國立臺灣大學電機資訊學院資訊工程學系
博士論文
Department of Computer Science and Information Engineering
College of Electrical Engineering and Computer Science
National Taiwan University
Doctoral Dissertation

揭開除錯暗物質：JavaScript 全知除錯器 Dbux 之研發
Uncovering the Dark Matter of Debugging with Dbux, an Omniscient Debugger for JavaScript

薛德明
Dominik Seifert

指導教授：許永真博士 & 葉丙成博士
Advisor: Jane Hsu, Ph.D. & Benson Yeh, Ph.D.

中華民國 111 年 10 月
October, 2022

doi:10.6342/NTU202210012

doi:10.6342/NTU202210012

誌謝

首先，我想感謝我的兩位指導教授許永真和葉丙成，他們總是非常有耐心地支持我和我的研究。萬俊彥積極主動地解決問題，對這個專案提供了很大的幫助。非常感謝提案和論文答辯委員會成員們的辛勞、回饋和深度的討論。也非常感謝陳冠宇和陳俊瑋的貢獻，以及 Alpha Camp 員工和工作坊參與者的回饋。

在過去的三年間，我的女朋友總是非常體諒我，也協助我打理各種我無暇顧及的事務。同時也感謝我的幾個非常好的朋友在這段時間裡不斷地聽我分享我的研究、給予回饋和忍受我的行為。

最後我要感謝我親愛的家人，即使他們遠在地球的另一端，當我需要幫助的時候他們總是在那裡支持我。當我失去繼續前進的力量時，想到你們還在支持著我，就能帶給我很大的安慰，讓我在生活和這份研究中能夠繼續前進。非常感謝你們！

揭露：這份研究有獲得一些我個人的非常小的公司：自主人生有限公司的經費支持。

doi:10.6342/NTU202210012

doi:10.6342/NTU202210012

Acknowledgements

With that out of the way, I would like to start by thanking both my advisors Jane Hsu and Benson Yeh for being incredibly patient with and always supporting of me as a person and my work. Michael Wan and his proactive approach to problem-solving was a tremendous help to this project. I am very grateful toward the proposal and dissertation defense committee members for all their hard work, feedback and in-depth discussions.

I would also like to thank Kuan-Yu Chen and Jyun-Wei Chen for their contributions, as well as the Alpha Camp staff and workshop participants for their participation and feedback.

In these past three years, my girlfriend has always been understanding and helping take care of things when I was not able to. Similarly, I want to thank my few, very good friends who have been listening to me talk about my stuff, giving feedback and just generally put up with me, during this time!

Lastly, I want to thank my beloved family. Despite living on the other side of the globe, they have always been there and offered to be there for me, when and if I ever needed them. Knowing that you will still be there even in the event that I lose my strength, always provided great comfort to me, and allowed me to keep going, in life and with this work in particular. Thank you all so much!

Disclosure: This work has received some financial support from Life is Learning Co., Ltd. (Taipei, Taiwan), my own, very small company.

doi:10.6342/NTU202210012

doi:10.6342/NTU202210012

摘要

對軟體開發者來說，最困難而耗時的工作便是除錯和理解他們的程式，因為這些工作幾乎都得在「黑暗中」進行。即使是富有經驗的開發者，也必須花費大量的時間來從一條可能很長的因果鏈中的一小段來識別和擷取出關鍵的資訊。

在這篇論文中，我們明確地指出會讓這些工作如此困難的原因，我們稱之為「除錯暗物質」。此外，我們提出了一個關於交互除錯程序的描述模型，來捕捉以往的模型中遺漏的重要細節。接著我們利用這個模型來為「除錯之旅」這個案例研究方法提供一個指引。我們提供了一個檢核表來總結這個方法，並且在針對現有的專案進行測試之後證明，這個檢核表對於評估案例研究的完整性是有其價值的。接下來在這篇論文中我們也會用這個方法來進行案例研究。

在除錯的過程中重要的程式資訊常常是隱形、無形的，為了解決這個問題，我們引入了一個 JavaScript 全知除錯器兼整合式除錯環境：Dbux。它的動態呼叫圖、低階事件分析器、程式碼編輯器整合環境和動態分析搜尋工具讓開發者能夠獲得許多不同的動態執行資料和洞察。這個工具已開始受到開發者社群的關注——它在 VSCode 的應用程式商店獲得超過 1500 個下載數，在 Github 上也已獲得超過 100 顆星星。

Dbux 從兩種角度為開發者提供協助，分別是 (1) 資料和 (2) 異步控制流：(1) 如排序的穩定性或堆積在圖遍歷演算法中的作用這類演算法的性質與實作是演算法教科書中的經典內容，但即使對於一個認真的學習者來說也無法突破其無形的本質。Dbux-PDG 的*程式依賴關係*

doi:10.6342/NTU202210012

圖能夠呈現演算法實作背後的資料流和依賴關係。對於某些演算法，它新穎的分層摘要技術能夠清楚顯示出其重要的內在特性，讓使用者能夠更進一步研究這些特性背後的成因。這也顯示出它在教育環境中的實用性。經過測試，這個功能在 94 種不同的演算法中有 48% 能夠發揮作用。(2) 在 JavaScript 或其他語言中，異步執行的程式常常是造成競賽情形的主要原因，也是程式理解中一個很大的障礙。我們在 Dbux-ACG 的*異步呼叫圖*採用了一個嶄新的演算法來將呼叫圖的根分成「分支」和「鏈」兩類。它能夠顯示出 11 個熱門的現實世界 JS 程式庫中的異步控制流、並發程度和並發模式。

