On Writing SMT-LIB Scripts: Metrics and a New Dataset

Soaibuzzaman, Jan Oliver Ringert

Bauhaus-University Weimar, Germany

Abstract

Satisfiability Modulo Theory (SMT) checking is concerned with checking the satisfiability of first-order formulas with respect to some background theories. The SMT-LIB format is a standardized language for scripts expressing SMT problems.

Popular datasets of SMT-LIB scripts have been collected for benchmarking SMT solvers. Rather than focusing on evaluating SMT solvers, our work focuses on exploring how novice users write SMT-LIB scripts. We present a dataset of SMT-LIB scripts with fine-grained editing paths. The dataset consists of 2,415 editing paths with a total of 18,133 SMT-LIB scripts. All scripts were collected from a web-based interface for the Z3 SMT solver in educational settings.

We analyze the dataset in terms of sizes of scripts, errors users make, similarities of consecutive scripts, editing distances, and edit steps required to fix errors. We make the dataset and the code for computing our metrics available for future research on language design, tool support, and teaching materials.

Keywords

SMT-LIB, evolution, dataset, novices

1. Introduction

Satisfiability Modulo Theory (SMT) checking [1] is concerned with checking the satisfiability of first-order formulas with respect to some background theories. SMT solvers [2, 3, 4] are tools for reasoning about SMT problems, e.g., for checking satisfiability or generating unsatisfiability proofs.

The SMT-LIB format is a standardized language [5] for scripts expressing SMT problems. The format of the language was designed as a standard input formal for SMT solvers and is continuously growing to foster and reflect advances in SMT. The SMT-LIB syntax has inspired various other languages from different domains, e.g., verification [6], modeling [7, 8], and syntax-guided synthesis [9]. The SMT-LIB standard is supported by most SMT solvers and API layers for using SMT solvers [10, 11].

Popular datasets of SMT-LIB scripts [12, 13], containing more than 400,000 SMT-LIB scripts, have been collected for benchmarking SMT solvers [14]. In addition to performance, many works address the correctness and quality of SMT solvers through testing on the input- [15] or API-level [16]. Much fewer works deal with the analysis of SMT-LIB scripts themselves. Some exceptions are tools and libraries for validating and translating scripts [17, 18] or tools for debugging these [19, 20][1]. Dolmen [17] offers parsing, validation, and type-checking capabilities, enabling real-time error detection in editors such as VS Code and Emacs. However, we are not aware of any tools offering contextual autocompletion, refactoring support, or complex dependency resolution. Similarly, we are not aware of any studies on quality aspects nor experiences of users writing SMT-LIB scripts.

Thus, rather than focusing on evaluating SMT solvers, our work focuses on exploring how novice users write SMT-LIB scripts. We present the $\mathsf{FMP}_{\mathsf{smt}}$ dataset of SMT-LIB scripts with fine-grained editing paths, i.e., sequences of SMT-LIB scripts that were consecutively written and analyzed by (novice) users. The dataset consists of 2,415 editing paths with a total of 18,133 SMT-LIB scripts. All scripts were collected from a web-based interface for the Z3 SMT solver [2] in educational settings. Similar datasets exist for other specification languages, e.g., the Alloy language [21, 22].

We analyze the dataset in terms of sizes of scripts, errors users make, similarities of consecutive scripts, editing distances, and edit steps required to fix errors. Our analyses are inspired by similar analyses

SMT 2025: 23rd International Workshop on Satisfiability Modulo Theories, August 10–11, 2025, Glasgow, UK
0000-0002-8971-5904 (Soaibuzzaman); 0000-0002-3610-3920 (J. O. Ringert)

[1] Admittedly, ddSMT's motivation is again focused on debugging SMT solvers rather than scripts.

conducted on the aforementioned Alloy datasets [23, 22]. We make the new FMP_{smt} dataset [24] and the code for computing our metrics [25] available for future research on language design, tool support, and teaching materials.

The remainder of this work is structured as follows. Section 2 briefly presents the foundations of our work. Section 3 lists our research questions, Sect. 4 presents our dataset and data processing. Section 5 presents the evaluation of our research questions on the datasets. We conclude in Sect. 6.

2. Preliminaries

We now give a brief overview of the SMT-LIB language specification and the Formal Methods Playground.

