Abstract Debuggers

Exploring Program Behaviors using Static Analysis Results

Karoliine Holter
University of Tartu
Tartu, Estonia

Juhan Oskar Hennoste
University of Tartu
Tartu, Estonia

Patrick Lam
University of Waterloo
Waterloo, Canada

Simmo Saan
University of Tartu
Tartu, Estonia

Vesal Vojdani
University of Tartu
Tartu, Estonia

Abstract

Traditional, or concrete, debuggers allow developers to step through programs and explore the corresponding concrete program states—developers can query current values of program variables. This exploration enables developers to formulate and refine hypotheses about program behaviors. We propose the novel notion of *abstract debuggers*, which allow developers to explore abstract program states, as computed by sound static analyzers. Giving developers the ability to interactively explore abstract states empowers them to work with hypotheses that are true for all program executions: they can examine and rule out false positives, or better understand a static analysis's declaration that some code is indeed safe. Abstract debuggers' interfaces, reminiscent of conventional debuggers, aim to make navigating and interpreting static analysis results more straightforward. We have formalized the concept, applied it by implementing a tool that leverages the static analyzer Goblint, and illustrate its usefulness through case studies.

CCS Concepts: **• Software and its engineering → Software testing and debugging**; **Automated static analysis**; **• Theory of computation → Abstraction**.

Keywords: Automated Software Verification, Abstract Interpretation, Explainability, Visualization, Data Race Detection

ACM Reference Format:
Karoliine Holter, Juhan Oskar Hennoste, Patrick Lam, Simmo Saan, and Vesal Vojdani. 2024. Abstract Debuggers: Exploring Program Behaviors using Static Analysis Results. In *Proceedings of the 2024 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! '24), October 23–25, 2024, Pasadena, CA, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3689492.3690053

Onward! '24, October 23–25, 2024, Pasadena, CA, USA
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1215-9/24/10
https://doi.org/10.1145/3689492.3690053

1 Introduction

According to Zeller [48], a best practice for debugging is to apply the scientific method: faced with a bug, developers ought to create a hypothesis about why their program fails, and then verify (or refine) this hypothesis by conducting experiments. Once a hypothesis has enough support, the developer can correct the underlying defect. In Zeller's book, the experiments consist of concrete program executions: the developer runs the program with particular inputs, and, potentially, instrumentation permitting better observability. Ko and Myers' Whyline [26] enables event-level observation of program behavior—in the context of the Alice programming system, developers could observe values at domain-relevant levels of abstraction.

In this work, we propose the notion of *abstract debugging*. As with concrete debugging, our approach enables developers to answer questions about program behaviors. Our insight is that it is possible to use static analysis tools to provide much more complete information about program behavior; specifically, at a higher level of abstraction. Sound static analysis tools, after all, reason about *all paths* in the program's execution. By moving away from concrete input values, abstract debugging liberates developers from the necessity to provide specific input values and allows them to reason about all possible behaviors of their software.

Fundamentally, static analysis can identify program properties by examining the program's code without executing it. This enables the pinpointing of errors that might only manifest under specific conditions (e.g., inputs or thread interleavings). Sound analyzers, such as abstract interpreters, aim to automatically verify the absence of errors under all conditions. While the primary output of such analysis is a list of potential program errors—or a claim that the program is free from errors—the foundation for these conclusions often rests on a vast array of other data, including potential program states, variable values, and their interrelationships. Our approach was motivated by the insight that these intermediate results can offer valuable information about a program's behavior and the origins of potential errors; however, the raw presentation of these results can be challenging to decipher, even with an in-depth understanding of the analyzer's internal processes. Abstract debuggers are our way of making static analysis results legible to developers.

Abstract debugging enables developers to interactively examine alerts raised by static analysis tools, thus confirming or refuting their hypotheses about program behavior. By exploring our interprocedural Abstract Reachability Graph (iARG), developers can rule out false positives, rule in true positives, and understand why a static analysis tool concludes that some code is guaranteed to meet a given property. Furthermore, this graph exploration is done in the familiar environment of a debugger: developers can set breakpoints, step (backwards and forwards) into functions and across statements, and explore the (abstract) state of the program.

We introduce abstract debugging with reference to concrete (traditional) debugging. Concrete debuggers allow for the step-by-step execution of a program, where users can directly observe the concrete program state at each step. We formalize the operational semantics of these stepping operations. To define abstract debugging, we thus also formalize the corresponding operational semantics for the abstract world, simulating step-by-step executions using the results from static analysis (rather than by executing the program and reporting concrete state values). User-visible abstract debugger states do not show concrete values as in traditional debuggers, but rather abstractions of these values. These abstract values encapsulate the program's state across all executions and can reflect a joint state of multiple traces.

We have developed a practical prototype that demonstrates this approach to abstract debugging, using the static analyzer Goblint to provide abstractions. Through this implementation, we aim to show the potential of abstract debugging in answering higher-level questions about programs, with the advantage that the answers are valid across all executions. Abstract debuggers aim to render static analysis more approachable, user-friendly and interpretable. By building on the familiar interface of traditional debuggers, we show that abstract debugging simplifies the navigation and interpretation of static analysis outcomes. Furthermore, using the *Debug Adapter Protocol (DAP)* supported by many Integrated Development Environments (IDEs), the implementation also leverages an existing framework instead of creating a new one, directly benefitting from IDE improvements and reducing implementation and maintenance efforts. The contributions of this work are:

- The notion of an *abstract debugger*, which enables developers to interactively pose questions about interprocedural program behavior and to receive answers valid for all program executions, using static analysis information.
- A series of examples showing the utility of an abstract debugger for understanding alerts from static analysis (true/false positives) and guarantees (true negatives).
- A formalization of the operational semantics of both the concrete and abstract debugger, ensuring soundness by guaranteeing that every debugging session in the concrete world has a corresponding session in the abstract world.
- An implementation of an abstract debugger for the abstract interpretation-based static analyzer Goblint.

2 Running Example

Consider the simple program in Fig. 1a, presented in a hypothetical C-like language to simplify understanding (the screenshot in Fig. 1b shows the actual C code for a flawed version of this program). We use this program to illustrate the challenges in interpreting analysis results. The program consists of two threads with identical implementations. Each thread attempts to acquire a mutex. If the mutex is acquired, the thread writes to a shared variable `global`; otherwise, it writes to a local cache.

The Goblint static analyzer would report that the version of the program in Fig. 1a is free from data races:

```
[Success][Race] Memory location "global" (safe):
  write with [lock:{mutex}, thread:[main, t1]] (L12)
  write with [lock:{mutex}, thread:[main, t2]] (L12)
```

However, if the user had forgotten the break statement on line 10, the analyzer would report a potential race[1].

```
[Warning][Race] Memory location "global" (unsafe):
  write with [thread:[main, t1]] (L12)
  write with [thread:[main, t2]] (L12)
```

By setting a breakpoint at the reported race location (see Fig. 1b) and stepping backward to the `CACHE` case, the *cause* of the race is immediately revealed. For a simple example, a person (or tool) could generate a concrete counter-example trace including the concurrent writes. However, when a concrete trace becomes too unwieldy (e.g., for complex multithreaded programs) or when a tool does not produce a concrete trace (as often happens), then interactively navigating through all possible traces in the program at once can help the user understand the cause of the issue more easily than poring through a lengthy counter-example trace.

Most analyzers allow inspection of their internal state, and looking at the Goblint analyzer's output for the program in Fig. 1a, we see that the switch statement on line 7 can be reached by the two threads, each thread either holding the mutex or not:

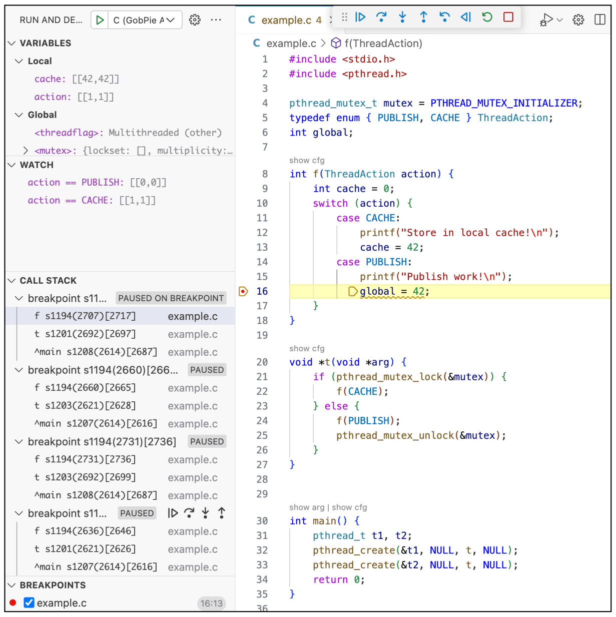
$$\begin{aligned}\{&\{\text{locals} : [\texttt{action} \mapsto \texttt{PUBLISH}], \text{tid} : \texttt{t1}, \text{lockset} : \{\texttt{mutex}\}\}\\ &\{\text{locals} : [\texttt{action} \mapsto \texttt{CACHE}], \quad \text{tid} : \texttt{t1}, \text{lockset} : \emptyset\}\\ &\{\text{locals} : [\texttt{action} \mapsto \texttt{PUBLISH}], \text{tid} : \texttt{t2}, \text{lockset} : \{\texttt{mutex}\}\}\\ &\{\text{locals} : [\texttt{action} \mapsto \texttt{CACHE}], \quad \text{tid} : \texttt{t2}, \text{lockset} : \emptyset\}\}.\end{aligned}$$

This is somewhat helpful. However, with the abstract debugger, we can step from the start of the program, and at the

[1]Compilers can warn about implicit fallthrough, but since fallthroughs can be deliberate, even `gcc -Wall` does not complain about them.

```
1  mutex_t mutex = MUTEX_INIT;
2  typedef enum { PUBLISH, CACHE } Action;
3  int global;
4
5  int f(Action action) {
6    int cache = 0;
7    switch (action) {
8      case CACHE:
9        cache = 42;
10       break; // remove for flawed version
11     case PUBLISH:
12       global = 42;
13   }
14 }
15
16 thread t1, t2 {
17   if (lock(&mutex)) {
18     f(CACHE);
19   } else {
20     f(PUBLISH);
21     unlock(&mutex);
22   }
23 }
```

(a) An example program (in stylized C-like code) that either writes to a shared variable, or, if mutex acquisition fails, writes to a local variable as a backup.



```
cache: [[42,42]]
action: [[1,1]]
```

```
<threadflag>: Multithreaded (other)
<mutex>: {lockset: [], multiplicity:…
```

```
action == PUBLISH: [[0,0]]
action == CACHE: [[1,1]]
```

```
breakpoint s11…                     PAUSED ON BREAKPOINT
  f s1194(2707)[2717]         example.c
  t s1201(2692)[2697]         example.c
  ^main s1208(2614)[2687]     example.c
breakpoint s1194(2660)[266…         PAUSED
  f s1194(2660)[2665]         example.c
  t s1203(2621)[2628]         example.c
  ^main s1207(2614)[2616]     example.c
breakpoint s1194(2731)[2736]        PAUSED
  f s1194(2731)[2736]         example.c
  t s1203(2692)[2699]         example.c
  ^main s1208(2614)[2687]     example.c
breakpoint s11…                     PAUSED
  f s1194(2636)[2646]         example.c
  t s1201(2621)[2626]         example.c
  ^main s1207(2614)[2616]     example.c
```

```c
#include <stdio.h>
#include <pthread.h>

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
typedef enum { PUBLISH, CACHE } ThreadAction;
int global;

show cfg
int f(ThreadAction action) {
    int cache = 0;
    switch (action) {
        case CACHE:
            printf("Store in local cache!\n");
            cache = 42;
        case PUBLISH:
            printf("Publish work!\n");
            global = 42;
    }
}

show cfg
void *t(void *arg) {
    if (pthread_mutex_lock(&mutex)) {
        f(CACHE);
    } else {
        f(PUBLISH);
        pthread_mutex_unlock(&mutex);
    }
}


show arg | show cfg
int main() {
    pthread_t t1, t2;
    pthread_create(&t1, NULL, t, NULL);
    pthread_create(&t2, NULL, t, NULL);
    return 0;
}
```

(b) A screenshot of the abstract debugger on the flawed version of the program in Fig. 1a. There is a breakpoint at the warning location. Stepping back will reveal the cause.

