

Automatic Pinpoint of Divergent Interpretations

Tom GOALARD

Raphaël MONAT

Julien FORGET

Abstract—Program interpreters are complex pieces of software where bugs are difficult to investigate. Given an observed divergence between two implementations of an interpreter, we want to determine the first diverging state. We present an approach to automatically find this divergence point in an interpreter and language-agnostic way. This approach has been applied to the Mopsa static analyzer on the GNU Coreutils.

I. INTRODUCTION

Differential testing [1] is a widely used technique to search for discrepancies in the behaviors of similar programs. This method has been successfully applied to find bugs between two different implementations of code analyzers/interpreters. However, this method presents some limitations in its applications. When two interpreters diverge – i.e. when they yield different results – it is often hard to determine the root cause of the divergence. We know that the divergence happens but not where and why it happens in the analysis. We present here a tool that aims to find the root cause of divergences between two analyzers as precisely as possible.

The presented tool uses the DAP (Debug Adapter Protocol) protocol [2], extended a bit, to control the evaluated programs. This gives it the ability to test any kind of program analyzers (static or dynamic) that can be controlled through DAP with only minor adaptations. For now, the tool has been tested with static program analyzers only. More specifically, the evaluation of relational abstract domain libraries (Apron [3], Elina [4], [5], [6], [7], PPLite [8]) uses the abstract debugging capabilities of Mopsa.

The search mechanism used is a binary-like search on a linearized representation of the program. This allows us to find divergences quickly while providing a trace of the execution path that led to the divergence. This approach consists in two main steps: first, the program is linearized according to the analysis order of the tool. Then, a binary-like search is performed on this linear representation to find the divergence point. Once the divergence point is found, the execution trace can be shown to the user to help understand the root cause of the divergence.

II. DEFINITIONS

This section defines the notations and concepts used throughout the document.

Let $\mathcal{F}$ be the set of functions defined in the analyzed program and $\mathcal{L}$ the set of labels used in the analyzed program.

$$\begin{aligned}
\text{Expr} &::= \mathbb{N} \\
&| \text{Var} \\
&| \text{Expr} \oplus \text{Expr} & \oplus \in \{+, -, /, \times\} \\
&| f^{\text{label}}(\text{Expr}^*) & f \in \mathcal{F} \\
&| \text{var} = \text{Expr} \\
&| \text{return Expr} \\
\text{Stmt} &::= \text{Expr}^{\text{label}}, \\
&| \text{if}^{\text{label}}(\text{Expr} \ltimes 0) \text{ Stmt else Stmt} & \ltimes \in \{<, >, \leq, \geq, \neq\} \\
&| \text{while}^{\text{label}}(\text{Expr} \ltimes 0) \text{ Stmt} & \ltimes \in \{<, >, \leq, \geq, \neq\} \\
&| \{\text{Stmt}^*\} \\
\text{FunDef} &::= \text{fun}^{\text{label}} \text{ fun_name}(\text{var}, \dots, \text{var})\{\text{Stmt}^*\} \\
\text{Program} &::= \text{FunDef}^*
\end{aligned} \tag{1}$$

Fig. 1: Toy imperative programming language. The notation x^{label} indicates that the statement x is labeled by $\text{label} \in \mathcal{L}$. The notation x^* indicates that the statement x can be repeated zero or more times.

Let $\mathcal{A}$ be the set of alarms that can be raised by the analyzers (e.g. errors encountered during analysis, potential runtime errors, etc.).

Consider a simple toy imperative programming language with labeled instructions Fig. 1.

This language is a simplification of common imperative programming languages such as C or Java. It contains the basic constructs needed, like expressions, conditionals, loops, and function definitions.

III. CONTROLLING THE ANALYZERS

The core functionality of the tool is the ability to control the analysis as desired. This control is performed using the DAP protocol [2].

A. Protocol definition

Let Σ be the set of program states. We consider an interpreter $\mathbb{S}_0[\llbracket \text{stmt} \rrbracket] : \Sigma \rightarrow \Sigma$ that, given a program state, returns the new program state after executing the current statement.

