Gillian Debugging: Swinging Through the (Compositional Symbolic Execution) Trees

Nat Karmios, Sacha-Élie Ayoun, and Philippa Gardner

Imperial College London, UK
n.karmios@ic.ac.uk

Abstract. In recent years, compositional symbolic execution (CSE) tools have been growing in prominence and are becoming more and more applicable to real-world codebases. Still to this day, however, debugging the output of these tools remains difficult, even for specialist users. To address this, we introduce a debugging interface for symbolic execution tools, integrated with Visual Studio Code and the Gillian multi-language CSE platform, with strong focus on visualisation, interactivity, and intuitive representation of symbolic execution trees. We take care in making this interface tool-agnostic, easing its transfer to other symbolic analysis tools in future. We empirically evaluate our work with a user study, the results of which show the debugger's usefulness in helping early researchers understand the principles of CSE and verify fundamental data structure algorithms in Gillian.

Keywords: Debugging, symbolic execution, verification

1 Introduction

Symbolic execution (SE) is a static program analysis technique that explores program behaviour by executing code over symbolic (rather than concrete) inputs. Although symbolic execution tools and frameworks based on first-order logic, such as CBMC [5] and KLEE [4], are widely used in academia and industry, their reasoning does not scale when faced with heap-manipulating programs.

Compositional symbolic execution [14, 15] (CSE) addresses this limitation by extending SE with function specifications expressed in a separation logic, such as the original separation logic (SL) [22] and the recent incorrectness separation logic (ISL)[24]. This enables functions to be analysed in isolation, independent of their calling contexts, and records the results as concise specifications for reuse during subsequent analyses. This compositionality significantly improves scalability while retaining precise reasoning about heap-manipulating programs.

The development and adoption of CSE-based tools has steadily increased both in academia and industry. For example, VeriFast provides semi-automatic compositional verification for C, Java and Rust using SL; CN supports verification of C programs using SL; the Viper [21] framework, based on implicit dynamic frames (akin to SL), underpins verification tools for Go, Java, Python

S. Junges and G. Katz (Eds.): TACAS 2026, LNCS 16506, pp. 195–214, 2026.
https://doi.org/10.1007/978-3-032-22749-2_10

and Rust [2, 6, 26]; Infer-Pulse [25], built on ISL, is deployed at Meta for automated multi-language bug finding at scale [17]; and Gillian [3, 8, 16] offers a unified framework for correctness and incorrectness reasoning based on SL and ISL respectively, applied to C, JavaScript and Rust. These tools have reached a level of maturity that enables their application to real-world software, including the AWS SDK (Gillian), Google's pKVM hypervisor (CN), internet routers (Viper), Meta's production codebase (Infer-Pulse), and the Rust standard library (Gillian, VeriFast). Several are also used in teaching program analysis principles to undergraduate and graduate students (Gillian, VeriFast, Viper).

With this growing and multi-faceted applicability comes a need for clearly presenting and efficiently debugging the output of a given CSE tool. To us, this need is threefold: (1) to allow tool developers to examine the inner workings of the tool in full detail so that they can improve its implementation; (2) to enable tool users to debug their programs by relating the output of the tool to the analysed code; and (3) to guide students, who are the next generation of tool users and developers, in understanding the fundamentals of CSE reasoning. This, however, still remains a relatively uninvestigated topic, typically addressed in an ad-hoc fashion, with user experience varying greatly from tool to tool: some only provide (overwhelming) textual feedback; some offer minimal language-server-based integration; others offer interactive exploration but limit it to generated counter-examples or are post-hoc; and the few that do have truly interactive debuggers either adopt broad editor integration at the cost of visual expression or tightly focus on integrating with a single editor, making the transfer to other tools difficult. A more detailed overview of the state-of-the-art is given in §2.

To address this gap, we introduce an interactive debugger for CSE-based verification, using Gillian as a case study. Our goal is to design a tool-agnostic debugger, identifying which of its components are general enough to support other CSE tools, and which are specific to Gillian. Gillian serves as an ideal test bed for this work due to its parametric treatment of memory, support for multiple source languages through compilation to a common intermediate representation (IR), and implementation of multiple analyses within a unified framework (§3). Our debugger is integrated with Visual Studio Code [18] to leverage a widely used and familiar development environment, again taking a generalised approach intended to facilitate future integration with additional editors beyond VSCode.

We give a tour of the debugging experience in §4 and present the principles underlying our debugger in §5, focusing on:

- *execution tree representation*, where we aim to create a useful, intuitive, and flexible representation of analysis with a tree-of-trees structure;
- *execution tree capture*, using this tree-of-trees structure to record information about the analysis in an unobtrusive way at the engine level;
- *execution tree lifting*, which lifts a low-level tree-of-trees structure to a source-level one, which can be read by the user; and
- *debugging interactivity*, giving the user full control of examining and directing the symbolic analysis on-the-fly rather than summarising it after completion.

Importantly, we take care to decouple our design choices from the Gillian implementation whenever possible, making them transferable to other CSE tools, and even potentially more traditional symbolic execution tools. Further, in §6, we highlight interesting lessons and observations that have come up during the substantial engineering work involved in this project.

We performed a basic user study at a PL summer school (§7), evaluating the usability of the Gillian debugger on CSE-base verification. Early researchers were introduced to SL, CSE and Gillian, before using Gillian to try to verify a number of data-structure algorithms. The examples were designed to require use of the main features of semi-automatic compositional verification: specification of pre- and post-conditions, loop invariants, and proof tactics. Afterwards, we performed a quantitative analysis that revealed that the users spent a substantial amount of time working with the debugger, indicating its usefulness. We also collected qualitative feedback using a Lickert-style questionnaire, with the aggregated results indicating that working with the debugger was helpful and intuitive, and its feedback informative and educational.

2 Related Work

We briefly review the debugging capabilities of state-of-the-art CSE tools. Gillian (prior to our work) and CN provide only text-based logs, which in real-world verification can reach tens of millions of lines. These logs linearise the inherently non-linear analysis into flat textual streams that, while exhaustive, are overly detailed and interpretable only by experts familiar with the tools.

Infer-Pulse generates HTML files that present similar information to text logs, organised into analysis "nodes" associated with source code locations. Although this offers greater clarity than plain logs, it remains difficult to follow the flow of the analysis through the program, and the need to leave the development environment disrupts the user's workflow.

