Symbolic Debugging with Gillian

Nat Karmios
Imperial College London
London, UK

Sacha-Élie Ayoun
Imperial College London
London, UK

Philippa Gardner
Imperial College London
London, UK

ABSTRACT

Software debugging for concrete execution enjoys a mature suite of tools, but debugging symbolic execution is still in its infancy. It carries unique challenges, as a single state can lead to multiple branches representing different sets of conditions, and symbolic states must be 'matched' against logical conditions. Some of today's otherwise mature symbolic-execution tools still rely on plain-text log files for debugging, which provide no good overview of the execution process and can quickly become overwhelming. We introduce a debugger for Gillian's verification mode—complete with a custom interface—and ponder the potential for this interface and the protocol behind it to be used outside of Gillian.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; **Formal software verification**.

KEYWORDS

debugging, symbolic execution, verification

ACM Reference Format:
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 '23), July 17, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3605155.3605861

PROPOSAL

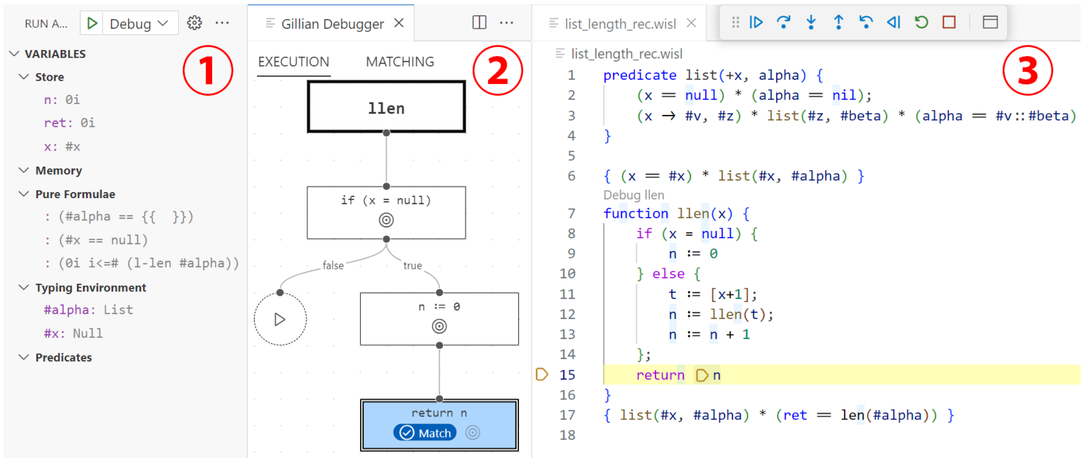
Gillian [2, 6] is a compositional symbolic execution platform, parametric on the memory model of the analysed language. It supports whole-program symbolic testing, compositional verification, and automatic compositional testing (ACT) powered by bi-abduction. Like some other symbolic-execution tools (e.g. Infer Pulse [1, 8], CBMC [5]), Gillian relies on verbose plain-text logging. Our work introduces a debugger for verification in Gillian, complete with a custom interface to ease the navigation of branching execution paths and state matching (cf. Figure 1); this interface presents the execution trace as a tree, and allows parts of the trace to be nested inside other nodes - for example, the body of a while-loop can be nested inside the loop definition's command node (cf. Figure 2). While this implementation is Gillian-specific, we intend to provide an intuitive UI for these purposes that could potentially be used with other tools and implement debugging for Gillian's ACT.

DEBT '23, July 17, 2023, Seattle, WA, USA
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0245-7/23/07.
https://doi.org/10.1145/3605155.3605861

Related work on symbolic debugger interfaces includes the VeriFast [4] tool, which also offers a visual interface. VeriFast performs verification to completion and presents the trace afterwards, whereas Gillian's debugger steps inside the verification process, allowing for potential performance gains in larger functions by performing each step on request and only exploring the desired path(s). VeriFast presents the trace as an unlabelled binary tree, whilst the Gillian debugger shows information about the command, whether it required state matching and whether the matching succeeded, and allows parts of a trace—or even the whole trace of another process—to be nested under any command. This provides a level of flexibility that other tools could make use of beyond Gillian. The ability to fold and unfold nests means that arbitrary levels of detail needn't unnecessarily pollute the interface or overwhelm a user. A symbolic execution debugger based on the KeY platform [3] has also been introduced, featuring an interface with some parallels to ours. Our work differs through its generality (KeY specifically verifies Java, and in the Eclipse IDE or their standalone UI), and the flexibility of the interface, with its aforementioned nesting capabilities.

A key part of our approach is the protocol with which the debugger process communicates with the user interface. We extend the Debug Adapter Protocol (DAP) [7], which is designed by Microsoft and provides a standard protocol for integrating debuggers and development environments, with the ability to manage the unique challenges of symbolic execution, such as tracking multiple execution branches; these extensions could form a "symbolic DAP" for use with multiple symbolic tools and IDEs. The visual map of execution is then presented in a web-view using a custom VSCode extension, though this could be decoupled from VSCode via a browser client.