Figure 1. Running example program and its abstract debugging session.

if-statement on line 17, we can force the abstract debugger to step into the true branch, where locking has failed. We know that the locking has failed in the true branch because the function `pthread_mutex_lock()` returns zero when successful. This is hard to test concretely (or in a concrete debugger) except by mocking the mutex acquisition or mutating the program. Yet, a sound analyzer must consider these cases, and the abstract debugger makes it easier to shed light on these dark corner cases using the power of the analyzer.

A user may also be interested in understanding the analyzer's justification for an absence of data races. In the correct program, we saw that the analyzer's output (above) claims that the writes to `global` are all protected by `mutex`, but no justification for this claim. The abstract debugger also allows users to step through the rest of the program, restricted to states reachable from the true (`CACHE`) branch, and verify that the program never reaches any writes to `global`. Thus, the user does not need to blindly trust the analyzer, but can understand what the analyzer is claiming about different paths through the program.

In this paper, we have a specific meaning in mind for an abstract debugger. It should be an abstraction of a concrete debugger, meaning that if a concrete debugger, such as *GDB*, is used in Visual Studio Code (VS Code), and the user performs a sequence of steps, there should be a corresponding sequence of steps that the user can witness by using the abstract debugger while pressing the same buttons in the VS Code interface. Under the abstract debugger, of course, the user would see abstract states (analysis abstractions) rather than concrete program states. The difference in button presses boils down to how multiple possible successor states are handled. Consider the following side-by-side comparison of a concrete debugger and our abstract debugger on the execution of the thread body (starting at line 17, denoted by the label L17 below), where line numbers refer to the stylized code in Fig. 1a:

L17 *Button: Step over.* In our real-world execution, locking succeeds and GDB moves to line 20. The fact that the lock is held is not easily visible to the concrete debugger's user. The abstract debugger asks the user for the target branch, and when choosing "success", the view of the abstract state shows the updated lockset and the result 0 (represented as the interval $[0, 0]$) from the call to `lock()`. It then also moves to line 20.

L20 *Button: Step into.* The concrete debugger enters function `f`, showing concrete values for the local variables. For `action`, GDB is capable of displaying the enum name `PUBLISH`; the abstract debugger, however, shows the abstract value $[0, 0]$, which is the underlying value of the enum. The value of the uninitialized local variable `cache` is arbitrary, so it is shown as $[-2147483648, 2147483647]$ in the abstract state.

L6 *Button: Step over.* The concrete debugger moves to line 7 and the state shows that the value of `cache` is 0. The abstract debugger also sets the value of `cache` to $[0, 0]$ and moves along.

L7 *Button: Step over.* The concrete debugger jumps to line 12, while our abstract debugger requires two button presses to reach line 12 due to the implementation's internal representation of switch as if-statement.

L12 *Button: Step out.* Both debuggers move out to line 21, the line after the function call, and show that the global is 42 and $[42, 42]$, respectively.

L21 *Button: Step over.* Both move to line 22, but after the unlock, the abstract debugger gives a less precise value, $[0, 42]$, for the now-unprotected global variable, due to its thread-modular abstraction of concurrent programs. The abstract state now shows that no locks are held; again, this information is not easily visible in the concrete state.

This example shows that the abstract debugger can, in principle, simulate the concrete debugging session. It enables the exploration of all possible program behaviors, either through over-approximation or by allowing the user to choose a path. In the following, we aim to formalize this claim: an abstract debugger can simulate any concrete debugging session, i.e., it accounts for all program inputs and interleavings.

We also point out the following advantage of abstract debuggers over concrete debuggers. Given any reachable program point p, it is possible to force an abstract debugger to navigate to p, by setting a breakpoint there and initiating debugging. With a concrete debugger, the user must supply program inputs leading to p (and possibly be lucky with interleavings), which may be challenging if p is hard to reach. Because the abstract debugger reasons about all possible inputs, the user does not need to (indeed, cannot) supply inputs.

3 Debugging in the Concrete World

This section provides a brief overview of the operations making up a typical debugging session and outlines their operational semantics to facilitate comparisons between conventional and abstract debuggers.

A debugger is a tool that allows one to run a program step-by-step and inspect the program's state after each step. Nearly all IDEs support debuggers in some form, and there is a debugger available for almost all popular programming languages.

From the variety of different conventional concrete debugging methods, we compare the abilities of our method with *live reverse debugging* [37], which has been implemented on top of LLDB. While our conceptual comparison is with the more powerful technology of live reverse debugging, our actual implementation builds on VS Code; most VS Code debugger backends provide traditional debuggers. Traditional debuggers allow a developer to execute a program in the (usual) forward direction, providing concrete inputs during its execution; furthermore, the developer may set a breakpoint and change the concrete program state. Record-and-replay debuggers further add the ability to reverse execution: stepping back rewinds the program to a previous state, and then stepping forwards plays the recorded execution until reaching another state of interest. Record-and-replay removes, to some extent, the ability to interactively change the state when in replay mode. Live reverse debugging, then, additionally allows the exploration of alternative paths in the program execution. *Liveness*, in particular, allows forward execution to resume live from a breakpoint. The concrete program execution proceeds, possibly exploring a different path if either (1) the state is modified by the developer or (2) the program is provided with different concrete input from the environment.

As alluded to above, in a debugger, breakpoints and stepping play a central role. Breakpoints mark a spot in the source code where the debugger halts execution to display the program's state at that point. Stepping allows users to execute the code step-by-step, advancing to the next program point with each step. Debugger stepping operations are essential for navigating the flow of program execution during debugging sessions. The common stepping operations of a conventional debugger, that our abstract debugger must support, are: step over, step into, step out, and step back. In the following paragraphs, we will outline the operational semantics of concrete debugging stepping operations.

Control Flow. We assume the source code is correctly parsed, and that each function is represented as a Control Flow Graph (CFG). A CFG of a function $f \in \mathit{Fun}$ is a directed graph $G_f = (N_f, E_f, st_f, ret_f)$ where the finite set of nodes N_f represents program locations of the function, st_f and ret_f refer to the function's unique start and end locations, and the finite set of edges $E_f = N_f \times O \times N_f$ represents control flow between the locations.

The edges use labels from a statement language $O = B \cup F$, which includes basic operations and function calls. The basic operations (B) include assignments and assume edges (modelling conditional branches). The function calls (F) contain expressions (potentially involving pointers) that evaluate to the target function's name during execution. The sets $N = \bigcup_{f \in \mathit{Fun}} N_f$ and $E = \bigcup_{f \in \mathit{Fun}} E_f$ capture the control flow of the entire program. Moreover, we denote by $E_B = \{(n_1, b, n_2) \in E \mid b \in B\}$ the set of all edges with basic operations and by $E_F = \{(n_1, p, n_2) \in E \mid p \in F\}$ the set of all edges with function calls.

Concrete Executions. A concrete program state $c \in C$ contains both control information (the program location $n \in N$ and call stack for each thread), as well as the memory state (e.g., the values of variables and the heap). We denote by $N(c) \in N$ the program location that is contained in concrete state c.

Balance (basic)
$$\frac{c \xrightarrow{b} c' \qquad b \in B \qquad \pi = \epsilon}{c \xrightarrow{\pi}{}^{\star} c'}$$

Balance (function)
$$\frac{c \xrightarrow{\downarrow f} c_1 \xrightarrow{\pi'}{}^{\star} c_2 \xrightarrow{f\uparrow} c' \qquad \pi = c_1 \pi' c_2}{c \xrightarrow{\pi}{}^{\star} c'}$$

Balance (append)
$$\frac{c \xrightarrow{\pi_1}{}^{\star} c_1 \xrightarrow{\pi_2}{}^{\star} c' \qquad \pi = \pi_1 c_1 \pi_2}{c \xrightarrow{\pi}{}^{\star} c'}$$

Step into
$$\frac{c \xrightarrow{e} c' \qquad e \in B \cup {\downarrow}F \cup F{\uparrow}}{\cdots c \xRightarrow{\text{into}} \cdots cc'}$$

Step over (basic, return)
$$\frac{c \xrightarrow{e} c' \qquad e \in B \cup F{\uparrow}}{\cdots c \xRightarrow{\text{over}} \cdots cc'}$$

Step over (entry)
$$\frac{c \xrightarrow{\downarrow f} c_1 \xrightarrow{\pi'}{}^{\star} c_2 \xrightarrow{f\uparrow} c' \qquad \pi = c_1 \pi' c_2}{\cdots c \xRightarrow{\text{over}} \cdots c\pi c'}$$

Step out (basic, entry)
$$\frac{c \xrightarrow{\pi'}{}^{\star} c_1 \xrightarrow{f\uparrow} c' \qquad \pi = \pi' c_1}{\cdots c \xRightarrow{\text{out}} \cdots c\pi c'}$$

Step out (return)
$$\frac{c \xrightarrow{f\uparrow} c'}{\cdots c \xRightarrow{\text{out}} \cdots cc'}$$

Step back (basic, entry)
$$\frac{c \xrightarrow{e} c' \qquad e \in B \cup {\downarrow}F}{\cdots cc' \xRightarrow{\text{back}} \cdots c}$$

Step back (return)
$$\frac{c \xrightarrow{\downarrow f} c_1 \xrightarrow{\pi'}{}^{\star} c_2 \xrightarrow{f\uparrow} c' \qquad \pi = c_1 \pi' c_2}{\cdots c\pi c' \xRightarrow{\text{back}} \cdots c}$$

Figure 2. Concrete operational semantics of a *live reverse* debugger.