Viper integrates with editors via the Language Server Protocol (LSP) [20], re-running analyses as the user types and reporting failures inline. This proves useful for receiving rapid feedback, though such feedback is often limited to brief error messages. We argue that this approach is complemented by the availability of a debugger when deeper insight into the analysis is needed.

Finally, VeriFast, through its VSCode extension [11], enables interactive inspection of the analysis' execution tree, akin to a debugging session. However, the tree nodes are unlabelled and difficult to relate back to the source code. Moreover, as analysis terminates at the first breakpoint or failure, the post-hoc tree is typically incomplete, leaving some execution branches unexplored.

Our Gillian debugger extends the Debug Adapter Protocol (DAP) [19] to provide an interactive symbolic debugging environment for its CSE analyses. Several other symbolic analysis tools, such as GobPie [10] and SecC [7], also use DAP to deliver an integrated debugging experience with many editors "for free". However, the DAP offers no intuitive mechanism for representing branching execution, leading ad-hoc solutions such as repurposing the DAP's 'threads' view, originally designed for visualising concurrent threads in concrete execution.

The debugger perhaps most similar to ours is the feature-rich Symbolic Execution Debugger [9] of the KeY project [1]. This debugger is capable of displaying detailed, annotated symbolic execution trees, but relies on a non-standard interface and is tightly integrated with both the Eclipse IDE and KeY's first-order Java analyses, limiting its portability to other IDEs and analysis frameworks.

3 Background

Symbolic Execution is a program analysis technique that executes a given program with symbolic instead of concrete inputs, where one symbolic state describes a set of concrete states, enabling general reasoning about program behaviour. Symbolic execution can branch (e.g. via an if-else statement), with each branch carrying a logical assertion called a *path condition* which describes the constraints on symbolic variables that have led the execution to that branch.

***Compositional* Symbolic Execution** (CSE) extends symbolic execution to use and create function specifications written in a separation logic. In particular, the reasoning allows function behaviour to be summarised on *partial* symbolic states using function specifications, which are Hoare triples that have meaning given by an underlying separation logic. With verification, the pre-condition describes a part of the symbolic state that is sufficient for the function to be evaluated; the post-condition describes all the possible results.

When calling a function, a specification of the callee can be used instead of inlining its body, providing compositionality and improving scalability. Executing a specification means *matching* the current symbolic state against its pre-condition, *consuming* (or 'exhaling') the matched part from the state, and *producing* (or 'inhaling') the post-condition. Verifying a function simply requires symbolically executing the function from its pre-condition until termination, then matching each final state against the post-condition. Since function calls are replaced by the execution of their specifications, a symbolic execution tree is as long as the number of statements in a single function, drastically reducing reasoning complexity compared to non-compositional SE.

Gillian is a multi-language platform that supports three types of analysis: whole-program symbolic testing, providing bounded verification guarantees similarly to the first-order CBMC tool [5]; semi-automatic compositional verification à la CN, VeriFast using SL, and Viper; and automatic bug-finding similar to Infer-Pulse using ISL. Gillian is parametric on the memory model of the source language analysed, which needs to be provided per instantiation together with a compiler from the source language to Gillian's intermediate representation, GIL. When it comes to real-world languages, Gillian has been instantiated to C, JavaScript, and Rust. For ease-of-presentation, this paper focuses on **WISL**, a demonstrator Gillian instantiation for a simple while language with a C-like block-offset memory model routinely taught to fourth-year undergraduates and MSc students. More detail on this language can be found in the extended version of this paper [12] .

4 The Gillian Debugging Experience

We illustrate the features of the Gillian debugger by using it to debug the verification of the WISL `llen()` function (Fig. 1, left), which computes the length of a null-terminated singly-linked list (SLL), each node of which consisting of a two-cell block containing the value carried by the node and the pointer to the next node. The function is specified using the standard separation-logic SLL predicate, `list(x, alpha)`, which states that the SLL starting from block with identifier `x` contains the values in the mathematical list `alpha`. Then, in the pre-condition we have just the list, and in the post-condition we additionally state that the return value equals its length. To illustrate the difference between the source- (WISL) and the IR-level (GIL), we also give a stylised compilation of `llen()` to GIL in Fig. 1 (right), where we can see how WISL's structured control flow becomes unstructured via GIL gotos, and also how WISL commands (such as pointer dereferencing, $\mathtt{t} := [\mathtt{x} + 1]$) get broken down into several GIL steps.

```
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)
  };
  return n
}
{ list(#x, #alpha) * (ret == len(#alpha)) }
```

```
proc llen(x) {
        goto? (x = null) then else;
  then: n := 0;
        goto end;
  else: _var0 := i_add(x, 1);
        goto? (_var0 is Ptr) cont fail;
  fail: fail "Invalid pointer";
  cont: t := load<_var0>;
        n := llen(t);
  end:  skip;
        ret := n;
        return
};
```

Fig. 1. Slightly simplified SLL predicate and a specified WISL recursive list-length function (left), and the corresponding compiled GIL code, simplified (right). The prefix `#` denotes a logical variable.

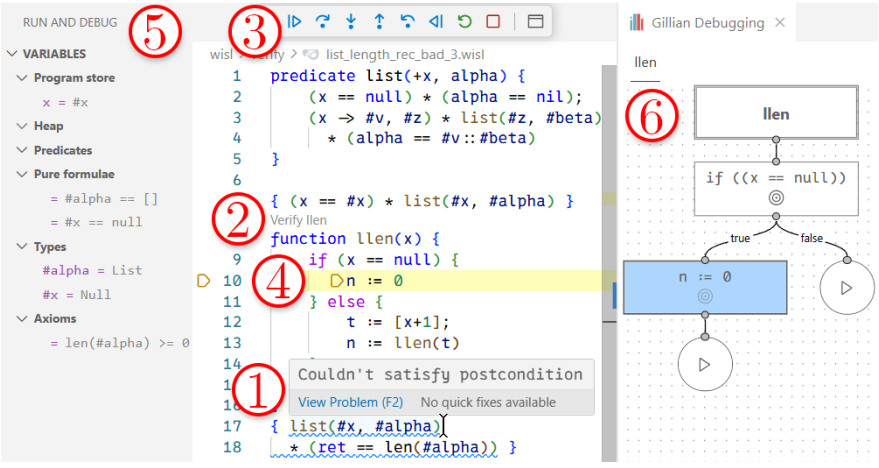

Fig. 2. Gillian's debugging and language server interface when verifying `llen()`.

