs, and therefore may be used for manual code analysis as well.

9.1 Future Work

* Fix up AviX's implementation and include it in Avicenna's current release. This will include an overhaul of how code is written, adhere to Avicenna's structure etc.
* Implement multiple file tests. See Avicenna's and AviX's performance in a much larger environment. Test debugging and code-understanding practices. For example, can these constraints help in a more realistic setting for a new person working on a project?
* Qualitative study using AviX's/Avicenna's constraints and fuzzed inputs for understanding code segments or bugs. Maybe in regard to comparing at what program size it is faster to use Avicenna.
* Compare energy consumption of Avicenna and compare it with other fault localization and code explanation methods, especially in regard to LLMs and their energy usage.
* Since context free grammars are such an important part of Avicenna's input, but also very hard to create for larger programs with more complex input classes, look into automating grammar creation, optimizing grammar performance, and check how optimizing a grammar may affect readability of a given grammar!
* Improve the semantic fuzzing of `ISLaSolver.solve()`.

Bibliography

[1] Semantic Debugging. 2023.

[2] P. Agarwal and A. P. Agrawal. Fault-localization techniques for software systems: a literature review. *ACM SIGSOFT Software Engineering Notes*, 39, Sept. 2014.

[3] A. Bacchelli and C. Bird. Expectations, outcomes, and challenges of modern code review. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 712–721, San Francisco, CA, USA, May 2013. IEEE.

[4] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering*, 41, May 2015.

[5] R. P. L. Buse and W. R. Weimer. Learning a Metric for Code Readability. *IEEE Transactions on Software Engineering*, 36(4):546–558, July 2010.

[6] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury. Directed Greybox Fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2329–2344, Dallas Texas USA, Oct. 2017. ACM.

[7] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs.

[8] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs.

[9] C. Cadar and M. Nowack. KLEE symbolic execution engine in 2019. *International Journal on Software Tools for Technology Transfer*, 23(6):867–870, Dec. 2021.

[10] L. Chazette, W. Brunotte, and T. Speith. Exploring Explainability: A Definition, a Model, and a Knowledge Catalogue, Aug. 2021. arXiv:2108.03012.

[11] J. Cohen, S. Teleki, E. Brown, B. DuRette, S. Brown, and B. Fuller. Best Kept Secrets of Peer Code Review.

[12] J. A. S. da Costa, R. Gheyi, F. Castor, P. R. F. de Oliveira, M. Ribeiro, and B. Fonseca. Seeing confusion through a new lens: on the impact of atoms of confusion on novices' code comprehension. *Empirical Software Engineering*, 28(4):81, May 2023.

[13] S. D'Mello and A. Graesser. Confusion and its dynamics during device comprehension with breakdown scenarios. *Acta Psychologica*, 151:106–116, Sept. 2014.

[14] R. M. Dos Santos and M. A. Gerosa. Impacts of coding practices on readability. In *Proceedings of the 26th Conference on Program Comprehension*, pages 277–285, Gothenburg Sweden, May 2018. ACM.

[15] M. Eberlein. Evaluating program behavior with different machine learning approaches.

[16] M. Eberlein. Explaining and debugging pathological program behavior. In *Proceedings of the 30th ACM Joint European Software Engineering Conference*

and Symposium on the Foundations of Software Engineering, pages 1795–1799, Singapore Singapore, Nov. 2022.

[17] M. Eberlein. Explaining and debugging pathological program behavior. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1795–1799, Singapore Singapore, Nov. 2022. ACM.

[18] M. Eberlein, Y. Noller, T. Vogel, and L. Grunske. Evolutionary Grammar-Based Fuzzing. In A. Aleti and A. Panichella, editors, *Search-Based Software Engineering*, pages 105–120. Springer International Publishing, Cham, 2020. Series Title: Lecture Notes in Computer Science.

[19] F. Ebert, F. Castor, N. Novielli, and A. Serebrenik. Confusion Detection in Code Reviews. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 549–553, Shanghai, Sept. 2017. IEEE.

