Efficient Grammar-based Exhaustive Testing of JSON Validation

YUTO TAKANO and AMELIA DOBIS, ETH Zürich, Switzerland

Additional Key Words and Phrases: grammar-based, exhaustive testing, ANTLR, JSON validation

1 ABSTRACT

Ubiquitous data formats such as JSON are crucial for enabling simple serialization, storage, and communication of important information. In this work, we devise a method for automatically verifying the correctness of JSON validators, such as the Node.JS library `jsonlint` and the Python 3 library `json`, in an exhaustive manner. This is done by employing a bounded grammar-based approach, which allows us to generate every possible JSON validation input up until our two controllable parameters: depth and width. Using these parameters, we generate fixed-width JSON grammars which are then handed to a program enumerator that generates all valid JSON strings based on the input grammar. This is done inside of a test harness that iteratively increases the two parameters following one of three possible path selection heuristics. We use the generated inputs to verify the validators using differential testing between the two aforementioned tools.

2 INTRODUCTION

Many applications rely on textual data formats in order to serialize, store, and communicate crucial information. JSON [2], being one these, is a widely used format for everything from web requests/responses to replacements for local databases. Its ease of legibility makes it a popular client-facing format, at the same time as it is simple to parse for computers. However, JSON input taken from clients can be misformatted or invalid, and applications often employ JSON validation [4] to make sure that the input is a valid JSON. These validators are expected to be trustworthy and output correct results. In order to guarantee that, we saw the need to figure out an appropriate and exhaustive way to automatically test these tools.

It is often challenging to write testing code in a way that does not require that itself to be tested. A solution to this never-ending testing cycle is to perform Differential Testing [5], where we compare the result of one JSON validator with that of another to determine whether a bug was found. In order to execute this, we need to thoroughly generate all possible inputs for the program to catch bugs, which is a technique also called Exhaustive Testing. We explored using a grammar-based exhaustive testing solution, which generates all possible input programs of a certain size using ANTLR [6] grammars, a format for specifying grammars in Extended Backus–Naur form (EBNF).

The main goal of our work is to efficiently test the Node.JS-based JSON validator `jsonlint` [1] by comparing it against Python's built-in `json` library (and its associated command-line tool). The main difficulty with applying grammar-based exhaustive testing to JSON is that the JSON format largely centers around unbounded structures such as arrays or objects. This requires care on search efficiency and methods of input enumeration. Naively using externally sourced grammars such as the one from the ANTLR repository can cause input enumeration to enter an infinite loop when enumerating arrays and objects. Excluding the EBNF grammar for these structures is not a viable option, given their central nature in JSON.

The principal contribution of this paper is a novel solution to this problem, which involves dynamically regenerating the JSON subset grammar by imposing an artificial but dynamically-adjusted limit on its breadth, thereby bounding

Authors' address: Yuto Takano, takanoy@ethz.ch; Amelia Dobis, adobis@ethz.ch, ETH Zürich, Zürich, Switzerland.

the input generation to terminate. We build upon a previous work on E.T., a research prototype for grammar-based exhaustive testing of SMT solvers, to extend it to JSON grammars. We implement two controllable parameters, width and depth, for controlling the size of the Abstract Syntax Tree (AST) used to generate JSON inputs to the two programs. Additionally, in order to explore the space of the new parameter efficiently to achieve our primary goal, we also implement three heuristics for grammar generation and input exploration (which we holistically call path exploration heuristics) for when the limits are adjusted dynamically during the testing campaign.

To our knowledge, this is the first work that uses a grammar-based tester with JSON grammar, and by that, we open the potential for applying grammar-based exhaustive testing to other grammars with unbounded structures.

This report is structured as follows: we first present our method in detail, including our failed initial solutions, we then evaluate our solution by presenting the results we obtained running a short testing campaign on our target JSON validator, we then end our report by presenting some related work and concluding on a summary of our project.

