Bringing Invariant Analysis to modern IDEs: The DIG+ Extension for VS Code

Stefania Piciorea
George Mason University
USA

ThanhVu Nguyen
George Mason University
USA

ABSTRACT

Program invariants, which are properties that hold at specific program locations, are important in formal program verification and analysis. Traditional invariant generation methods using dynamic and static analyses are abundant and powerful, supporting a wide range of applications. However, these tools often remain underutilized due to their complex command-line interfaces and the technical expertise required for usage.

To bridge the gap between research and practical application, we have developed DIG+, which integrates the DIG invariant generator and the CIVL symbolic execution tool using the Language Server Protocol (LSP) design used in modern IDEs, such as VS Code. DIG+ simplifies the process of invariant generation by automating setup tasks and providing an intuitive and familiar interface for developers in VS Code. This integration allows users to generate and check invariants directly within their favorite IDEs, enhancing accessibility and usability. We hope DIG+ will inspire researchers to develop similar IDE integration for their research tools, making them more attractive to end users.

DIG+ can be downloaded from GitHub at **https://github.com/dynaroars/dig/tree/dig-vscode**. A video demonstrating DIG+ is available at **https://youtu.be/ZqbjLZptbeE**.

KEYWORDS

Invariant Generation and Checking, LSP, VSCode Extension, IDE

ACM Reference Format:
Stefania Piciorea and ThanhVu Nguyen. 2022. Bringing Invariant Analysis to modern IDEs: The DIG+ Extension for VS Code. In . ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Program invariants are properties that always hold at specific program locations, such as pre- and post-conditions, loop invariants, and assertions. Invariants are frequently used in formal verification (e.g., loop invariants in Hoare logic), and program synthesis, but have also been found to be useful in many other programming tasks, such as documentation, testing, debugging, code generation, and synthesis [1, 3, 4].

Invariants can be discovered or inferred using dynamic or static analyses. A static analysis can reason about all program paths soundly, but doing so is often expensive and is only possible for relatively simple forms of invariants. Dynamic analyses limit their attention to only some of a program's paths, and as a result can often be more efficient and produce more expressive invariants, but provide no guarantee that those invariants are correct.

In recent years, several systems have started to take a hybrid approach that uses dynamic analysis to infer *candidate invariants* and a static verifier to confirm their validity. For example, the DIG [4] approach targets rich numerical invariants by integrating dynamic inference and symbolic checking. Dynamic inference allows DIG to efficiently discover many useful and rich classes of invariants from program traces, while symbolic and static checking helps remove and refine spurious results. The DIG approach has been used to support a wide range of application domains, e.g., program termination, heap analysis, program rewriting and transformation, complexity reasoning, configuration analysis, and algebraic specifications.

Despite fruitful and abundant research on invariant discovery, DIG and existing techniques and tools are not widely used in practice, e.g., in industry, research labs, or even in classrooms for teaching. One of the main reason is that research and prototype tools are created mainly to demonstrate technical research ideas, thus invariant generation and program analysis tools are developed as command line utilities and can be difficult to setup, use, and understand. Moreover, they are often not accessible to engineers who may not be familiar with research tools or may not have the time to learn them.

To bridge the gap between research and practice in invariant research, we have developed DIG+, an extension that leverages Language Server Protocol technology to integrate the capabilities of the DIG invariant generator and the CIVL symbolic execution tool into the VS Code IDE (Integrated Development Environment). Specifically, DIG+ has quick and useful automated assertion generation features that one would expect from a modern IDE e.g., user selects a location and DIG+ automatically infers invariants as assertions at that location. Moreover, DIG+ exploits LSP client and server interaction design to send information between the backend DIG and CIVL tools and the user through the VSCode editor (e.g., changing invariant types or symbolic execution depths, checking and removing spurious results). The goal of DIG+ is to provide a user-friendly interface for developers to generate and test invariants directly within a modern IDE, streamlining the process of invariant inference and checking.