[20] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *Proceedings of the 21st international conference on Software engineering*, Los Angeles California USA, May 1999.

[21] X. Gao, B. Wang, G. J. Duck, R. Ji, Y. Xiong, and A. Roychoudhury. Beyond Tests: Program Vulnerability Repair via Crash Constraint Extraction. *ACM Transactions on Software Engineering and Methodology*, 30(2):1–27, Apr. 2021.

[22] R. Gopinath, A. Kampmann, N. Havrikov, E. O. Soremekun, and A. Zeller. Abstracting failure-inducing inputs. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, Virtual Event USA, July 2020.

[23] A. R. Ibrahimzada, Y. Varli, D. Tekinoglu, and R. Jabbarvand. Perfect is the enemy of test oracle. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2022, pages 70–81, New York, NY, USA, Nov. 2022. Association for Computing Machinery.

[24] A. Kampmann, N. Havrikov, E. O. Soremekun, and A. Zeller. When does my program do this? learning circumstances of software behavior. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Virtual Event USA, Nov. 2020.

[25] A. Kampmann, N. Havrikov, E. O. Soremekun, and A. Zeller. When does my program do this? learning circumstances of software behavior. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1228–1239, Virtual Event USA, Nov. 2020. ACM.

[26] S. Kang, G. An, and S. Yoo. A Quantitative and Qualitative Evaluation of LLM-Based Explainable Fault Localization. *Proceedings of the ACM on Software Engineering*, 1:1424–1446, July 2024.

[27] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, July 1976.

[28] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19, July 1976.

[29] V.-M. Le, A. Felfernig, M. Uta, D. Benavides, J. Galindo, and T. N. T. Tran. DIRECTDEBUG: Automated Testing and Debugging of Feature Models. pages 81–85, May 2021.

[30] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering*, 38(1):54–72, Jan. 2012. Conference Name: IEEE Transactions on Software Engineering.

[31] C. Le Goues, M. Pradel, and A. Roychoudhury. Automated program repair. *Communications of the ACM*, 62(12):56–65, Nov. 2019.

[32] J. Li, B. Zhao, and C. Zhang. Fuzzing: a survey. *Cybersecurity*, 1(1):6, Dec. 2018.

[33] H. Liang, X. Pei, X. Jia, W. Shen, and J. Zhang. Fuzzing: State of the Art. *IEEE Transactions on Reliability*, 67(3):1199–1218, Sept. 2018.

[34] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, Chicago IL USA, June 2005. ACM.

[35] C. C. S. Liem and A. Panichella. Oracle Issues in Machine Learning and Where to Find Them. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, ICSEW'20, pages 483–488, New York, NY, USA, Sept. 2020. Association for Computing Machinery.

[36] K. Liu, L. Li, A. Koyuncu, D. Kim, Z. Liu, J. Klein, and T. F. Bissyandé. A critical review on the evaluation of automated program repair systems. *Journal of Systems and Software*, 171:110817, Jan. 2021.

[37] S. M. Lundberg and S.-I. Lee. A Unified Approach to Interpreting Model Predictions. In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[38] K.-K. Ma, K. Yit Phang, J. S. Foster, and M. Hicks. Directed Symbolic Execution. In E. Yahav, editor, *Static Analysis*, Lecture Notes in Computer Science, Berlin, Heidelberg, 2011. Springer.

[39] S. Mechtaev, J. Yi, and A. Roychoudhury. DirectFix: Looking for Simple Program Repairs. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, pages 448–458, Florence, Italy, May 2015. IEEE.

[40] S. Mechtaev, J. Yi, and A. Roychoudhury. Angelix: scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, New York, NY, USA, 2016. Association for Computing Machinery.

[41] S. Mechtaev, J. Yi, and A. Roychoudhury. Angelix: scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 691–701, New York, NY, USA, 2016. Association for Computing Machinery.