When trying to verify `llen()` using Gillian and VSCode (cf. Fig. 2), the language server informs us that there is an error (①), stating that the function post-condition could not be satisfied. While other existing LSPs, such as Viper's, may provide slightly more precise information, such as which subset of the post-condition failed, the LSP interface can only point to the failure, not explain it. At this point, we turn to the Gillian debugger to trace the root cause of the error, as the verification could have failed for a variety of reasons, such as an incorrect specification, a missing proof annotation, or a bug in the program itself.

Above each function, the Gillian debugger displays a 'Verify' button (②) that initiates an interactive debugging session for that function. While in a session, several pieces of interface are presented to the user: controls for stepping through the executions (③); a highlight of the line of code being currently executed (④); a breakdown of the current symbolic state (⑤); and a tree view that visualises the symbolic execution presented in a separate panel (⑥).

The controls at ③ are inherited from the VSCode DAP interface. They contain: contain: a 'continue' button, which continues execution along the current branch until the next breakpoint or end of execution; a 'step over' button which executes the next WISL statement; a 'step in' button which steps inside a function call if present; a 'step out' button which continues execution until the current function returns; a 'step back' button which undoes the last statement; a 'continue backwards' button which returns to the previous breakpoint or start of execution; a 'restart' button which restarts the entire symbolic execution from the beginning; and a 'stop' button, which ends the debugging session.

When navigating through the symbolic execution using these controls, the line of code currently being executed is highlighted with a small arrow pointing to the sub-expression being evaluated (④). In addition, the current symbolic state is displayed on the left-hand side of the screen, using VSCode's built-in variable explorer to display the current program variable store, heap, predicates, path conditions, and other relevant information (⑤).

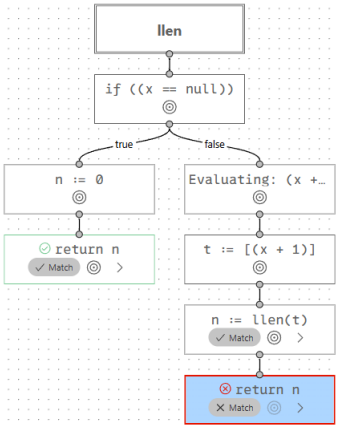

Fig. 3. Full symbolic execution tree of `llen()`

Finally, and most importantly, the tree view (⑥) enables interactive visualisation of the symbolic execution. Statements are represented as nodes in a tree structure, with edges representing the flow of execution. Clicking on a node jumps to that point in the execution, updating the highlighted code and symbolic state, providing a way to navigate *between* branches as well as forward and backward within the same branch. If a branch has not yet terminated, a 'play' button is displayed as a leaf of that branch and can be clicked on to execute the next statement on that branch. When examples increase in complexity and verification times grow longer, this interface allows users to contain branch exploration, avoiding long waiting times.

Fig. 3 shows the full symbolic execution tree of `llen()`, from which we identify that the path leading to a verification failure is the one taking the 'false' branch of the `if` statement, as identified by the red cross next to the return statement. This node also displays a small arrow pointing to the right, indicating that there is *nested* information within that node.

Clicking that arrow expands the node to reveal the steps of the post-condition matching that failed (Fig. 4), and the magnifying glass icon in the nested tree 'pops it out' to a new tab to view in isolation. Notice how the visual branching remains relevant in matches, where Gillian has backtracked and attempted to recover by unfolding the list predicate. Within this tree, we see that Gillian failed to match `ret == len(#alpha)` (leftmost branch) before backtracking and applying a *recovery tactic* (here, the 'unfold' tactic), where the match still fails for the same reason (two rightmost branches). Recovery tactics are triggered by heuristics and sometimes lead to confusion; this example illustrates how our debugger clearly displays this process to the user.

By clicking on the leftmost failing node and then inspecting the symbolic state in detail (Fig. 4, right), we see that `ret` equals `len(#lvar_4)`, but `#alpha` is the list (`[#lvar_7]@#lvar_4`), and that therefore `ret` is off by one. This is the final step that leads to the understanding of the root cause of the error: we forgot to increment `n` in the program itself. In particular, after line 12 of Fig. 1 (left), `n := n + 1` is required. While this last step must still be taken by the user, the information required to get them there was accessible within a few clicks.

We note that, at the top level, the maximum depth of the execution tree is the number of statements in the function being verified (modulo expression evaluation, which is an implementation choice); as CSE permits verifying functions in isolation, execution tree sizes are constrained. While real-world conditions—e.g. larger functions, more complex source languages—will naturally produce trees larger than we encounter in this work, we expect them to remain orders of magnitude smaller than those seen in non-compositional symbolic analyses á la KLEE.

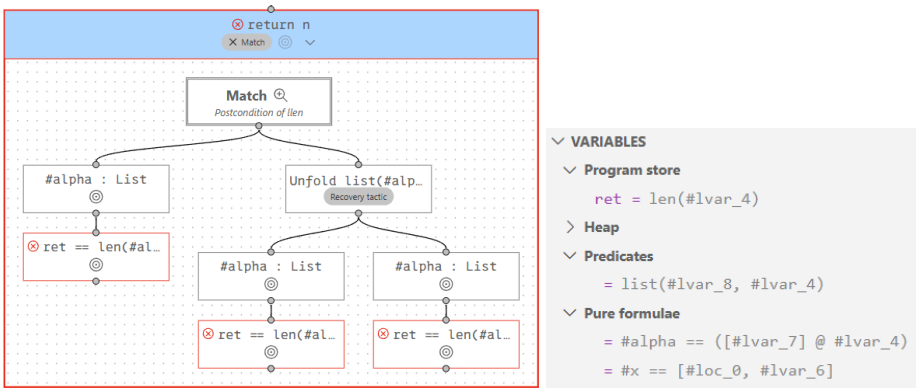

Fig. 4. Expanded failing node (left) and symbolic state at the failing match (right).

