Static analyzer debugging and quality assurance approaches

Maxim Menshikov
Department of Software Engineering
Saint Petersburg State University
Saint Petersburg, Russia
info@menshikov.org

***Abstract*—Writing static analyzers is hard due to many equivalent transformations between program source, intermediate representation and large formulas in Satisfiability Modulo Theories (SMT) format. Traditional methods such as debugger usage, instrumentation, and logging make developers concentrate on specific minor issues. At the same time, each analyzer architecture imposes a unique view on how to represent the intermediate results required for debugging. Thus, error debugging remains a concern for each static analysis researcher.**

In this paper, our experience debugging a work-in-progress industrial static analyzer is presented. Several most effective techniques of constructive (code generation), testing (random test case generation) and logging (log fusion and visual representation) groups are presented. Code generation helps avoid issues with the copied code, we enhance it with the verification of the code usage. Goal-driven random test case generation reduces the risks of developing a tool highly biased towards specific syntax construction use cases by producing verifiable test programs with assertions. A log fusion merges module logs and sets up cross-references between them. The visual representation module shows a combined log, presents major data structures and provides health and performance reports in the form of log fingerprints.

These methods are implemented on a basis of Equid, the static analysis framework for industrial applications, and are used internally for development purposes. They are presented in the paper, studied and evaluated. The main contributions include a study of failure reasons in the author's project, a set of methods, their implementations, testing results and two case studies demonstrating the usefulness of the methods.

***Index Terms*—static analysis, debugging, goal-driven random test case generation, code generation, log file analysis, visual representation.**

I. Introduction

Software engineering is ailing from quality assurance issues. Many approaches are aiming at achieving runtime error absence (reviewed in Section II), but logical correctness is not trivially reachable even if it is guaranteed statically. This is often the case because the correctness is not automatically derived from internal consistency.

The static analysis is aimed at verification of the real-world problems written in the form of programs. Analyzers follow the generic debugging and quality assurance trends, however, there is specificity which should be taken into account [1]. The input program undergoes several transformations, and each of them has a significant impact on the validity of the final result. Moreover, several transformations, when combined, may have cumulative effects. The issues during transformations are usually *not* runtime errors, which are easy to narrow down using traditional methods, but rather logical defects. Since transformations are unique products of each static analyzer, quality assurance is the sole responsibility of the analyzer's author.

In the Equid project [2], the author had several observations. First, feature testing looked biased towards the developer's interpretation: there is a tendency to test constructions in a way they are used by the person. Second, thorough bugs may be hidden behind multiple abstractions and appear unexpectedly as analyzer grows. Third, a *significant amount* of errors could have been found if there was a simple measure indicating that action was required. Inserting heuristics for every aspect of a large software project log is barely achievable (consider limited system resources when making such advanced logging), so log analysis is the foremost goal for analysis debugging. Fourth, contracts and formal requirements undeniably contribute to the quality of the product, however, they cover integrity and consistency rather than the absence of logical issues. In that sense, static analyzers have no specificity. And the last, static analyzers have a rare environment with little to no requirements for issue reproduction. For example, reproducing the issue in network router software may require days and months just to repeat the pattern. That makes it possible to increase logging verbosity until the issue is detected, so the developer may put efforts into making logs as informative as possible. With that in mind, the paper suggests an approach for increasing the visibility of issues and/or reducing the likeliness of bugs, based on *code generation*, *log file improvements*, *goal-driven random test case generation*, and *visual representation*. These four methods make up the author's *static analysis debugging and quality assurance approach*. The study starts with the description of third-party approaches (Section II), continues with the key issue sources identified by the author (Section III), the method is presented in Section IV and evaluated in Section V.

Motivation. The debugging field concentrates on runtime issues and doesn't answer the question how to debug deep logical issues in the static analyzer. Thus, solving this problem and employing the right methods brings many practical benefits, such as improvements in stability, reduced risk of problems after implementing new features, a faster development pace.

Novelty. The suggested approach covers a significant

amount of issues found in the real-world analyzer's development. The code generation stage is enhanced with the post-processing phase in which the internal use cases are loosely verified, making integration of new objects faster. The log fusion adopts hypertext-like approach, making the output more linked and indexable, allowing for better search and filtering. To the knowledge of the author, the logging had never been integrated with the hypertext. Random test case generation usually covers trivial input data or just *compilable* programs, but in this study, it produces programs with specified verification goals, which is an improvement over completely random programs. The visual representation is usually aimed at the visualization of control flow graphs, but in this paper, it is intended for data structures and internal health/performance reports.