2.1. SMT-LIB

The SMT-LIB standard [5] defines a textual scripting language for sorted first-order logic terms and formulas and commands to interact with SMT solvers, e.g., for creating assertions, checking their validity, and inspecting generated models and proofs. The standard also provides languages for specifying logics and background theories less relevant our analysis presented here. An example SMT-LIB script is shown in Listing 1. It encodes part of the Agatha puzzle [26, P.55] consisting of various statements relating Agatha, her butler, and Charles to determine who killed Agatha.

```
1  (set-logic UF)
2  (declare-datatype Person (Agatha Butler Charles)) ; datatype for people in mansion
3  (declare-const Killer Person)
4  (declare-fun killed (Person Person) Bool) ; a function/predicate to represent killing
5  (declare-fun hates (Person Person) Bool)
6  (declare-fun richer (Person Person) Bool)
7
8  ; encoding of formula (1) someone killed Agatha
9  (assert (exists ((x Person)) (killed x Agatha)))
10 (assert (forall ((x Person)) (forall ((y Person))
11     (=> (killed x y)
12         (and (hates x y) (not (richer x y)))))))
13 (assert (forall ((x Person)) (=> (hates Agatha x) (not (hates Charles x)))))
14 ; ...
15 (assert (killed Killer Agatha))
16 (check-sat)
17 (get-model)
```

Listing 1: Excerpt of an example SMT-LIB script from the FMP_{smt} dataset

SMT-LIB scripts are sequences of commands, e.g., the `set-logic` command selecting the logic of uninterpreted functions in Listing 1, l. 1, the command `assert` for asserting formulas in Listing 1, ll. 9-15, or the command `check-sat` for checking satisfiability of the asserted formulas. Additional commands with prefix `declare-` introduce datatypes (Listing 1, l. 2), constants, and functions (Listing 1, ll. 3-6). The command `get-model` (Listing 1, ll. 17), if successful, retrieves an interpretation of constants and functions satisfying all asserted formulas and presents them as function definitions, e.g., the interpretation of constant `Killer` will be defined as the value `Agatha`, `Butler`, or `Charles`.

2.2. Formal Methods Playground

The Formal Methods Playground[2] is a web application for writing and analyzing models, specifications, and scripts in various modeling and specification languages. We have developed this application mainly for teaching, e.g., slides in our Formal Methods for Software Engineering lecture [27] contain permalinks to example scripts on the Formal Methods Playground for direct analysis in the browser. Currently, the Formal Methods Playground supports Limboole[3], Z3 [2], Alloy [28], nuXmv [29] and Spectra [30] specifications. For analyzing SMT-LIB scripts, the input pane of the Formal Methods Playground consists

[2]See https://play.formal-methods.net and https://www.youtube.com/playlist?list=PLGyeoukah9NYq9ULsIuADG2r2QjX530nf
[3]See https://fmv.jku.at/limboole/

of an editor with syntax highlighting and basic code completion and search capabilities. The output pane shows the output of the Z3 solver executed on the input SMT-LIB script verbatim. The analysis of SMT-LIB scripts is usually performed locally in the users' browser using Web Assembly [31] without any timeout or as a fall-back on our server with a timeout of 60s. We have added basic language support, e.g., syntax checking and referencing constant and function symbols, as an optional feature in February 2025 (we have not evaluated the use and effect of these features so far).

3. Research Questions

We aim to gain a better understanding of how novice users write and evolve SMT-LIB scripts.

We define the following research questions:

- **RQ1**: What are the key characteristics of the $\mathsf{FMP}_{\mathsf{smt}}$ dataset?
- **RQ2**: Where do users most commonly introduce syntactic errors in SMT-LIB scripts?
- **RQ3**: How do consecutive SMT-LIB scripts differ?
- **RQ4**: How large are the edit distances between consecutive SMT-LIB scripts?
- **RQ5**: How do users fix errors over multiple edit steps?

4. Data Processing and Metrics Computation

4.1. Experimental Data

We analyze our proposed Formal Methods Playground SMT-LIB ($\mathsf{FMP}_{\mathsf{smt}}$) dataset [24]. It contains SMT-LIB scripts executed on the Formal Methods Playground from November 2023 to May 2025. Students at Bauhaus-University Weimar use this platform as part of our Formal Methods for Software Engineering [27] module. Students taking this module are mainly from the international programs Digital Engineering (DE, an interdisciplinary 2-year MSc program) and Computer Science for Digital Media (CS4DM, a 2-year MSc program). While the background of the CS4DM students is a classical BSc degree in computer science, the second, and larger group of DE students are from diverse engineering, e.g., civil, mechanical, or even electrical engineering, (∼50%) and computer science (∼50%) backgrounds. Based on user activity and the models authored, at least one additional university uses the platform. We capture scripts, timestamps, and historical derivations structured in a parent-child relationship. The $\mathsf{FMP}_{\mathsf{smt}}$ dataset contains 18,133 SMT-LIB scripts.

4.2. Edit Paths

The $\mathsf{FMP}_{\mathsf{smt}}$ datasets maintain records of the previous revision of each SMT script. Each revision is a user submission i.e., the current script whenever the user executes the solver. We utilize this information to reconstruct the *edit path*[4], allowing us to capture the sequences of edits/submissions made by users (these edits are typically small, see Sect. 5.4).