Given the novelty of our debugger, we believe that comprehensive user feedback is pivotal for guiding future development. To this end, we conducted a two-hour lab session in Gardner's 4th-year and M.Sc course on Separation Logic at Imperial College London (cf. Figure 3), where students used the debugger to diagnose and fix list algorithms in the Gillian tool instantiated with a small while language. This debugger is the result of a long journey associated with this course, which started with Ayoun's M.Sc project in 2018, and continued with three excellent M.Sc projects by Radu Lacraru (2020), Matthew Ho (2021) and Nat Karmios (2022), primarily under the supervision of Ayoun. In November 2022, the debugger reached a standard where we could present it to the students in a lab session. The positive feedback and useful suggestions (received via feedback forms) and overall excitement in the laboratory was inspirational, with many of the students and demonstrators working far past the end of the lab. Following this success, we have continued to refine the interface, with the aim to present a formal coursework task during the course in 2023. We have also presented the debugger in an informal workshop at Meta, and are now in discussion about the possibility of adapting the debugger to Infer Pulse.



```
VARIABLES
  Store
    n: 0i
    ret: 0i
    x: #x
  Memory
  Pure Formulae
    : (#alpha == {{  }})
    : (#x == null)
    : (0i i<=# (l-len #alpha))
  Typing Environment
    #alpha: List
    #x: Null
  Predicates
```

```
predicate list(+x, alpha) {
    (x == null) * (alpha == nil);
    (x → #v, #z) * list(#z, #beta) * (alpha == #v::#beta)
}

{ (x == #x) * list(#x, #alpha) }
function llen(x) {
    if (x = null) {
        n := 0
    } else {
        t := [x+1];
        n := llen(t);
        n := n + 1
    };
    return n
}
{ list(#x, #alpha) * (ret == len(#alpha)) }
```

Figure 1: The debugger interface, including: ① the symbolic state at the current step; ② the 'map' of symbolic execution (custom extension); and ③ the source code.

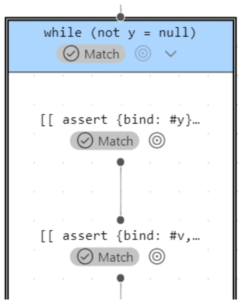


Figure 2: An example of nesting; the body of the while-loop is nested inside the node for the loops definition.

Figure 3: The student lab held in November 2022

This project is unusual for our team, whose primary focus is mathematical theory and well-engineered research tools for compositional symbolic analysis. The debugger has changed our ambition: the Gillian tool is now usable by the 4th-year and M.Sc students, and we hope, in future, code developers with an interest in verification. We look forward to attending the DEBT workshop and gaining valuable insight from the debugger community in order to further refine our priorities in future development.

ACKNOWLEDGMENTS

Thanks go to Radu Lacraru and Matthew Ho for previous work, the 2022 students of the course on Scalable Software Verification at Imperial College London for lab participation and feedback, and our reviewers for their comments. We also thank Andreas Lööw and Petar Maksimović for their invaluable input. This work was supported by funding from Gardner's UKRI fellowship 'Verified Trustworthy Software Specification', Ayoun's Department of Computing PhD Scholarship from Imperial College London, a faculty gift from Meta and a grant from the UK National Cyber Security Centre.

REFERENCES

[1] Cristiano Calcagno and Dino Distefano. 2011. Infer: An Automatic Program Verifier for Memory Safety of C Programs. In *NASA Formal Methods: Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18–20, 2011, Proceedings (Lecture Notes in Computer Science, Vol. 6617)*, Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi (Eds.). Springer, 459–465. https://doi.org/10.1007/978-3-642-20398-5_33
[2] José Fragoso Santos, Petar Maksimović, Sacha-Élie Ayoun, and Philippa Gardner. 2020. Gillian, Part I: A Multi-Language Platform for Symbolic Execution. In *Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/3385412.3386014
[3] Martin Hentschel, Richard Bubel, and Reiner Hähnle. 2019. The Symbolic Execution Debugger (SED): a platform for interactive symbolic execution, debugging, verification and more. *International Journal on Software Tools for Technology Transfer* (2019). https://doi.org/10.1007/s10009-018-0490-9
[4] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. 2011. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In *NASA Formal Methods Symposium*. https://doi.org/10.1007/978-3-642-20398-5_4
[5] Daniel Kroening, Peter Schrammel, and Michael Tautschnig. 2023. CBMC: The C Bounded Model Checker. arXiv:2302.02384 [cs.SE]
[6] 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 (CAV)*. https://doi.org/10.1007/978-3-030-81688-9_38
[7] Microsoft. [n. d.]. Debug Adapter Protocol - Overview. https://microsoft.github.io/debug-adapter-protocol/overview [Accessed 2023/06/27].
[8] The Infer Team @ Meta. [n. d.]. Infer Pulse. https://fbinfer.com/docs/checker-pulse [Accessed 2023/06/27].

Received 2023-05-22; accepted 2023-06-20