Abstract Debugging with GobPie

Karoliine Holter
University of Tartu
Tartu, Estonia

Juhan Oskar Hennoste
University of Tartu
Tartu, Estonia

Simmo Saan
University of Tartu
Tartu, Estonia

Patrick Lam
University of Waterloo
Waterloo, Canada

Vesal Vojdani
University of Tartu
Tartu, Estonia

ABSTRACT

GobPie is an IDE integration designed to enhance the usability and explainability of the abstract interpretation-based static analyzer Goblint. GobPie features abstract debugging, a novel approach to presenting static analysis results, which complements traditional debugging methods by making program analysis results visible. Its goal is to help resolve rare but real software issues. Unlike traditional debugging, which proceeds step-by-step to observe concrete states, abstract debugging uses static analysis results to simulate the same steps, offering insights into all possible execution paths.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; **Automated static analysis**.

KEYWORDS

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

ACM Reference Format:
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 (DEBT '24), September 19, 2024, Vienna, Austria.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3678720.3685320

1 INTRODUCING GOBPIE

GobPie[1] is a Visual Studio Code extension designed to provide an intuitive user interface for Goblint[1], an abstract interpretation-based static analyzer for C code. GobPie lets developers explore Goblint's results directly within the IDE, using both the code editor and debugger as visualization interfaces. The analysis integration leverages the MagpieBridge framework [7] to support the Language Server Protocol (LSP), while the abstract debugger communicates through the Debug Adapter Protocol (DAP). Goblint has a server mode that communicates with GobPie via IPC sockets, allowing the analyzer to remain active throughout the editing session without needing to restart for each analysis [2].

[1] https://github.com/goblint/GobPie and https://github.com/goblint/analyzer

GobPie's visualization makes Goblint's warnings about potential safety property violations visible in the code editor and offers an abstract debugging feature for identifying the causes of these potential issues. Thus, on one hand, GobPie facilitates the interactive use of Goblint to explore the warnings and the abstract program states in a robust and professional user interface. On the other hand, the abstract debugging feature can be seen as an augmentation of a debugger with sound static analysis results, bringing a formal verification tool into a setting already familiar to developers.

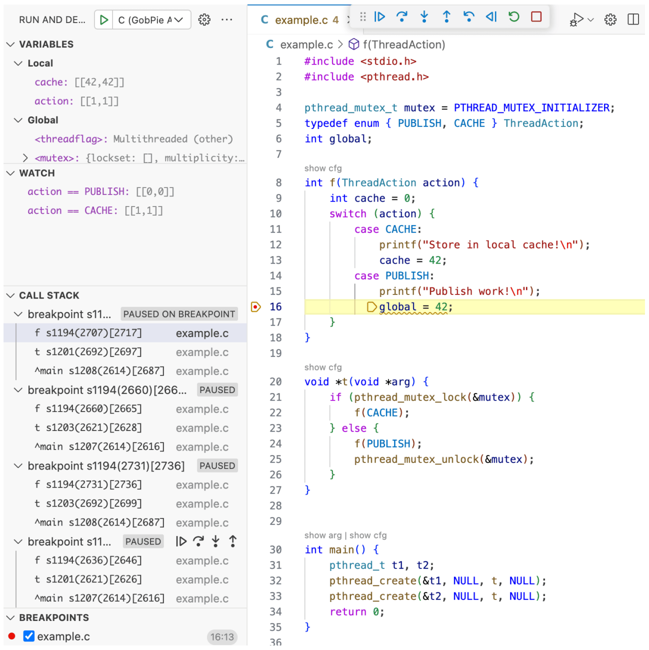
2 ABSTRACT DEBUGGING WITH GOBPIE

In this demo, we show a fix to a race condition using GobPie, where we use GobPie's debugger interface to navigate Goblint's static analysis results. GobPie implements the concept of abstract debugging [4]. An abstract debugger operates similarly to a conventional debugger, presenting the same well-known user interface. However, unlike traditional debuggers, which allow step-by-step execution of a program to observe changes in the concrete state, the abstract debugger simulates these steps using results from static analysis. Instead of working with concrete states, the abstract debugger enables users to observe abstract states, which represent information from all possible executions. The user can explore potential issues that may arise in different execution scenarios without the need for specific input values.

The abstract debugger navigates an Abstract Reachability Graph (ARG) of the program, which models an over-approximation of the possible concrete states. This navigation mimics the execution of statements in a traditional debugger, with stepping operations functioning similarly. The features of the abstract debugger using the built-in user interface of the IDE are summarized below, highlighting its ability to step through all execution scenarios and the similarities it shares with traditional debugging.

Stepping and Breakpoints. GobPie's abstract debugging supports all basic stepping operations—step into, step over, step out, and step back—and the ability to set breakpoints. As Goblint's analysis is path- and context-sensitive, as well as thread-modular, a single breakpoint may correspond to multiple abstract states, which means that the program point could be reached by taking different paths in the program.