[42] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. SemFix: program repair via semantic analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, San Francisco, CA, USA, 2013. IEEE Press.

[43] S. Scalabrino, M. Linares-Vasquez, D. Poshyvanyk, and R. Oliveto. Improving code readability models with textual features. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, pages 1–10, Austin, TX, USA, May 2016. IEEE.

[44] S. Scalabrino, M. Linares-Vásquez, R. Oliveto, and D. Poshyvanyk. A comprehensive model for code readability. *Journal of Software: Evolution and Process*, 30(6):e1958, June 2018.

[45] T. Sedano. Code Readability Testing, an Empirical Study. In *2016 IEEE 29th International Conference on Software Engineering Education and Training (CSEET)*, pages 111–117, Dallas, TX, USA, Apr. 2016. IEEE.

[46] K. Sen. Concolic testing. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 571–572, New York, NY, USA, Nov. 2007. Association for Computing Machinery.

[47] R. Shariffdeen, Y. Noller, L. Grunske, and A. Roychoudhury. Concolic program repair. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, pages 390–405, New York, NY, USA, June 2021. Association for Computing Machinery.

[48] M. Smytzek and A. Zeller. SFLKit: a workbench for statistical fault localization. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Singapore Singapore, Nov. 2022.

[49] M. Smytzek and A. Zeller. SFLKit: a workbench for statistical fault localization. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2022, pages 1701–1705, New York, NY, USA, Nov. 2022. Association for Computing Machinery.

[50] D. Steinhoefel and A. Zeller. Input invariants. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Singapore Singapore, Nov. 2022.

[51] D. Steinhöfel and A. Zeller. Input invariants. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 583–594, Singapore Singapore, Nov. 2022. ACM.

[52] Y. Tashtoush, Z. Odat, I. Alsmadi, and M. Yatim. Impact of Programming Features on Code Readability. *International Journal of Software Engineering and Its Applications*, 7:441–458, Nov. 2013.

[53] D. Vince, R. Hodován, D. Bársony, and Kiss. The effect of hoisting on variants of Hierarchical Delta Debugging. *Journal of Software: Evolution and Process*, 34, 2022.

[54] M. Weiser. Program slicing.

[55] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa. A survey on software fault localization, 2016.

[56] A. Z. H. Yang, C. Le Goues, R. Martins, and V. Hellendoorn. Large Language Models for Test-Free Fault Localization. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–12, Lisbon Portugal, Feb. 2024. ACM.

[57] A. Zeller. Isolating cause-effect chains from computer programs. *ACM SIGSOFT Software Engineering Notes*, 27, Nov. 2002.

[58] R. B. M. F. G. H. C. Zeller, Andreas; Gopinath. The fuzzing book. 2019.

[59] T. Zhang, P. Wang, and X. Guo. A Survey of Symbolic Execution and Its Tool KLEE. *Procedia Computer Science*, 166:330–334, 2020.

[60] T. Zhang, P. Wang, and X. Guo. A Survey of Symbolic Execution and Its Tool KLEE. *Procedia Computer Science*, 166, 2020.

[61] A. Zheng, M. Jordan, B. Liblit, and A. Aiken. Statistical Debugging of Sampled Programs. In *Advances in Neural Information Processing Systems*, volume 16. MIT Press, 2003.

Appendix

middle
markup
calculator
expression

B Middle

Predictor Table
Producer Table

Listing 10: middle Grammar

```
grammar_middle = {
    "<start>": ["<stmt>"],
    "<stmt>": ["<x>,<y>,<z>"],
    "<x>": ["<integer>"],
    "<y>": ["<integer>"],
    "<z>": ["<integer>"],
    "<integer>": ["<digit>", "<digit><integer>"],
    "<digit>": [str(num) for num in range(1, 10)]
}
```

Listing 11: middle, Line 4

```
forall <z> elem_1 in start:
  exists <y> elem_2 in start:
    (> (str.to.int elem_1) (str.to.int elem_2))
```