5 Principles of CSE Debugging

We give an overview of our three-layered architecture in Fig. 5 and discuss the key processes we compose to create a usable CSE debugger. We will refer to the execution of the WISL statements `t := [x+1]; n := llen(t)` on line 12 of `llen()`, which correspond to lines 5–8 and 9 of the GIL code respectively (see Fig. 1). We abstract these concepts from Gillian-specific implementation details of Gillian to encourage wider applicability; some implementation details specific to Gillian are discussed in §6.

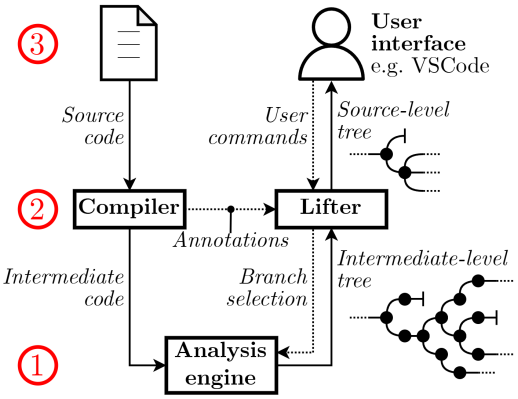

Fig. 5. Overview: architecture of the Gillian debugger.

At Layer 1, we create a tree structure that describes the analysis process at the level of the intermediate representation (IR). This is a low-level tree that includes all the execution steps of the analysis, and, ideally, any associated fundamental processes (for Gillian, for example, these would be matching (supported) and expression and state simplifications (currently unsupported)). This tree must accurately represent one's intuition of the 'shape' of the analysis, and be extensible over time in the context of an interactive analysis. Working with this layer alone is only for the expert tool developer, giving them the ability to fine-tune tool performance and understand unexpected behaviours or bugs in the tool itself.

To present an understandable and actionable execution tree to a tool user working with the analysed source language, one must also bridge the disconnect between the internal IR of the tool and said source language. This is the purpose of Layer 2, at which the debugger employs various lifting mechanisms to abstract the low-level tree from Layer 1 into a source-level execution tree.

Finally, Layer 3 concerns an important but rarely discussed aspect of tool development: the infrastructure through which a user interacts with the tool. UI development, especially when aiming for reuse, maintainability, and longevity, can be an arduous engineering task, as it requires a deep understanding of existing developer tooling.

5.1 Low-level Tree-of-trees Structure

An intuitive, interactive debugging experience requires a structure that accurately represents the 'shape' of the analysis process, without knowing the execution order ahead of time. One could argue that symbolic execution can be represented simply as a tree, where a node with multiple children denotes symbolic branching. We represent analyses at both the intermediate and source level with a *tree-of-trees* structure, where each node can contain nested trees providing further detail. This has the benefit of capturing the intuition of breaking execution and analysis down into smaller steps, while hiding excessive details until desired by the user. Nesting is used, for example, to represent Gillian's matching process or the body of an inlined function call; recall `llen()`'s execution tree given in Fig. 3, where the failing `return` node expanded to reveal the nested matching tree shown in Fig. 4.

A CSE tool will typically implement a particular execution strategy (e.g. depth-first) for traversing the execution tree. Interactive debugging requires recording an execution tree that is agnostic to any such strategy, in such a way that it can handle interrupting and later returning to an execution path. The tools' analysis engine will therefore need some modification to provide this information necessary to build the tree-of-trees. However, tracking non-trivial control flow in symbolic execution, such as entering and leaving function and loop bodies, can dramatically increase engine complexity, making a critical part of the tool more difficult to maintain. We found that a minimal approach is sufficient, and can straightforwardly apply to many CSE tools: instead of just writing text logs describing the state of analysis at each step, like most tools do, the engine can produce a machine-readable report with similar information and a unique identifier. A report also stores the identifier of the previous execution step, and the enclosing step that nests it (if relevant); this requires only light bookkeeping from the engine and can easily be stored alongside the execution context. These reports can be freely accessed later in the debugging pipeline, where the heavy lifting of constructing a more intricate tree can be performed with no further input from the engine. Fig. 6 shows a GIL trace segment for the WISL statements on line 12; to keep engine logic simple, function bodies such as that of `i_add()` are not nested under the calling command. Nesting is still utilised at the intermediate level where convenient, such as matching for the recursive call to `llen()`.

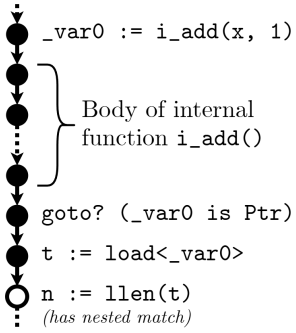

Fig. 6. A snippet of `llen()`'s GIL trace.

5.2 Lifting to the Source Level

After applying the prior concepts, one is rewarded with a perfectly adequate interactive execution tree, but only for the relevant tool's IR. To facilitate debugging at the level of the source language, a CSE debugger needs to include a

lifter, whose responsibility is to tell the analysis engine how to transform the tree-of-trees for a specific IR into a tree-of-trees more reminiscent of a particular source language—in our debugger, from GIL to WISL. A lifter can, for example, transform the shape of a tree by combining the nodes of multiple IR-level commands into a single node for the originating source-level statement, it could alter and combine branch cases for greater clarity, and it could introduce nesting where appropriate. A lifter can also improve the information associated with individual nodes by, for example, producing node display text and transforming the displayed state to be more recognisable at the source level.

The source-aware information needed to construct a source-level tree-of-trees is most easily gathered during compilation and can be passed to the lifter via **annotations** on IR code. These annotations may include the display text for the source statement, flags denoting which groups of IR statements comprise a source statement, or hints at the reason a branch may occur. As an example, the WISL statement `t := [x+1]` compiles to lines 5–8 of the GIL code: we choose to count expression evaluation as a distinct step, so line 5 is flagged as terminating a node; line 6 is flagged as *not* terminating a node; and lines 7 and 8 are flagged as terminating. In the source-level tree, this results in one step for evaluating `x+1`, and one for performing the lookup. See the extended paper for some further details on WISL's annotations and their assignments to `llen()`'s GIL code.

5.3 User Interface

When it comes to UI development, given the amount of engineering work it requires, one should try to avoid 'reinventing the wheel' to create an interface with a slightly different functionality or in a slightly different environment.