Main contributions. Main contributions comprise the study of the issues in the Equid project, four debugging and quality assurance techniques, implementations for 3 of the mentioned platform-independent methods, the testing results, and two case studies presenting the usefulness of the method.

II. Related work

Most works are about analyzing complex systems with static analyzers rather debugging analyzers, however, there is a study of defects in static analyzers [1], which gives useful insight to the opinions of other researchers. According to the paper, visualization and handling of intermediate results is still not satisfactory for the most developers, as well as handling of data structures. While the author hadn't synchronized with this research when the development of the analyzer started, a significant match with the practical experience had been determined.

There are well-known complex tools for debugging of complex systems, e.g. GDB [3], supporting all major Central Processing Unit architectures. The LLDB [4] is an LLVM-based GDB analogue, which aims to provide reusable infrastructure. Of course, there is a number of language-specific tools, but GDB and LLDB are among the most universal ones (e.g. UndoDB [5], which is based upon GDB, can be used to debug Java and Go applications). Such debuggers provide a way to debug tools in the direction from the beginning to the end and support analyzing core files.

The reverse direction debugging (or "omniscient debugging") is covered by GDB itself; the Mozilla's RR [6], the record and replay framework; the Undo Debugger [5], which is claimed to be one of the first commercial reverse-debuggers [7]. The reverse-debugging tools are useful for runtime errors, but the majority of the defects, at least in our project, don't fall into this category, so the usefulness is limited.

Random program generation has been first shown in [8], this method then evolved into CSmith [9] framework, which had extremely successful applications to industrial compilers. An interesting result was achieved in the paper [10], in which the author tries to avoid generating dead code by using all the temporary computations for the final result. Intel has also prepared its random program generator [11] capable of triggering compiler optimization bugs, with over 140 defects found in LLVM and GCC. The research [12] covers Orange4 random program generator with an idea of equivalent transformations — which is, by the framework, similar to goal-based generation from our study, however, the generated programs are completely random without any goals set. The MicroTESK [13] project generates test programs for various microprocessors (ARM, MIPS, RISC-V and other architectures), however, it is aimed at a lower level than the tool described in our research.

The static analysis visualization is a highly specialized topic, only applicable to concrete tool developers. Still, the authors of the paper [14] explored the ways to animate the static dependence analysis, and the result is very different from standard still visualizations. The closest generic solutions so far are brought by code visualization tools, e.g. Sourcetrail [15], CVSscan — code evolution visualizer [16], which are at least capable of visualizing programs. The negative part is that local intermediate representations aren't supported in such tools. Graph-based program evolution is estimated by GEVOL [17] project, which is important for tracking defects produced by specific authors and functions. Software performance evolution had been examined in [18], as a list of functions along with performance results.

The logging is well-investigated in [19], with useful insight about the usage of logging in open-source projects. The study [20] concentrates on characterizing when do developers log the information. Additionally, researchers performed the survey on how to improve the logging. Three suggestions are relevant to our study: log filtering, categorization and analysis/visualization.

The current methods for information retrieval are mostly for natural languages. For unstructured data, it implies the usage of n-grams, machine learning and other text search methods [21]. One of the examples is DeepLog [22] system, which aims to find anomalies using deep neural network model utilizing Long Short-Term Memory (LSTM). Paper [23] reconstructs control flow graph of the distributed system to find anomalies. Anomaly detection in computer systems using decision trees is performed in [24]. Those are valid methods for unstructured log analysis, however, these methods are more suitable for malware action detection on sets of third-party applications, while the study concentrates the single application development, where defects are detected by the developer. Our log handling approach is closer to classic hypertext [25].

III. Sources of issues

Throughout 2019 author had been analyzing defects for the issues in more than 1500 commits in the closed static analysis project. Key issue sources had been identified:

- *Missing support for the specific syntax/intermediate representation (IR) construction in submodules.*
- *Small differences in implementations for repeating parts (classes).*