The $\mathsf{FMP}_{\mathsf{smt}}$ dataset consists of 2,415 unique edit paths. In particular, the top 25% of the edit paths have a length greater than 13 for $\mathsf{FMP}_{\mathsf{smt}}$, with median lengths of 6. The $\mathsf{FMP}_{\mathsf{smt}}$ dataset has 272 unique initial scripts[5] within the edit paths.

Note that the number of scripts in edit paths exceeds the total number of scripts in the dataset. The exchange of permalinks to individual scripts via the Formal Methods Playground allows the branching of edit paths leading to common prefixes.

[4]We adopt the terminology of [23] and [22].

[5]While many edit paths start from scratch, most edit paths share initial models provided by instructors [27].

Table 1
(a) Characteristics of SMT-LIB scripts in the FMP_{smt} dataset and (b) Lengths of edit paths

	Q1	Median	Q3	Max
ELOC	10	26	65	1,531
Max Nesting Depth	5	5	6	42
# assert commands	2	7	23	287
# declare-const commands	1	4	14	371
# declare-fun commands	0	0	3	299
Time taken (s) (timeout of 600 s)	0.02	0.02	0.03	318.27

(a)

	Length
Min	1
Q1	3
Median	6
Q3	13
Max	321
Length ≥ 5 (%)	58.26%

(b)

4.3. Semantic Comparison of SMT-LIB scripts

For semantic comparison, we utilize a lightweight procedure (implemented using Z3 [2] APIs that we make available from [25]) that verifies the semantic entailment between the assertion sets collected from two compared scripts. This process classifies their semantic relationship as either equivalent, incomparable, or a refinement in either direction. Specifically, we evaluate two queries: whether script 1 semantically entails script 2, and vice versa. If both hold, the scripts are semantically equivalent; if neither does, they are incomparable; and if only one direction holds, we report a refinement relation.

It is important to note that this semantic comparison approach is naive and has several limitations. The comparison is oblivious to variable renaming, ignores `push` and `pop` scopes, and can misclassify scripts that contain inconsistent or unsatisfiable assertions. Consequently, the reported relationships are approximate and may not accurately reflect subtle semantic nuances. However, relatively small edit distances (see Sect. 5.4), i.e., syntactically very similar scripts in edit paths, and low numbers of incremental scripts (see Sect. 5.1) suggest a potential for the analysis on the FMP_{smt} dataset.

5. Evaluation

We now present data to answer our research questions from Sect. 3. All experiments used Z3 v4.13.4 and were executed on a server with an Intel Xeon Platinum 8260 CPU @ 2.4GHz running Ubuntu 22.04.

5.1. RQ1: Dataset Characteristics

To characterize the FMP_{smt} dataset, we conducted a quantitative analysis based on several structural and syntactic features. These include effective lines of code (ELOC), the frequency of common SMT-LIB commands such as `assert`, `declare-const`, `check-sat`, etc., as well as the maximum nesting depth of expressions within each script.

Table 1a summarizes the distribution of key structural features in the FMP_{smt} dataset, highlighting 25th percentile, median, 75th percentile, and maximum values. Effective lines of code (ELOC) have a median of 26 and a maximum of 1,531, indicating most scripts are small, but a few are much larger. The maximum nesting depth of operators is generally low, with a median maximum nesting depth of 5, with some outliers reaching 42, suggesting instances of highly nested formulas. The counts of different commands per script increase with ELOC.

The FMP_{smt} dataset comprises 1,221 (6.73%) scripts with multiple `check-sat` commands (incremental scripts in the terminology of the SMT-LIB benchmark [12]). Execution times vary considerably; as indicated in Table 1a, 75% of the scripts finish in under 0.03 seconds, though some outliers extend up to 318.27 seconds. A uniform timeout of 600 seconds was applied based on our expectation of how long users might wait for the in-browser SMT execution on the Formal Methods Playground. Only 28 scripts in the dataset timed out.

The dataset spans a wide range of SMT logics—38 in total (including typos)—with QF_UF (1,819 scripts), QF_LIA (1,447 scripts), and QF_AUFLIA (401 scripts) being the most common. Note that many

scripts do not declare a logic, as Z3 does not enforce this. Less frequent logics include specialized or rarely used ones, such as QF_SLIA, appearing only once. This variety emphasizes the dataset's extensive scope, encompassing both quantifier-free and quantified fragments.