3 OUR METHOD

In this section we will present the methods we employed to achieve our goal. The intermediate solutions before the final method yield poor and non-functioning results, but we mention them here for completeness. The final method we present, based on dynamic grammar sub-set generation, is our working solution that will be used for the evaluation.

3.1 First Attempts

We initially started with a naive implementation, that adapted the existing SMT solution (in the E.T. prototype repository) so that it functioned on the full JSON grammar. We retrieved the grammar from the official ANTLR repository [8]. We quickly found a problem with this approach, which was that many parts of the grammar were written in such a way that E.T. stalled on them during input enumeration. Specifically, arrays and objects were defined using complex EBNF grammar utilising the Kleene operator, as seen in Listing 1. E.T. enumerates inputs using the state machine (the ANTLR Transition Network = ATN) that ANTLR generates from the EBNF grammar, but this state machine treats occurences of the Kleene operator as a single state, instead of using transitions to link the possibly infinite number of states it represents in reality. Since E.T.'s depth parameter controls the maximum number of transitions and is not aware of any presences of the Kleene operator, it stalls on them and endlessly enumerated arrays and objects in increasing sizes, without violating its internal depth constraint. As a solution to this, we rewrote the grammar to use recursive BNFs, as seen in Listing 2.

```
arr
   : '[' value (',' value)* ']'
   | '[' ']'
   ;
```

Listing 1. Original EBNF array definition as present in the ANTLR-provided JSON grammar, which makes E.T. not terminate because adding array elements does not increase the depth. Depth in E.T. is defined as the number of transitions it within the ATN, which stays constant during Kleene stars.

```
arr
   : '[' arrcontent ']'
   | '[' ']'
   ;
arrcontent
   : value
   | value comma arrcontent
   ;
```

Listing 2. Rewritten recursive BNF array definition, which makes E.T. treat an increase in the number of array elements as an increase in depth. comma is defined as having a single literal “,”, which is reused for the definition of JSON objects as well.

A similar phenomenon was seen with numbers and strings. With these however, we did not see any benefit to keeping their input space so large, so we modified the grammar to only use the string literals “a” and “b”, and the number literals “0” and “1”. This is the same approach taken by the original authors of E.T. for their SMT grammar, which also contains string and number literals.

Having solved the main problem that the E.T. prototype had with the full JSON grammar, we then developed upon this solution to enable a higher level of control, in terms of how our exhaustive tester explores the JSON validator input space. To do so, we needed to add another dimension to our parameter space, which we called `width`. This represents the number of elements allowed in JSON’s unbounded data-structures, such as objects or arrays.

We originally attempted to add the `width` parameter by dynamically pruning the program tree generated by ANTLR. The goal with this approach was to limit the size of array and object nodes, so that they could not contain more than a specific amount of direct children. To do so, the idea was to modify how E.T. transitions through the program ATN by conditionally terminating the input generation when a certain width was reached. This approach was limited by the fact that we could not use ANTLR’s runtime to differentiate between transitions that increase the tree’s width and those that increase the depth. Another limitation with this approach is that it requires the dynamic modification of dynamically generated input generation scripts. For these reasons, we opted for a higher level approach based on dynamic fixed-width grammar generation.

3.2 Final Solution

In order to solve the problems presented in the previous subsection, we decided to change our approach to limiting the width of the array and object constructs. To do so, we created a grammar generator which, given a width, generates a fixed grammar, as a subset of the full JSON grammar, which only allows for specific array and object widths. This allows us to limit the width without needing to dynamically modify the generated ANTLR scripts. The behaviour of our grammar generator is showcased in Listings 3 and 4.

```
arr
   : '[' value ',' value ']'
   | '[' value ']'
   | '[' ']'
   ;
```

Listing 3. The definition of a JSON array, generated using our grammar generator with `width: 2` as the parameter.

```
arr
   : '[' value ',' value ',' value ']'
   | '[' value ',' value ']'
   | '[' value ']'
   | '[' ']'
   ;
```

Listing 4. The definition of a JSON array, generated using our grammar generator with `width: 3` as the parameter.