Let $\mathcal{B} = \mathcal{L} \cup \mathcal{F}$ be the set of possible breakpoint locations (either on function entry or on a label).

Let C be a set of commands defined as follows:

- STEP: execute the next statement.

- CONTINUE: continue execution until a breakpoint is reached or the program ends.
- BREAKPOINT $\llbracket 2^{\mathcal{B}} \rrbracket$: set the breakpoints listed in the set, replacing any previously set breakpoints.
- RESTART: restart the analysis from the beginning.

There is also a special command, BACKWARD, that allows going back to the previous stack frame. This command is only used as an optimization because it allows us to avoid re-analyzing the whole program when going back in the analysis. However, this command is not strictly necessary for the functioning of the tool.

We define the interpreter controlled by DAP commands as follows:

$$\mathbb{S}[\text{stmt}] : \Sigma \times C \times 2^{\mathcal{B}} \rightarrow \Sigma \times 2^{\mathcal{B}} \times 2^{\mathcal{A}} \quad (2)$$

Where, given a program state $\sigma \in \Sigma$, a command $c \in C$, and a set of breakpoints $\text{bp} \subseteq \mathcal{B}$, the interpreter returns the new program state, the updated set of breakpoints, and the set of alarms after executing the command c .

The definition of $\mathbb{S}[\text{stmt}]$ is not detailed here as it depends on the analyzer/interpreter being controlled. However, we need one more component to achieve our goal: a way to determine whether at one point of the analysis two analysis states are diverging. We therefore define an equivalence relation:

$$\sim \subseteq \Sigma \times \Sigma \quad (3)$$

That gives us a way to know when two analysis states are diverging. Using the previous, we can now define the diverged predicate as follows:

$$\text{diverged} : \begin{cases} \Sigma \times \Sigma \times \mathcal{A} \times \mathcal{A} \rightarrow \{\text{true}, \text{false}\} \\ (\sigma_1, \sigma_2, a_1, a_2) \mapsto \neg(\sigma_1 \sim \sigma_2) \vee (a_1 \neq a_2) \end{cases} \quad (4)$$

B. The DAP protocol

The DAP (Debug Adapter Protocol) [2] is a communication protocol that standardizes the control of debuggers. It allows a standard way to send commands to a debugger. We use it for two main reasons. The first one is that it is already implemented in a lot of tools. We can hence adapt the tool to control other analyzers without any huge effort. The second reason is that it is not language-dependent. In theory, we can apply the divergence finder to Python analyzers as well as C analyzers without any change to the core part of the tool (it was originally intended for C code).

Regarding most of the previously defined commands, the adaptation to DAP commands is straightforward: they are defined as they are in the DAP protocol. There is, however, a change regarding the state of the analyzer. In DAP, we have to explicitly request it. The implementation is hence slightly different: after executing a command, we have to send a request to get the current state and alarms of the analyzer.

As noted before, we added a new command to the protocol. This command allows retrieving the current active alarms. It is necessary because some alarms can be canceled afterward, especially in loops with static analyzers. This command hence allows us to get the current alarms at any point of the analysis without the canceled ones.

IV. CODE LINEARIZATION

In order to navigate through the code structure as one would with a binary search, we linearized the code structure. This section presents the formalization of this linearization.

A. Definition

Our goal is to create a linear representation (5) as a mapping from statements to a partial mapping of an index to the statements' corresponding labels and parent index in the linearized structure (Listing 1).

$$A : \text{stmt} \rightarrow (\mathbb{N} \rightarrow (\text{label} \times (\mathbb{N} \cup \text{Main}))) \quad (5)$$

Where:

- stmt is the set of all statements in the program.
- label is the set of all labels in the program.
- $\mathbb{N} \cup \text{Main}$ is the set of natural numbers augmented with a special element Main representing the root of the analyzed program (in C, this would be the function main).

The mapping A associates to each statement in the program a tuple containing its label and the index of its parent statement in the linearized structure (Listing 1) (i.e., it is not contained within any other statement), its parent is represented by Main .