Table 2
(a) Syntactically unique scripts in the FMP_{smt} dataset (100% refers to the total number of scripts) and (b) Share of syntactically invalid scripts in edit paths

	#	%
Syntactically Unique Scripts	9,150	50.46%
Syntactically Correct Scripts (in unique)	5,971	65.26%
Syntactically Incorrect Scripts (in unique)	3,012	32.92%

(a)

	#
Edit Paths (100%)	2,415
With Invalid Scripts (%)	59.13%
Without Valid Scripts (%)	1.41%

(b)

Table 1b summarizes the distribution of edit path lengths. The dataset comprises 2,415 distinct edit paths, with lengths from 1 to 321 edits. The median length is 6, and the 75th percentile is 13, suggesting that while some users finish their analyses quickly, many engage in longer editing processes. Notably, over 58% of edit paths have five or more steps, highlighting the significant time and effort users invest in refining their scripts.

We also investigated the frequency with which users encounter syntactic errors in these paths. Table 2b shows that 59.13% of edit paths contain at least one invalid script, suggesting users often make erroneous iterations before reaching a correct version. Only 1.41% of paths consist entirely of invalid scripts, indicating instances where users cannot construct a syntactically valid SMT-LIB script.

Another important characteristic of the dataset is the degree of syntactic uniqueness across scripts. As shown in Table 2a, 9150 scripts (50.46%) are syntactically unique, indicating that more than half of the dataset comprises distinct SMT-LIB formulations rather than duplicates or near-duplicates. Among these unique scripts, 5971 (65.26%) are syntactically correct, highlighting a reasonably high rate of well-formed inputs within the diverse subset.

> **In summary**, the FMP_{smt} dataset consists of SMT-LIB scripts of mainly small to medium sizes (median 26 ELOC) and complexities, encompassing a broad range of 38 different logics (including typos). The execution times are typically brief (within tens of milliseconds), and users often refine scripts iteratively. The dataset includes a wide range of problem instances, as well as diverse modeling styles and patterns.

5.2. RQ2: Where do users most commonly introduce syntactic errors in SMT-LIB scripts?

Among the 18,133 SMT-LIB scripts analyzed within the FMP_{smt} dataset, we observed that approximately 40% contain at least one syntax error, which is distributed across various SMT commands. To analyze common syntactic errors in SMT-LIB scripts, we developed a parser to identify and categorize errors. For each identified syntactic error, we recorded the top-level SMT-LIB command, the absolute number of errors, the total occurrences of the command, and calculated the relative percentage of errors for each command.

Table 3 presents the ten commands most susceptible to user-induced syntactic errors, evaluated by both absolute and relative frequencies of errors. The `assert` command accounts for the highest absolute number of errors, with 35,132 instances (11.01%), likely due to the inherent complexity of the logical expressions it often encompasses. In contrast, the `declare-const` command shows 10,920 errors, but with a lower error rate of 4.85%, indicating that it is mostly used correctly despite its frequent use. Interestingly, the commands `get-value` and `eval` exhibit the highest relative error rates, at 30.62% and 27.96%, respectively. These commands are less common but appear to be more error-prone. However,

Table 3
Top 10 commands with the highest absolute number of errors, number of elements across scripts in the FMP_{smt} dataset, and percentage of erroneous commands per type

command	abs. # error	# total elements	rel. % of command
assert	**35,132**	**319,049**	11.01%
declare-const	10,920	224,947	4.85%
declare-fun	6,870	46,489	14.78%
get-value	5,404	17,648	**30.62%**
define-fun	3,237	22,548	14.36%
get-model	2,071	12,762	16.23%
declare-datatype	1,787	18,152	9.84%
check-sat	134	28,318	0.47%
eval	104	372	27.96%
quantifiers	49	1,342	3.65%

note that our automated analyses of errors in SMT-LIB scripts did not distinguish between static syntax errors, e.g., invoking command `get-value` for an undeclared constant, and runtime errors, e.g., invoking command `get-value` without a model. We leave a more detailed investigation to future work.

The absolute and relative numbers of errors in Table 3 suggest that while `assert` is frequently used and therefore contributes a large raw number of errors, commands like `get-value` and `eval` might proportionally be more challenging for novice users to implement correctly.

Building on the analysis of error distribution across SMT-LIB commands, we investigated the specific types of syntactic errors encountered by users. Table 4 summarizes the top 10 categories of syntactic errors observed in the FMP_{smt} dataset. These errors are classified according to frequent patterns observed in the error messages. Detailed information regarding the exact and complete error messages for each script[6] can be found in the replication package [25].

The most prevalent error, accounting for over 50% of all cases, is "Unknown constant", which typically arises when users reference symbols that have not been previously declared. This suggests a recurring issue with undeclared variables or scope-related oversights in SMT-LIB scripts. The following most common categories involve incorrect or malformed declarations, such as "Invalid constant declaration" (9.8%) and "Parsing function declaration" errors (7.16%), indicating that sort mismatches and incorrect syntactic forms in declarations are also significant sources of failure.