Listing 12: middle, Line 5

```
(forall <z> elem_1 in start:
  exists <y> elem_2 in start:
    (> (str.to.int elem_1) (str.to.int elem_2)) and
forall <y> elem_0 in start:
  exists <x> elem_3 in start:
    (> (str.to.int elem_0) (str.to.int elem_3)))
```

Listing 13: middle, Line 6

```
# Two equivalent constraints were found.
#
# Constraint 1
(forall <z> elem_1 in start:
   exists <y> elem_2 in start:
     (> (str.to.int elem_1) (str.to.int elem_2)) and
forall <x> elem_0 in start:
  exists <y> elem_3 in start:
    (>= (str.to.int elem_0) (str.to.int elem_3)))

# Constraint 2
(forall <x> elem_1 in start:
```

```
   exists <y> elem_2 in start:
     (>= (str.to.int elem_1) (str.to.int elem_2)) and
forall <z> elem_0 in start:
  exists <y> elem_3 in start:
    (> (str.to.int elem_0) (str.to.int elem_3)))
```

Listing 14: `middle`, Line 7

```
# Three constraints were found.
# Constraint 1 and Constraint 2 are equivalent.
# Constraint 3 is weaker than the other two.
#
# Constraint 1
(forall <z> elem_1 in start:
   exists <x> elem_2 in start:
     (> (str.to.int elem_1) (str.to.int elem_2)) and
forall <x> elem_0 in start:
  exists <y> elem_3 in start:
    (>= (str.to.int elem_0) (str.to.int elem_3)))

# Constraint 2
(forall <x> elem_1 in start:
   exists <y> elem_2 in start:
     (>= (str.to.int elem_1) (str.to.int elem_2)) and
forall <z> elem_0 in start:
  exists <x> elem_3 in start:
    (> (str.to.int elem_0) (str.to.int elem_3)))

# Constraint 3
# This is the weaker constraint.
(forall <z> elem_1 in start:
   exists <y> elem_2 in start:
     (> (str.to.int elem_1) (str.to.int elem_2)) and
forall <y> elem in start:
  (<= (str.to.int elem) (str.to.int '48')))
```

Listing 15: `middle`, Line 9

```
forall <y> elem_1 in start:
  exists <z> elem_2 in start:
    (>= (str.to.int elem_1) (str.to.int elem_2))
```

Listing 16: `middle`, Line 10

```
(forall <y> elem_1 in start:
   exists <z> elem_2 in start:
     (>= (str.to.int elem_1) (str.to.int elem_2)) and
forall <x> elem_0 in start:
  exists <y> elem_3 in start:
    (> (str.to.int elem_0) (str.to.int elem_3)))
```

Listing 17: `middle`, Line 11

```
# Two equivalent constraints were found.
#
# Constraint 1
(forall <y> elem_1 in start:
```

```
   exists <z> elem_2 in start:
     (>= (str.to.int elem_1) (str.to.int elem_2)) and
forall <y> elem_0 in start:
  exists <x> elem_3 in start:
    (>= (str.to.int elem_0) (str.to.int elem_3)))

# Constraint 2
(forall <y> elem_1 in start:
   exists <x> elem_2 in start:
     (>= (str.to.int elem_1) (str.to.int elem_2)) and
forall <y> elem_0 in start:
  exists <z> elem_3 in start:
    (>= (str.to.int elem_0) (str.to.int elem_3)))
```

Listing 18: `middle`, Line 12

```
(forall <x> elem_1 in start:
   exists <z> elem_2 in start:
     (> (str.to.int elem_1) (str.to.int elem_2)) and
forall <y> elem_0 in start:
  exists <x> elem_3 in start:
    (>= (str.to.int elem_0) (str.to.int elem_3)))
```