作為一個工具套件，Dbux 為開發者們提供了大量過去所看不到的資料和洞察，使 JavaScript 應用程式的「除錯暗物質」變得更加清楚而具有互動性。也展現了它能夠幫助在教育環境中的學習者和在工作環境中的專業開發者的潛力。

doi:10.6342/NTU202210012

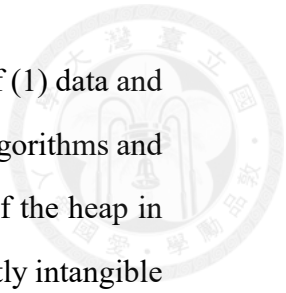
Abstract

Debugging and Program comprehension are among the most difficult and time-consuming tasks for software developers due to the fact that developers are largely operating “in the dark”. When investigating a piece of code or locating a bug, even a skilled developer can spend a significant amount of time identifying and extracting crucial data from only a single location in a potentially long investigative chain.

In this work, we make explicit a major contributor to debugging difficulty, which we, somewhat brazenly, denote “the dark matter of debugging”. Furthermore, a new descriptive model of the interactive debugging process is proposed to more accurately capture important details left out by previous attempts. We then use that model to provide guidelines on the “debugging journey” case study methodology. A resulting checklist summarizing the methodology proves valuable in assessing case study completeness when tested on existing works. The methodology is also used in case studies throughout this dissertation.

To address the invisibility and intangibility of vital program information during debugging, we introduce Dbux, an omniscient debugger and integrated debugging environment for JavaScript. Its dynamic call graph, lower-level event analysis tools, code editor integration and dynamic analysis search tools allow developers to easily access many types of dynamic execution data and insights. It’s over 1500 downloads on the VSCode marketplace and over 100 stars on GitHub serve as a first indicator of public interest.

doi:10.6342/NTU202210012



Two Dbux extensions help make tangible important types of (1) data and (2) asynchronous control flow, respectively: (1) Properties of algorithms and their implementations, such as stability in sorting, or the role of the heap in graph traversal, are a staple in Algorithm textbooks, yet inherently intangible even to a diligent learner. Dbux-PDG's *Program Dependency Graph* renders algorithm implementation data flow and dependencies. Its novel layered summarization technique reveals important intrinsic algorithmic properties when applied to several types of algorithms. Users can further investigate the cause behind some of these properties, showing its utility for educational settings. We have tested it successfully on 48% of 94 different algorithms. (2) Asynchrony is one of the biggest impediments to program comprehension and also the primary source of race conditions in JavaScript and other languages. Dbux-ACG's *Asynchronous Call Graph* uses a novel algorithm to classify call graph root relationships into FORKs and CHAINs. It reveals asynchronous control flow, degree of concurrency and concurrency patterns in eleven popular real-world JS codebases.

Dbux, as a tool suite, provides developers with a plethora of previously intangible data and insights, making the "dark matter of debugging" of JavaScript applications a little more visible and interactive. It shows potential to aid learners in an educational setting and professionals on the job.

Keywords: debugging, program comprehension, program visualization, dynamic analysis, asynchrony, algorithms, data flow, call graph, JavaScript, VSCode, dark matter, Dbux

doi:10.6342/NTU202210012

Contents

doi:10.6342/NTU202210012

doi:10.6342/NTU202210012

doi:10.6342/NTU202210012

doi:10.6342/NTU202210012

doi:10.6342/NTU202210012

doi:10.6342/NTU202210012

doi:10.6342/NTU202210012

doi:10.6342/NTU202210012

List of Figures

doi:10.6342/NTU202210012

doi:10.6342/NTU202210012

doi:10.6342/NTU202210012

doi:10.6342/NTU202210012

List of Tables

doi:10.6342/NTU202210012

doi:10.6342/NTU202210012

Chapter 1

Introduction

1.1 Debugging Story Time – A Motivating Example

Let us set the scene with a common scenario: Alma has discovered a bug in her frontend application. When pressing button `B`, widget `W` should appear in the top right corner of the screen, but it does not, and there is no error message to speak of. The bug is hidden in a single line, located in one of the thousands of files that make up her 100k lines of code. A needle in a haystack – a search problem.

As Alma will find out the hard way: the bug is hidden inside what is referred to as a "cross-cutting concern" – a part of the program that uses or is being used by many other parts of the program, a busy hub of sorts, a train station. On top of that, the bug is nestled deep inside the call graph, and only observable as a symptom of this specific button `B`'s click event. **This is not an easy bug**. Beller et al. [27] found strong consensus among practitioners "that the hardest problems to debug are ones where interfaces or transactions between components are involved".