These errors highlight a common challenge that novice users encounter with sort declarations and naming conventions in SMT-LIB. Many of these issues are avoidable and could likely be minimized through improved language support, such as context-aware autocompletion, scoping, referencing, validation, and informative diagnostics that guide users toward correct usage as they write. Partially, these analyses are already supported by existing tools, e.g., Dolmen [17] supports validation and type-checking, and evaluating their impact might be interesting future work.

Errors such as "Logic does not support" and "Model is not available" indicate issues related to the selected logic or solver state. If the commands are misused in a given context, these issues can also appear as syntactic problems.

> **In summary**, syntactic errors in the FMP_{smt} dataset (mainly referencing unknown constants or writing invalid declarations) are both frequent and concentrated around specific SMT-LIB commands and constructs (mainly in assertions and declarations). Common errors include malformed or mismatched declarations, pointing to challenges in understanding SMT-LIB's type and declaration system. These insights suggest a need for improved tooling and guidance, such as contextual validation, better scoping support, and more intelligent error detection and error messages to assist users.

[6] See file https://raw.githubusercontent.com/se-buw/smt-metrics/refs/tags/SMT2025/results/fmp_error_category.csv

Table 4
The 10 most common categories of errors, along with their corresponding percentages of the total errors recorded in the FMP_{smt} dataset (100% refers to the total count of errors)

Category	Count	Percentage
Unknown constant **constant_name**	35,509	50.13%
Invalid constant declaration **sort_name**	6,941	9.8%
Parsing function declaration **sort_name**	5,070	7.16%
Logic does not support	4,325	6.11%
Invalid declaration	3,629	5.12%
Model is not available	3,506	4.95%
Invalid sort	2,921	4.12%
Unknown sort *	2,587	3.65%
Unexpected character	928	1.31%
Invalid function declaration	856	1.21%

Table 5
Syntactic Identical and Semantic (Non-)Consecutive Table

	Syntactically Identical	Semantically $\equiv$	$\neq$	$S1 \sqsubset S2$	$S2 \sqsubset S1$
Consecutive	4,319	6,332	2,805	1,149	1,748
Non-Consecutive	2,121	877	2,125	908	1,542

5.3. RQ3: How do consecutive SMT-LIB scripts differ?

To understand the evolution of SMT-LIB scripts during user editing sessions, we analyzed the differences between "consecutive" (i.e., directly successive) and "non-consecutive" (i.e., scripts related through longer edit paths) script pairs along each edit path (see Sect. 4.2). Our analysis addresses both syntactic identity and various semantic relationships, including equivalence ($\equiv$), incomparability ($\neq$), and refinement relationships ($S1 \sqsubset S2$ and $S2 \sqsubset S1$) as mentioned in Sect. 4.3.

As shown in Table 5, a substantial number of consecutive script pairs (4,319 pairs) were syntactically identical, which is significantly higher than for non-consecutive pairs (2,121 pairs). This suggests users often submit the same script without modifications, e.g., when reviewing their own earlier work or examples provided in permalinks.

This analysis revealed that only 23.86% of consecutive script pairs (6,332 pairs) in the edit paths are semantically equivalent, while over 10% (2,805 pairs) are incomparable, and roughly 11% (1,149 and 1,748 pairs) exhibit refinement relations. This indicates significant changes in specification behavior, even among consecutive edits. Furthermore, we have identified 216 unknowns. Note that these percentages do not add up to 100% as one syntactically incorrect script in a pair prevents our semantic comparison of that pair.

In contrast, non-consecutive script pairs on the same edit path demonstrate lower semantic equivalence (8.04%) but higher rates of incomparability (19.48%). These trends suggest that as users make more edits over time, scripts tend to diverge semantically, reflecting an increasing complexity in the SMT script. Interestingly, refinement relationships remain prevalent even in non-consecutive pairs, implying that users incrementally strengthen or weaken specifications throughout the editing process.

Table 6
Levenshtein Distance Statistics

Statistic	Levenshtein Distance
Q1	6
Median	51
Q3	315
Max	38,659

```
(set-logic QF_UF)
(declare-const q Bool)
(declare-const r Bool)
(declare-const s Bool)
(declare-const t Bool)
(assert (and (=> q r) (=> r s) (=> s t)
    (=> t (not q)) q))
(check-sat)
```

Listing 2: Stronger specification with cyclic Boolean implications (unsat)

```
(set-logic QF_UF)
(declare-const p Bool)
(declare-const q Bool)
(declare-const r Bool)
(declare-const s Bool)
(assert (=> (or (and p q) (and r s)) (and
    (or p r) (or q s))))
(check-sat)
```

Listing 3: Weaker specification with general Boolean implication (sat)

As an example, Listings 2 and 3 demonstrate a refinement relation between two SMT-LIB scripts. Listing 2 has a stronger specification that creates a cyclic chain of Boolean implications leading to a contradiction. Conversely, Listing 3 asserts a weaker property that is always satisfiable. Our semantic comparison reveals that Listing 2 $\sqsubset$ Listing 3: all models of the listing 2 (none, in this case) satisfy the assertions in Listing 3, but not vice versa.