For each basic operation edge $e = (n_1, b, n_2) \in E_B$, the concrete program semantics induce a labeled transition relation $\rightarrow_e \subseteq C \times B \times C$ between pre- and post-states. These relations capture intraprocedural control flow: $c_1 \xrightarrow{b}_e c_2$ implies $N(c_1) = n_1$ and $N(c_2) = n_2$.

Function calls, corresponding to interprocedural control flow, are executed by proceeding into the function body. Let ${\downarrow}F = \{{\downarrow}f \mid f \in \mathit{Fun}\}$ and $F{\uparrow} = \{f{\uparrow} \mid f \in \mathit{Fun}\}$ be sets of special operations indicating the entry to and the return from function f, respectively. As functions are called dynamically, each function call edge $e = (n_1, p(), n_2) \in E_F$ induces the relation $\xrightarrow{\downarrow f}_e \subseteq C \times \{{\downarrow}f\} \times C$ for every function $f \in \mathit{Fun}$. Here, $c_1 \xrightarrow{\downarrow f}_e c_2$ implies that p evaluates to f in the state c_1 and execution enters the body of f, i.e., $N(c_1) = n_1$ and $N(c_2) = st_f$.

Functions return to the caller based on the call stack. Thus, we have a relation $\xrightarrow{f\uparrow} \subseteq C \times \{f{\uparrow}\} \times C$ for every function $f \in \mathit{Fun}$. Being in the relation $c_1 \xrightarrow{f\uparrow} c_2$ implies $N(c_1) = ret_f$ and $N(c_2)$ is the return location indicated by the call stack of c_1. We obtain the complete (infinite) transition relation $\rightarrow \subseteq C \times (B \cup {\downarrow}F \cup F{\uparrow}) \times C$ as the union of all these relations:

$$\rightarrow = \Big(\bigcup_{e \in E_B} \rightarrow_e\Big) \cup \Big(\bigcup_{e \in E_F, f \in \mathit{Fun}} \xrightarrow{\downarrow f}_e\Big) \cup \Big(\bigcup_{f \in \mathit{Fun}} \xrightarrow{f\uparrow}\Big).$$

This relation captures the validity of individual operations without considering their reachability from an initial state.

For debugging, we are interested in finite prefixes of concrete executions. Let C^* denote the set of all finite sequences with elements from C. An execution prefix is a sequence of states $c_0 c_1 \cdots c_n \in C^*$ where c_0 is an initial state and for each $0 < i \leq n$, we have $c_{i-1} \rightarrow c_i$. This restricts executions to those actually reachable from c_0.

Concrete Operational Semantics. A state of a live reverse debugger is an execution prefix $c_0 c_1 \cdots c_n$ reaching state c_n, where execution is paused for inspection. We define a transition relation $\Rightarrow \subseteq C^* \times \mathcal{S} \times C^*$ for the debugger based on the operations it can perform: $\mathcal{S} = \{\text{into}, \text{over}, \text{out}, \text{back}\}$. As for pushdown systems, we define this relation based on the relevant latter parts of the execution history, often the last element in the sequence. Thus, when we write $\cdots \tau \Rightarrow \cdots \tau'$, the $\cdots$s on both sides must be the same sequences of states.

Using the outlined definitions, Fig. 2 defines the concrete operational semantics as follows:

Balance: Following Reps et al. [35], we define the notion of same-level runs. These are executions $c\pi c'$ from a concrete state c to another state c' where all entered functions in the execution π have correctly returned. Furthermore, π does not return from functions it did not enter. In such executions, function entry and return operations are balanced like parentheses. This is expressed in the rules named Balance.

Step into: For a basic operation $e \in B$, this step transitions to a next interprocedural location. If the next operation is a function call $e \in {\downarrow}F$, it steps into (enters) the corresponding function, and if $e \in F{\uparrow}$, it leaves the current function and returns to the calling function. This behavior is defined by the rule Step into.

Step over: For a basic operation or a function return, this is identical to step into: it executes a statement or leaves the current function (Step over (basic, return)). For a function call, i.e., the label of the next transition is a function enter, the debugger executes the whole function and stops at the next location in the current function (Step over (entry)), using the Balance rules.

Step out: Executes the remaining statements of the current function and returns to the calling function, defined by Step out (basic, entry). If there are no remaining statements to execute, i.e., the next transition is function return, the debugger will simply step out of the function (Step out (return)), just like step into/over.

Step back: The essence of reverse debugging. If the label of the last transition is either a basic operation or a function entry, the step back operation discards the last element of the execution sequence. This behavior is defined by the rule Step back (basic, entry). If the label of the last transition is a function return, step back will step to the location of that function's call, as defined by Step back (return). Note that these are dual to step over.

While the above rules define individual stepping operations on debugger states, a complete debugging session is a sequence $\pi_0 \overset{s_1}{\Longrightarrow} \pi_1 \overset{s_2}{\Longrightarrow} \cdots \overset{s_n}{\Longrightarrow} \pi_n$ of such steps, starting from an initial debugging state π_0. The developer may inspect the execution prefix π_i, e.g., values of variables, to decide on the next stepping operation $s_i \in \mathcal{S}$. The initial state may correspond to the program start or a breakpoint. For simplicity, we assume the former, although it is straightforward to extend our concrete operational semantics with a set of breakpoints and a continue operation.

4 Debugging in the Abstract World

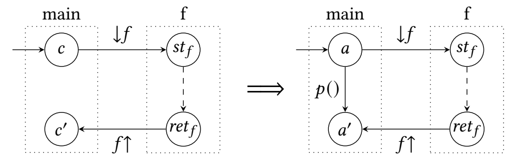
While different sound program analyzers may compute and represent results slightly differently (e.g., abstract interpretation over program syntax versus state space exploration), they all fundamentally compute an over-approximation of the set of reachable states of a program. Thus, we designed our abstract debugging machinery to accept an abstract domain $\mathcal{A}$ and a concretization function that maps abstract states to sets of concrete states $\gamma : \mathcal{A} \to \mathcal{P}(C)$. Since we aim to display the analysis results in a debugger view, we demand that each abstract state $a \in \mathcal{A}$ is associated with a unique program location $N(a) \in N$, and for any $c \in \gamma(a)$, we have $N(c) = N(a)$.

Abstract Executions. To step through the results of the analysis computation, we need a representation of the analysis results that is both faithful to the analyzer's reasoning and amenable to the stepping operations of the concrete debugger. An ideal structure for this is our interprocedural Abstract Reachability Graph, which represents how the analyzer's results are derived from the program's control flow depending on how aggressively it has merged different paths into single abstract states.

Definition 1. *An interprocedural Abstract Reachability Graph (iARG) is a directed graph $G' = (N', E', a_0)$ with*

- *a set $N' \subseteq \mathcal{A}$ of abstract states visited by the analyzer,*
- *a set $E' \subseteq N' \times O' \times N'$ of edges using operations $O' = B \cup F \cup \downarrow F \cup F\uparrow$,*
- *an initial state $a_0 \in \mathcal{A}$.*

We denote edges in the iARG as $a \overset{o}{\rightsquigarrow} a'$ where $o \in O'$. Unlike Beyer et al. [8], this relation does not denote the immediate abstract successor relation; for us, these are transitions in the final analysis result where some abstract states are joined with other states based on a merge operation. Unlike concrete traces, the iARG retains the call edges $(a, p(), a')$ that transition from the call site to the next (intraprocedural) location that the function returns to. In concrete execution, reaching a state after a function call involves computing it by traversing the statements within the function. However, in the case of the iARG, the analysis has already computed the states for each location, and has its own way of handling recursive calls, so it should retain the call edge in the iARG that respects same-level runs. This condition can be expressed intuitively as follows:



If there is a concrete execution through a function, reaching a state c', we should have a corresponding call edge in the iARG, allowing stepping over the function to the abstract state a'. More formally, the soundness conditions for the iARG are as follows:

Definition 2. *An iARG is sound w.r.t. the concrete semantics of the program if it satisfies the following two conditions:*

1. *For every concrete execution $c_0 \xrightarrow{e_1} c_1 \xrightarrow{e_2} \cdots \xrightarrow{e_n} c_n$, the iARG contains a corresponding abstract execution path $a_0 \overset{e_1}{\rightsquigarrow} a_1 \overset{e_2}{\rightsquigarrow} \cdots \overset{e_n}{\rightsquigarrow} a_n$ where $c_i \in \gamma(a_i)$ for $0 \leq i \leq n$.*
2. *For every call edge of a function pointer p, if there exists a path through the function f in the concrete world $c \xrightarrow{\downarrow f} c_1 \xrightarrow{\pi}{}^{\star} c_2 \xrightarrow{f\uparrow} c'$, then the iARG contains a direct call edge $a \overset{p()}{\rightsquigarrow} a'$ between all abstract states a and a' with $c \in \gamma(a)$ and $c' \in \gamma(a')$.*

Abstract Operational Semantics. An abstract debugger operates similarly to a conventional debugger and, critically, uses the same user interface. While a conventional debugger lets the user step through the statements executed by a program and observe changes to its concrete state, the abstract debugger allows the user to step through the statements in the program and observe changes to the abstract state (which summarize all possible changes to the concrete state).

In other words, instead of executing program statements, the abstract debugger moves along the iARG of that program and mimics the execution of statements. The iARG models the possible concrete states of the actual program and thus, instead of showing the concrete states of a running program, shows the results of an over-approximating analysis—the abstract states.

$$\textsc{Intraprocedural Path (empty)}\quad \frac{}{a \rightsquigarrow^{\star} a} \qquad\qquad \textsc{Intraprocedural Path (basic)}\quad \frac{a \overset{e}{\rightsquigarrow} a'' \rightsquigarrow^{\star} a' \qquad e \in B \cup F}{a \rightsquigarrow^{\star} a'}$$

$$\textsc{Step into}\quad \frac{a \overset{e}{\rightsquigarrow} a' \qquad e \in {\downarrow}F \cup F{\uparrow} \cup B}{a \overset{\text{into}}{\leadsto} a'} \qquad \textsc{Step over}\quad \frac{a \overset{e}{\rightsquigarrow} a' \qquad e \in F \cup F{\uparrow} \cup B}{a \overset{\text{over}}{\leadsto} a'}$$

$$\textsc{Step out}\quad \frac{a \rightsquigarrow^{\star} a'' \overset{f\uparrow}{\rightsquigarrow} a'}{a \overset{\text{out}}{\leadsto} a'} \qquad \textsc{Step back}\quad \frac{a \overset{e}{\rightsquigarrow} a' \qquad e \in {\downarrow}F \cup F \cup B}{a' \overset{\text{back}}{\leadsto} a}$$

Figure 3. The operational semantics of the abstract debugger directly corresponds to traversal of the iARG.