As a first strategy, Alma tries setting a breakpoint at the beginning of and then **stepping** through the click event handler function of `B`, but since the bug is in the deep end of the call graph, step-based debugging using the traditional debugger is largely infeasible for three reasons: (i) The amount of steps required is considerable. To make matters worse, (ii) stepping is a highly error-prone process, since at every step Alma has to decide: do I step IN or OVER? It is easy to miss the right place to step in. It is easy to over-step.

doi:10.6342/NTU202210012

Since the traditional debugger does not allow stepping backward in time, over-stepping forces the developer to restart the process. (iii) It turns out that the traditional debugger will not lead Alma to the faulty location, when starting in the click event handler, since the actual fault is triggered asynchronously, and the debugger is incapable of following asynchronous causal links. Even if there was a direct path, this experience would still be horrendous. As an analogy, imagine using Google Street View to navigate between two distant cities, but... (i) You can not use Google Maps; i.e. you cannot skip most of that distance by clicking into Maps, nor (ii) use Google Maps to orient yourself. And now add one more constraint of (iii) not being allowed to navigate backwards, but only forward.

Alma realizes her predicament. In a second attempt, she uses her understanding of the program to guess a more relevant location in the code, closer to where she believes the bug is hiding, and sets a **breakpoint** inside "the busy hub code". However, narrowing down the search and determining where to place a breakpoint turns out to be an "extremely difficult task" [122]. Due to the nature of that piece of code being cross-cutting, the breakpoint stops many times because the function is called by an unrelated feature.

To prevent breaking on unrelated events, Alma tries to set a **condition** for the breakpoint, s.t. execution would only halt in the relevant context. This turns out to be impossible since she is unable to access necessary context data for her condition because that data is stored in local variables higher up on the call stack and thus not available to the debugger's condition evaluation engine.

The traditional debugger has failed her. Like most other developers of her time, Alma is forced to resort to a mix of trial-and-error and print debugging strategies. After a lengthy session of tracking down several more places that showed traces of the bug, she finally finds and fixes that one line that was responsible for this long-winded and difficult journey.

1.2 The Problem: Fax Machines and Blueprints

Debugging is one of the most time-intensive and costly endeavors in software engineering. Ko et al. [57] point out that 70% of cost in software development is spent on program understanding and debugging. Beller et al. [27] claimed to have observed developers

doi:10.6342/NTU202210012

only spend 14% on “debugging”, but they only measure time spent in the “traditional debugger”, while a very substantial parts of the debugging process, such as “program comprehension”, print- and other types of debugging, are ignored.

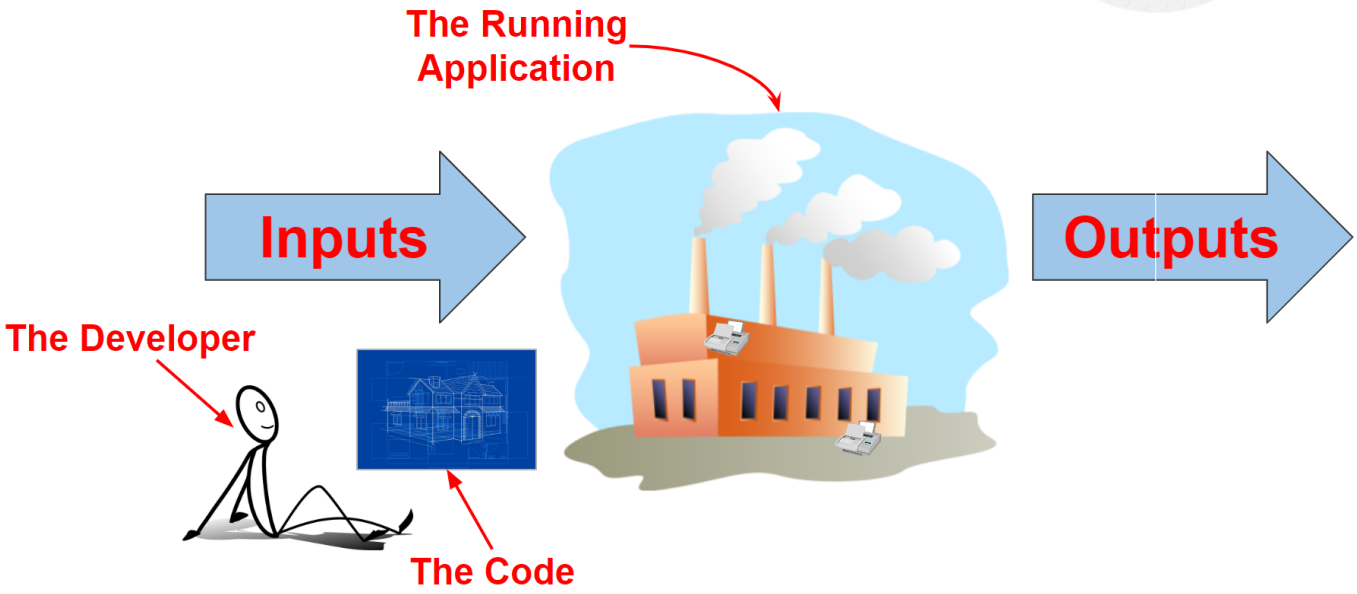


Figure 1.1: An Analogy: How not to Run a Factory

To this date, developers are still required to entirely rely on their mental models to fully capture the complex nuances of their applications and the dependencies within, with little to no tool support. To better understand how borderline ridiculous that is, let us compare debugging of an application to troubleshooting in a large, complex factory: Fig. 1.1 is an illustration of the manager (i.e. the developer). They do their job sitting outside the factory after having turned off all the lights inside. They are only able to see inputs and outputs of the factory, as well as the factory’s blueprint and occasional faxes that are automatically sent by “fax machines,” hooked up to sensors in the factory hall.