Additionally, we present a test harness to incorporate the grammar generation into our testing pipeline. In order to more efficiently traverse the space that the new parameters create, we implement the automatic growth of both parameters (width and depth) directly inside of the test harness. We achieve exploration of the JSON grammar space and the JSON validator input space by growing the parameters in a way that follows one of three possible heuristics, which we call path exploration heuristics. These include `DFS`, `BFS`, and a random walk, and function as follows:

- **DFS:** Explores the depth parameter first by increasing it at the end of every enumeration, until the user-defined maximum depth is reached. After this the depth is reset, and the width is incremented.

- **BFS:** Explores the width parameter first by increasing it at the end of every enumeration, until the user-defined maximum width is reached. After this the width is reset and the depth is incremented.
- **RANDOM:** At every end of the enumeration, we randomly decide whether to increase the width or the depth until a limit is reached on either one, in which case the other parameter is increased. This gives our exploration a more organic evolution and we expect this method to be most effective at generating bug-triggering inputs.

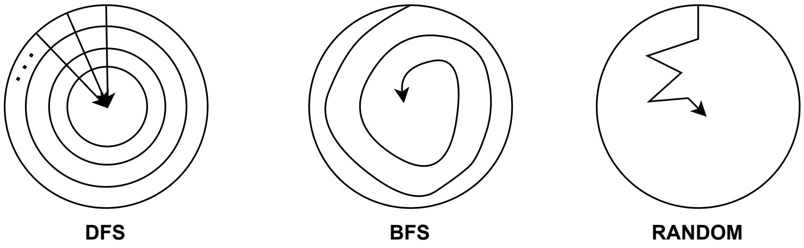
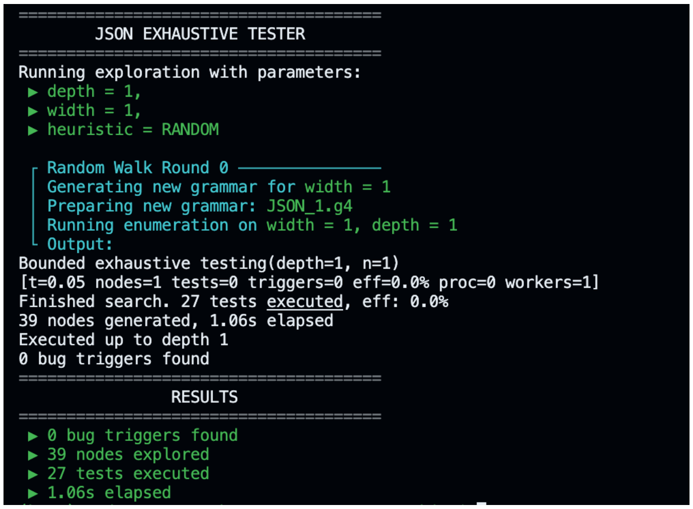


Fig. 1. Illustration of the three heuristics we implemented for input-space exploration. Here an increase in the width is illustrated by a move in the tangential direction, and an increase in the depth is illustrated by a move towards the center.

Figure 1 illustrates how our different heuristics explore the input space. We must note that the grammar generator, as well as the grammar preparation which generated the ANTLR scripts, is called every time the width is increased.



```
======================================
        JSON EXHAUSTIVE TESTER
======================================
Running exploration with parameters:
 ▶ depth = 1,
 ▶ width = 1,
 ▶ heuristic = RANDOM

 ┌ Random Walk Round 0 ───────────────
 │ Generating new grammar for width = 1
 │ Preparing new grammar: JSON_1.g4
 │ Running enumeration on width = 1, depth = 1
 └ Output:
Bounded exhaustive testing(depth=1, n=1)
[t=0.05 nodes=1 tests=0 triggers=0 eff=0.0% proc=0 workers=1]
Finished search. 27 tests executed, eff: 0.0%
39 nodes generated, 1.06s elapsed
Executed up to depth 1
0 bug triggers found
======================================
                RESULTS
======================================
 ▶ 0 bug triggers found
 ▶ 39 nodes explored
 ▶ 27 tests executed
 ▶ 1.06s elapsed
```