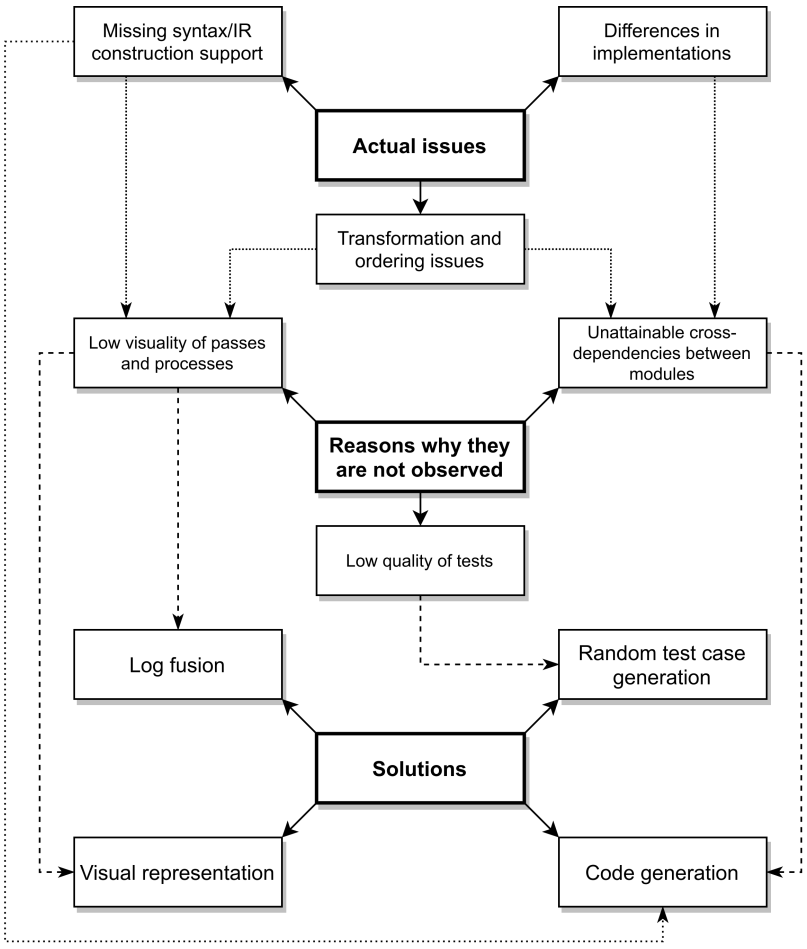

Fig. 1. Sources of the issues, reasons why these issues are not observed and their solutions

- *Transformation and ordering issues*.

The developers fix such issues promptly if they are observed, but they are not trivial to find. The author had determined the following three main reasons:

- *Low visuality of the transformation passes and the development process*. The developer sees the input, the result, but intermediate transformations might be incorrect. That might lead to false testing results.
- *Unattainable cross-dependencies between modules*. The engineer creates a new feature and edits modules to integrate it, yet different parts might be broken.
- *Low quality of tests*. This is the main reason why non-trivial issues are often not found. For example, if tests verify separate handling of `if` and `switch` constructions, there might be problems in their combinations if their implementations have interchanged code.

The following ideas were evaluated to make these issues more visible:

- Automatic generation of major cross-linked data structures to avoid unattended defects.
- Making an interface for viewing log that would detect cross-references between pass logs, visualize the steps, the internal error rate. That's similar to the approach suggested by the paper [1].
- Improving test cases: make random tests that would be more representative than those written by hand.

No single solution can be engineered to solve these problems. All typical quality assurance techniques were used in the project, which is not extraordinary, considering that the ultimate goal for the static analyzer is to promote good software engineering practices. The *recipes* suggested by this paper are extensions of the typical methods, designed specifically for static analyzers. In section V-E, we predict the classes of the programs which might also benefit from the approach.

Fig. 1 summarizes the findings, matches the issues with the reasons why they are not found and the solutions.

In the next section, the author's solutions to these problems are demonstrated, along with their implications.

IV. Methods

As stated in the introduction, the paper aims to develop methods assisting in debugging of logical issues in static analyzers. Unlike other kinds of issues, these issues do not cause runtime failures and thus are often unattended. The form of assistance varies for every method.

A. Code generation

As the code base grows along with the number of syntax constructions, it gets important to ensure that repeatable fragments are written strictly and concisely, not breaking the stability of the whole program. Code generation reduces the risk of adding logical mistakes by producing modules with high integrity and compatibility. This technique itself is not new and can be applied to any software project, however, the analysis had shown that verification of the usage is vital as static analyzers introduce transformations between different models. Also, the plurality of the models implies the need for the model instantiation for different occasions. This way, the risk of logical issues in this domain reduces due to common code generation base.