Displaying multiple program states at once. The main advantage of the abstract debugger is its ability to display multiple program states simultaneously, facilitating the debugging of all possible execution paths in parallel. As DAP does not have dedicated support for displaying multiple program states at once, the functionality for displaying different threads is repurposed for showing different



```
VARIABLES
  Local
    cache: [[42,42]]
    action: [[1,1]]
  Global
    <threadflag>: Multithreaded (other)
  > <mutex>: {lockset: [], multiplicity:…
WATCH
  action == PUBLISH: [[0,0]]
  action == CACHE: [[1,1]]

CALL STACK
  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
BREAKPOINTS
  ● ✓ example.c                     16:13
```

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

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

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;
    }
}

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


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

Figure 1: A screenshot of the abstract debugger on a flawed program. There is a breakpoint at the warning location. Stepping back will reveal the cause.

states of the same breakpoint. The screenshot in Fig. 1 shows all four states as separate threads in the "Call Stack" view on the bottom left, even though in reality, there are two threads with two contexts each. Repurposing the thread display feature is a reasonable compromise because Goblint's thread-modular analysis focuses on the local view of a single thread, referred to as the *ego thread*, while handling other threads as if they run freely [9]. Thus, while stepping, we see the abstract states from the perspective of the ego thread. In this way, GobPie reuses the existing infrastructure to provide a visualization tool for debugging multi-threaded C programs, ensuring that users can navigate and analyze multiple thread-modular states without requiring a completely new interface.

Overview of Additional Features. Several additional features appear in the demonstration. The call stack for each observable state is visible in the bottom-left corner of Fig. 1, displaying the function calls and thread creations leading to the current program point. Besides regular breakpoints, GobPie supports conditional breakpoints that pause only at states that meet a set condition, effectively filtering the states to be explored. Variable values are displayed in the "Variables" view using abstract domain values. For example, interval domain values are shown for local variables such as `cache` and `action`. For the shared (global) variables, the local view of the ego thread shows the potential values if they were to be read at the current program point. Additionally, the raw values of Goblint analyses' abstract domains are displayed, such as the information about the non-value analyses like the set of currently held locks, labeled with `<mutex>`. The debugger can also evaluate side-effect-free C expressions added to a list of *watch expressions*, which are automatically evaluated at each program point change. This is illustrated in the "Watch" panel beneath the "Variables" view in Fig. 1, where the expression `action == PUBLISH` is displayed.

Related work. Related work on static analysis-based debuggers using DAP for source code includes Gillian's symbolic debugger [5] based on symbolic execution, SecC integration of a debugging feature [3] into its autoactive verifier of C programs based on symbolic execution, and TLC model checker tool's VS Code extension debugger [6] based on TLA+ models and specifications.

Some static analysis based debuggers leverage DAP for debugging the static analysis itself. These include VisuFlow [1], which allows tracing the source code of a static analyzer simultaneously with the program based on data flow analysis, cross-level debugging for static analysis [10], and a GDB-like interface to the abstract interpretation of the program [8].

3 DEMONSTRATION

In this demonstration, we showcase GobPie and its abstract debugging capabilities through some use cases and introduce its features. We have also included a video showcasing how to debug and fix a data race in the source code of SMTP Relay Checker using GobPie: https://youtu.be/KtLFdxMAdD8.

ACKNOWLEDGMENTS

This research was co-funded by the European Union and the Estonian Research Council via project TEM-TA119.

REFERENCES

[1] 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)*. ACM, New York, NY, USA, 89–92. https://doi.org/10.1145/3183440.3183470

[2] 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]

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

[4] 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! 2024)*. ACM. To appear.

[5] 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 (DEBT 2023)*. ACM, New York, NY, USA, 1–2. https://doi.org/10.1145/3605155.3605861

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

[7] 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 (LIPIcs, Vol. 134)*. Schloss Dagstuhl - LZI, Dagstuhl, Germany, 21:1–21:25. https://doi.org/10.4230/LIPIcs.ECOOP.2019.21

[8] Raphaël Monat, Abdelraouf Ouadjaout, and Antoine Miné. 2024. Easing Maintenance of Academic Static Analyzers. arXiv:2407.12499 [cs.PL]

[9] Michael Schwarz, Simmo Saan, Helmut Seidl, Kalmer Apinis, Julian Erhard, and Vesal Vojdani. 2021. Improving Thread-Modular Abstract Interpretation. In *Static Analysis (LNCS, Vol. 12913)*. Springer, Cham, 359–383. https://doi.org/10.1007/978-3-030-88806-0_18

[10] 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 (SLE 2023)*. ACM, New York, NY, USA, 138–148. https://doi.org/10.1145/3623476.3623512

Received 2024-06-21; accepted 2024-07-22