Another goal of DIG+ is to provide researchers with an easy way to make their research more visible and accessible to users who are familiar with IDEs but not with command-line based research tools. Researchers can adapt their static or dynamic analysis tools to work within the Language Server Protocol (LSP), enabling these tools to provide services (like error checking and code completion) directly within IDEs. For example, in addition to creating a command-line

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

fault localization or program repair tool, the developers can also create an LSP connecting the tool with the IDE to allow users to repair code directly through any code editor that supports LSP. Given the popularity of IDEs such as VSCode and their powerful extension ecosystem, we believe this will make research tools more attractive to end users, who are often excited to try new extensions that are easy to install and use.

Targeted Users. We target industrial developers interested in discovering program specifications and invariants for documentation and debugging purpopses. We also aim to introduce formal methods and invariant generation to students and software developers in program analysis courses hoping to inspire them to leverage and contribute to DIG+.

Availability. DIG+ has been developed as a VS Code extension and can be downloaded from GitHub at **https://github.com/dynaroars/dig/tree/dig-vscode**. A video demonstrating DIG+ is available at **https://youtu.be/ZqbjLZptbeE**.

2 BACKGROUND

2.1 Invariant Generation and Checking Tools

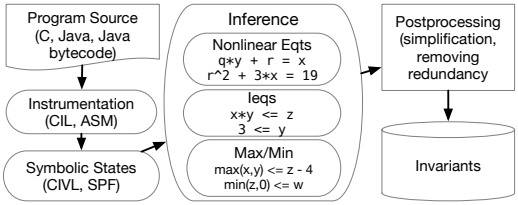

Fig. 1: DIG Invariant Generation

DIG. Fig. 1 gives an overview of the dynamic invariant generation tool DIG [4]. DIG *finds invariants* by iterating between (i) *dynamic analysis*, which infers candidate invariants from program execution traces obtained by running the program from sample inputs, and (ii) *symbolic checking*, which checks candidates against the program using a symbolic execution tool. Reporting too many invariants, even if they are all valid, would be a burden to the user. Thus, DIG uses a *post-processing step* to reduce the number of reported invariants (e.g., using SMT checking to eliminate weaker or implied invariants).

The DIG inference approach has been used to analyze many important classes of invariants, e.g., nonlinear numerical relations, array and heap properties, termination and temporal properties, and interactions among configuration options.

CIVL. CIVL [5] is a formal verification tool that checks the correctness of programs using symbolic execution to explore program states. If the tested assertion fails in any state, the asserted property is invalid. For complex programs, symbolic execution cannot explore all possible states but can still be highly effective in finding bugs, e.g., computing an input that causes the program to violate an asserted property. DIG and DIG+ both use CIVL to find such counterexample inputs to determine spurious invariants and invalid assertions.

```
Violation 0 encountered at depth 5:
CIVL execution violation in p0 [Library: assert.h, Function: assert] (prope
at assert.cvl:7.2-14
  $assert(expr);
  ^^^^^^^^^^^^^

Assertion: expr
 -> false

Input:
  x=X_x
  y=X_y
Context:
  0<=(X_x+(-1*X_y))
  0<=(X_x-1)
  0<=(X_y-1)
Call stacks:
process 0:
  assert@4 assert.cvl:7.2-8 "$assert" called from
  mainQ@21 output.c:29.6-11 "assert" called from
  main@2 output.c:47.2-6 "mainQ"

Logging new entry 0, writing trace to CIVLREP/output_0.trace
Terminating search after finding 1 violation.
```

Fig. 2: CIVL command-line output

While both tools are powerful, they might not be accessible to developers who are not familiar with program analysis techniques. Running DIG from the command line requires a certain level of technical knowledge, as users must navigate through a series of manual setup steps, including cloning the DIG repository and building the system. Additionally, the terminal output from DIG is often verbose and complex, making it challenging for users to interpret the results accurately. Similarly, for CIVL, the user would need to write specific CIVL code and commands and invoke the CIVL command line tool (which itself requires the Java JDK and an SMT solver). Moreover, as shown in Fig. 2, the result of CIVL can be difficult to interpret and misguiding, e.g., the failed assertion is *not* on line 29 of the original input file. These limtiation motivate the development of DIG+ for a more streamlined and user-friendly experience of using DIG and CIVL.