That is what is still happening in software development today. The behavior of a running application –as we argue– is similar to “dark matter”, in that we cannot directly see or interact with it. The developer must invest tremendous time and effort in uncovering and reverse-engineering that “dark matter” using printf (i.e. employing a “fax machine”), interleaved with reading and re-reading the code (the “blueprint”). Some developers use the traditional debugger, which is equivalent to troubleshooting by stepping into the vast and dark factory hall in order to scan the floor, one step at a time, equipped only with “a

doi:10.6342/NTU202210012

flashlight and magnifying glass".

1.3 A Solution?

In 1997, in what we like to refer to as an "informal call to arms", Lieberman [76] lamented that the pervasiveness of print debugging is "a sad commentary on the state of the art". Ko [56] echoes that sentiment, commenting that "every day, millions of software developers work" are left with "little more than a breakpoint and a print statement" to tackle big challenges. Lieberman envisioned a future where new innovative debugging tools (like his own ZStep 95 [77] LISP debugger) will help address this issue, and surely become a staple of developer toolboxes around the world.

However, 25 years later, print debugging and the very limited "traditional debugger" are still the most commonly used debugging tools among practitioners [81] [60] [92] [27]. While a lot of progress has since been made in (i) enterprise-level monitoring tools, (ii) domain-specific debugging tools, (iii) testing tools, and even (iv) automatic debugging research, general-purpose debugging tools used by practitioners have not changed for decades. We argue that it is this lack of better tools and the general lack of understanding of the debugging process that belies the difficulty of Alma's (see §1.1) and so many other developers' jobs.

Some argue that better testing would eliminate the problem, but, we agree with Lieberman when he posits that "program verification" cannot magically prevent all bugs, considering how software keeps evolving, in scope, scale and complexity; and better debugging tools are still required.

1.4 A Solution!

A big part of the solution to the "debugging dilemma" would probably lie in the demand-side of the equation: we need better training and a culture that promotes interactive debugging as a much more fundamental component of the development process and celebrates debugging skill, so as to even have a chance at mitigating the immense cost of debugging.

doi:10.6342/NTU202210012

While we discuss that part of the problem to a smaller extent in §3, the rest of this work focuses on the supply side. We present Dbux, a novel omniscient debugger and integrated debugging environment. As we shall explain in §3.3, its goal is to increase tangibility of program behavior facts, thereby assisting investigative processes of software engineers in debugging and program comprehension tasks.

We further propose two extensions Dbux-PDG and Dbux-ACG, aimed at helping developers uncover some of the most difficult types of the causal chains in their application's dynamic execution behavior, targeting data flow and asynchronous control flow, respectively.

1.5 Contributions

A list of the major contributions of this dissertation follows. This research has also resulted in two publications, which are marked in this list.

- A descriptive model of the interactive debugging process §3.2 — We build on our experience in the field to present a model of interactive debugging that captures key facets of the process in the form of a search problem. This is the first informal model to define debugging as an iterative process while accounting for perceived factoids of physical locations, and the crucial role of fact tangibility. We believe that, in the long run, these types of models and its (hopefully ever more accurate) future iterations can be helpful in better reasoning about the low- and high-level processes that practitioners engage in, thereby allowing for more well-defined debugging subprocesses, strategies, skills and maybe even educational scaffolding.
- Definition of "Debugging Journey" case study methodology §3.4 — This is a commonly used methodology in interactive debugger research but lacks clear definition. We compiled a checklist of three goals, five structural guidelines and six process guidelines. When testing the checklist against two existing case studies from the literature, it helps explain aspects of case study comprehensibility and repeatability. This methodology and its checklist are also used repeatedly in case studies

doi:10.6342/NTU202210012

throughout this dissertation.

- A first omniscient debugger for JavaScript (Dbux) §4 — In contrast to previous works, Dbux is the first omniscient debugger to: (1) Support JavaScript, (2) render code decorations to indicate recorded execution behavior, integrated in the standard text editor of an IDE and (3) offer a playground and curated list of popular open-source projects and bugs that can be executed at a single button click (Dbux-Projects). Dbux has been successfully tested in a multitude of real-world scenarios and also serves as the foundation for the following extensions.

- **(Published [107])** A Program Dependency Graph to reveal hidden properties of data structures and algorithms (Dbux-PDG) §5 — Dbux-PDG's *Program Dependency Graph* is the first tool to render summarizable dynamic algorithm data flow and dependencies, and make it interactive. Its novel layered summarization technique reveals important intrinsic algorithmic properties when applied to several types of algorithms and data structure operations. Users can further investigate the cause behind some of these properties, showing its utility for educational settings. We have tested it successfully on 48% of 94 different algorithms.