This mapping allows three things:

- To know the “length” of the program, i.e., the number of statements it contains.
- To know which statement is at position n in the linearized structure.
- To get the list of parents of a given statement in the program.

C

```

1  foo(a) {
2      if (a % 2 == 0) {
3          return a / 2;
4      } else {
5          return a * 3 + 1;
6      }
7  }
8  main() {
9      x = 14;
10     while (x < 20) {
11         x = foo(x);
12         printf("%d\n", x);
13     }
14     return x;
15 }

```

Index	Type	Range	Parent
0	Expr	line 9	Main
1	Loop	line 10 to 14	Main
2	Function	line 11, foo	1
3	If	line 2 to 12	2
4	Expr	line 3	3
5	Expr	line 5	3
6	Expr	line 12	1
7	Expr	line 14	Main

Listing 1: Example of code linearization. On the left is a simple C program with a function `foo` that returns the absolute value of its argument, and a `main` function that calls `foo` twice and prints the result. On the right is the corresponding linearized mapping, where each entry represents a statement in the order it would be interpreted, along with its label and parent index.

B. Mapping construction

The construction of the mapping (5) is done through a recursive traversal of the program’s abstract syntax tree (AST). The traversal follows the tool’s analysis order, ensuring that the linear representation accurately reflects the analysis flow of the program.

The mapping is built as follows:

- If the statement is a simple expression (e.g., assignment, return), it is added to the mapping with its corresponding label and parent.
- If there is a function call in the expression, the function is recursively traversed, and its statements are added to the mapping in the order they would be analyzed. In this case, the expression statement itself is not added to the mapping.
- If the statement is a control flow structure (e.g., if, while), the condition expression is processed first, followed by the body of the structure. Each statement within the body is added. The condition expression itself is labeled in a way that indicates it is the condition of the control flow structure.
- Blocks are processed by recursively traversing each statement within the block and adding them to the mapping in order. They are not added as separate entries in the mapping.

An example of the linearization process is shown in Listing 1. The mapping reflects the analysis order of the statements, including function calls and control flow structures. Note that the `printf` function is a Mopsa built-in and is hence not expanded in the linearization (i.e., considered as an expression). It is built as follows:

- 1) The analysis starts at line 9 (expression `x = 14;`), which is added to the mapping as an expression with parent `Main`.
- 2) Next, the while loop at line 10 is processed. We add it as a loop ending at line 14 with parent `Main`.
- 3) The function call to `foo(x)` at line 11 is encountered next. The function `foo` is recursively traversed, adding its statements to the mapping in the order they would be analyzed. The function entry is added as a function with parent 1 (the while loop).
- 4) And so on for the rest of the program.

Note that the linearization gets rid of the concept of blocks. This is because they do not have any impact on the analysis order of the tool. Thus, the `if` statement in the function `foo` has an ending range at line 12 because the next statement to be analyzed after the `if` is the one at line 12.

C. Implementation

The OCaml implementation follows the described construction process closely. The mapping is of type `instr ref array`, where each entry corresponds to a statement in the linearized program Listing 1.

The implementation uses a reference instead of an index for ease of manipulation after construction; the structure is not modified after being built.

Instruction labeling reflects each instruction's role in the program structure. This simplifies the code while preserving the information required for analysis, in particular function-call site columns and loop end locations.

The main issue with the analysis is the program comparison. It is the part that takes the most time because it requires getting the current state and alarms from both analyzers, which is slow. We hence need to minimize the number of comparisons made.

V. STEP-BY-STEP SEARCH

The first search strategy implemented was a simple step-by-step search. This strategy searches for divergences by repeating `STEP` and analyzing the obtained results each time until the end. It does not require linearization because it only needs to know the next statement to analyze. This

```

1 type instruction =
2   | Expr of location
3   | LoopCondition of location * location (* pos * end
      location *)
4   | Loop of
5     location * location * location (* pos * start location
      * end location *)
6   | Condition of location * location (* pos * end location
      *)
7   | Function of string * location * int (* name * callsite *
      column *)
8   | StubbedFunction of string * location * int * location
9 (* name * callsite * column * stubbed comment location *)

```

Listing 1: OCaml type definition for the linearized instruction representation. Each variant corresponds to a different type of statement in the program, capturing necessary information such as location, function names, and control flow details.

approach has the drawback of being very slow for large programs because it greatly increases the number of comparisons. It hence hasn't been extensively tried and is not presented in detail here.