Thus, in the case of Equid, there are three major considerations for the code generation:

- **Enumerations**. When a new element is added to the enumeration, there is a high chance that dependent enumerative functions are invalidated. A mechanism that indicates the expected use of enumeration within the source code had been developed.
  For example, if a selected code needs to handle just specific enumeration elements, then, before using the enumeration `EnumerationName`, the developer may indicate the all-variant usage of the enumeration:

  ```
  core_indicate_use(EnumerationName,
    CoreEnumUse::AllVariants)
  ```

  If the code intentionally uses just selected enumeration values, then the developer indicates it:

  ```
  core_indicate_use(EnumerationName,
    CoreEnumUse::Selected)
  ```

  The post-code generation phase verifies whether all or selected elements are used. This is a cheap yet effective mechanism to ensure that all enumeration values are processed. Noteworthy is that GCC has a switch-checking approach (`-Wswitch-enum` or `-Wswitch`), which can be used with `#pragma GCC diagnostic push` for the region selection, however, GCC's approach is compiler-dependent.
- **Repetitive classes**. A big software project doing many data transformations inevitably gets many classes representing nodes participating in different analysis passes. In most cases, nodes have a similar structure. The rule of thumb is that classes that can be described declaratively should be written this way. For instance, in the project, we cover not only syntax structures but also data classes, language semantics.
- **Multi-model data**. Input data sets may be cross-linked and can be used for different purposes. There should be a way to interpret the data differently.

1) The practical implementation: To get code generation, the author had written a standalone C++ tool[1]. The input is in customized YAML[2] format. Several models were employed: enumerations, expressions and intermediate representation commands. The tool produces a not strictly formal syntax tree, which is transformed into the real code file.

The *Enumeration* model has the following format:

```
type: <A complete type, can have
  reference to a different
  namespace>
clean_type: <A type name without
  namespace references>
namespace: <Namespace in which
  enum is introduced>
dont_create_enum: [false/true]
header:
  - <extra header entry>
  - ...
field:
  - name: <Field name>
    token: <string representation>
  - ...
unknown: <Unknown field for
  default alternative>
mapping:
  - name: <MappingName>
    from: <Source type>
    to: <Target type>
    unknown: <default result for
      unmapped values>
    map:
      - from: <Source value>
        to: <Destination value>
```

The product is a C++ `enum class` with the given name, a set of fields, a mapping function between `enum` and `std::string`, and several custom mappings.

The *Expression* model borrows many ideas from enumerations, but it is slightly more oriented towards expression model:

```
type: <Short expression type>
value_type:
    direct: <fixed type>
    indirect: <expression to borrow
      type from>
constructor:
  - name: <Internal constructor name>
    parameter:
        - <list of parameters, named as
          members>
        - ...
```

[1]https://github.com/maximmenshikov/eq_codegen. This version is limited regarding supported syntactical constructions, only intended for demonstration.

[2]https://yaml.org/

```
  - ...
member:
  - name: <Friendly member name>
    internal: <Private member name>
    type: <Member type>
    default: <Default value>
header:
  - <extra header entry>
  - ...
operation:
  <enumeration-like list of operations>
function:
  - name: <custom function name>
    signature: <function's signature>
    body: <function's body>
override:
  ToString: <Code that will return
 entry's string representation>
```

This is a short description of expression model, in fact, the model has more parameters for handling minor cases, e.g. children handling, whether the entry is LValue, and a more sophisticated return type handling.

The source for the command model (not presented in the paper) is provided at GitHub[3].

B. Log fusion

A typical log is a flat file with thousands of lines. Developers often struggle to find an optimal balance between verbosity and conciseness [19], but as mentioned in Section I, static analyzers are in the unique position in which running debugging versions is possible without complete reproduction of the environment. In that case, it is possible to make tools as verbose as possible. This method does not find logical issues by itself, but, in conjunction with the fact that logs present a significant part of intermediate objects being the inputs and the outputs of transformations, it assists in making the log analysis quicker.

An approach based on the following two concepts is suggested:

1) **Module-specific logs**. Each module writes to its log, however, the core keeps track of unique timestamps for each log entry. Thus, it is possible to view separate logs if required or to view combinations of logs. This approach has important implications from a practical perspective. First, it decreases the resources required for categorization and search. A single log would have to be parsed from the beginning to the end — which would inevitably mean a slowdown. Second, the reader gets an opportunity to select points of interest and completely avoid unrelated parts.
2) **Internal bookmarks/tags**. Each object (in our classification, a resource, a fragment or a virtual machine command) has unique ID within the object pool. Objects are referenced by a tuple $(name : id : type)$, and each action on the object is bookmarked by a tuple $(id : type : action)$. The log viewer allows quick navigation between these objects and by that reduces the cognitive load on log reader. This implies that the log is less of *flat* structure, but rather a technical document, more oriented towards understanding.

1) Implementation: Separate logging is very implementation-specific and novelty-free. The analyzer simply opens a number of streams and provides a debug context object allowing for access to all of these streams.

The bookmark approach requires attention to the implementation of Uniform Resource Name (URN) producing methods. URN must be both readable and short, since reading log files in plain text is still an option. In the author's implementation, the URN doesn't adhere to RFC 2141[4] to save space. The practical URN grammar is presented in the corresponding GitHub repository[5].