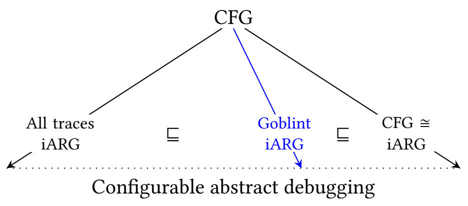
As in the concrete case, Fig. 3 defines the operational semantics of the abstract debugger on the same stepping operations as a transition relation $\leadsto \subseteq \mathcal{A} \times \mathcal{S} \times \mathcal{A}$. Notably, the abstract debugger operates on single abstract states, not their sequences. The presence of function call edges significantly simplifies these rules compared to the concrete ones, due to the second condition from Definition 2. Only stepping out requires extra care using the notion of intraprocedural paths—the abstract version of same-level (balanced) runs. Definition 2 ensures the following soundness statement. For a given program, any debugging session in the concrete world can be replicated in the abstract world.

Theorem 4.1. *Let $c_0 \overset{s_1}{\Longrightarrow} \pi_1 c_1 \overset{s_2}{\Longrightarrow} \cdots \overset{s_n}{\Longrightarrow} \pi_n c_n$ be a debugging session in the concrete world. Then, there exists a corresponding debugging session $a_0 \overset{s_1}{\leadsto} a_1 \overset{s_2}{\leadsto} \cdots \overset{s_n}{\leadsto} a_n$ in the abstract world such that $c_i \in \gamma(a_i)$ for $0 \leq i \leq n$.*

Proof. The proof is by induction on the number of steps n in the debugging session. The base case $n = 0$ holds as we assume $a_0 = \gamma(c_0)$. For each concrete debugging rule, we can extend the abstract debugging session to maintain the invariant $c_i \in \gamma(a_i)$. The first condition ensures that whenever the execution prefix is extended by a concrete state $c \xrightarrow{e} c'$, we can find an iARG transition to a corresponding abstract state $a \overset{e}{\rightsquigarrow} a'$. The second condition covers traces extended via function calls. It directly ensures that any step over action can be soundly replicated by traversing the iARG call edge. More generally, we can show (by induction on the depth of the paths) that for any same-level run $c \xrightarrow{\pi}{}^{\star} c'$ reachable in a concrete execution, there is a corresponding intraprocedural path $a \rightsquigarrow^{\star} a'$ in the iARG. This ensures the correctness of stepping out. □

In regular program execution, a program is traversed along a single path. For instance, in the case of a conditional statement, a specific branch of the conditional is always taken, and during function calls, a specific function is always called. In the abstract debugger, a program is traversed along a subset of all possible paths at once. Thus, there are situations where it is not unambiguously determined which branch of a conditional is taken or which function is called. The semantics in this section is non-deterministic; Section 6 describes how the implementation handles these ambiguous actions.

Configurable Abstract Debugging. Beyer et al. [8] introduced the concept of *configurable program analysis*, wherein configurations can impact the precision and cost of the analysis of a program. Similarly, our approach presents *configurable abstract debugging*, where the choice of underlying analysis method influences the precision and cost of the calculated results, thereby affecting the abstract debugger's functionality. This flexibility offers numerous possibilities for abstract debugging sessions:



The left-hand side showcases the set of all possible traces from the concrete world, representing the most precise and costly approach. This results in a tree of infinite size (which could be constructed lazily), of which a debugging session can only explore a finite portion. Conversely, the right-hand side displays the iARG resulting from the most imprecise analysis. This analysis has to ensure that states for separate program points are kept apart because locations are crucial to the debugging view. In other words, the rightmost iARG is close to the Control Flow Graph: $N' = N$, $a_0 = st_{\text{main}}$ and E' consists of E and an over-approximation of function entry and return edges.

We hypothesize that any configurable program analysis, whose operations satisfy the soundness conditions of Beyer et al. [8], will satisfy our conditions on the iARG, and thus can result in sound abstract debugging. Proving this lies outside the scope of this paper because it would require formalizing the reachability graph construction, which is not explicit in any paper we are aware of. Here, we will instead show how sound iARGs can be constructed based on the approach used in the Goblint analyzer, yielding one instance of abstract debugging.

5 Constructing the iARG in Goblint

Goblint is a static analysis tool for multi-threaded C code based on abstract interpretation [46]. Goblint takes a C program as input and uses CIL [32] to compile it into a simplified subset of C. CIL constructs one CFG per function and these CFGs represent the program's control flow needed for data flow analysis.

The tool comprises various analyses that interact with each other, querying one another within the analyzer through a general interface. Each analysis defines its abstract domain that models the program properties of interest to the analysis, and the corresponding transfer functions that describe changes in program state in that abstract domain. It is possible to turn different analyses on and off as needed [36].

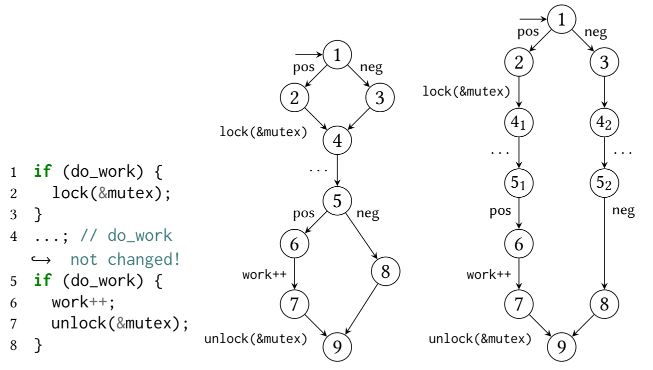
A primary analysis used by Goblint is the base analysis, which investigates the possible values of variables in a program. The abstract domain of the base analysis combines several different abstract domains, including the interval domain, to approximate various types of variables. In addition to the base analysis, Goblint includes other analyses that model the state of the program. For example, the lockset analysis monitors the locks that are definitely acquired at a particular location in the program [46]. The example in Section 2 illustrated both the interval domain and the lockset analysis.

The CFGs and the combined analyses yield a constraint system [1], which is solved using a local generic solver [40]. The solved constraint system encodes a set of produced warnings and the interprocedural Abstract Reachability Graph (plus other results which are less important for this paper).

Two key aspects of Goblint's data flow analysis play a significant role in constructing the reachability graph. Firstly, the analysis is *path-sensitive*, distinguishing different program paths under certain conditions. Secondly, the analysis is *context-sensitive*, distinguishing function calls based on parameter values within an abstract domain, following the functional approach of Sharir and Pnueli [41]. The following paragraphs elaborate on the significance of path- and context-sensitivity in constructing the iARG in Goblint.

Theorem 5.1. *The iARG constructed from the path- and context-sensitive analyses of the Goblint analyzer satisfy the conditions of Definition 2 for a sound iARG construction.*

Expressing path-sensitivity in the iARG. A path-sensitive analysis can distinguish between feasible and infeasible paths within a program. For instance, consider the program shown in Fig. 4a, where a mutex is acquired on line 2 based on the condition `do_work`, and later a variable is accessed based on the same condition [45]. In this example, there are four possible paths through the branches. However, only two of these paths are viable: one where `do_work` evaluates to true and another where it evaluates to false. That is because `do_work` evaluates to the same boolean value at both lines 1



```
1 if (do_work) {
2   lock(&mutex);
3 }
4 ...; // do_work
↪  not changed!
5 if (do_work) {
6   work++;
7   unlock(&mutex);
8 }
```

(a) Program **(b)** CFG **(c)** iARG

Figure 4. An example of a path-sensitive iARG of a program with conditional locking. Note that the iARG does not include the infeasible path crossing from 2 to 8.

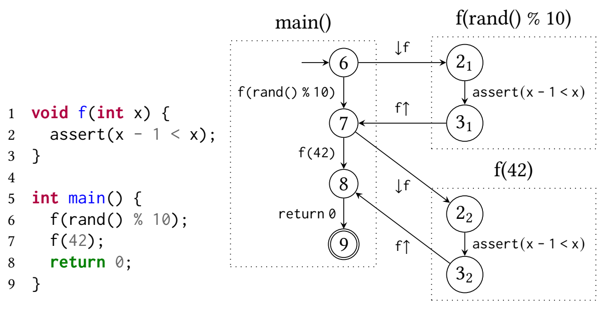
and 5. As a result, a path-sensitive analysis can eliminate the impossible paths, a capability that is reflected in the iARG as shown in Fig. 4c. In comparison to the CFG in Fig. 4b, where only one node represents program locations 4 and 5, the iARG contains separate nodes for these locations for each allowed path, maintaining separate states.

Unlike model checkers, where paths are either fully separated or merged, Goblint's analysis distinguishes paths based on a specific property of interest. In Fig. 4, paths are maintained separately as long as the property of interest—such as the set of held locks—differs. Once the mutex is unlocked on line 7, causing the locksets of the paths to become identical, the paths are merged back together. This precision choice is also why Goblint's iARG tends to lean towards the right side of the configurable abstract debugging spectrum (page 7). While we maintain paths separately in certain instances, we merge them again when there is no difference in the values of the relevant properties that would ensure the paths remain distinct.

Lemma 5.2. *The iARG constructed from the above property-simulation approach to path-sensitive analysis satisfies soundness condition 1.*

Expressing context-sensitivity in the iARG. Whenever the analysis encounters a function call, it notes down the called function along with the context where the function is invoked. The recorded invocation contexts and the CFGs corresponding to the called functions are used to construct segments of the iARG related to the called functions. These segments are constructed using the analysis results of the called function f, as shown in condition 2 of Definition 2.

An iARG may include multiple calls of the same function for different contexts. Take for instance the program depicted in Fig. 5a, where the same function is invoked multiple times with different arguments (lines 6 and 7). In such scenarios, different contexts (arguments) of the same function are



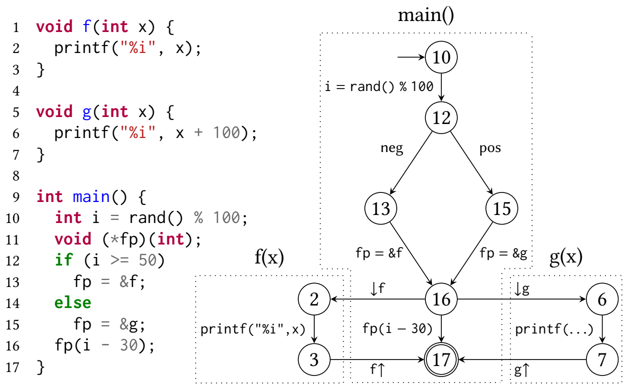
```
1 void f(int x) {
2   assert(x - 1 < x);
3 }
4
5 int main() {
6   f(rand() % 10);
7   f(42);
8   return 0;
9 }
```

(a) Program **(b)** iARG

Figure 5. A program with two calls to the same function with different contexts, at lines 6 and 7, and its iARG.

treated separately, i.e., context-sensitively. This results in the creation of an iARG as depicted in Fig. 5b, which includes different nodes corresponding to the same program location. That is, the nodes 2_1 and 2_2 include the program point on line 2, and nodes 3_1, 3_2 include the program counter of line 3, respectively. Each such iARG vertex represents a state at a program point with a distinct calling context, distinguished by the context recorded in the analysis results.