VI. BINARY-LIKE SEARCH

A. Main idea

This search strategy finds divergences by performing a binary-like search on the linearized code structure. Two values, min and max, represent the current search interval. The algorithm Algorithm 1 follows the standard pattern: compute $\text{mid} = \lfloor \frac{\text{min} + \text{max}}{2} \rfloor$ and go to mid to check for divergence. If a divergence is found, the divergence point lies between min and mid, so set $\text{max} = \text{mid}$; otherwise set $\text{min} = \text{mid}$. Repeat until min and max are consecutive, then check the statement at max to determine whether it is the divergence point.

Algorithm 1: Binary-search

```

1: procedure BINARY-SEARCH( $A$ , interpreter1, interpreter2)
2:   min  $\leftarrow$  0
3:   max  $\leftarrow$  length( $A$ ) - 1
4:   while max - min > 1 do
5:     mid  $\leftarrow$  floor( $\frac{\text{min} + \text{max}}{2}$ )
6:     Restart(interpreter1)
7:     Restart(interpreter2)
8:     GoToStatement( $A$ , mid)
9:      $\sigma_1, a_1 \leftarrow$  GetStateAndAlarms(interpreter1)

```

```

10:   $\sigma_2, a_2 \leftarrow \mathbf{GetStateAndAlarms}(\text{interpreter}_2)$ 
11:  if diverged( $\sigma_1, \sigma_2, a_1, a_2$ ) then
12:     $\text{max} \leftarrow \text{mid}$ 
13:  else
14:     $\text{min} \leftarrow \text{mid}$ 
15:  end
16: end
17:  $\triangleright$  Final check at position  $\text{max}$ 
18: if diverged( $\sigma_1, \sigma_2, a_1, a_2$ ) then
19:   return "max"
20: else
21:   return -1
22: end
23: end

```

The algorithm is, however, more complex in practice due to the need to support loops. We have to check each iteration of each loop for divergence. The point is evaluated multiple times, once for each iteration of the loops it is contained in. To get the information about the loops when the divergence point is found, we store the number of iterations performed.

B. Reaching the midpoint

In order to check for divergence at the midpoint, we need to reach it first. To this end, we make use of the parent information stored in the linearized mapping. We make a chain of all the parents of the midpoint. Then, going to the midpoint consists in going through this chain from the root to the midpoint. However, there are some edge cases to consider.

a) Going to an expression:

To go to an expression at line n , we simply have to execute `BREAKPOINT({n})` followed by `CONTINUE`. We then reach the expression at line n . If the program ends or reaches another breakpoint, it means that line n is not reachable.

b) Going to a function:

To go to a function, we first need to go to its call site. This is done the same way as going to an expression. Once at the call site, we set a function breakpoint `BREAKPOINT({functionName})`. Then, we execute `CONTINUE` again to reach the function entry point. Because there can be multiple function call sites on the same line, we need to check that we reached the correct function by comparing the call site column with the one stored in the linearized mapping. If they do not match, we run `CONTINUE` again until we reach the correct function or the program ends.

c) *Going through conditions:*

Conditions are handled by simply ignoring them. When going through a condition, we simply recursively go to the next parent in the chain. This is because, when we need to reach a condition child, either the branch is reachable and we can reach the child directly without further action, or the branch is not reachable and we will handle it in the recovery mechanism.

d) *Going through loops:*

This is the most complex case. When going through a loop, we need to consider multiple cases. The simplest case is a loop surrounded by reachable code like the one

```
1 int main() {  
2     int i = 0;  
3     while (i < 10) {  
4         > Condition Breakpoint  
5         printf("%d\n", i);  
6         > Start Breakpoint  
7         i = i + 1;  
8     }  
9     > End breakpoint  
10 }
```