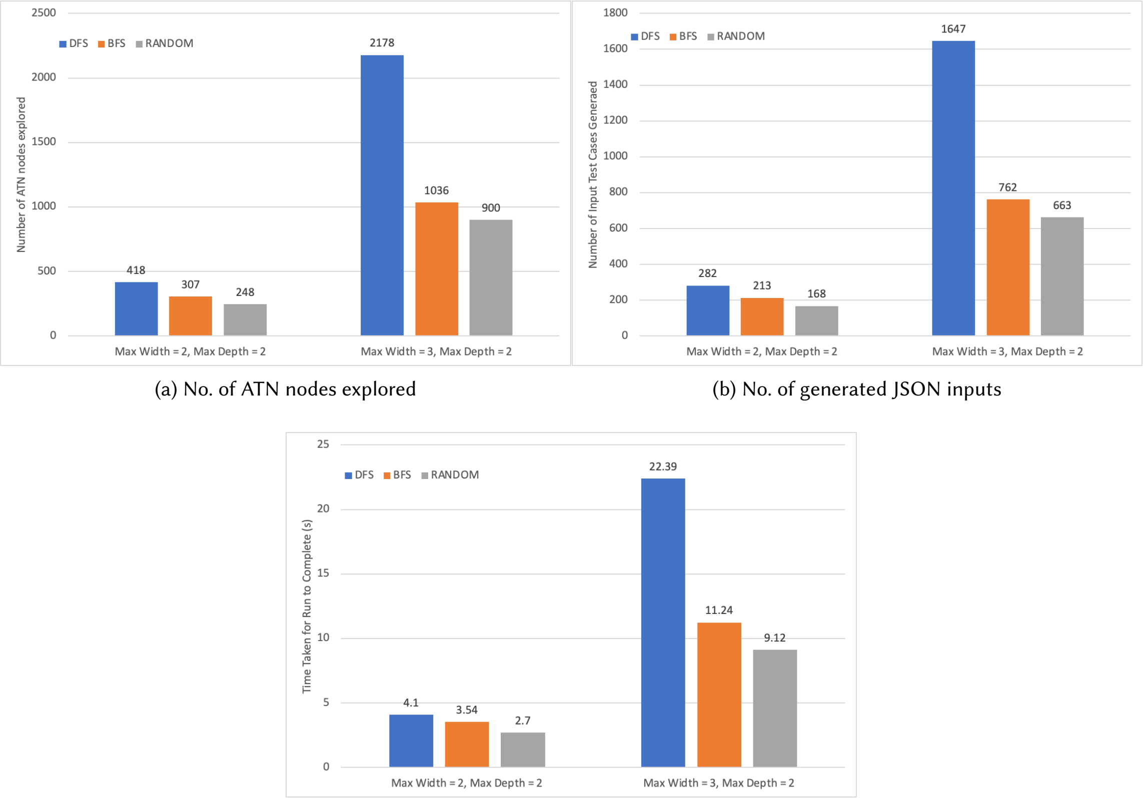
Fig. 2. Example of a output produced by our grammar-based exhaustive testing tool.

In order to test our solution, we modified our test harness to capture bug-reports from the enumeration script. Figure 2 shows an example of what our test-harness outputs during a test-campaign. The number of ATN nodes explored, the number of JSON test inputs generated and executed, and the total time elapsed are sums across the multiple rounds of executing E.T, and may contain duplicates.

4 OUR RESULTS

The solution presented in this work allows us to perform grammar-based exhaustive testing on a complex grammar such as JSON. The solution we presented can be used to adapt any existing grammar containing unbounded data-structures, and allow them to be used for grammar-based exhaustive testing, which was previously not possible. The proposed introduction of a new parameter to control width also enables a more controlled exploration of the generated input space.