Furthermore, when a function is invoked through a pointer, the analysis results for each of the functions that may be pointed-to are combined with the calling context and connected to the vertices of the call edge. To illustrate, consider the code snippet in Fig. 6a, where on line 16, there is a function call through a pointer that can point to either function f or g. To get the iARG segment for that function call, the iARG segments of both of the functions are connected to nodes 16 and 17. That is, there are connections 16 $\overset{\downarrow f}{\rightsquigarrow}$ 2 and 16 $\overset{\downarrow g}{\rightsquigarrow}$ 6 for the function enters, and 3 $\overset{f\uparrow}{\rightsquigarrow}$ 17 and 7 $\overset{g\uparrow}{\rightsquigarrow}$ 17 for function returns, as shown in Fig. 6b.



```
1  void f(int x) {
2    printf("%i", x);
3  }
4
5  void g(int x) {
6    printf("%i", x + 100);
7  }
8
9  int main() {
10   int i = rand() % 100;
11   void (*fp)(int);
12   if (i >= 50)
13     fp = &f;
14   else
15     fp = &g;
16   fp(i - 30);
17 }
```

(a) Program **(b)** iARG

Figure 6. A program with calls to different functions through pointers and its iARG.

Additionally, as implied by its name, the reachability graph represents solely the reachable state space of the program. Thus, during the construction of the iARG, any segments of the program that are unreachable are omitted. The unreachable portions of the program are effectively pruned away, ensuring that the resulting iARG represents only the reachable states within the program.

Lemma 5.3. *The iARG constructed from the above functional approach to context-sensitive analysis satisfies soundness condition 2.*

Revisiting the running example. We are now equipped to understand the iARG, as depicted in Fig. 7, for the program discussed in Section 2. We have abbreviated variable names `action` and `mutex` and enum names `CACHE` and `PUBLISH` to their first letter. The most important aspect here is that path splitting occurs on line 17, where the program locks the mutex `m`, resulting in the two nodes $L17^{\{m\}}$ and $L17^{\emptyset}$. The result of the possibly-failing locking operation is stored in a temporary boolean r. This is the only place where there is a potential choice of paths. If locking fails, r gets value true, determining which if branch is taken, and subsequently the argument to the function call. In the iARG, the nodes for the function f are split, and we denote their difference based on the arguments P and C, visible in the node contents, although

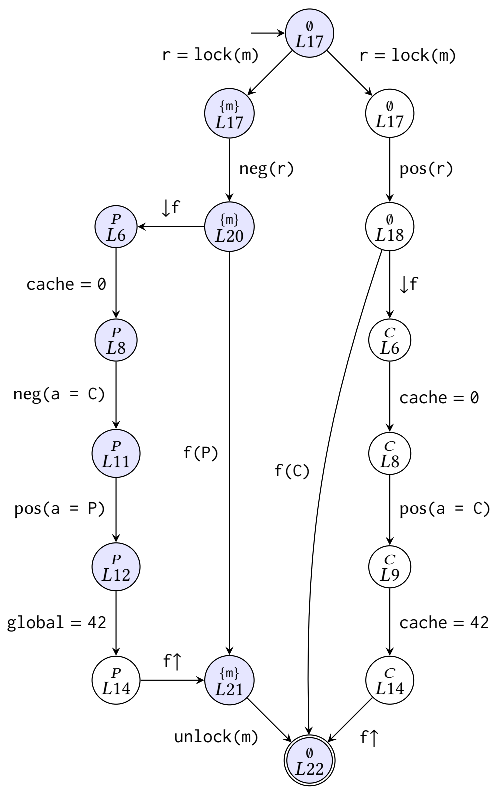


Figure 7. iARG for the example program from Section 2, with nodes visited in the example debug session highlighted.

the context also includes the differing locksets, which we do not show in f's contents. We can now see that the debugging session from Section 2 proceeded along the leftmost path in the iARG, except that we stepped out of the function at node $L12^P$ and thus did not investigate the return node $L14^P$.

6 Implementation

The Goblint abstract debugger implementation[2] leverages *Debug Adapter Protocol (DAP)*, a protocol that allows the IDEs to communicate with various debuggers using a unified, standardized interface. The Goblint abstract debugger is implemented as part of the Visual Studio Code extension GobPie to reuse the existing logic for communicating with Goblint and running analyses [23]. GobPie serves as a user interface for Goblint, displaying potential program errors in the code editor [18]. GobPie relies on the MagpieBridge framework, a general solution for displaying results from static analyzers in IDEs [28]. Currently, GobPie and the newly created abstract debugger are only provided as a Visual Studio Code extension. However, in the future, support can be extended to other IDEs that support DAP. The basic features of the abstract debugger fall into categories:

- Stepping;
- Setting breakpoints;
- Displaying variable values;
- Evaluating expressions (watchlists and console);
- Displaying the call stack.

We discuss some more advanced features in Section 7:

- Debugging multi-threaded code;
- Displaying multiple program states at once;
- Stepping with multiple program states at once;
- Setting conditional breakpoints;
- Visualizing the iARG.

From the analyzer's perspective, the features can be categorized into two groups: those where the analyzer must provide a state when queried for a location, e.g., for stepping or setting a breakpoint, and those that require the analyzer to be able to evaluate expressions in the current state, e.g., for watchlists. We describe how these features are accessible in the Goblint abstract debugger and elaborate on their technical implementation.

Stepping. The abstract debugger supports the stepping operations defined in Section 4. Recall that when stepping in the abstract world, taking a conditional branch or calling a function can be non-deterministic. In such cases our implementation thus requires the user to manually choose the branch or function they wish to enter (via the "step into target" command in DAP). If a stepping operation has no possible targets, the debugger displays an error.

[2]The abstract debugger is available at https://github.com/goblint/GobPie as an open-source project released under the MIT license.

Breakpoints. To select a location of interest for debugging, it is possible to set breakpoints in the program's source code. When debugging is initiated, the abstract debugger queries the states corresponding to the program locations of the set breakpoints from the analyzer. At a breakpoint, the debugger pauses and allows the user to observe the program states and traverse the control flow step-by-step. As Goblint's analysis is path- and context-sensitive, a single breakpoint may correspond to multiple nodes in the reachability graph, which in traditional debugger terminology implies that the same breakpoint is traversed multiple times.

In a conventional debugger, situations where the same breakpoint is crossed multiple times during execution are handled by pausing at the breakpoint each time. However, this has a significant downside: if a breakpoint is hit frequently, it can be difficult for the user to identify the instance of interest, since each pause at the breakpoint can only be observed once. (Conditional breakpoints mitigate this to some extent). To facilitate the identification of interesting cases, the abstract debugger displays the states of all iARG nodes that correspond to the same breakpoint simultaneously.

Displaying Variable Values. The debugger allows the user to view the values of variables at the current state of the program being inspected. In the abstract debugger, the values are represented by abstract domain values. Given the modular structure of Goblint, numerous analyses model different values with different domains. In the upper-left corner of Fig. 1b, an example of displaying variable values in the "Variables" view is provided. Under "Local", the interval domain values for local variables `cache` and `action` are displayed. In addition to local and global variable values, the raw values of Goblint analyses' abstract domains are displayed in the variable view, e.g., the information about the non-value analyses like the set of currently held locks, with synthetic variable name `<mutex>`. When stepping, the debugger automatically highlights variables whose abstract domain values have changed, making it easier to spot state changes.

Evaluating expressions. Apart from variable values, the abstract debugger can also monitor the values of side-effect-free C-language expressions. Users can add an expression of interest to a list of *watch expressions*, causing the expression to be automatically evaluated at the selected state of a current program point each time the point of observation changes. For example in the "Watch" panel beneath the "Variables" view in Fig. 1b, the value of the expression `action == PUBLISH` is displayed. It is also possible to evaluate expressions at a selected state of the current program point by entering the expression into the debugger's console.

Consider setting a breakpoint on line 2 in the program shown in Fig. 5a and running the debugger. Here, the user can evaluate expressions involving the variable x by entering them into the debugger's console and the debugger will show their abstract values. For example, evaluating x could return the interval $[0, 9]$, while evaluating $2x+3$ would yield $[3, 21]$.

Furthermore, the debugger will display all possible states of the program point in the call stack panel (described below). In this scenario, there are two different states, corresponding to the calling contexts of function calls on lines 6 and 7. By default, one of these states will be selected (in the above example, line 6), but the user can switch between them and evaluate the expression in another state. For the state corresponding to the function call on line 6, the debugger evaluates x to $[0, 9]$. However, the user can also select the other state corresponding to the function call from line 7 instead, where evaluating x yields $[42, 42]$.

Call Stack. A conventional debugger displays the call stack for each thread, showing which calls led to the invocation of the currently observed function. In the abstract debugger, the call stack is constructed by finding a path in the iARG leading from the program's starting point to the currently observed node. The call stack view is visible in the bottom-left corner of Fig. 1b. When a breakpoint is set in a function that can be reached through multiple paths in the iARG, the user is presented with the longest common suffix of the paths leading to the node. We chose this compromise because displaying all call stacks in their entirety is impractical. Firstly, there is no good way to display branching call stacks in a conventional debugger interface; and, secondly, in certain cases, the number of possible call stacks can be very large or even infinite.

The call stack view also exposes DAP's "restart frame" operations, which our implementation supports. This command resets execution to the beginning of the chosen call frame by jumping back to the entry point of the function.

7 Advanced Features

To make the discussion in Section 6 clearer, we reserved the less-obvious implementation details and features for the present section.

Debugging multi-threaded code. Thread-modular analysis, as used in Goblint [38, 39], supports a useful mode of debugging multi-threaded programs based on stepping a single thread while resuming and allowing other threads to run freely. Concrete debuggers, such as GDB, also support this feature, and the Debug Adapter Protocol provides the ability to freeze and thaw threads. In this mode, we focus on a single thread, referred to as the *ego thread*, which is stepped while other threads are running. Consequently, the debugger presents the *local view* of a given thread, showing the potential values of shared variables if they were to be read at the given program point. Information about the ego thread is part of the local state and the call context of the analysis. Thread spawning is also considered a call in the call stack view, allowing users to see where the thread was created. This ensures that the entire path from the program's starting point up to the current point is traceable, even for multi-threaded code.

At Breakpoints: Displaying multiple program states at once. While DAP does not natively support the display of multiple program states simultaneously, we repurpose its functionality for displaying different threads to show different states at the same breakpoint. These may or may not belong to the same program thread, as thread identity is also part of the context and state. The screenshot in Fig. 1b shows four different states in the thread view because the function f is called by both program threads, each time with two different actions. Although this "misuse" may initially seem misleading, it serves the purpose well as long as the user understands the thread-modular concept.

Stepping with multiple program states at once. As we repurposed the DAP's thread view due to the thread-modularity of Goblint, we do not permit independent stepping of the different states. This restriction ensures that all states under inspection are always at the same control flow node, having traversed an equivalent path during stepping. That is, when a step is taken in one path, an equivalent step is taken in all other paths under inspection. If an equivalent step is not possible because the required control flow node is not reachable, that state becomes inaccessible. Stepping backward can restore inaccessible paths: if, while stepping back, the node is passed where a path became inaccessible, that path becomes accessible again. This design choice allows users to switch between inspected states without remembering each state's specific path.