C. Goal-driven random test case (program) generation

Order of actions and bias towards specific use case for syntax structures is the major source of the issues during the development. They are not detectable mainly because preparing a reasonable selection of tests proving the issue source is hard. The random test program generation is helpful in such cases due to its ability to test software against large volumes of varying input data. In result, not only logical issues are detected, but also a number of runtime issues, as seen in compilers.

A random program generator creating a set of tests with the following properties was prepared. First, all tests have one goal defined and asserted in `main()`, this is unlike other random program generators, which produce *compilable* programs without assertions. Second, all of the test cases are identical in terms of final results.

Our algorithm revolves around the idea of a *verification goal*. The straight-forward algorithm for `main()` is as follows:

1) Create a function `main()` with empty block.
2) Insert a randomly named variable. Let it be x.
3) Assume a random goal as a target value for variable x. Let it be a.
4) Generate a random block or a random function returning a or modifying the input pointers so that x is receiving a.
5) Insert an assertion $x = a$.

A random block or a random function is generated accordingly. The goal is transformed into a final statement (e.g. `return a` or `x = a`, based on whether the block or a function is being generated).

1) The statement is transformed into a different statement based on the random value (which chooses the next operation from the list below):

[3]https://github.com/maximmenshikov/eq_codegen_models

[4]https://tools.ietf.org/html/rfc2141

[5]https://github.com/maximmenshikov/eq-urn-grammar

a) If the statement isn't a block, it is transformed into a block of statements.
b) If the statement is a block, then a new variable is added to a block.
c) If the statement is a block, then a random (not goal) variable is assigned.
d) The statement is rolled into the `if-else if-else` statement, where the goal is put into the first `if` *then* statement. For the rest of the branches, the false goal is generated and rolled into a random block.
e) The statement is rolled into the `switch/case` statement, where the goal is put into the first `case` body. For the rest of the cases, the false goal is generated and rolled into a random block. The false cases are randomly ended with `break`.
f) The statement is rolled into `for` or `while` loop. In the author's implementation, these constructions were of minor interest, so they were implemented trivially, similarly to Orange4 [12] same-assignment.
g) ... the rest of constructions.

After executing the first two procedures, the outcome is a valid program which must be well-parsed (syntactically correct by construction), should be analyzable and, if compiled, must satisfy all assertions. To get a set of programs, the shuffling is performed on all constructions allowing for it (the showcase is for `if` and `switch`). `if` branches are shuffled if the condition expressions are not intersecting (i.e. swapping branches doesn't change the semantics), `cases` are switched based on `break` existence. All-**break** switch-case statements can be shuffled completely.

The practical implementation is located in corresponding GitHub repository[6].

D. Visual representation

The visual introspection assists in finding logical issues by using a graphical view. A number of issues, especially those involving formulas and type conversions, are not distinguishable in textual forms. The following directions were in the focus of the research. First, making passes visually observable. Second, provide reasonable health reports for performance figures, error occurrence rates.

As for the first goal, our main intention was to make an internal Virtual Machine intermediate representation (IR) viewable. The IR can be represented using a tree view since branches may contain child entries. To simplify interface development, a plain tree view from Qt[7] framework had been used. Clicking on any IR command brings a list of related parts: a simplified SMT representation, a simplified abstract interpretation view (Fig. 2). At the moment of writing, the IR representing module is not directly linked to the debugger e.g. via GDB/MI interface [26], but the project's debugging framework is capable of generating the commands to set the breakpoints at the specific execution points (like IR transformation phases) for the RR [6] replaying.

[6]https://github.com/maximmenshikov/eq_fuzzytest
[7]https://www.qt.io

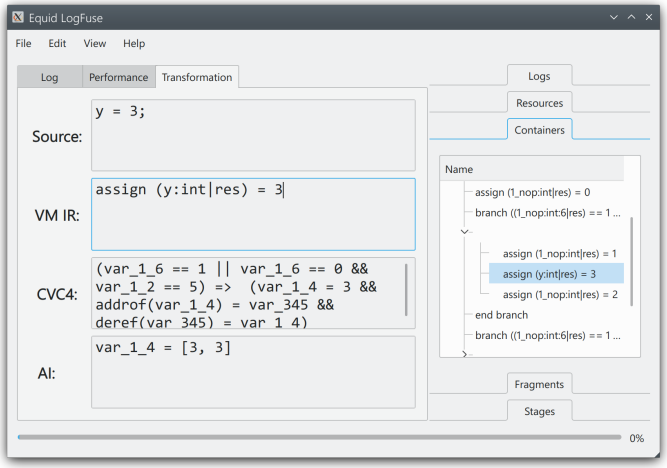

Fig. 2. The transformation view of IR commands

As for the second, the following formulas had been used to prepare a chart. The first formula is trivial. Consider t_i a start time for $i-$ verification phase, where $i \in [1, n]$, and t' is the final execution time. If, for simplicity, $t' = t_{n+1}$, then durations are calculated accordingly: $\Delta t_i = t_{i+1} - t_i$, where $i \in [1, n]$. However, this computation gives a very rough approximation of internal time spans.