Figure 3 presents the results of comparing the efficiencies of the different path exploration strategies that we propose in our work. We ran the **DFS**, **BFS**, and **RANDOM** path exploration heuristics with our test harness, with two sets of parameters: once with maximum depth and width both set to 2, and once with the maximum width increased to 3. The first subfigure to the left shows the number of nodes explored by E.T. runs within the ATN for the grammar, the second shows the number of generated JSON test cases, and the third shows the time taken in seconds for the full run to complete. It is important to note that regardless of the strategy performed, the same total input space is explored in the end. Therefore, a lower number of duplicated test cases and re-explored ATN nodes signifies a more efficient strategy.



(a) No. of ATN nodes explored

(b) No. of generated JSON inputs

(c) Time taken for campaign runs

Fig. 3. Summary of the results obtained from a small test campaign, comparing differences across the three heuristics presented in our work, and across different choices in parameters.

Results shown in the figures clearly demonstrate that the **RANDOM** path exploration heuristic performs the fastest, with the lowest number for all metrics across both choices of maximum bounds. This result makes sense intuitively, as it is less likely for the strategy to generate many duplicate test cases compared to DFS or BFS when dynamically adjusting its parameters. The behaviour shown in these figures demonstrate our proposed solution's viability and effectiveness in exploring the input space of JSON grammars efficiently.

In order for our work to find any potential bugs, we think that a much larger test-campaign is required, given the highly tested nature of these JSON validators. We ran another test campaign with the **RANDOM** heuristic and a maximum width and depth of 10. This ran for over three days and did not produce any bugs. Any larger campaigns will be left up to future work, as our main goal for efficient grammar-based exhaustive testing has been achieved.

One further discussion that may be had, is related to our modification of the string and number definitions in our generated grammars. These limit the scope of the strings and numbers that may be generated in our inputs. We think that this is necessary in the scope of grammar-based exhaustive testing, as the input space must be finite in order for our exhaustive generation to terminate. Additionally, we think that most bugs in our target programs are more likely to come from complex structures in the inputs and not necessary from the contents of the values of the input JSON. This type of large content exploration is a problem that may be more suited for a fuzzing-based solution, rather than an exhaustive testing-based one.

5 RELATED WORK

After some research, we found that grammar-based exhaustive testing is a very new topic, that hasn't been explored much outside of the E.T. prototype provided by the lab. There is however, some existing work on simple grammar-based test generators. Gramtest [9] is a grammar-based test case generator, which, given a BNF grammar, can generate a certain, user-defined, number of test cases of a random program tree depth in that grammar. This is useful when wanting to generate simple tests to more generally test a design, but it is not exhaustive. On the other hand, our goal is to test every possible program trees of a given depth, thus giving a less general but stronger guarantee of code correctness.

Fuzzing is a field that closely resembles the work presented in this paper. Fuzzing is a type of black-box testing which randomly explores the input-space of a program under test (PUT), often by generating inputs in a mutation-based fashion based on a certain metric such as a coverage metric. A popular, and well implemented, coverage-guided mutation-based fuzzer is the American Fuzzy Lop (AFL) [3]. This fuzzer simply requires the user to wrap their PUT in a custom test harness which converts the fuzzed inputs into a format that works for their specific PUT. AFL using edge-cover, a more sophisticated version of path coverage, to guide the mutation of its input seeds to generate inputs that are more likely to explore parts of the input space that maximize that metric. While this is similar to our solution, it takes a different approach. Our solution is inherently a grey-box solution, as it requires sophisticated knowledge about the import format (although not the internals of the validator), while fuzzers do not care about the semantics of the input and operate at the level of bit-strings. Additionally our method aims to explore the entirety of a sub-set of our input space, while fuzzing more generally randomly traverses the space.

We do however think that our work would work well in combination with fuzzing. Specifically, we think that fuzzing can be used to generate the internal literals, such as strings or numbers, which we limited in our solution. This would allow for bugs based around the validation of badly-formatted literals to be triggered, which is currently not possible with our more strict literal generation.