As discussed in §2, some tools leverage the DAP to take advantage of debugging interfaces provided by IDEs such as VSCode. This includes, for example, highlighting the current code step, controls for stepping through execution, and a view of the current state. The DAP, however, is not expressive enough to represent branching execution, much less navigate a complete execution tree, leading to ad-hoc solutions such as using the 'threads' view to list execution branches, or abandoning the DAP altogether in favour of a bespoke post-analysis tree viewer.

Our approach achieves the best of both worlds by *extending* the DAP to support interaction with a custom tree view in the editor. Referring to Fig. 2, the standard debugging UI elements are provided as standard for a DAP-implementing debugger; users can see the line of code (④) and the state (⑤) at the current step, and use the (linear) stepping controls (③). The bespoke tree view (⑥) is supported by a custom event `mapUpdate` to relay changes to the tree, and the custom commands `jump` to travel to a different point in the tree and `stepSpecific` to step forward via a specific branch when multiple are available. This does detract from the DAP's editor agnosticism—due to requiring a VSCode extension—but we believe this to be a worthwhile compromise for the moment, and are working on re-establishing DAP editor independence.

6 Implementation Insights

While the specifics Gillian Debugging's implementation are not a primary contribution of this work, we would like to highlight a number of observations that have emerged and that could be of interest to the wider community.

Structured logs. The structured logs required by the debugger are necessarily more complex to keep track of than regular logs. After experimentation, we decided that the burden of maintaining the structure should be minimised for the symbolic execution engine, an already complex piece of technology with strong correctness requirements. Specifically, we separate concerns: the engine performs minimal bookkeeping, storing all log reports in an SQLite database with only two additional fields, `previous` and `parent`; the lifter later uses these reports to construct a tree with more elaborate control flow. The former operation is language-independent and is factored out from Gillian instantiations.

Rebuilding source-level commands. The WISL lifter aggregates executed GIL steps into WISL commands. Each executed GIL step is assigned to a *partial command* representing a source-level WISL command that has not yet been fully explored. Annotations are used to mark GIL commands that terminate a WISL command, allowing the lifter to finalise a partial command into a source-level tree node when all paths have been explored. As we had full control over the WISL compiler, we could straightforwardly extend it to add such annotations to the GIL code. This approach is applicable to other compliers: for example, we successfully applied the same pattern to a new Gillian-C instantiation that uses CBMC's parser and IR. We could not adapt the original Gillian-C instantiation, as the CompCert compiler it employs is written in Rocq and is therefore difficult to modify. This demonstrates that our approach can be portable across different language implementations if the frontend preserves sufficient source-level information during compilation.

Advanced nesting. The tree-of-trees structure and its visual nesting (cf. Fig. 4) were particularly useful in reifying source-level nesting where the IR-level tree had none. For instance, WISL loops annotated with invariants compile to GIL functions with pre- and post-conditions, but the WISL lifter is able to re-integrate the loop body by initiating its separate verification and nesting the resulting tree in the tree of the outer function. This demonstrates both the flexibility of the tree-of-trees structure and the ability of the lifter to reconstruct source-level control flow that fundamentally changes during compilation.

Everything is interactive. A key difference between our approach and post-analysis tree viewers is the ability for users to interactively guide the analysis as it progresses. To implement this interactivity, we modified Gillian's interpreter to return a continuation function after each GIL execution step. In normal execution mode, these continuations are simply called repeatedly until completion, preserving the original behaviour. In debugging mode, however, the continuations are called with a branch identifier, selecting which execution path to

continue. This approach shows promise for integration with other symbolic execution tools that already operate in continuation-passing style, such as VeriFast and Viper, or with continuation monads, such as CN.

The lifter cooperates with this engine-level interactivity by translating between user-level commands and engine operations. For each high-level instruction ('step in', 'step over', etc.), the lifter determines the appropriate GIL steps required to extend the tree.

A generalised UI toolkit. Gillian's debugging UI is implemented in VSCode by combining extensions to the DAP with a dedicated tree-viewing interface presented via a web-view panel. Early iterations of the debugger revealed that both the extensions to the DAP and the custom tree viewing interface had few conceptual dependencies on Gillian or VSCode, and could be separated entirely; the interface deals in abstract steps, requiring no knowledge of Gillian, its IR, or the language being analysed, and said interface is contained in a single web-based UI component. This realisation has been brought to fruition with the working title of SEDAP (Symbolic Execution DAP), a prototype for a standard extension to the DAP with libraries providing the interactive tree view as an extensible web component, and helpers for integrating this interface with VSCode debug sessions. Just as the DAP bridges editors with traditional debuggers, further work on SEDAP should bridge editors with analysis tools like those discussed in §2; Gillian's debugger in VSCode now serves as the first example of this in practice.

Benchmark	**Time to verify (ms)**		
	No logs	**File**	**Database**
`SLL_iterative.wisl`	242	680 (2.8x)	487 (2.0x)
`DLL_recursive.wisl`	91	259 (2.8x)	174 (1.9x)
`sll.c`	303	390 (12.8x)	668 (2.2x)
`dll.c`	1875	81501 (43.5x)	6335 (3.4x)
`priQ.c`	292	5730 (19.6x)	7815 (2.7x)
`sort.c`	236	2369 (10.0x)	475 (2.0x)
`SLL.js`	149	1192 (8.0x)	1188 (8.0x)
`DLL.js`	202	1624 (8.1x)	1672 (8.3x)
`ExprEval.js`	246	6415 (26.1x)	4513 (18.3x)

Table 1. Benchmark results on a selection of Gillian's verification examples across WISL, Gillian-C and Gillian-JS. This compares no logging, verbose file logging, and structured database logging, showing slowdown factor for the latter two. Benchmarks were performed on a laptop with an Intel i7-1065G7 processor and 32 GiB of memory.

Performance impact. As shown in Tbl. 1, structured logging to the on-disk database incurs a performance cost over no logging, but is no worse than that of Gillian's default text log. Interestingly, database logs fare much better against file logs in C verification, owing to the complicated logic used to print the heap; this demonstrates a benefit of separating logging concerns from the engine. We note that no effort has yet been made to optimise structured logs, and there is much

low-hanging fruit for performance improvements: database queries could be performed asynchronously or in batches; a different database model could be more appropriate (say, a graph database rather than relational); and a more efficient serialisation method such as protocol buffers could be used instead of JSON.