- **(Partially Published [106])** An Asynchronous Control Graph for revealing concurrency in JavaScript (Dbux-ACG) §6 — We present a new model of asynchronous control flow which superimposes (virtual) threads upon basic units of non-preemptive control flow, which we call "Call Graph Roots" (CGRs). For that, we employ a novel algorithm that uncovers CHAINs and FORKs between CGRs. CGRs, FORKs and CHAINs are then used to produce the world's first "Asynchronous Call Graph". It is also the first asynchronous event model to unify all three of JavaScript's primary concurrency-inducing semantics. We show its ability to visualize inherently concurrent properties of the producer-consumer problem, a classic problem in parallel programming. We further show its efficacy by manually tackling two real-world bugs, and share results from successfully testing it on eleven real-world projects.

doi:10.6342/NTU202210012

1.6 Notes on Notation

Some keywords are written in cursive and have a superscript to link to their definition or primary mention in this thesis, e.g.: *notation*[§1.6].

1.7 Open Source Development & One-click Installer

All tools produced for this research are open source and available on Dbux's Github repository[1] since its inception on 2019/11/16. The repository also contains the Dbux documentation and an issue tracker. Dbux is one-click-installable from the VSCode marketplace[2].

As of 10/2022, Dbux has 1700 downloads on the VSCode marketplace and 120 stars on GitHub.

[1] `https://github.com/Domiii/dbux`
[2] `https://marketplace.visualstudio.com/items?itemName=Domi.dbux-code`

doi:10.6342/NTU202210012

doi:10.6342/NTU202210012

Chapter 2

Related Work

This chapter serves to contextualize this dissertation within the body of existing research on interactive debugging tools and processes.

2.1 Models of Program Comprehension and Debugging

Much of the research on **program comprehension and debugging processes** happened in the 1980s [30] [127] [72] [54] and more models of said processes emerged in the 1990s [22] [45]. Culminating in many variations of models that mostly concern themselves with "mental representations", interest in this area of research seems to have declined. Bidlake et al. [29] (in 2020) counted only 12 empirical studies on mental representations of programs since 2000. While having made important contributions to the field, these early models are considered simplistic, as they eliminate crucial variables, only consider the codebase itself as the sole source of knowledge, and were only tested against very simple pieces of code. Most of them also failed to capture the incremental nature of the process.

Davies [35] and later, Ko et al. [59], claim to rectify this by accounting for multiple knowledge sources, and also loosening "numerous restrictions" which were imposed upon developers "to isolate the measurement of a single variable". Davies makes explicit the fact that these indeed are "incremental processes". Ko et al. account for human error, capture the fact that different contexts require different actions and makes explicit the use

doi:10.6342/NTU202210012

of the debugger in the process, but only captures a high-level description of actions (e.g. "Understanding" being one action).

Böhme et al. [31] found a promising amount of consensus among practitioners in terms of debugging outcomes. That discovery served as a strong motivator for this work.

More similar to our work, LaToza et al. [61] define program comprehension as fact finding. In an empirical study, they found evidence that much of the process comprises (i) seeking, (ii) learning, (iii) critiquing, (iv) explaining, (v) proposing and (vi) implementing facts about code.

The model proposed by Lawrance et al. [66] probably comes closest to ours. They employed **information foraging theory** to model the debugging process in form of, as the name suggests, a task that primarily involves finding and correlating of information. However, unlike ours, their model does not make explicit (i) the role of the debugger or print debugging, (ii) difference between true and perceived state space, (iii) factoids other than links or (iv) the importance of fact tangibility. Nevertheless, much of their terminology can be mapped to ours:

- Their "topology" represents a subset of all our "location links", but constrained to only account for links that can be "navigated" through the UI.
- Their "information patches" are similar to what we denote "locations".
- "Prey" seems to refer to the set of all information patches (i.e. our "locations") "relevant" to the current debugging journey; the most important prey being the set of what we define as "goal locations".
- Their "proximal cues" appear to comprise the set of all locations relevant to finding prey, similar to Brooks' beacons [30].
- Their "scent" could be interpreted as the "search fringe" in our model: the investigator's mental model of the set of most likely targets to investigate next, and their prioritization.

Several works study practitioners in their work environment to uncover aspects of their processes and difficulties therein (e.g. [65] [101] [67]). Their insights gained are helpful

doi:10.6342/NTU202210012

for understanding certain aspects of the debugging process, but are not generalized to model the entire debugging process.

Other works attempt to explore and destructure the debugging process, but focus on specific aspects of tool usage, rather than debugging as a problem-solving process [93] [27] [99] [16].

Several studies have attempted to understand novice debugging behavior [86] [80] [83] [85]. Many of their results do not extrapolate to the expert level, since, as we argue in §2.3.3, novices and experts face inherently different challenges.