2.2 Language Server Protocol (LSP)

LSP is a standard protocol that allows development tools (text editors such as Emacs and VIM and IDEs such as VSCode and Eclipse) to communicate with language servers (or in this case, program analysis tools). DIG+ uses LSP and consists four main components shown in Fig. 3.

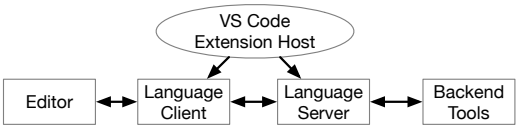

Fig. 3: Overview of DIG+

① The **editor** provides a graphical user interface (GUI) for the user. It hosts features such as syntax highlighting, autocomplete and menu popups. It interfaces with the language client (②) through its API to display information such as inferred invariants or validation/error messages from the DIG and CIVL backends.

② The **language client** interacts with the editor and the language server by sending and receiving information, e.g., getting the content of an opened file, text/cursor position, invariant results or error messages. Whenever the client is

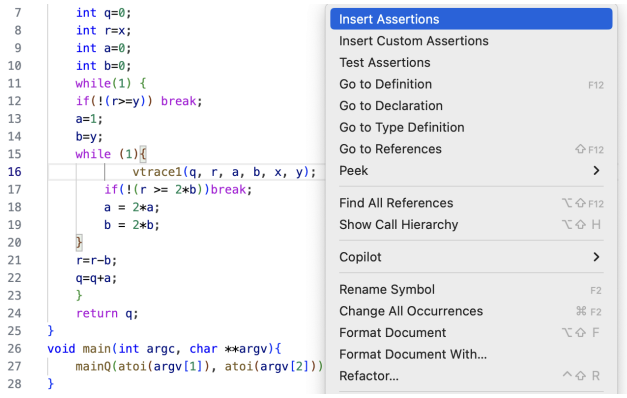

Fig. 4: Pop-up Menu for Invariant Generation

activated, it creates a language server to which data is sent to.

③ The **language server** is a proxy to our backend tools. When a request from the client is received, the server retrieves the source code and sends it to DIG and CIVL for analysis. The analysis results are composed into the LSP response format and returned to the client through the editor.

④ DIG+'s **backend** consists of the DIG invariant generation and CIVL symbolic execution tools. DIG infers candidate invariants and CIVL checks the inferred (or user-supplied) invariants. Both tools are integrated into the language server and communicate with the client through it (e.g., sending results and error feedback).

Implementation. DIG+ is implemented in TypeScript, the default language for VSCode extensions. DIG+ uses a script to install and setup DIG and CIVL on a Linux environment using Docker, ensuring the tools are running on a consistent and isolated Linux Docker instance (i.e. DIG+ can run on Windows, Mac, or Linux). DIG+ consists of separate scripts that facilitate the interaction between the backend, language clients and servers, and the frontend editor. Other scripts preprocess C files into formats accepted by the backend tools and handle specific operations. When beneficial to performance, we leverage multiprocessing and run tasks in parallel. For example, DIG+ runs multiple instances of CIVL to check invariants, which allows for comprehensive feedback on all assertions in a single run, contrasting with the traditional command-line method where CIVL stops at the first invalid assertion, requiring manual intervention to proceed further.

3 USING DIG+

To improve the usability and adoption of invariant analysis, DIG+ provides a user-friendly interface to generate (discover) and test invariants.