Conditional Breakpoints. In addition to regular breakpoints, conditional breakpoints are also supported in the abstract debugger. These pause only at nodes in the iARG that meet a set condition. One way to think about conditional breakpoints in the context of abstract debugging is as a filter on states to be explored. This condition is a C-language expression evaluated at the node in the reachability graph. The expression can use variables from the program being analyzed to check if a desired condition holds at a particular program point. Additionally, users can choose between two modes for checking conditions:

\may: condition holds if true is *among* the possible values of the expression in the observed state.

\must: condition holds if true is *the only* possible value of the expression in the observed state.

The mode can be chosen by prefixing the expression with the corresponding mode symbol. For instance, the condition "\must $a > 0$" holds if the value of variable a is always greater than 0 in the observed state. The backslash at the beginning of the mode symbol makes it clearly distinguishable from a C-language expression. If no mode symbol is provided, the default mode \may is used.

Checking of conditions is implemented using Goblint's server capability to evaluate C-language expressions. The provided expression is evaluated in each node corresponding to the breakpoint's program location and the possible values of the expression are checked according to the chosen mode.

Let us consider the program illustrated in Fig. 5a once again. We can set a conditional breakpoint using the expression $x > 0$ at line 2. In this scenario, the debugger will show the same states as it would with a nonconditional breakpoint or a conditional breakpoint with the expression "\may $x > 0$". That is because, in one state (function call from line 6), the possible values for x fall within the range $[0, 9]$, while in the other state (during the function call from line 7), the value of x is 42. In both states, there exist values of x that are greater than 0, i.e., x *may* be greater than 0. However, if we set a conditional breakpoint with the expression "\must $x > 0$" instead, only the state corresponding to the function call from line 7 is shown. This difference occurs because for the function call from line 6, the condition $\forall x.\ x > 0$ does not hold and, hence, "x *must* be greater than 0" is not satisfied.

Visualizing the iARG. The implementation includes an option to visualize the iARG of the analyzed program using Graphviz, creating a representation similar to that shown in Fig. 7, as is common in many other static analysis tools. This feature is still a work in progress, as its true value will only be realized when the implementation is extended to link the visualized iARG nodes to their corresponding abstract states, allowing navigation through the iARG alongside the debugger's step operations. Moreover, currently, the iARG of the entire program is shown at once, which can quickly become unwieldy and lead to performance issues, particularly in larger programs with code spread across multiple files. A more scalable solution would focus on showing only the relevant fragment of the program being debugged. Since the debugger inherently queries the current, previous, and next iARG nodes, limiting the visualization to these fragments would help address scalability concerns.

8 Examples

In this section, we illustrate the potential of abstract debugging through additional use cases.

Inspecting path-sensitive results. Stepping between states is as essential to understanding warnings in the abstract world as it is for diagnosing failures in the concrete, dynamic world. At a given program point, a failure might occur in only some concrete states but not in others. Similarly, only some abstract states for that program point may correspond to the failure. The abstract debugger facilitates examination of the paths leading to a failing state separately from those paths that do not reach the state, to focus the debugging process on the specific statements leading to the failure.

```
1 void start_scan(void) {
2 for(o.cur_host=0;o.cur_host<o.no_hostnames;o.cur_host++){
3   pthread_mutex_lock(&main_thread_count_mutex);
4   while(o.cur_threads>=o.number_of_threads) {
5     pthread_mutex_unlock(&main_thread_count_mutex);
6     debug("...", o.cur_threads, o.number_of_threads);
7     nanosleep(&tv, NULL);
8   }
9   pthread_mutex_unlock(&main_thread_count_mutex);
```

Figure 8. Path-sensitive fault from smtprc.

Take, for example, a case study extracted from the Smtp Open Relay Checker (smtprc[3]), shown in Fig. 8. The static analyzer emits a data race warning on line 4 because there exists a state where the set of held locks is empty, reflecting the fact that a data race is possible. When a breakpoint is set on line 4, where the warning is indicated, the debugger displays two states: one with an empty set of held locks and another with the element `main_thread_count_mutex` included in the set of held locks. We choose the state where the lock is absent, and a data race is possible; however, the source of the warning remains unclear. As we step through the loop using the "step back" operation, we observe that along the path, the mutex was unlocked on line 5 in the previous loop iteration but not re-acquired. We successfully debugged the program and identified a fault at the end of the loop body, which caused the true positive warning.

Inspecting context-sensitive results. The abstract debugger can also simplify navigation through contexts when searching for the cause of a bug that only appears in specific calling contexts of a function.

Consider a code snippet from The Silver Searcher[4] shown in Fig. 9. To simplify, we have heavily sliced away the non-relevant parts of the code. A static analyzer flags "Must dereference NULL pointer" on line 14. However, this warning only applies to 3 out of 9 calling contexts where `buf_c` is `NULL`. Through the debugger, we observe that all three instances occur during function calls to `is_binary` from `search_buf`, on lines 8 and 10. Stepping back, we can trace the cause all the way back to the assignment of `buf` to `NULL` on line 2, and we observe no `NULL` checks before the call to `is_binary` in the `search_buf` function. Through the abstract debugger's conditional breakpoints and navigation across different contexts, we successfully pinpoint the cause of the warning; however, the static analyzer itself could have done much more to help this process. In particular, it should expose use-def chains to allow the user to navigate along the data flow between the source and sink of such flow patterns that are common in program analysis.

[3]https://sourceforge.net/projects/smtprc/files/smtprc/smtprc-2.0.3/

[4]A fast code searching tool, https://github.com/ggreer/the_silver_searcher

```c
void search_file(const char *file_full_path) {
  char *buf = NULL;
  matches_count = search_buf(buf, f_len, file_full_path);

ssize_t search_buf(const char *buf, ...) {
  int binary = -1;
  if (!opts.search_binary_files && opts.mmap)
    binary = is_binary((const void *)buf, buf_len);
  if (!opts.print_nonmatching_files && ...)
    binary = is_binary((const void *)buf, buf_len);

int is_binary(const void *buf, const size_t buf_len) {
  const unsigned char *buf_c = buf;
  if (buf_len >= 3 && buf_c[0] == 0xEF && ...)
```

Figure 9. Context-sensitive bug from The Silver Searcher.

Understanding the cause of false alarms. Another use case for abstract debugging is to help analysis developers understand unexpected results from their static analyzer. For example, when analyzing the source of EasyLogger[5], the analyzer flags numerous potential races with many accesses, both with and without a mutex. Using the abstract debugger, we could resolve this mystery far more easily than our previous tooling allowed. We stepped forward to the location where the different locksets appear: a branching on a configuration flag `output_lock_enabled` in a structure called `elog`. We searched for where the flag was set, placed a breakpoint on that line, and by stepping out to the calling function and then stepping around, we realized that the initialization of `elog` occurs after other threads have been created. Thus, although the flag is set to true in every path of the main thread, the analyzer infers that other threads could potentially read the default false value for the `output_lock_enabled` flag before it is changed. Such problematic scenarios seem to be avoided by the code, however. EasyLogger uses another flag, `init_ok`, to signal when initialization is complete, but the analyzer cannot automatically show that this flag eliminates the race. Overall, the ability of the abstract debugger to navigate the iARG, stepping outward from a given call context, was particularly helpful in understanding why the analyzer believed the lock might not have been taken.

9 Discussion

Having tested our experimental prototype on a few realistic use cases, we can now reflect on the broader context of this work and highlight the benefits and drawbacks of the design choices we made.

Broader context. While the immediate benefit of abstract debuggers, as demonstrated in our examples, may appear to be quicker bug detection and diagnosis, the broader impact lies in enabling a more intuitive and insightful engagement with static analysis results. By focusing on making these results more accessible and understandable, we can open new possibilities for developers to better navigate and comprehend the complex behaviors of their programs.

The usability aspects of sound static analyzers deserve more research attention, as empirical studies suggest that poor explainability of analysis results is one serious obstacle preventing the wider adoption of static analysis tools [11, 15, 24, 31, 42]. The real challenge is to make these tools truly practical and valuable in everyday programming. As static analysis tool designers, our primary focus is on developing new techniques. However, the evolution of techniques and the usability of these tools must progress together, ensuring that novel methods are developed alongside user interfaces that effectively communicate the results of static analysis. Actively using the interfaces in the tool development process itself is a litmus test for their usability. Our long-term aspiration is to see abstract debuggers form a part of static analysis designers' toolkits in terms of making these tools usable by the broader public; understanding static analysis output is one of the challenges to its everyday deployment, and we believe that our approach can help here.

Reverse Debugging. For a reverse debugger that operates at the concrete level, its implementation may impose significant extra overhead on program executions. This has been an obstacle to the widespread adoption of reverse debugging. Our abstract debugger is based on static analysis results, which is analogous to recording a representation of all possible program executions. Thus, we can simulate the behavior of a live reverse debugger with no runtime overhead, as the analysis has already been performed. The abstract debugger, however, requires its users to take a different perspective on their code, one more centered on program correctness than on a specific sequence of events. Our (admittedly biased) stance is that this perspective is valuable for understanding important program properties, specifically those that concern what could possibly happen. Such a shift is required if we hope to prevent software vulnerabilities, rather than patching them as they are detected in already-deployed software.

Shoehorning into a standardized protocol. While shoehorning a non-deterministic abstract interpretation into DAP may seem like a challenging fit, our experience has still demonstrated notable benefits by offering practical advantages over our previous (custom) interface for visualizing analysis results. For instance, our implemented abstract debugger leverages several features inherent to IDEs through DAP that would otherwise require significant engineering effort. The IDE provides many valuable functionalities such as displaying states in relation to source code locations using breakpoints, highlighting state changes, and tracking values of interest with watch expressions. Utilizing DAP, these

[5]An efficient logging library for C projects ensuring thread-safe operations, https://github.com/armink/EasyLogger

features come essentially for free, making our abstract debugger easier to use and more effective for the same purposes compared to our custom interface.

The first two use cases in our Examples section showcased the benefits of the debugger interface when navigating results from a path- and context-sensitive analysis. For a context-insensitive analysis, the usability benefits of working within an industrial-strength IDE remain; however, the benefits of step-based execution are less pronounced. Our method especially shines for context-sensitive analysis results, where the call history plays a significant role. The ability to interactively navigate this history is a key advantage of the abstract debugging approach, allowing the interface to display only relevant contexts, unlike static visualizations where all call contexts are displayed at once.

Visualizing abstract states. In this work we focused on developing a sound method for easy navigation of the static analysis results, and implemented an instance using this method into an IDE extension. Our prototype includes predefined representations for abstract values, like intervals or structs, displayed as variable values. However, to increase versatility, provisions for adding visualizations for new abstract domains are needed. Future development can focus on improving the visual representations to accommodate a broader range of abstract domains and more complex data, such as abstract values for relational domains.