2.2 Interactive Debuggers

Empirical evidence shows that even in recent years, the **traditional debugger (TD)** and print debugging were still the **most pervasive** debugging tools among practitioners [27] [92]. That dominance is mind-boggling, especially when considering the many alternatives proposed and implemented by researchers. One reason might be versatility. While its competitors can easily outperform TD in some subset of all possible bugs, at the end, none of them is (and usually does not aim to be) the simple-to-use jack of all trades that TD is. While TD has many shortcomings, its simplicity, wide availability and "enormous engineering effort" [74] involved in debugger development make it difficult for any competing solution to gain traction. When considering the demand-side of the equation, we note that most programmers are not aware of the many ideas and possible solutions out there and "debugging training" is virtually non-existent [92].

One major shortcoming of TD is the fact that it can only step forward in time. **Reverse or replay debuggers** enable the coveted "step backward" button by recording a program execution in form of a small set of events and side effects necessary to eliminate sources of non-determinism [33]. When stepping backwards, a faulty program is re-run in a sandbox that simulates all system calls and events from the recordings, thereby making any execution of a program repeatable.

Algorithmic debugging has originally been proposed by Saphiro [110], and extended many times since. Feedback-based debuggers [78] claim to be a generalization of Algo-

doi:10.6342/NTU202210012

rithmic Debugging, but the generalizations already seems to have been claimed by Algorithmic Debugging advocates [115]. A generalized algorithmic debugger (or feedback-based debugger) implements a feedback loop with two steps: (i) it first applies an automatic technique that prunes the search space, that is the set of potentially buggy lines of code, and then (ii) asks the user for input to overcome some of its own shortcomings. It repeats until either search space size has been sufficiently minimized for the developer to trivially find the bug, or the automatic method fails to satisfy all its constraints. Recent examples include Enlighten [74] and Microbat [78]. Feedback-based debugging still shares the weaknesses of automatic methods, but to a lesser extent. On the other hand, this approach is more susceptible to "communication problems" where the user misinterprets the query presented to them.

Query-based debuggers were defined by Lencevicius et al. [71] in 1997 to allow developers debug their program by writing queries about runtime behavior in a formal language. The general idea also exists in other important debugger features, such as conditional breakpoints, as well as model-based debuggers.

In contrast, Dbux is an **omniscient debugger** (OD). ODs record, visualize and allow interacting with a large chunk of a program's dynamic execution behavior, i.e. debugging's "dark matter". As Pothier et al. [95] put it: "questions that would otherwise require a significant effort can be answered instantly". The term "omniscient debugger" is often used interchangeably with "back-in-time debuggers", "trace-based debuggers" and "history-based debuggers". Among the more noteworthy implementations are probably Lieberman's ZStep [77] for LISP, as well as ODB ("Omniscient DeBugger") by Lewis [73], the Whyline for Java [57] and TOD ("Trace-Oriented Debugger") [97] [95] [96], all targeting Java. Lewis credits EXDAMS [24], a TTY system from 1969, to be the first omniscient debugger. Omniscient debuggers record all (or most) of the program's execution. That log is often referred to as the "trace". Every recorded execution of an AST node is commonly referred to as an "event". The recorded data is then presented to the developer, allowing them to easily inspect values of expressions and/or variables without needing a breakpoint. They can also navigate between the recorded events to follow

doi:10.6342/NTU202210012

program execution, similar to the traditional debugger, but both, backward and forward in time, and with random access. Omniscient debuggers tend to have the strongest suite of interactive analysis tools, since, as their name suggests, they "know everything" (or more aptly: they know most of the low-level events) of the program's execution behavior. That comes at the price of reduced scalability, as well as noise and clutter, as traces can be prohibitively large.

2.3 Other Types of Debuggers

This work primarily concerns itself with a specific category of debuggers, that are (i) interactive, (ii) general-purpose code-level debugger (iii) for expert use. With that in mind, here, we briefly explore three deviations from this category that are out of scope of this document. We believe this to be helpful for properly embedding our work in the greater context of all of "debugger research".

2.3.1 Non-Interactive Debuggers

As the name implies, **automatic debuggers** attempt finding bug locations automatically. In the research, rank-based automatic debuggers seem to be the most dominant subcategory. These debuggers use algorithms that are commonly referred to as "fault localization techniques" which primarily output a list of ranked program statements based on some suspiciousness metric. This allows the developer to focus on the most suspicious statements first. Given an accurate ranking, this can significantly improve debugging time. On the other hand, inaccurate ranking can have the opposite effect, leading the developer into a wild goose chase. Additionally, while these debuggers have the ability to locate a bug, the state of the art still lacks another important quality, that is, explanatory power: users can identify statements that are (likely to be) at fault, but not WHY it is faulty. For that and several other reasons, despite its promise and great strides in the research, automatic techniques still face very low adaption rates among practitioners [60]. We refer the interested reader to Le Goues et al. [68], Pearson et al. [91] and Liang et al. [135] for literature

doi:10.6342/NTU202210012

studies of such techniques. Some strides have been made in attempts to not just find, but go so far as to fix bugs automatically [68] [69] [55].

Model-based debuggers is an application of Model-based Diagnosis (MBD) techniques to locating errors in computer programs [84]. A model-based debugger scans either the code (static) or the log of execution events (dynamic) for pre-defined models describing common problematic program patterns or behaviors (i.e. "recipes of disaster"), before eventually reporting its findings back to the developer. These types of debuggers, especially linters [123] [124], are commonly used in the industry.