Module log separation provides two other useful empirical formulas. One formula is based on *log size*. Consider that the logging is more or less uniformly spread around the modules. In that case, module log size is a simple profiler for the module execution times, without an actual profiler running.

The other formula requires building a time series. The complete execution time t' is collected, and it is divided up to 40 chunks: $\Delta t_i = \frac{t'}{40}$, where $i \in [1, 40]$. The graph with timestamp occurrence frequencies for each Δt_i group is built for every module. The result of the implementation can be roughly described as a digital fingerprint for the execution (Fig. 3). The source code is located at GitHub[8].

Concluding, these three formulas provide insight into performance. They consider (a) total phase time, (b) total time spent in the module, (c) time distribution. They are useful if logging invocation distribution is uniform.

V. Evaluation

The implementation of the proposed approaches had been tested in the Equid project. For random test case generation, it is common to measure how many errors of which severity had been found using the technique, that's what determines the real usefulness of the method.

For log fusion, the developer's time to find an error was continuously monitored. This way is not accurate since the issues might differ at every testing instance.

[8]https://github.com/maximmenshikov/loghealth

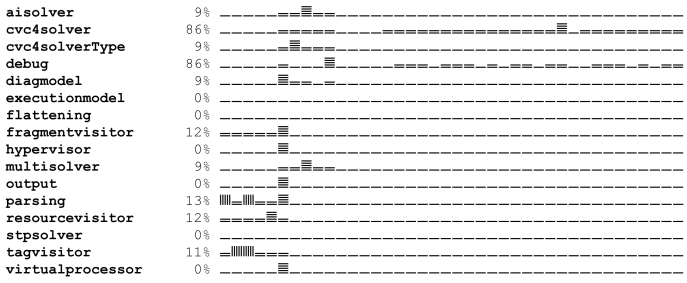

Fig. 3. The "fingerprint" of the execution

For code generation, we investigated the time needed to bring up a new syntax structure. We were lucky to have had a syntax processing refactoring right after tool bringup due to new language requirements to the static analyzer (which are out of the scope of the paper), that made the evaluation easier.

Visual representation isn't trivially testable. The only evaluation the author could do is the subjective contribution to the issue investigation.

A. Random test case generation

After introducing the random test case generation, the author observed the decrease in a number of issues with both existing and newly added syntax constructions. Several defects were found, which could be classified as logical, performance, ordering and runtime issues. Logical issues comprise of verification-breaking issues, not related to ordering (which is a separate group). Performance issues are due to slow handling of syntax constructions. For this group, the time required for the execution of generated source had been evaluated and tested for sanity. The time twice larger than the empiric average for the syntax construction had been considered an error. Ordering issues appear during transformations when specific elements can't be trivially reordered in a destination form. When resulting messages differ for reordered sources, the case is considered a failure.

The results of testing are provided in Table I. In total 10 issues had been detected during the evaluation, 6 of them had high severity, and 4 of which had medium severity. The author considers the method applicable to finding mistakes in static analyzers but needs significant improvement to cover all language features. However, it is hard to judge the usefulness for compilers because no investigation had been done.

B. Log fusion

The logging engine can be practically evaluated only by checking the time to resolve a typical issue. The evaluation time (1 month) had been divided into two periods, in one period no logging features were used during issue-resolving, the other period is characterized by intensive usage of log fusion. The time to resolve the typical issue reduces twice or thrice (see table II). The improvement rates are 1.92, 5, 3.3, 1, which result in an average of 2.8 among these test groups. These results also indicate that the method is feasible for static analyzer development tasks, however, the effect is vastly different for different groups (and, supposedly, tests). But, at least, the method doesn't make the process slower.

C. Code generation

The code generation covers a significant part of the process of adding a new syntax structure. For the project, class support had to be implemented again due to customer's requirements & changed the project's infrastructure. The result is as follows. It was determined that adding class support has taken 7 times less time than the same feature several years before this test. Moreover, it was noticed that previous attempts had a month-long trail of commits revising the architectural modules and minor issues, however, the attempt after introducing code generation didn't have so many visible effects. The representability of this empirical test is very low: after all, the project has become more mature over years, however, it is hard to perform a more fair comparison to see the improvement.