> **In summary**, over 4,300 consecutive pairs are syntactically identical, indicating user habits, such as rerunning unchanged scripts. Consecutive scripts show high numbers of cases of semantic equivalence and refinement, implying that users refine their logic incrementally. The persistence of refinement in both consecutive and non-consecutive pairs highlights a frequent pattern of strengthening or weakening specifications, suggesting a trial-and-error process in developing SMT scripts.

5.4. RQ4: How large are the edit distances between consecutive SMT-LIB scripts?

To explore the evolution of SMT-LIB scripts across edit paths, we calculate the Levenshtein distance [32] between successive scripts. This metric quantifies the number of single-character insertions, deletions, or substitutions required to transform one script into another, providing a straightforward yet insightful assessment of syntactic variation.

The findings are summarized in Table 6, illustrating that most script edits tend to be minor. The 25th percentile of edit distances is 6, while the median stands at 51, suggesting that users implement only slight adjustments between successive versions in over half of the instances. These adjustments can involve minor tweaks such as refining logical assertions, correcting constants, adding or removing a single command, or fixing formatting issues.

Nonetheless, the distribution shows a lengthy tail of significant edits. The 75th percentile sits at 315 characters, while the maximum distance reaches 38,659, indicating infrequent yet significant rewrites. These large distances correspond to major structural changes or cases where the user replaces the existing script entirely, potentially because they are starting anew.

Table 7
Edit Steps observed when fixing parsing errors and to reach a SAT script from UNSAT

Type	Q1	Median	Q3	Max
Fix Syntax Error	1	1	3	52
UNSAT to SAT	1	1	2	58

> **In summary**, the edit distances between consecutive SMT-LIB scripts indicate that the majority of user edits are small (median 51 characters). This suggests that users generally implement incremental adjustments such as refining assertions, correcting errors, or modifying individual commands– rather than undertaking comprehensive overhauls of entire scripts.

5.5. RQ5: How do users fix errors over multiple edit steps?

Our analysis indicates that approximately 40% of all scripts exhibit at least one syntax error (see Sect. 5.1), while approximately 60% of the edit paths are characterized by the presence of at least one erroneous script (see Table 1b).

To quantify the effort required for error resolution, we calculated the number of consecutive edit steps necessary to rectify specific types of issues. Table 7 presents the quartiles and maximum values for the steps involved in correcting syntax errors and transforming an unsatisfiable (UNSAT) script into a satisfiable (SAT) script.

The data indicates that most syntax errors are corrected swiftly: both the 25th percentile (Q1) and the median stand at 1, meaning that in at least half of the instances, users address a parse error in just one edit step. The 75th percentile (Q3) is 3, whereas the most prolonged fix recorded took 52 edit steps. This suggests that while fast recovery is typical, some users face considerable challenges, particularly with longer or more intricate scripts.

Similarly, the process of resolving semantic unsatisfiability, which involves changing a script from UNSAT to SAT, also demonstrates a high degree of efficiency. The median number of steps required is 1, with 75% of transitions occurring within two steps. This suggests that users often make precise logical adjustments in a single step once they recognize unsatisfiability. However, some complex semantic issues resulted in up to 58 steps for correction.

These findings indicate that novice users are typically able to address syntactical issues and advance towards satisfiability with low effort; however, they also underscore the existence of non-trivial challenges in a minority of instances.

> **In summary**, most users resolve syntax and satisfiability issues quickly, often within a single edit. This indicates users typically identify and fix problems with minimal effort. However, some cases require extensive revisions, highlighting that, while rapid recovery is common, some users face significant challenges with complex debugging tasks.

5.6. Threats to Validity

We now identify and discuss various threats to the validity of our analyses. For transparency and reproducibility we provide the implementation of our processing in [25].

First, we assume that the use of the Formal Methods Playground across the dataset is similar to that of our students, i.e., small models are created and analyzed. However, we have no control over how other users use the publicly available platform, e.g., it could be used for teaching with very narrow task definitions. To assess this uncertainty, we have assessed unique initial nodes of edit paths in Sect. 4.2, giving some confidence in the models' variability.