For this case, we need three breakpoints. The first one is at the loop condition. We first set a breakpoint there and run CONTINUE to reach the condition the first time. Then, we set two breakpoints: one at the start of the loop body and one at the end of the loop. We then CONTINUE. If we reach the start breakpoint, we are in the loop body and can continue to the next element in the chain. If we reach the end breakpoint, this means that the loop body is dead code and cannot be reached. To count the number of iterations, we count the number of times the condition breakpoint is reached. We have to hit CONTINUE after each execution of the next element in the chain. Once the end breakpoint is reached, the loop analysis is finished.

Then, there are two edge cases to consider: loops where the end breakpoint is not reachable and loops encapsulated in other loops without code surrounding them.

The first case is quite easy to handle. We just need to check if the interpreter is finished. If it is, we consider that equivalent to reaching the end breakpoint.

The second case is more complex. Consider the following code:

```
1 int main() {  
2     int i = 0;
```

```

3   while (i < 10) {
4       int j = 0;
5       i = i + 1;
6       while (j <= 10) {
7           printf("%d %d\n", i, j + 1);
8           j = j + 1;
9       }
10  }
11  return 0;
12 }

```

Here, we cannot place a breakpoint at the end of the inner loop. We need a different strategy. The approach used is to consider the condition breakpoint of the outer loop as the end breakpoint of the inner loop. This way, when we reach the condition breakpoint of the outer loop while analyzing the inner loop, we consider that we reached the end of the inner loop. In this case, line 11 is also considered as the end of the inner loop. This way, if the outer loop finishes at the same time as the inner loop, we can still consider that we reached the end of the inner loop.

C. Recovery mechanism

The binary search algorithm assumes that the midpoint can always be reached. However, there are cases where the midpoint cannot be reached because it points to dead code. In that situation, we need a recovery method to continue the search. The recovery mechanism consists of adding breakpoints between min and max until a reachable statement is found. When such a statement is reached, we check it for divergence and update either max or min accordingly. If no reachable statement can be found, this indicates an invalid state and the search is stopped.

Algorithm 2: Recovery Mechanism

```

1: procedure RECOVER( $A$ , min, max, interpreter1, interpreter2)
2:   reached  $\leftarrow$  false
3:   bps  $\leftarrow$  2
4:   while  $\neg$  reached do
5:     incr  $\leftarrow$  ceil( $\frac{\text{max} - \text{min}}{1 + \text{bps}}$ )
6:     Restart(interpreter1)
7:     Restart(interpreter2)
8:     ClearAllBreakpoints(interpreter1)
9:     ClearAllBreakpoints(interpreter2)
10:    for  $i$  from min + 1 to max - 1 by incr do
11:      AddBreakpoint(interpreter1,  $i$ )
12:      AddBreakpoint(interpreter2,  $i$ )
13:    end
14:    GoToStatement( $A$ , min)

```

```

15:    Continue(interpreter1)
16:    Continue(interpreter2)
17:    reached  $\leftarrow \neg$  Finished(interpreter1) and  $\neg$  Finished(interpreter2)
18:    bps  $\leftarrow 2 \times$  bps
19:  end
20:   $\sigma_1, a_1 \leftarrow$  GetStateAndAlarms(interpreter1)
21:   $\sigma_2, a_2 \leftarrow$  GetStateAndAlarms(interpreter2)
22:  if diverged( $\sigma_1, \sigma_2, a_1, a_2$ ) then
23:    max  $\leftarrow$  CurrentPosition(interpreter1)
24:  else
25:    min  $\leftarrow$  CurrentPosition(interpreter1)
26:  end
27: end

```

Let's take the following toy example to illustrate the recovery mechanism:

C

```

1  int main() {
2    printf("Hello");
3    if (0) {
4      printf("This is dead code");
5    }
6    printf("World!");
7    return 0;
8  }

```

It would be linearized as follows:

Index	Type	Range	Parent
0	Expr	line 2	Main
1	Condition	line 3 to 6	Main
2	Expr	line 4	1
3	Expr	line 6	Main
4	Expr	line 7	Main

The binary search method would first try to go to position $\lceil \frac{5}{2} \rceil = 3$. It will fail because it is dead code. The recovery mechanism will then set breakpoints at lines 2 and 4. The analysis will stop at line 2, which is not diverging. The search interval is then updated to $[3, 5]$. The next midpoint is $\lceil \frac{3+5}{2} \rceil = 4$, which is reachable. The analysis then continues as normal.