D. Visual representation

For the visual representation, the low improvement for maintenance development phases was observed. The reason is that no developer or tester would ever look at visual reports for everyday testing. However, it is profitable for the active development phase. At least 2 performance issues were discovered using the performance chart implemented in our log viewing tool. They were related to different timings between stages, while the overall result was about the same: this situation happened due to substantially simplified processing of structures due to all of them getting the same visibility level. The deeply nested test had much longer table lookup time with much shorter propagation stage. While the visual representation testing implies little representability, the whole method can benefit if the developer is taught to have a critical look on charts.

E. The classes of programs

During the evaluation, the techniques had been tested not only on the main static analyzer project but also on various software packages surrounding it, to a possible extent. Author's experience shows that not only static analyzers may benefit from these methods. The class of "compatible" software comprises the programs performing a significant number of transformations. It includes the compilers, their optimization passes, refactoring, code obfuscation tools, archivers, encryption tools. The improvement would be seen in case both the input and the output are in the readable representation and if the intermediate representations are cross-linked. The approach is not cost-effective if the project is small due to a high level of an initial investment.

F. Case studies

1) The case of GNU statement expressions: In this case study, we would like to stress how the developed software package helps add new functionality. The GNU Compiler Collection (GCC) has the support for *statement-expressions*, which represent code blocks with the last statement being

TABLE I
DISCOVERED ISSUES & THEIR SEVERITY

Defect type	***Number of issues***	***Severity***	***Comment***
Performance	3	Medium	Slow handling of specific combinations of syntax constructions, branches, especially with a high number of objects.
Ordering	5	High	Ordering of syntax constructs affects the processing. This kind of issues appears during the transition from AST to IR form due to change in linearity
Runtime failure	1	High	Other critical issues with failing statements
Logical issues	1	Medium*	Problems with expression-to-formula mapping. * - This issue usually has high severity, but this concrete case was not as critical.

TABLE II
TIME TO RESOLVE TYPICAL DEFECTS

Defect type	***Time to resolve before (h)***	***Time to resolve after (h)***	***Comment***
Performance	25	13	Performance issues require significant refactoring, but it was taking time to properly diagnose where does the issue appear.
Ordering	5	1	Bookmarks make ordering issues easier to diagnose.
Runtime failure	1	0.3	Runtime failures are easy to work around, but harder to fix completely. All information in one place makes it quicker.
Logical issues	1	1	No large difference — when a logical issue is expected, you watch the log with this information. Technically, it reduces the need to find the mapping, but we haven't found it measurable.

the result of the block. The infrastructure of the analyzer was highly biased towards blocks and statements, and the statement expression was an example of the construction which could be used on the unexpected levels.

By simply adding a new compound type ("BlockWithResult") to the model, the code generator-related tools had shown the places which had to be touched. These areas included *parsing*, *expression flattening*, *expression cloning*, *type deduction* and *IR conversion*. However, the IR conversion stage was not ready for the adoption of the statement expressions, it took around 7 working days to refactor the algorithm for it. The visualization approach let the author find the issue with the incorrect placement of internal statements: e.g. conditions were set on entering *wrong* fragments. The random program generation supported the process by providing a suitable number of examples. In total, the addition of statement expressions took approximately 10 working days.

2) The case of wrong constructors: This case is more towards mechanical mistakes when writing the code. The author did a mistake making a constructor with `std::string` parameter and a constructor with `bool` parameter. When passing regular strings, they are internally represented by `const char *` object, and the closest implicit casting for the argument was to `bool`. This mistake flowed from a Directed Acyclic Graph (DAG) level to VM intermediate representation and then materialized in missing predicate check during the verification stage. The issue had been noticed after using visual representation: it was determined that the object in DAG was missing a minor property only after reading the expression dump linked to the VM IR command. The omniscient debugging wasn't of help because the time to break the execution was unclear.

VI. CONCLUSION AND FUTURE WORK

In the paper, the sources of errors in the author's static analyzer project were studied. Defects are mostly related to logical issues plaguing from missing syntax/IR support, minor issues in repeating parts, transformation defects and ordering problems. To cope with them, four sustainable solutions were prepared and shown. They include random test case (program) generation, log fusion, code generation and visual representation. These methods allowed finding at least 10 defects and decreased the time to resolve defects by 2.8 on average. The response differs for different test groups or even tests, from 5x for ordering issues, down to 1x (no improvement) for logical issues. Two presented case studies support the thesis of applicability of these methods.

In future, we expect to continue improving the functionality of the logging package and increasing the number of cross-links between log parts. Code generation will experience further generalization of the models. More metrics will be investigated to make health reports more useful.