We would also like to briefly mention other solutions such as Ying-Yang [11], which is a novel tool for stress testing SMT solvers using a mutation-based testing approach. This tool introduces the concept of semantic fusion to generate semantically coherent mutants based around the merger of two previous inputs. This allows for correct, yet complex, inputs to be generated, allowing for the efficient triggering of SMT solver bugs. This tool is presented in conjunction with another solution for type-aware operator mutation called OpFuzz [10]. This solution allows for semantic fusion to be augmented with an operator mutation that takes into account the operands of the original operator. This allows for a much more complex mutation to take place, increasing the distance covered within the input-space. Using these tools, the authors were able to find several bugs in Z3, a popular and widely used SMT solver. This solution is closer to what we present in this work than fuzzing is, as it attempts to exhaustively stress-test the PUT in an efficient manner, also using grammar-based input generation. The main difference is in how the input-space is explored. While we explore the space using our high-level parameters controlling the structure of the generated inputs, Ying-Yang with Opfuzz use generates new inputs based on intelligent mutations and fusion of two previous inputs. While some of its works on fusion may be effective in JSON due to its highly composable grammar, the lack of operator expressiveness limits the viability of these approaches. It also requires more specific tailoring to the JSON grammar, which bounds the adaptation of our solution to other input languages.

Finally, we want to briefly mention `creduce` [7], which is a tool that enables efficient test-case reduction. This was originally meant for reducing C compiler bug triggers, however it has been shown to also be useful for reducing other structured inputs in different languages. While this doesn't directly relate to our solution, it can be used to reduce the generated JSON inputs found by our exhaustive tester to make the bug more reproducible.

6 CONCLUSION

In this work, we presented our solution to enabling the use of grammar-based exhaustive testing on unbounded grammars such as JSON. To enable this, we proposed a method based around the generation of fixed-width grammar subsets to limit the scope of the exhaustive code generation. Our solution augments the E.T. research prototype by introducing a new dimension to the previously depth-only based approach. This dimension, called width, limits the scope of unbounded data-structures such as objects or arrays. We then used our solution inside of a test harness with explores parameter space using one of three path selection heuristics. Doing so enabled us to run a successful exhaustive testing run using differential testing verify the correctness of two JSON validators. In the future, it would be beneficial to launch a large scale testing campaign using our novel solution, in order to potentially find bugs in the selected validators. This was left as future work due to time constraints.

REFERENCES

[1] Carter, Z. Json lint. https://github.com/zaach/jsonlint.

[2] Crockford, D. Introducing JSON. https://www.json.org/json-en.html.

[3] Fioraldi, A., Maier, D., Eissfeldt, H., and Heuse, M. AFL++: Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)* (Aug. 2020), USENIX Association.

[4] Jackson, W. *The JSON Schema: JSON Structure Validation.* Apress, Berkeley, CA, 2016, pp. 21–29.

[5] McKeeman, W. M. Differential testing for software. *Digital Technical Journal 10*, 1 (1998), 100–107.

[6] Parr, T. J., and Quong, R. W. Antlr: A predicated-ll (k) parser generator. *Software: Practice and Experience 25*, 7 (1995), 789–810.

[7] Regehr, J., Chen, Y., Cuoq, P., Eide, E., Ellison, C., and Yang, X. Test-case reduction for c compiler bugs. *SIGPLAN Not. 47*, 6 (jun 2012), 335–346.

[8] Repository, A. O. Full JSON Grammar. https://github.com/antlr/grammars-v4/blob/master/json/JSON.g4.

[9] Sharma, A. GramTest. https://github.com/codelion/gramtest.

[10] Winterer, D., Zhang, C., and Su, Z. On the unusual effectiveness of type-aware operator mutations for testing smt solvers. *Proc. ACM Program. Lang. 4*, OOPSLA (nov 2020).

[11] Winterer, D., Zhang, C., and Su, Z. Validating smt solvers via semantic fusion. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2020), PLDI 2020, Association for Computing Machinery, p. 718–730.