2.3.2 Domain-specific Debuggers

In this work, we are only interested in general-purpose code-level debugging. In contrast, domain-specific debuggers aid developers in a smaller set of application domains. We give a quick overview over some domains where specialized debugging tools have become commonplace:

- Frontend/GUI debugging — When working on websites or GUIs, bugs often have a visual manifestation in the interface that might be more easily resolved when consulting specialized GUI inspection tools, such as Chrome Dev tools [3] or NetBeans Visual Debugger [6]. Commonly used web frontend frameworks React, Angular and Vue, each offer their own "dev tools" to inspect the high-level virtual DOM and the data flow between components [7] [2] [9]. The WhyLine [57] tells the developer which piece of code is responsible for any property in Java AWT GUIs.
- Profilers — Performance bugs are difficult to pin down, especially since they are often not just small problems, but rather the sum of all the parts of the software system. These days, profilers, first conceived by Graham [46], are able to visualize an application's performance profile and potential bottlenecks. They ship with most browsers, as well as many specialized development environments, such as Matlab [10] and Tensorboard [8].
- State management debugging — There are many types of state management debug-

doi:10.6342/NTU202210012

gers. For frontend development, entire frameworks have been developed, dedicated to data and state management in complex interactive applications. On top of these frameworks, dedicated tools have been developed for monitoring the framework's internal processes. For example, Redux Dev Tools [11] allows monitoring and manipulating (i) the global application state, as well as (ii) all write actions upon it.

- Memory debugging — Valgrind's Memcheck [12] is probably one of the more famous candidates in this category. It assists in locating a variety of memory-related problems, ranging from illegal memory access to memory leaks. While garbage-collected languages, such as JavaScript usually do (or should) not suffer from illegal memory access problems, memory leak debuggers can certainly be found [129].
- Distributed and system-level debugging — Complex systems involving many nodes of computation, ranging from multi-process to cloud-based websites to high performance computing, benefit from additional system-level debugging tools. Chronus [132] for example allows system administrators to provide a script ("software probe") that keeps regular system snapshots, while testing for whether the system is still working correctly, and if not, allows an administrator and automatic analysis programs to determine the time of failure, allowing them to compare the difference in snapshots between "working" and "failed".

2.3.3 Debuggers for Novices

Early education in computational thinking and programming has become a staple of educational systems worldwide. The UK adopted a nationwide computing curriculum in 2013, and many countries have followed suite since. Likewise, research in tools to aid novices in their debugging journey has been picking up steam [48] [75] [52] [121] [21]. However, the problems novices face are inherently different from the problems that experts face. Novices struggle with basic syntax and semantic constructs while working on small programs where fault and symptom are not very far apart. Experts, on the other hand, struggle with a large search space and a complex entanglement of different types of knowledge domains.

doi:10.6342/NTU202210012

2.4 Algorithm Visualizations

Many strides have been made in the area of **Algorithm Visualization (AV)**. Grissom et al. [47] found that student learning can be improved by algorithm visualizations, especially if visualizations allow for meaningful interactions. Shaffer et al. [109] analyzed hundreds of AVs and found that many of them were of poor quality and also skewed toward few topics. AVs are costly to develop and can only deal with one type or family of algorithms. To aid creators, general purpose AV authoring tools have been proposed. JHAVE [87] is one of the earliest AV editors, allowing AV authors to build their own visualization. JSAV [53] provides a more modern web-based solution, but, as of writing, has not seen updates in five years. When googling "algorithm visualization", the first result was VisuAlgo [50], another AV collection that seems mostly maintained through student projects, but it is not open source and does not explicitly invite outside collaboration. Clearly, the authoring process of domain-specific AVs, despite their great value to learners, suffers from silos, expensive development cost and risk of low quality.

2.5 Memory and Data Flow Visualizations

Zimmermann et al. [134] proposed one of the earliest works that address the issue of **memory visualization**. They target C programs and implement it by querying GDB for variable and pointed-to memory chunks. Heapviz [15] visualizes the Java heap, while allowing the user to interact with and also summarize it. Python Tutor [48] has become commonplace in many CS classrooms. It integrates stack and heap visualization with several debugging features, such as forward and backward stepping as well as call stacks.

Many tools have been proposed to visualize and track the flow and dependencies between the different interrelated parts of a program, but most of them focus on **control flow**; despite the fact that, 40 years ago, Weiser [131] already noted that "program state constructs" have more meaning in isolation than "flow-of-control" constructs. More recently, Ko et al. [58] posit that most Java bugs are bugs of **data flow**, rather than control flow. Yet, there still is a notable lack of tools that focus on data flow. Why? When ana-

doi:10.6342/NTU202210012

lyzing the Whyline's features for control flow with features for data flow, Ko et al. [58] found that users were more reluctant to follow data than control dependencies. The authors conjectured that the reason lies either in users' lack of familiarity with data flow tools or insufficient understanding of data flow concepts. Furthermore, we would like to add that control flow tools are easier to build than tools that capture data flow. Several debuggers have been proposed for visualizing data dependencies of reactive programs [103] [25]. However, that is a simpler problem, since explicit data dependencies are a core feature of reactive programming.