7 Empirical Evaluation

We made a formative attempt at evaluating Gillian debugging as part of a summer school attended by late-stage undergraduate and early-stage PhD students interested in formal methods and PL research. Following 3 hours of lectures on SL, CSE and Gillian, participants were invited to spend two 90-minute lab sessions during which they would attempt to verify a number of WISL programs using Gillian's language server and debugger. For this lab, we created a suite of 13 introductory and 6 advanced exercises: the introductory ones use simple programs on linked lists to introduce various aspects of Gillian verification, whereas the advanced ones consist of larger programs using more complex data structures (e.g., sets, doubly-linked-lists, and binary search trees) as well as a WISL translation of a part of the Collections-C library [23]; see the extended paper for the full list of exercises.

The lab session was initially guided in that we solved the first few exercises together with the participants, introducing them to the debugger interface and the Gillian proof tactics that they had at their disposal: folding and unfolding user-defined predicates, the `assert` and `apply` proof tactics, and loop invariants.

At the end of proceedings, participants were invited to complete an anonymised survey about their experience and to optionally submit usage logs. These logs were generated and stored locally on the participants' devices and tracked a small amount of timestamped information per participant interaction with the debugger and language server. Of the ≈30 lab participants, 19 completed the survey, of which 18 submitted usage logs. We use this data to evaluate the usefulness of Gillian debugging in understanding CSE and completing the provided exercises. We first evaluate qualitatively by analysing responses to a series of Lickert-scale statements about the lab to gauge subjective participant impressions on the usefulness of the debugger. We then contrast these responses against a quantitative metric by analysing the provided usage logs to understand the actual participant use of the debugger and language server.

The provided Lickert-style statements and their combined responses are shown in Tbl. 2. The overall sentiment towards the usefulness of the debugger was overwhelmingly positive (statements 1/5/6), with 63% of responses strongly in favour and 26% somewhat so. Further, the participants felt that the debugger was well-integrated with the verification process (statement 10) and that other tools could benefit from similar debuggers (statements 11/12). The participants also recognised the usefulness of more specific properties arising from the interactive, step-by-step nature of the debugger, including the exploration of individual execution paths (statement 7) and viewing the state at each execution step (statement 9). They were less positive, however, about the viability of the language server alone (statements 2/8), with neutral-to-slightly-negative responses.

		Strongly Disagree	Somewhat Disagree	Neutral	Somewhat Agree	Strongly Agree
1	I didn't need to use the debugger to complete the exercises	11	6	1	1	0
2	The language server was sufficient for me to complete the exercises	1	6	5	6	1
3	The debugger helped me to understand CSE	0	2	5	5	7
4	The debugger helped me understand what Gillian is doing	0	0	2	8	9
5	I found the debugger helpful in completing the exercises	0	0	3	4	12
6	The debugger helped me to understand why verification failed	0	2	3	6	8
7	I found it useful to explore each path of execution individually using the debugger	0	0	1	5	13
8	I understood the errors that the language server presented me with	1	7	3	4	4
9	I found it useful to see the state at each point in execution using the debugger	0	0	3	5	11
10	It felt natural to use the debugger as part of the verification workflow	0	1	2	4	12
11	This debugging interface would be applicable to similar tools I have used	0	1	4	3	6
12	I would benefit from a debugger like this in similar tools I have used	0	0	1	4	9

Table 2. Summed Lickert scale responses of the user survey.

On the quantitative side, the usage logs consist of timestamped entries for each participant interaction with the debugger and the language server; language server entries show the changes to the exercise file and the analysis result (success, or the reason for a failure), and debugger entries denote whether the user started or stopped a session or made a step through the tree. From this, we calculate the amount of time each participant spent on each question (accounting for switching between exercises and long periods of inactivity), and how much of that time was spent actively using the debugger (interpreted as having interacted with the debugger within the last 30 seconds), to build a rough estimate of what proportion of time participants spent using the debugger. The distribution of these proportions, shown in Fig. 7, suggests that the majority of participants spent between 40% and 60% of their time having recently interacted with the debugger, giving a strong indication that the participants indeed made frequent and active use of the debugger.

While this user study provided solid evaluation for Gillian debugging, a number of limitations prevented deeper insight. Ultimately, the lab primarily aimed to be engaging and informative for the participants, meaning the controlled conditions and intense observation desirable of a more formal user study were not

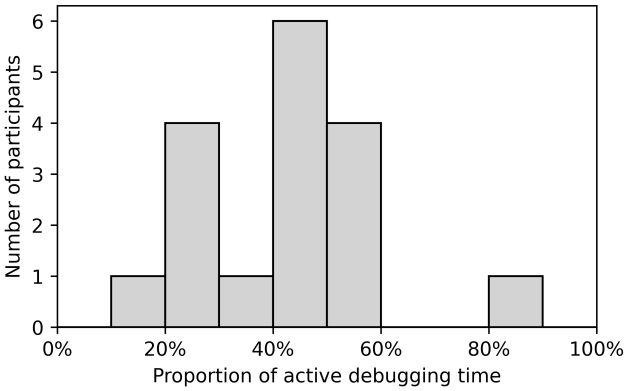

Fig. 7. Distribution: proportion of participant time in an active debugging session.

feasible. Participants were not required to focus entirely on the exercises at hand, nor work individually, nor stay for the whole runtime. Collecting more detailed information about participant behaviour would be difficult without more invasive techniques such as screen recording and diaries, which are arguably inappropriate in this context. Additionally, the students were still at an early stage in their PL-specific eduction; at least 60% of participants reported no prior experience with each of proof assistants, symbolic execution –based tools, CSE –based tools, and separation logic (see the extended paper). In contrast, the suite of exercises was created based on prior demonstrations of Gillian debugging to attendees of a fourth-year undergraduate course on separation logic. This resulted in less progress through the exercises than desired, with participants completing up to 6 of the 13 introductory exercises, and not reaching the advanced exercises.