3.1 Invariant Generation

DIG+ enhances the development experience in VSCode by enabling the automatic inference of invariants directly on a working C file. The user initiates the inference process by calling a special method

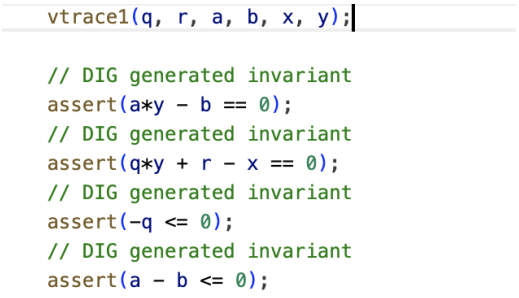

Fig. 5: Generated Invariants as Assertions

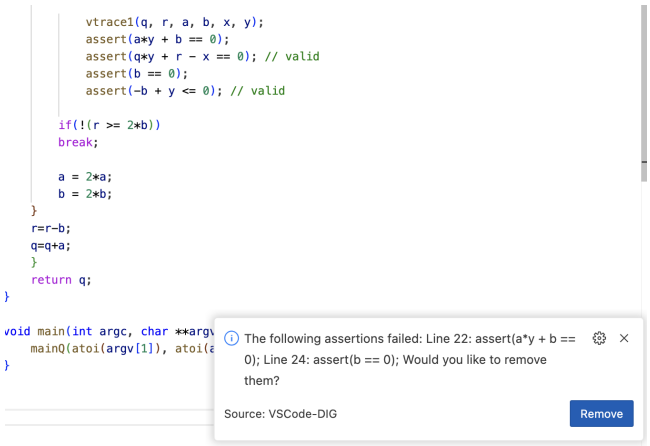

Fig. 6: Checking Invariants with CIVL

vtrace() at desired locations, and selects the **Insert Assertion** option from a pop-up menu as shown in Fig. 4.

This action triggers the language client to send the code to the language server, which then invokes DIG to infer invariants at the vtrace-marked locations. After DIG generates invariants, the process is reversed, and the language server sends these results to the language client, which then inserts them into the C file as *assertions* as shown in Fig. 5. The user can use these assertions to increase their confidence in the correctness of their code.

DIG+ also gives various visual cues to the user during the process. For example, a progress bar is displayed to indicate that the inference process of DIG is running. This feature is particularly important for longer operations, as it helps users understand that the delay is normal and not indicative of a failure or crash. Note that if DIG+ encounters issues, such as a failure to generate invariants, users are promptly notified through the IDE's notification system.

3.2 Invariant Checking

DIG+ uses CIVL to check invariants or assertions. This is important to confirm the validity of dynamically inferred or manually inserted invariants, e.g., introduced by the user or through external tools such as GitHub Copilot which is readily integrated with VSCode.

Feedback from CIVL is integrated into the IDE, offering developers real-time insights into the outcomes. When CIVL successfully checks an assertion, DIG+ automatically annotates these lines with

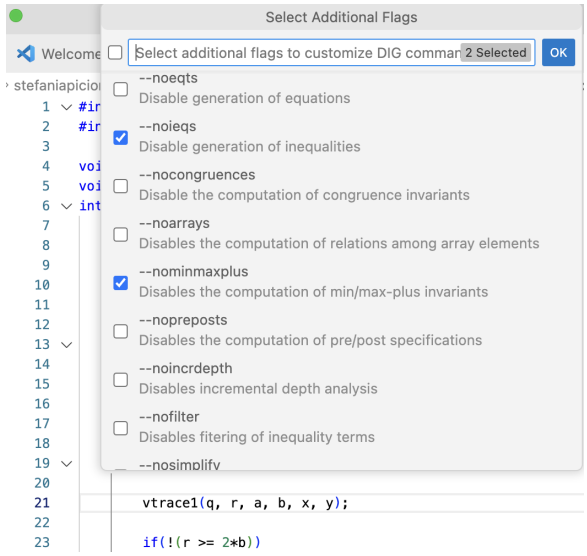

Fig. 7: QuickPick UI for customizing DIG

a comment stating "//valid" to indicate their validity. Conversely, if CIVL finds a violation, DIG+ displays a notification detailing the location and assertion that is not valid. The notification also prompts the user with options to remove the incorrect assertion as shown in Fig. 6. Moreover, DIG+ allows checking single or multiple assertions simultaneously, offering flexibility for bulk checking and reducing the need for manual intervention.