REFERENCES

[1] L. N. Q. Do, S. Krüger, P. Hill, K. Ali, and E. Bodden, "Debugging static analysis," *IEEE Transactions on Software Engineering*, 2018.

[2] M. Menshikov, "Equid—a static analysis framework for industrial applications," in *International Conference on Computational Science and Its Applications*. Springer, 2019, pp. 677–692.

[3] GDB: The GNU Project Debugger. [Online]. Available: https://www.gnu.org/software/gdb/

[4] The LLDB Debugger¶. [Online]. Available: https://lldb.llvm.org

[5] Reproducing software defects finally made easy. [Online]. Available: https://undo.io/

[6] R. O'Callahan, C. Jones, N. Froyd, K. Huey, A. Noll, and N. Partush, "Engineering record and replay for deployability," in *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*, 2017, pp. 377–389.

[7] J. Engblom, "A review of reverse debugging," in *Proceedings of the 2012 System, Software, SoC and Silicon Debug Conference*, Sep. 2012, pp. 1–6.

[8] E. Eide and J. Regehr, "Volatiles are miscompiled, and what to do about it," in *Proceedings of the 8th ACM international conference on Embedded software*, 2008, pp. 255–264.

[9] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in c compilers," in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, 2011, pp. 283–294.

[10] G. Barany, "Liveness-driven random program generation," in *International Symposium on Logic-Based Program Synthesis and Transformation*. Springer, 2017, pp. 112–127.

[11] V. Livinskij, A. Mitrohin, and D. Babokin, "Yet Another Random Program Generator - generator sluchajnyh testov dlja verifikacii optimizacij v kompiljatorah jazykov C/C++."

[12] S. Takakura, M. Iwatsuji, and N. Ishiura, "Extending equivalence transformation based program generator for random testing of c compilers," in *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, 2018, pp. 9–15.

[13] M. Chupilko, A. Kamkin, A. Kotsynyak, and A. Tatarnikov, "Microtesk: Specification-based tool for constructing test program generators," in *Haifa Verification Conference*. Springer, 2017, pp. 217–220.

[14] D. Binkley, M. Harman, and J. Krinke, "Characterising, explaining, and exploiting the approximate nature of static analysis through animation," in *2006 Sixth IEEE International Workshop on Source Code Analysis and Manipulation*. IEEE, 2006, pp. 43–52.

[15] "Sourcetrail - documentation," https://www.sourcetrail.com/documentation/, (Accessed on 03/15/2020).

[16] L. Voinea, A. Telea, and J. J. Van Wijk, "Cvsscan: visualization of code evolution," in *Proceedings of the 2005 ACM symposium on Software visualization*, 2005, pp. 47–56.

[17] C. Collberg, S. Kobourov, J. Nagra, J. Pitts, and K. Wampler, "A system for graph-based visualization of the evolution of software," in *Proceedings of the 2003 ACM symposium on Software visualization*, 2003, pp. 77–86.

[18] J. P. S. Alcocer, F. Beck, and A. Bergel, "Performance evolution matrix: Visualizing performance variations along software versions," in *2019 Working Conference on Software Visualization (VISSOFT)*. IEEE, 2019, pp. 1–11.

[19] D. Yuan, S. Park, and Y. Zhou, "Characterizing logging practices in open-source software," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 102–112.

[20] Q. Fu, J. Zhu, W. Hu, J.-G. Lou, R. Ding, Q. Lin, D. Zhang, and T. Xie, "Where do developers log? an empirical study on logging practices in industry," in *Companion Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 24–33.

[21] D. Jurafsky, J. Martin, P. Norvig, and S. Russell, *Speech and Language Processing*. Pearson Education, 2014.

[22] M. Du, F. Li, G. Zheng, and V. Srikumar, "Deeplog: Anomaly detection and diagnosis from system logs through deep learning," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 1285–1298.

[23] A. Nandi, A. Mandal, S. Atreja, G. B. Dasgupta, and S. Bhattacharya, "Anomaly detection using program control flow graph mining from execution logs," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2016, pp. 215–224.

[24] O. Sheluhin, V. Rjabinin, and M. Farmakovskij, "Anomaly detection in computer system by intellectual analysis of system journals," *Voprosy kiberbezopasnosti*, no. 2 (26), 2018.

[25] B. Smith, John and F. Weiss, Stephen, "Hypertext," *Communications of the ACM*, vol. 31, no. 7, pp. 816–819, 1988.

[26] R. Stallman, R. Pesch, and S. Shebs, *Debugging with GDB: The GNU Source-Level Debugger*. 12th Media Services, 2018.