C Markup

Predictor Table
Producer Table

Listing 19: `markup` Grammar

```

 grammar_markup = {
    "<start>": ["<structure>"],
    "<structure>": ["<string>", "<html><structure>", "<string><html><
    structure>"],
    "<html>": ["<open><structure><close>"],
    "<open>": ["<LPAR><string><RPAR>"],
    "<close>": ["<LPAR>/<string><RPAR>"],
    "<LPAR>": ["<"],
    "<RPAR>": [">"],
    "<string>": ["", "<str>"],
    "<str>": ["<chars>"],
    "<chars>": ["<char>", "<char><chars>"],
    "<char>": [
       ASCII-char()
    ],
}
```

Listing 20: `markup`, Line 8

```
# Two equivalent constraints were found.
#
# Constraint 1
exists <html> elem_xy in start:
    inside(elem_xy, start)
```

```

# Constraint 2
exists <RPAR> elem in start:
    (= elem >)
```

Listing 21: `markup`, Line 10

```
exists <html> elem_xy in start:
  inside(elem_xy, start)
```

Listing 22: `markup`, Line 14

```
exists <chars> elem_xy in start:
  inside(elem_xy, start)
```

D Calculator

Listing 23: `calculator` Grammar

```
grammar_calc = {
    "<start>": ["<arith_expr>"],
    "<arith_expr>": ["<function>(<number>)"],
    "<function>": ["sqrt", "sin", "cos", "tan"],
    "<number>": [
        "<maybe_minus><one_nine><maybe_digits><maybe_frac>",
        "<zero>"
    ],
    "<maybe_minus>": [
        "",
        "-"
    ],
    "<maybe_frac>": ["", ".<digits>"],
    "<maybe_digits>": ["", "<digits>"],
    "<digits>": [
        "<digit>",
        "<digit><digits>"
    ],
    "<zero>":["0"],
    "<one_nine>": [str(num) for num in range(1, 10)],
    "<digit>": [digit for digit in string.digits],
}
```

Listing 24: `calculator`, Line 8

```
# Three constraints have been found. Constraints 2 and 3 are equivalent.
#
# Constraint 1
(forall <digits> elem in start:
    (<= (str.to.int elem) (str.to.int '89')) and
forall <number> elem_0 in start:
    (>= (str.to.int elem_0) (str.to.int '0')))

```

```
# Constraint 2
exists <function> elem in start:
    (= elem 'sqrt')

# Constraint 3
forall <arith_expr> container in start:
    exists <function> elem in container:
        (= elem 'sqrt')
```

Listing 25: `calculator`, Line 9

```
# Two equivalent constraints were found.
#
# Constraint 1
(exists <number> elem in start:
    (= elem '0') and
forall <arith_expr> container in start:
    exists <function> elem_0 in container:
        (= elem_0 'sqrt'))

# Constraint 2
(exists <number> elem in start:
    (= elem '0') and
exists <function> elem_0 in start:
    (= elem_0 'sqrt'))
```

Listing 26: `calculator`, Line 15

```
# Two constraints were found. They are not equivalent.
#
# Constraint 1
(exists <one_nine> container in start:
    exists <number> length_field in start:
        (<= (str.len container) (str.to.int length_field)) and
exists <function> elem in start:
    (= elem 'sqrt'))

# Constraint 2
(forall <arith_expr> container in start:
    exists <one_nine> elem in container:
        (= (str.len elem) (str.to.int '1')) and
exists <function> elem_0 in start:
    (= elem_0 'sqrt'))
```

Listing 27: `calculator`, Line 20

```
forall <arith_expr> container in start:
    exists <function> elem in container:
        (= elem 'tan')
```

Listing 28: `calculator`, Line 24

```
forall <arith_expr> container in start:
    exists <function> elem in container:
        (= elem 'cos')
```

Listing 29: `calculator`, Line 28

```
forall <arith_expr> container in start:
    exists <function> elem in container:
        (= elem 'sin')
```