As we enhance the clarity of abstract states, we can also explore automated stringification methods or provide configurable options for users to extract important information from raw states. Inspiration could be drawn from the work of Apriyadi et al. [4], who proposed a configurable method for state visualization using a domain-specific language (DSL). As mentioned above, the development and integration of new visual representations and methods should go hand in hand with the development of new analyses and domains themselves.

On some design choices. In contrast to the forward stepping operations (step into, step over, and step out), which each have distinct functions and buttons, DAP only defines *one* operation for backward stepping. Thus, in our work we designed our step back operation to be dual to step over. Alternatively, step back could be defined to be the inverse of step into or step out.

Defining step back as the dual of step into would mean that, when the last transition was a function return, step back would move to the location of the function's return instead of the location of the function's call. The concrete operational semantics are as follows:

$$\textsc{Step back into (basic, entry, return)}\quad \frac{c \xrightarrow{e} c' \qquad e \in B \cup {\downarrow}F \cup F{\uparrow}}{\cdots cc' \xRightarrow{\text{back into}} \cdots c}$$

Defining step back as the inverse of step out would behave differently from our semantics in the cases where the last transition was either a return or basic operation—stepping back and out of the function instead of just returning to the previous location within the same function. The concrete operational semantics of this are defined as:

$$\textsc{Step back out (entry)}\quad \frac{c \xrightarrow{\downarrow f} c'}{\cdots cc' \xRightarrow{\text{back out}} \cdots c} \qquad \textsc{Step back out (basic, return)}\quad \frac{c \xrightarrow{\downarrow f} c_1 \xrightarrow{\pi'}{}^{\star} c' \qquad \pi = c_1\pi'}{\cdots c\pi c' \xRightarrow{\text{back out}} \cdots c}$$

To offer the same capabilities and options for backward stepping as for forward stepping, the Debug Adapter Protocol would need to be extended to better accommodate debuggers with backward debugging features. In the absence of such extensions, the most practical options are to define step back as either "step back over" or "step back into". In our implementation, we opted for the former. The same effect as the third choice, "step back out", can be achieved by restarting the current call frame and then stepping back from the entry point.

10 Related Work

Here, we discuss the term "abstract debugging" and relate our study to several works that have focused on improving the usability and explainability of static analyzers, highlighting how our contributions differ from prior efforts.

The definition of an abstract debugger. In the literature, there are alternative methods described as *abstract debugging*. For instance, Bourdoncle [9] characterizes abstract debugging as employing an abstract interpretation-based static analysis tool for program debugging. Monat et al. [30] define abstract debugger as a method for debugging the implementation of an abstract interpretation-based tool—thus, the word "abstract" referring to the target that is being dynamically debugged. In our approach, however, the term "abstract" refers to the abstraction of concrete debugger behaviors, similar to how it refers to the abstraction of concrete interpretation in the term "abstract interpretation" itself.

Explainable Verification. This work was originally motivated by our attempts to explain the correctness claims of our analyzer. When a tool identifies a flaw in the program, it may be able to produce a counterexample execution trace that is useful for debugging the program and understanding the flaw. For instance, this has been critical to the success of model checking [13]. In contrast, when a sound analyzer verifies the absence of errors in a program, it does not produce an equivalent human-readable artefact to explain this verdict. An important goal is to support automated verifiers in proving that a property holds along all possible executions of the program, but in a way that is interpretable by humans [5].

Apinis and Vojdani [3] provide a framework for explaining the results of abstract interpretation based on approaches to decomposing variable dependencies within a domain [2]. The explanations produced are shown in a custom user interface, so it would be interesting to attempt to shoehorn these explanations into a standard protocol like DAP. This falls under the general approach to deriving meta-analyses developed by Cousot et al. [14]. Meta-analyses could further improve usability of verification tools, although the current focus is on quantification of precision loss [10, 21].

Verifier-Based Debuggers. We highlight some works that rely on formal verification tools as the engine for step-based program execution or simulation through a debugger user interface. Karmios et al. [25] introduced a symbolic debugger built on Gillian, a multi-language platform for developing symbolic analysis tools based on separation logic [20, 29]. Their debugger features a tailored interface for navigating branching execution paths and state matching. For non-deterministic and probabilistic programs—which are not our focus—multiverse analysis and their visualization via multiverse debugging has been proposed [22, 34, 43].

There are also innovative and creative uses of the Debug Adapter Protocol to display the proof state of a deductive verifier. Ernst et al. [19] integrated a debugging feature into the SecC deductive program verification tool, which also relies on symbolic execution. TLC, a model checker for TLA$^+$ models and specifications [47], offers a debugger through its VS Code extension [27].

There have also been proposals for performing debugging-like stepping through static analysis results using custom graphical user interfaces (GUIs). For instance, CBMC-UI by Clarke et al. [12] allows users to step through counterexample traces in a manner similar to traditional debuggers. Similarly, Beyer and Dangl [7] proposed an interface where the counterexample traces are highlighted within an ARG and can be replayed by stepping through the trace.

Similar to our approach, the aforementioned methods use static analysis results to perform debugging-like stepping through those results within an IDE or GUI. However, instead of debugging with results from a specific analysis technique, our approach is configurable and enables debugging using analyses based on abstract interpretation, model checking, or something in between. This abstraction of concrete debuggers is based on our formalization of a concrete debugger's operational semantics. While we have seen some formalizations of programming tools [6, 33], we are not aware of any other operational semantics for a standard debugger.

Debugging Static Analyzers. While the above works primarily target the end-users of the analysis tools, and thus aim to make the analysis computation less visible, there is also an important area of research focused on the debugging and maintenance of the analysis tools themselves. VisuFlow [16, 17] and Mopsa [30] allow simultaneous tracing of both source code and the static analyzer, which is ideally suited to find static analyzer bugs or precision loss during fixpoint computation. Mopsa [30] offers debugging using a GDB-like interface integrated into an IDE using the Debug Adapter Protocol, whereas VisuFlow is a separate Eclipse plugin that does not aim to mimic a traditional debugger. Van Molle et al. [44] introduced a technique called cross-level debugging, specifically designed to enhance the debugging of static analyzers.

In contrast to the aforementioned tools and techniques, our focus is on inspecting analysis results to understand the results computed for a given program under analysis, rather than on debugging the static analyzer itself. We still found the abstract debugger to be useful for understanding the causes of false alarms; however, our tool cannot be used to debug computational aspects. For example, if our analysis does not terminate or crashes, there is no output for our abstract debugger to inspect.

11 Conclusion

We have introduced the concept of configurable abstract debugging and an instantiation of it based on the static analyzer Goblint. Abstract debugging allows developers to confirm or refine hypotheses about the behavior of their programs by interactively exploring the abstract states computed by static analysis tools. Unlike concrete debugging, which only shows results for executions that actually run, abstract debugging leverages static analysis to enable reasoning over all executions.

We have derived the operational semantics of abstract debugging from the operational semantics of concrete debugging, and implemented an abstract debugging tool that makes Goblint's analysis results available to the VS Code debugger via the Debug Adapter Protocol. Our approach ensures the soundness of abstract debugging w.r.t. the concrete semantics of the program. We aim to make state-of-the-art over-approximating analyses more accessible to developers and suggest the potential for further research in this direction. While additional evaluation is needed, our proposed approach could enhance both developer understanding and the effectiveness of these tools.

Acknowledgments

This research was co-funded by the European Union and the Estonian Research Council via project TEM-TA119. We gratefully acknowledge SIGPLAN-M for introducing the authors of this paper, thus making this collaboration possible.

References

[1] Kalmer Apinis, Helmut Seidl, and Vesal Vojdani. 2012. Side-Effecting Constraint Systems: A Swiss Army Knife for Program Analysis. In *Programming Languages and Systems*, Ranjit Jhala and Atsushi Igarashi (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 157–172. https://doi.org/10.1007/978-3-642-35182-2_12

[2] Kalmer Apinis, Varmo Vene, and Vesal Vojdani. 2018. Demand-driven interprocedural analysis for map-based abstract domains. *Journal of Logical and Algebraic Methods in Programming* 100 (Nov. 2018), 57–70. https://doi.org/10.1016/j.jlamp.2018.06.003

[3] Kalmer Apinis and Vesal Vojdani. 2023. Context-Sensitive Meta-Constraint Systems for Explainable Program Analysis. In *Tools and Algorithms for the Construction and Analysis of Systems (Lecture Notes in Computer Science)*, Sriram Sankaranarayanan and Natasha Sharygina (Eds.). Springer Nature Switzerland, Cham, 453–472. https://doi.org/10.1007/978-3-031-30820-8_27

[4] Rifqi Adlan Apriyadi, Hidehiko Masuhara, and Youyou Cong. 2023. Program State Visualizer with User-Defined Representation Conversion (WIP). In *Proceedings of the 1st ACM International Workshop on Future Debugging Techniques* (Seattle, WA, USA) *(DEBT 2023)*. Association for Computing Machinery, New York, NY, USA, 5–10. https://doi.org/10.1145/3605155.3605863

[5] Christel Baier and Holger Hermanns. 2021. From Verification to Explanation (Track Introduction). In *Leveraging Applications of Formal Methods, Verification and Validation: Tools and Trends*, Tiziana Margaria and Bernhard Steffen (Eds.). Springer International Publishing, Cham, 1–7. https://doi.org/10.1007/978-3-030-83723-5_1

[6] Karen L. Bernstein and Eugene W. Stark. 1995. Operational Semantics of a Focusing Debugger. *Electronic Notes in Theoretical Computer Science* 1 (1995), 13–31. https://doi.org/10.1016/S1571-0661(04)80002-1 MFPS XI, Mathematical Foundations of Programming Semantics, Eleventh Annual Conference.

[7] Dirk Beyer and Matthias Dangl. 2016. Verification-Aided Debugging: An Interactive Web-Service for Exploring Error Witnesses. In *Computer Aided Verification*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer International Publishing, Cham, 502–509. https://doi.org/10.1007/978-3-319-41540-6_28

[8] Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. 2007. Configurable software verification: concretizing the convergence of model checking and program analysis. In *Proceedings of the 19th International Conference on Computer Aided Verification* (Berlin, Germany) *(CAV'07)*. Springer-Verlag, Berlin, Heidelberg, 504–518. https://doi.org/10.1007/978-3-540-73368-3_51

[9] François Bourdoncle. 1993. Abstract debugging of higher-order imperative languages. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation* (Albuquerque, New Mexico, USA) *(PLDI '93)*. Association for Computing Machinery, New York, NY, USA, 46–55. https://doi.org/10.1145/155090.155095

[10] Marco Campion, Mila Dalla Preda, and Roberto Giacobazzi. 2022. Partial (In)Completeness in abstract interpretation: limiting the imprecision in program analysis. *Proceedings of the ACM on Programming Languages* 6, POPL (Jan. 2022), 59:1–59:31. https://doi.org/10.1145/3498721

[11] Maria Christakis and Christian Bird. 2016. What Developers Want and Need from Program Analysis: An Empirical Study. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*. ACM, New York, NY, USA, 332–343. https://doi.org/10.1145/2970276.2970347

[12] Edmund Clarke, Daniel Kroening, and Flavio Lerda. 2004. A Tool for Checking ANSI-C Programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, Kurt Jensen and Andreas Podelski (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 168–176. https://doi.org/10.1007/978-3-540-24730-2_15

[13] Edmund M. Clarke, E. Allen Emerson, and Joseph Sifakis. 2009. Model checking: algorithmic verification and debugging. *Commun. ACM* 52, 11 (Nov. 2009), 74–84. https://doi.org/10.1145/1592761.1592781

[14] Patrick Cousot, Roberto Giacobazzi, and Francesco Ranzato. 2019. A^2I: abstract^2 interpretation. *Proceedings of the ACM on Programming Languages* 3, POPL (Jan. 2019), 42:1–42:31. https://doi.org/10.1145/3290355

[15] Lisa Nguyen Quang Do and Eric Bodden. 2022. Explaining Static Analysis With Rule Graphs. *IEEE Transactions on Software Engineering* 48, 2 (Feb. 2022), 678–690. https://doi.org/10.1109/TSE.2020.2999534 Conference Name: IEEE Transactions on Software Engineering.