VII. EXPERIMENTAL EVALUATION

As of now, the tool hasn't been extensively evaluated. However, some preliminary results have been obtained when trying to find divergences in the Mopsa static analyzer [3] on the GNU

Coreutils. They show that the tool is able to find divergence points in a reasonable time (compared to the analysis time of Mopsa). For instance, on the `comm` utility, the analysis is able to converge in 12 iterations and about 50 minutes of analysis time.

VIII. FUTURE WORK

The tool being in early stages of development, there are many possible improvements and extensions that could be made. Some of the most pressing ones are among the following:

- Extensive evaluation on a wider range of programs and analyzers to assess the tool’s effectiveness and performance.
- Timing the different parts of the tool to identify what takes time (e.g., waiting for the analyzer, computing the next action, etc.)
- Adding more optimizations to reduce the number of times we need to restart the analysis

The tool could also benefit from being integrated with other tools than Mopsa. This would require adapting a bit the way those tools are controlled through DAP but would allow a lot more use cases to be covered.

IX. CONCLUSION

We presented a tool that is able to find divergence points between two program analyzers in an automatic way. The tool is analyzer-agnostic thanks to the use of the DAP protocol to control the analyzers. The search for divergence points is done through a binary-like search on a linearized representation of the program. This approach allows us to find divergence points quickly while providing a trace of the execution path that led to the divergence. While the tool is still in early stages of development, preliminary results show that it is able to find divergence points in real-world, reasonably sized programs. It thus opens the way to easier debugging of program analyzers through differential testing approaches.

REFERENCES

- [1] C. Klinger, M. Christakis, and V. Wüstholtz, “Differentially testing soundness and precision of program analyzers,” in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, Beijing China: ACM, July 2019, pp. 239–250. doi: 10.1145/3293882.3330553.
- [2] Microsoft, “Official page for Debug Adapter Protocol.” Accessed: Dec. 01, 2025. [Online]. Available: <https://microsoft.github.io/debug-adapter-protocol/>
- [3] R. Monat, M. Milanese, F. Parolini, J. Boillot, A. Ouadjaout, and A. Miné, “Mopsa-C: Improved Verification for C Programs, Simple Validation of Correctness Witnesses (Competition Contribution),” Springer Nature Switzerland, Apr. 2024, p. 387. doi: 10.1007/978-3-031-57256-2_26.
- [4] G. Singh, M. Püschel, and M. Vechev, “Making numerical program analysis fast,” in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, Portland OR USA: ACM, June 2015, pp. 303–313. doi: 10.1145/2737924.2738000.

- [5] J. He, G. Singh, M. Püschel, and M. Vechev, “Learning fast and precise numerical analysis,” in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, London UK: ACM, June 2020, pp. 1112–1127. doi: 10.1145/3385412.3386016.
- [6] G. Singh, M. Püschel, and M. Vechev, “A practical construction for decomposing numerical abstract domains,” *Proceedings of the ACM on Programming Languages*, vol. 2, no. POPL, pp. 1–28, Jan. 2018, doi: 10.1145/3158143.
- [7] G. Singh, M. Püschel, and M. Vechev, “Fast polyhedra abstract domain,” in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, Paris France: ACM, Jan. 2017, pp. 46–59. doi: 10.1145/3009837.3009885.
- [8] A. Becchi and E. Zaffanella, “PPLite: Zero-overhead encoding of NNC polyhedra,” *Information and Computation*, vol. 275, p. 104620, Dec. 2020, doi: 10.1016/j.ic.2020.104620.