After discussion with students and analysing these results, we believe that Gillian debugging has greater potential than simply assisting with verification tasks. We have formed new hypotheses on its use as a powerful tool for learning about CSE and its practical applications. For instance, we believe that newcomers to Gillian would use the debugger much more extensively than advanced users, who would reserve its use for more complex debugging tasks. While this hypothesis seems consistent with the positive responses to statements 3, 4 and 6, further study is required to assert it with greater confidence. Similarly, the participants' inexperience with other CSE tools, together with the responses to statements 11 and 12, suggest that interactive visualisation could prove helpful for other analysis techniques such as theorem proving or abstract interpretation.

8 Lessons Learned & Future Work

Our experience developing and evaluating the symbolic execution debugger has revealed potential in making CSE tools more accessible, while highlighting work that needs to be done. We reflect on the lessons learned through this project, and outline directions for future research.

User study. Performing a user study is a rare occurrence in this space, and the efforts to perform one in this work were perhaps unexpectedly valuable; despite an evaluation of Gillian debugging being its primary motivation, it resulted in interesting discussions on the nature of tool usability and how it could vary for different analyses and levels of user experience. In addition to demonstrating educational value, future studies might target experienced tool users performing real-world verification tasks, observing in finer detail to perform focused evaluations of specific design choices. It may also be prudent to observe more objective metrics, such as the overall size or amount of branching in resulting trees.

Gillian. Due to its highly modular nature, Gillian proved to be a useful test bed for exploring debugging in this style, naturally leading to the separation of responsibilities between the compiler, analysis engine, and lifter. Its language-agnosticism provides the opportunity to explore the creation of symbolic execution trees for other languages without re-implementing the debugger internals, particularly to investigate debugging analyses of real-world programs using Gillian's implementations for JavaScript and C. To this effect, a prototype lifter for our second Gillian-C has been created with the same techniques used for WISL. In addition to semi-automatic verification, Gillian can perform automatic, bounded, whole-program symbolic testing (with non-compositional symbolic execution) and true bug-finding with bi-abduction; further work will be able to explore the applicability of the debugging interface introduced in this work to these analyses.

Scaling analyses. Our focus on small, educational verification examples naturally raises the question of scaling to real-world programs, both in terms of lifter complexity for a real-world language and the interface's ability to elegantly handle larger execution trees. Future work will apply our approach to analyses on non-trivial C programs, via both Gillian and other analysis tools such as CN.

Presenting the symbolic state. While this work primarily focuses on presenting an intuitive shape of analysis, more attention will be needed on the presentation of the symbolic state. For example, the WISL lifter prints the GIL-level expressions produced by the engine in a more WISL-esque syntax, but the state still includes intermediate variables introduced during compilation. These variables can be confusing to users as they seemingly have no basis in the source program, but are a necessary part of Gillian's analysis on its intermediate representation. An investigation into how to meaningfully display CSE states at the source level could prove useful in improving user experience.

Error messages. Error messages are a notable pain point of this implementation, corroborated by the user survey (see Tbl. 2, statement 8). Verification error messages are difficult to implement, as it is often impossible to differentiate between an erroneous implementation, an erroneous specification, or insufficient proof annotations. Nonetheless, we do believe that Gillian's error messages have room for improvement. Interactive debugging could turn an immutable error message into a conversation with the tool, exploring the entire analysis tree to refine the error into something more specific.

UI generality. Our DAP extensions and custom UI have already been decoupled from Gillian and VSCode to form the SEDAP. Further work will realise the goal of SEDAP to provide debugging for many symbolic analysis tools, connected to many well-known editors. Our hope is that this will in turn expand SEDAP's capability to represent different internal processes, types of analyses, or volumes of information. How would SEDAP fare with abstract interpretation? Could it be extended to display symbolic heap visualisations? Could interactivity be leveraged to, for example, apply proof tactics in a live debugging session?

9 Conclusion

We have introduced a novel debugging interface for Gillian, addressing the underexplored challenge of creating intuitive debugging interfaces for compositional symbolic execution tools. We focus on providing a familiar user experience by leveraging VSCode's existing debugging interface for standard UI elements, while developing custom components specifically designed to present compositional symbolic execution information in an intuitive way that accurately captures the analysis flow, including branching, nested information and state matching.

We have given the conceptual principles underlying our design, emphasising transferability of the approach to other tools, and discussed interesting technical insights that arose during implementation in the hope that they may inform future endeavours in this space. Importantly, we have decoupled the UI from Gillian and VSCode as much as possible, resulting in an independent library which we hope to continue developing and see adopted by other tools and editors.

Finally, we have conducted a preliminary evaluation of the debugger through a quantitative and qualitative analysis of its use by early researchers during a lab session, providing validation of its usefulness in understanding and working with compositional symbolic execution.

While the amount of engineering work involved to start proper research on improving the usability of compositional symbolic execution tools is significant—and several attempts have been made in the past and later abandoned—we hope that this work will serve as a stepping stone for future research in this space.

Data Availability Statement

An artifact containing Gillian binaries, the debugger VSCode extension, and example WISL programs (including the exercises used in evaluation) is available at [13].

Acknowledgements

We extend a heartfelt thanks to Opale Sjöstedt, Simon Park and Diego Cupello for their tireless work in creating exercises for the user study, as well as the participants of the study for making our evaluation possible; Jessica Shi and Alastair Donaldson for their advice on constructing the user study and this paper respectively; Petar Maksimović for his support in this project's inception; and Radu Lacraru and Matthew Ho, whose MEng projects formed the engineering foundation that this project was built upon.

This work was supported by funding from Gardner's UKRI fellowship 'Verified Trustworthy Software Specification' and grants from Meta and Amazon.