[16] Lisa Nguyen Quang Do, Stefan Krüger, Patrick Hill, Karim Ali, and Eric Bodden. 2018. VisuFlow: A Debugging Environment for Static Analyses. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*. Association for Computing Machinery, New York, NY, USA, 89–92. https://doi.org/10.1145/3183440.3183470

[17] Lisa Nguyen Quang Do, Stefan Krüger, Patrick Hill, Karim Ali, and Eric Bodden. 2020. Debugging Static Analysis. *IEEE Transactions on Software Engineering* 46, 7 (July 2020), 697–709. https://doi.org/10.1109/TSE.2018.2868349

[18] Julian Erhard, Simmo Saan, Sarah Tilscher, Michael Schwarz, Karoliine Holter, Vesal Vojdani, and Helmut Seidl. 2022. Interactive Abstract Interpretation: Reanalyzing Whole Programs for Cheap. arXiv:2209.10445 [cs.PL] https://arxiv.org/abs/2209.10445

[19] Gidon Ernst, Johannes Blau, and Toby Murray. 2021. Deductive Verification via the Debug Adapter Protocol. In *Proc. of Formal Integrated Development Environment (F-IDE)*. arXiv:2108.02968 [cs.LO] https://arxiv.org/abs/2108.02968

[20] José Fragoso Santos, Petar Maksimović, Sacha-Élie Ayoun, and Philippa Gardner. 2020. Gillian, part i: a multi-language platform for symbolic execution. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 927–942. https://doi.org/10.1145/3385412.3386014

[21] Roberto Giacobazzi, Francesco Logozzo, and Francesco Ranzato. 2015. Analyzing Program Analyses. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*. Association for Computing Machinery, New York, NY, USA, 261–273. https://doi.org/10.1145/2676726.2676987

[22] Ken Gu, Eunice Jun, and Tim Althoff. 2023. Understanding and Supporting Debugging Workflows in Multiverse Analysis. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems (CHI '23)*. Association for Computing Machinery, New York, NY, USA, 1–19. https://doi.org/10.1145/3544548.3581099

[23] Karoliine Holter, Juhan Oskar Hennoste, Simmo Saan, Patrick Lam, and Vesal Vojdani. 2024. Abstract Debugging with GobPie. In *Proceedings of the 2nd ACM International Workshop on Future Debugging Techniques* (Vienna, Austria) *(DEBT 2024)*. Association for Computing Machinery, New York, NY, USA, 2 pages. https://doi.org/10.1145/3678720.3685320

[24] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why Don't Software Developers Use Static Analysis Tools to Find Bugs?. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 672–681. https://doi.org/10.1109/ICSE.2013.6606613

[25] Nat Karmios, Sacha-Élie Ayoun, and Philippa Gardner. 2023. Symbolic Debugging with Gillian. In *Proceedings of the 1st ACM International Workshop on Future Debugging Techniques* (Seattle, WA, USA) *(DEBT 2023)*. Association for Computing Machinery, New York, NY, USA, 1–2. https://doi.org/10.1145/3605155.3605861

[26] Amy J. Ko and Brad A. Myers. 2004. Designing the whyline: a debugging interface for asking questions about program behavior. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Vienna, Austria) *(CHI '04)*. Association for Computing Machinery, New York, NY, USA, 151–158. https://doi.org/10.1145/985692.985712

[27] Markus Alexander Kuppe. 2021. TLA+ for Visual Studio Code: Add TLC Debugger. https://github.com/tlaplus/vscode-tlaplus/pull/214

[28] Linghui Luo, Julian Dolby, and Eric Bodden. 2019. MagpieBridge: A General Approach to Integrating Static Analyses into IDEs and Editors (Tool Insights Paper). In *33rd European Conference on Object-Oriented Programming, ECOOP 2019, July 15-19, 2019, London, United Kingdom*

(LIPIcs, Vol. 134), Alastair F. Donaldson (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 21:1–21:25. https://doi.org/10.4230/LIPIcs.ECOOP.2019.21

[29] Petar Maksimović, Sacha-Élie Ayoun, José Fragoso Santos, and Philippa Gardner. 2021. Gillian, Part II: Real-World Verification for JavaScript and C. In *Computer Aided Verification*, Alexandra Silva and K. Rustan M. Leino (Eds.). Springer International Publishing, Cham, 827–850.

[30] Raphaël Monat, Abdelraouf Ouadjaout, and Antoine Miné. 2024. Easing Maintenance of Academic Static Analyzers. arXiv:2407.12499 [cs.PL] https://arxiv.org/abs/2407.12499

[31] Marcus Nachtigall, Lisa Nguyen Quang Do, and Eric Bodden. 2019. Explaining Static Analysis - A Perspective. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*. 29–32. https://doi.org/10.1109/ASEW.2019.00023 ISSN: 2151-0830.

[32] George C. Necula, Scott McPeak, Shree P. Rahul, and Westley Weimer. 2002. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *Compiler Construction*, R. Nigel Horspool (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 213–228. https://doi.org/10.1007/3-540-45937-5_16

[33] Cyrus Omar, Ian Voysey, Michael Hilton, Jonathan Aldrich, and Matthew A. Hammer. 2017. Hazelnut: a bidirectionally typed structure editor calculus. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL '17)*. Association for Computing Machinery, New York, NY, USA, 86–99. https://doi.org/10.1145/3009837.3009900

[34] Matthias Pasquier, Ciprian Teodorov, Frédéric Jouault, Matthias Brun, Luka Le Roux, and Loïc Lagadec. 2023. Temporal Breakpoints for Multiverse Debugging. In *Proceedings of the 16th ACM SIGPLAN International Conference on Software Language Engineering (SLE 2023)*. Association for Computing Machinery, New York, NY, USA, 125–137. https://doi.org/10.1145/3623476.3623526

[35] Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '95)*. ACM, New York, NY, USA, 49–61. https://doi.org/10.1145/199448.199462

[36] Simmo Saan, Michael Schwarz, Kalmer Apinis, Julian Erhard, Helmut Seidl, Ralf Vogler, and Vesal Vojdani. 2021. Goblint: Thread-Modular Abstract Interpretation Using Side-Effecting Constraints. In *Tools and Algorithms for the Construction and Analysis of Systems*, Jan Friso Groote and Kim Guldstrand Larsen (Eds.). Springer International Publishing, Cham, 438–442. https://doi.org/10.1007/978-3-030-72013-1_28

[37] Anthony Savidis and Vangelis Tsiatsianas. 2021. Implementation of Live Reverse Debugging in LLDB. arXiv:2105.12819 [cs.SE] http://arxiv.org/abs/2105.12819

[38] Michael Schwarz, Simmo Saan, Helmut Seidl, Kalmer Apinis, Julian Erhard, and Vesal Vojdani. 2021. Improving Thread-Modular Abstract Interpretation. In *Static Analysis (Lecture Notes in Computer Science)*, Cezara Drăgoi, Suvam Mukherjee, and Kedar Namjoshi (Eds.). Springer International Publishing, Cham, 359–383. https://doi.org/10.1007/978-3-030-88806-0_18

[39] Michael Schwarz, Simmo Saan, Helmut Seidl, Julian Erhard, and Vesal Vojdani. 2023. Clustered Relational Thread-Modular Abstract Interpretation with Local Traces. In *Programming Languages and Systems (Lecture Notes in Computer Science)*, Thomas Wies (Ed.). Springer Nature Switzerland, Cham, 28–58. https://doi.org/10.1007/978-3-031-30044-8_2

[40] Helmut Seidl and Ralf Vogler. 2021. Three improvements to the top-down solver. *Mathematical Structures in Computer Science* 31, 9 (2021), 1090–1134. https://doi.org/10.1017/S0960129521000499

[41] Micha Sharir and Amir Pnueli. 1981. Two approaches to interprocedural data flow analysis. *Program flow analysis: theory and applications* (1981), 189–234.

[42] Daniil Tiganov, Lisa Nguyen Quang Do, and Karim Ali. 2022. Designing UIs for static-analysis tools. *Commun. ACM* 65, 2 (Jan. 2022), 52–58. https://doi.org/10.1145/3486600

[43] Carmen Torres Lopez, Robbert Gurdeep Singh, Stefan Marr, Elisa Gonzalez Boix, and Christophe Scholliers. 2019. Multiverse Debugging: Non-Deterministic Debugging for Non-Deterministic Programs. In *33rd European Conference on Object-Oriented Programming (ECOOP 2019) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 134)*, Alastair F. Donaldson (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 27:1–27:30. https://doi.org/10.4230/LIPIcs.ECOOP.2019.27

[44] Mats Van Molle, Bram Vandenbogaerde, and Coen De Roover. 2023. Cross-Level Debugging for Static Analysers. In *Proceedings of the 16th ACM SIGPLAN International Conference on Software Language Engineering* (Cascais, Portugal) *(SLE 2023)*. Association for Computing Machinery, New York, NY, USA, 138–148. https://doi.org/10.1145/3623476.3623512

[45] Vesal Vojdani, Kalmer Apinis, Vootele Rõtov, Helmut Seidl, Varmo Vene, and Ralf Vogler. 2016. Static race detection for device drivers: the Goblint approach. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (Singapore, Singapore) *(ASE '16)*. Association for Computing Machinery, New York, NY, USA, 391–402. https://doi.org/10.1145/2970276.2970337

[46] Vesal Vojdani, Kalmer Apinis, Vootele Rõtov, Helmut Seidl, Varmo Vene, and Ralf Vogler. 2016. Static race detection for device drivers: the Goblint approach. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. Association for Computing Machinery, New York, NY, USA, 391–402. https://doi.org/10.1145/2970276.2970337

[47] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. 1999. Model Checking TLA+ Specifications. In *Correct Hardware Design and Verification Methods*, Laurence Pierre and Thomas Kropf (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 54–66. https://doi.org/10.1007/3-540-48153-2_6

[48] Andreas Zeller. 2005. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.