Second, the permalinks provided in the lecture materials might have inadvertently encouraged users to begin their models from these pre-existing structures. This could affect the root nodes of edit paths, complicating the comparison of model evolution across various initial states and potentially underestimating the diversity of evolution paths. Additionally, users may forget to reset the edit path after completing one model and before starting another, i.e., the edit paths of the new model may incorrectly link to the previous one. This could distort the sequence of changes and hinder our ability to analyze the edit paths accurately.

Third, the dataset captures only a small portion of the SMT-LIB standard and is primarily written by novice users introduced to SMT-LIB through lecture materials. Any findings are restricted to these assumptions and may not be generalized. In particular, while it might seem surprising that unsatisfiability is often resolved in a single edit step, this could be due to the prevalence of small SMT scripts with few assertions.

6. Conclusion

We have presented and analyzed the Formal Methods Playground SMT-LIB ($\mathsf{FMP}_{\mathsf{smt}}$) dataset. We analyzed the dataset in terms of sizes of scripts, errors users make, syntactic and semantic similarity of consecutive scripts, editing distances, and edit steps required to fix errors. Syntactic errors are highly prevalent in writing SMT-LIB scripts, and our analysis reveals common syntax error classes that could likely be avoided with improved tool support, such as code completion and reference and scope checking. Interestingly, debugging unsatisfiability did not appear to present a problem to users, and scripts returned to being satisfiable often within a single editing step.

7. Data Availability

We have made the Formal Methods Playground SMT-LIB ($\mathsf{FMP}_{\mathsf{smt}}$) dataset publicly available on Zenodo as [24]. In addition, to support reproducibility of our metrics computations and the preprocessing, we have made the implementation used for our analyses available in a GitHub repository as [25].

Acknowledgments

We want to acknowledge Salar Kalantari's MSc thesis [33] on a preliminary analysis (inspiring Table 4 and including a difficulty metric, not presented here) of a smaller, earlier subset of the $\mathsf{FMP}_{\mathsf{smt}}$ dataset.

Declaration on Generative AI

The authors have not employed any Generative AI tools.

References

[1] C. W. Barrett, R. Sebastiani, S. A. Seshia, C. Tinelli, Satisfiability modulo theories, in: Handbook of Satisfiability - Second Edition, volume 336 of *Frontiers in Artificial Intelligence and Applications*, IOS Press, 2021, pp. 1267–1329. doi:10.3233/FAIA201017.

[2] L. M. de Moura, N. S. Bjørner, Z3: an efficient SMT solver, in: TACAS 2008, volume 4963 of *LNCS*, Springer, 2008, pp. 337–340. doi:10.1007/978-3-540-78800-3_24.

[3] C. W. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, C. Tinelli, CVC4, in: CAV 2011, volume 6806 of *LNCS*, Springer, 2011, pp. 171–177. doi:10.1007/978-3-642-22110-1_14.

[4] B. Dutertre, Yices 2.2, in: CAV 2014, volume 8559 of *LNCS*, Springer, 2014, pp. 737–744. doi:10.1007/978-3-319-08867-9_49.