E Expression

Listing 30: `expression` Grammar

```
grammar_expression = {
    "<start>": ["<arith_expr>"],
    "<arith_expr>": [
        "<arith_expr><operator><arith_expr>",
        "<number>",
        "<par>",
    ],
    "<par>": ["<lpar><arith_expr><rpar>"],
    "<lpar>": ["("],
    "<rpar>": [")"],
    "<operator>": ["<plus>", "<minus>", "<mul>", "<div>"],
    "<plus>": [" + "],
    "<minus>": [" - "],
    "<mul>": [" * "],
    "<div>": [" / "],
    "<number>": [
        "<maybe_minus><non_zero_digit><maybe_digits>",
        "0"
    ],
    "<maybe_minus>": ["", "~ "],
    "<non_zero_digit>": [
        str(num) for num in range(1, 10)
    ],
    "<digit>": list(string.digits),
    "<maybe_digits>": ["", "<digits>"],
    "<digits>": ["<digit>", "<digit><digits>"],
}
```

Listing 31: `expression`, Line 10

```
exists <operator> elem_xy in start:
    inside(elem_xy, start)
```

Listing 32: `expression`, Line 40

```
exists <maybe_minus> elem in start:
    (= elem '~ ')
```

Listing 33: `expression`, Line 59

```
exists <par> elem_xy in start:
    inside(elem_xy, start)
```

Listing 34: expression, Line 73

```
exists <par> elem_xy in start:
    inside(elem_xy, start)
```

Listing 35: expression, Line 85

```
exists <maybe_minus> elem in start:
    (= elem ""~ "")
```

Listing 36: expression, Line 87

```
forall <digit> elem_1 in start:
  exists <number> elem_2 in start:
    (>= (str.to.int elem_1) (str.to.int elem_2))
```

Listing 37: expression, Line 93

```
exists <operator> elem_xy in start:
    inside(elem_xy, start)
```

Listing 38: expression, Line 95

```
# Two equivalent constraints were found.
#
# Constraint 1
exists <operator> elem in start:
    (= elem ' * ')

# Constraint 2
exists <mul> elem in start:
    (= elem ' * ')
```

Listing 39: expression, Line 97

```
exists <operator> elem in start:
    (= elem ' / ')
```

Listing 40: expression, Line 105

```
# Four different constraints were found.
# Constraint 3 and 4 are equivalent, the rest are all not equivalent.
#
# Constraint 1
exists <operator> elem_xy in start:
    inside(elem_xy, start)

# Constraint 2
(exists <operator> elem_xy in start:
   inside(elem_xy, start) and
forall <digits> elem_1 in start:
    exists <number> elem_2 in start:
        (>= (str.to.int elem_1) (str.to.int elem_2)))

# Constraint 3
(exists <non_zero_digit> container in start:
    exists <non_zero_digit> length_field in start:
        (<= (str.len container) (str.to.int length_field)) and
```

```
exists <operator> elem_xy in start:
   inside(elem_xy, start))

# Constraint 4
(exists <operator> elem_xy in start:
    inside(elem_xy, start) and
exists <maybe_minus> elem_xy_0 in start:
    inside(elem_xy_0, start))
```

Listing 41: `expression`, Line 107

```
# Two equivalent constraints were found.
#
# Constraint 1
exists <plus> elem_xy in start:
    inside(elem_xy, start)

# Constraint 2
exists <plus> elem in start:
    (= elem ' + ')
```

Listing 42: `expression`, Line 109

```
# Two equivalent constraints were found.
#
# Constraint 1
exists <minus> elem_xy in start:
    inside(elem_xy, start)

# Constraint 2
exists <operator> elem in start:
    (= elem ' - ')
```

Selbständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und noch nicht für andere Prüfungen eingereicht habe. Sämtliche Quellen einschließlich Internetquellen, die unverändert oder abgewandelt wiedergegeben werden, insbesondere Quellen für Texte, Grafiken, Tabellen und Bilder, sind als solche kenntlich gemacht. Mir ist bekannt, dass bei Verstößen gegen diese Grundsätze ein Verfahren wegen Täuschungsversuchs bzw. Täuschung eingeleitet wird.

Berlin, den October 14, 2024 .....