3.3 Customizations

Command line tools such as DIG often require users to specify various flags to customize the invariant generation process. These flags can be challenging to remember and use, especially for developers who are not familiar with the tool.

To address this issue, DIG+ provides a user-friendly interface that allows developers to customize the behavior of the DIG command through an intuitive VSCode-built in QuickPick UI. This interface presents a list of flags that users can select to fine-tune the invariant generation process. Each flag is documented within the QuickPick UI, providing clear descriptions and, when necessary, prompting for additional input. This enables developers to tailor the invariant generation process to better align with their project requirements, ensuring that the generated assertions are relevant and useful. Fig. 7 shows the QuickPick UI for customizing DIG.

4 RELATED WORK

Well-known related invariant tools include Daikon [1] and Infer [2]. Daikon is a pure dynamic invariant detection tool that infers properties from program executions. Facebook Infer performs interprocedural static analysis to compute abstraction (a form of invariants) to identify bugs in Java, C, and Objective-C code. Despite their capabilities, both are command-line interface tools similarly to DIG and thus are limited in accessibility and adoption among developers who prefer modern integrated development environments (IDEs). In contrast, commercialized products such as SonarQube [6] and Coverity [7] offer comprehensive GUI systems that enhance usability. SonarQube provides continuous code quality inspection with a focus on detecting bugs, vulnerabilities, and code smells, supporting multiple languages and integrating with CI/CD pipelines. Coverity specializes in finding security and reliability defects, offering a web-based dashboard and IDE integration. However, neither system focuses on LSP or invariant analysis.

To address these limitations, DIG+ automates setup tasks and provides an intuitive interface, allowing users to generate and check invariants directly within their IDE. This LSP-based integration ensures that DIG+ can be used with any editor or IDE supporting LSP, such as Emacs, Vim, and VSCode, significantly enhancing its accessibility and practical usability for developers and researchers.

Finally, modern AI/LLM-based technologies such as Github Copilot have the ability to suggest interesting invariants and assertions directly in the IDE. However, these suggestions are not always reliable and may require manual verification. DIG+ provides a streamlined process for checking these suggestions, ensuring the validity of these ML-generate invariants.

5 CONCLUSION

We presented DIG+, an LSP that extends modern IDEs to support invariant generation and analysis. The current implementation of DIG+ uses the DIG invariant generation and CIVL symbolic execution tools and integrates them into the Visual Studio Code IDE. DIG+ automates the setup, execution, and usage of DIG and CIVL, reducing the need for command-line interactions and allowing non-experts to easily leverage these tools. We hope that DIG+ will make invariant generation and analysis more accessible to developers, and that DIG+ will inspire researchers to develop similar user-friendly IDE integration for other program analysis tools.

REFERENCES

[1] Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *Science of computer programming* 69, 1-3 (2007), 35–45.

[2] Meta Platforms, Inc. 2024. Infer. https://fbinfer.com/. Accessed: February 12, 2025.

[3] ThanhVu Nguyen, Deepak Kapur, Westley Weimer, and Stephanie Forrest. 2012. Using Dynamic Analysis to Discover Polynomial and Array Invariants. In *International Conference on Software Engineering*. IEEE, 683–693.

[4] Thanhvu Nguyen, KimHao Nguyen, and Matthew Dwyer. 2021. Using Symbolic States to Infer Numerical Invariants. *Transactions on Software Engineering (TSE)* (2021).

[5] Stephen F Siegel, Manchun Zheng, Ziqing Luo, Timothy K Zirkel, Andre V Marianiello, John G Edenhofner, Matthew B Dwyer, and Michael S Rogers. 2015. CIVL: the concurrency intermediate verification language. In *International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–12.

[6] SonarSource. 2024. SonarQube. https://www.sonarqube.org/. Accessed: 2024-06-26.

[7] Synopsys, Inc. 2024. Coverity. https://www.synopsys.com/software-integrity/security-testing/static-analysis-sast.html. Accessed: 2024-06-26.