[5] C. Barrett, P. Fontaine, C. Tinelli, The SMT-LIB Standard: Version 2.7, Technical Report, Department of Computer Science, The University of Iowa, 2025. Available at https://www.SMT-LIB.org.
[6] A. Cimatti, A. Griggio, S. Tonetta, The VMT-LIB language and tools, in: SMT 2022, volume 3185 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2022, pp. 80–89. URL: https://ceur-ws.org/Vol-3185/extended9547.pdf.
[7] A. Vakili, N. A. Day, Avestan: a declarative modeling language based on SMT-LIB, in: MiSE 2012, IEEE Computer Society, 2012, pp. 36–42. doi:10.1109/MISE.2012.6226012.
[8] N. A. Day, A. Vakili, Representing hierarchical state machine models in SMT-LIB, in: MiSE@ICSE 2016, ACM, 2016, pp. 67–73. doi:10.1145/2896982.2896990.
[9] S. Padhi, E. Polgreen, M. Raghothaman, A. Reynolds, A. Udupa, The sygus language standard version 2.1, CoRR abs/2312.06001 (2021). arXiv:2312.06001.
[10] M. Gario, A. Micheli, Pysmt: a solver-agnostic library for fast prototyping of smt-based algorithms, in: SMT Workshop 2015, 2015.
[11] D. Baier, D. Beyer, K. Friedberger, Javasmt 3: Interacting with SMT solvers in java, in: CAV2021, volume 12760 of *LNCS*, Springer, 2021, pp. 195–208. doi:10.1007/978-3-030-81688-9_9.
[12] M. Preiner, H.-J. Schurr, C. Barrett, P. Fontaine, A. Niemetz, C. Tinelli, Smt-lib release 2024 (incremental benchmarks), 2024. doi:10.5281/zenodo.11186591.
[13] M. Preiner, H.-J. Schurr, C. Barrett, P. Fontaine, A. Niemetz, C. Tinelli, Smt-lib release 2024 (non-incremental benchmarks), 2024. doi:10.5281/zenodo.11061097.
[14] D. R. Cok, A. Stump, T. Weber, The 2013 evaluation of SMT-COMP and SMT-LIB, J. Autom. Reason. 55 (2015) 61–90. doi:10.1007/S10817-015-9328-2.
[15] D. Winterer, C. Zhang, Z. Su, On the unusual effectiveness of type-aware operator mutations for testing SMT solvers, Proc. ACM Program. Lang. 4 (2020) 193:1–193:25. doi:10.1145/3428261.
[16] A. Niemetz, M. Preiner, A. Biere, Model-based API testing for SMT solvers, in: SMT 2017, volume 1889 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2017, pp. 3–14. URL: https://ceur-ws.org/Vol-1889/paper1.pdf.
[17] G. Bury, Dolmen: A validator for SMT-LIB and much more, in: SMT 2021, volume 2908 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2021, pp. 32–39.
[18] D. R. Cok, jsmtlib: Tutorial, validation and adapter tools for smt-libv2, in: NFM 2011, volume 6617 of *LNCS*, Springer, 2011, pp. 480–486. doi:10.1007/978-3-642-20398-5_36.
[19] O. Guthmann, O. Strichman, A. Trostanetski, Minimal unsatisfiable core extraction for SMT, in: FMCAD 2016, IEEE, 2016, pp. 57–64. doi:10.1109/FMCAD.2016.7886661.
[20] G. Kremer, A. Niemetz, M. Preiner, ddsmt 2.0: Better delta debugging for the smt-libv2 language and friends, in: A. Silva, K. R. M. Leino (Eds.), CAV Proceedings, Part II, volume 12760 of *Lecture Notes in Computer Science*, Springer, 2021, pp. 231–242. doi:10.1007/978-3-030-81688-9_11.
[21] N. Macedo, A. Cunha, J. Pereira, R. Carvalho, R. Silva, A. C. R. Paiva, M. S. Ramalho, D. C. Silva, Experiences on teaching alloy with an automated assessment platform, Sci. Comput. Program. 211 (2021) 102690. doi:10.1016/J.SCICO.2021.102690.
[22] Soaibuzzaman, S. Kalantari, J. O. Ringert, On writing alloy models: Metrics and a new dataset, in: ABZ 2025, volume 15728 of *LNCS*, Springer, 2025, pp. 1–18. doi:10.1007/978-3-031-94533-5_5.
[23] A. Jovanovic, A. Sullivan, Right or wrong - understanding how users write software models in alloy, in: SEFM 2024, volume 15280 of *LNCS*, Springer, 2024, pp. 309–327. doi:10.1007/978-3-031-77382-2_18.
[24] Soaibuzzaman, J. O. Ringert, Formal methods playground smt-lib dataset, 2025. doi:10.5281/zenodo.15488370.
[25] Soaibuzzaman, J. O. Ringert, Smt-lib metrics replication package, 2025. Available from https://github.com/se-buw/smt-metrics.
[26] F. J. Pelletier, Seventy-five problems for testing automatic theorem provers, J. Autom. Reason. 2 (1986) 191–216. doi:10.1007/BF02432151.
[27] Soaibuzzaman, J. O. Ringert, Introducing github classroom into a formal methods module, in: FMTea 2024, volume 14939 of *LNCS*, Springer, 2024, pp. 25–42. doi:10.1007/

978-3-031-71379-8_2.
[28] D. Jackson, Alloy: a language and tool for exploring software designs, Commun. ACM 62 (2019) 66–76. doi:10.1145/3338843.
[29] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, S. Tonetta, The nuxmv symbolic model checker, in: CAV, volume 8559 of *LNCS*, Springer, 2014, pp. 334–342.
[30] S. Maoz, J. O. Ringert, Spectra: a specification language for reactive systems, Softw. Syst. Model. 20 (2021) 1553–1586. doi:10.1007/S10270-021-00868-Z.
[31] R. L. Huang, A. Monroe, P. de Halleux, S. Lerner, N. S. Bjørner, Z3guide: A scalable, student-centered, and extensible educational environment for logic modeling, CoRR abs/2506.08294 (2025). doi:10.48550/ARXIV.2506.08294.
[32] V. I. Levenshtein, Binary codes capable of correcting deletions, insertions, and reversals, Proceedings of the Soviet physics doklady (1966).
[33] S. Kalantari, Exploration of Specifications written by Novices: SAT, SMT, Alloy, NuSMV, Master's thesis, Bauhaus-University Weimar, 2024.