Bibliography

[1] Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.): Deductive Software Verification – The KeY Book, Lecture Notes in Computer Science, vol. 10001. Springer International Publishing, Cham (2016), ISBN 978-3-319-49811-9 978-3-319-49812-6, https://doi.org/10.1007/978-3-319-49812-6

[2] Astrauskas, V., Bílý, A., Fiala, J., Grannan, Z., Matheja, C., Müller, P., Poli, F., Summers, A.J.: The prusti project: Formal verification for rust (invited). In: NASA Formal Methods (14th International Symposium), pp. 88–108, Springer (2022), URL https://link.springer.com/chapter/10.1007/978-3-031-06773-0_5

[3] Ayoun, S., Denis, X., Maksimović, P., Gardner, P.: A hybrid approach to semi-automated rust verification. Proc. ACM Program. Lang. **9**(PLDI), 970–992 (2025), https://doi.org/10.1145/3729289, URL https://doi.org/10.1145/3729289

[4] Cadar, C., Dunbar, D., Engler, D.: KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation, pp. 209–224, OSDI'08, USENIX Association, USA (Dec 2008)

[5] Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004), Lecture Notes in Computer Science, vol. 2988, pp. 168–176, Springer (2004), ISBN 3-540-21299-X

[6] Eilers, M., Müller, P.: Nagini: a static verifier for python. In: Computer Aided Verification: 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I 30, pp. 596–603, Springer (2018)

[7] Ernst, G., Blau, J., Murray, T.: Deductive verification via the debug adapter protocol. Electronic Proceedings in Theoretical Computer Science **338**, 89–96 (Aug 2021), ISSN 2075-2180, https://doi.org/10.4204/eptcs.338.11, URL http://dx.doi.org/10.4204/EPTCS.338.11

[8] Fragoso Santos, J., Maksimović, P., Ayoun, S.E., Gardner, P.: Gillian, part I: A multi-language platform for symbolic execution. In: Programming Language Design and Implementation, PLDI (2020), ISBN 9781450376136, https://doi.org/10.1145/3385412.3386014, URL https://doi.org/10.1145/3385412.3386014

[9] Hentschel, M., Bubel, R., Hähnle, R.: The symbolic execution debugger (sed): a platform for interactive symbolic execution, debugging, verification and more. International Journal on Software Tools for Technology Transfer (2019), ISSN 1433-2787, https://doi.org/10.1007/s10009-018-0490-9, URL https://doi.org/10.1007/s10009-018-0490-9

[10] Holter, K., Hennoste, J.O., Saan, S., Lam, P., Vojdani, V.: Abstract debugging with gobpie. In: Proceedings of the 2nd ACM International Workshop

on Future Debugging Techniques, p. 32–33, DEBT 2024, Association for Computing Machinery, New York, NY, USA (2024), ISBN 9798400711107, https://doi.org/10.1145/3678720.3685320, URL https://doi.org/10.1145/3678720.3685320

[11] Jacobs, B.: verifast-vscode (2023), https://github.com/verifast/verifast-vscode [Accessed 2025/10/17]

[12] Karmios, N., Élie Ayoun, S., Gardner, P.: Gillian debugging: Swinging through the (compositional symbolic execution) trees, extended version (2026), URL https://arxiv.org/abs/2602.07742

[13] Karmios, N., Ayoun, S.E., Gardner, P.: [Artifact] Gillian Debugging: Swinging through the (Compositional Symbolic Execution) trees (Jan 2026), https://doi.org/10.5281/zenodo.18311740, URL https://doi.org/10.5281/zenodo.18311740

[14] Lööw, A., Nantes-Sobrinho, D., Ayoun, S.E., Cronjäger, C., Maksimović, P., Gardner, P.: Compositional Symbolic Execution for Correctness and Incorrectness Reasoning. In: Aldrich, J., Salvaneschi, G. (eds.) 38th European Conference on Object-Oriented Programming (ECOOP 2024), Leibniz International Proceedings in Informatics (LIPIcs), vol. 313, pp. 25:1–25:28, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2024), ISBN 978-3-95977-341-6, ISSN 1868-8969, https://doi.org/10.4230/LIPIcs.ECOOP.2024.25, URL https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2024.25

[15] Lööw, A., Park, S.H., Nantes-Sobrinho, D., Ayoun, S.E., Sjöstedt, O., Gardner, P.: Compositional symbolic execution for the next 700 memory models. Proc. ACM Program. Lang. **9**(OOPSLA2) (Oct 2025), https://doi.org/10.1145/3763151, URL https://doi.org/10.1145/3763151

[16] Maksimović, P., Ayoun, S.E., Santos, J.F., Gardner, P.: Gillian, part ii: Real-world verification for javascript and c. In: Computer Aided Verification (CAV) (2021), ISBN 978-3-030-81687-2, https://doi.org/10.1007/978-3-030-81688-9_38, URL https://doi.org/10.1007/978-3-030-81688-9_38

[17] Meta: Infer Static Analyzer (2025), https://fbinfer.com/ [Accessed 2025/10/17]

[18] Microsoft: Visual Studio Code (2024), https://code.visualstudio.com [Accessed 2025/10/17]

[19] Microsoft: Debug Adapter Protocol (2025), https://microsoft.github.io/debug-adapter-protocol/ [Accessed 2025/10/17]

[20] Microsoft: Language Server Protocol (2025), https://microsoft.github.io/language-server-protocol/ [Accessed 2025/10/17]

[21] Müller, P., Schwerhoff, M., Summers, A.J.: Viper: A Verification Infrastructure for Permission-Based Reasoning. In: Jobstmann, B., Leino, K.R.M. (eds.) Verification, Model Checking, and Abstract Interpretation, vol. 9583, pp. 41–62, Springer Berlin Heidelberg, Berlin, Heidelberg (2016), ISBN 978-3-662-49121-8 978-3-662-49122-5, https://doi.org/10.1007/978-3-662-49122-5_2

[22] O'Hearn, P.: Separation logic. Commun. ACM **62**(2), 86–95 (jan 2019), ISSN 0001-0782, https://doi.org/10.1145/3211968, URL https://doi.org/10.1145/3211968
[23] Panić, S.: Collections-C (2025), https://github.com/srdja/Collections-C [Accessed 2025/10/17]
[24] Raad, A., Berdine, J., Dang, H.H., Dreyer, D., O'Hearn, P., Villard, J.: Local Reasoning About the Presence of Bugs: Incorrectness Separation Logic. In: Lahiri, S.K., Wang, C. (eds.) Computer Aided Verification, pp. 225–252, Springer International Publishing, Cham (2020), ISBN 978-3-030-53291-8, https://doi.org/10.1007/978-3-030-53291-8_14
[25] The Infer Team @ Meta: Infer Pulse (2025), https://fbinfer.com/docs/checker-pulse [Accessed 2025/10/17]
[26] Wolf, F.A., Arquint, L., Clochard, M., Oortwijn, W., Pereira, J.C., Müller, P.: Gobra: Modular specification and verification of go programs (extended version) (2021)