Ferrante et al. [41] proposed tools to combine data flow and control flow and their inter-dependencies into a single graph, that is the **program dependency graph (PDG)**. PDGs are commonplace, but almost exclusively confined to the field of compilers and compiler-related development. JavaPDG [112] was proposed to visualize PDGs to help researchers with dependency analysis.

2.6 Asynchronous JavaScript

Many works have explored the **Dynamic Call Graph (DCG)** which spans directed edges from call sites to executed functions and files. DCG research largely focuses on the goal of performance profiling as proposed by Graham et al. [46]. However, the effect of asynchrony, or concurrency in general, on the DCG has garnered little attention in the research. Eichinger et al. [37] used call graph mining for automatic debugging in multithreaded programs, with a focus on reducing rather than enhancing the graph, to faster compare them and narrow down likely fault sites.

More recently, **asynchronous event graphs** have been proposed to tackle asynchrony in JavaScript. Loring et al. [79] propose a formalization of relationships between asynchronous events in JavaScript in terms of `links` and `causes` where the latter observes partial overlap with our notion of "synchronization" (§6.5). Chang et al. [32] infer atomic event pairs that are similar to our CHAIN relationship. They do not support async functions, and focus on IO resources and Node.js queue priority. Sotiropoulos et al. [118] use static analysis to establish a happens-before relationship between potential callbacks

doi:10.6342/NTU202210012

to produce a "callback graph". Madsen et al. [82] propose a promise graph model for static analysis of ES6 promises. Alimadadi et al. [19] extend that promise graph with dynamic analysis. It also supports ES6 promise specs. They use it to implement a model-based debugger PromiseKeeper that scans that graph for common promise anti-patterns. They propose "synchronization edges", but only to represent edges of promises created by `Promise.all` or `Promise.race`, unlike our "inter-thread synchronization" which captures a dependency between virtual threads.

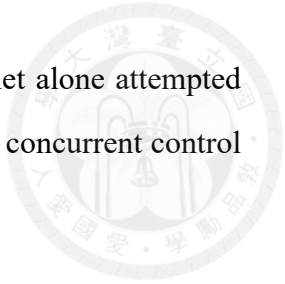
A natural application of asynchronous control flow analysis is **concurrency bug detection**. To that end, Davis et al. [36] and Wang et al. [130] present studies of concurrency bugs in Node.js. Davis et al. additionally implement a first fuzzing mechanism by modifying Node.js to help uncover these types of bugs by shuffling event queues. NodeRacer [40] claims to be more efficient than [36] in detecting race conditions across callbacks and promises in Node.js by first extracting and then imposing a happens-before constraint, thus narrowing the search space.

Only few works have the primary goal to **visualize asynchronous control flow** in JS for comprehension purposes. None of them are interactive for developers. Sahand [18] visualizes an asynchronous event graph over an entire full-stack application, straddling server-client communication. They only support callback-based asynchrony. AsyncG [120] visualizes an async event graph. The authors claim to support real-world projects, but do not demonstrate an investigation thereof. The demonstrated sample graph only seems to handle single file programs, since file names do not make an appearance. Their goals include "Automatic Bug Detection", while not mentioning developer interactivity. AwaitViz [125] visualizes a timeline of promise and await events, but does not support callbacks, and only works on toy problems.

Despite evidence showing that non-concurrent program comprehension mechanisms do "not migrate well to a concurrent equivalent" [51], little has been done to improve interactive debugging capabilities for concurrent programs. Few debuggers support the novel async/await construct. Implementations do not offer integrations with IDEs, and often not even visualization. Furthermore, no previous tool integrates the asynchronous

doi:10.6342/NTU202210012

relationships induced by all three types of JS asynchronous events, let alone attempted to capture degree of concurrency or synchronization barriers between concurrent control flows.



doi:10.6342/NTU202210012

doi:10.6342/NTU202210012

Chapter 3

Interactive Debugging Revisited – Dark Matter, A New Model and Methodology

Much of the life of a software engineer is about finding needles in haystacks. The haystacks are large codebases and the needles bugs, often hiding in a single line, hidden between thousands if not millions of other lines of code. Searching for those needles is frustrating, difficult, and also expensive. Ko et al. [57] estimate the cost of program understanding and debugging to be as high as 70% of the overall cost of software development. Yet, we know surprisingly little about the nature of this "interactive debugging" process.

Conversely, in many **other domains**, as domain knowledge is more deeply understood and processes are refined, humans tend to hone their skills and master their craft, and as a consequence, continually beat previous records. For example, since the beginning of the last century, Marathon world records have gone from almost 3h down to just over 2h[1]. Chess is another famous example of a domain where human excellence shines, in that, not only is the skill ceiling of Chess high, but all the insights into chess have been used to teach a machine to beat the best humans in it. But of course, Chess seems a much simpler problem than "interactive debugging". The real-time strategy (RTS) game StarCraft II might serve as an example in a more complex domain. In 2019, AlphaStar was the first AI to beat a professional player at their game [23]. One (if not the most) important aspect contributing to this milestone was the deciphering and subsequent encoding of the state

[1] https://en.wikipedia.org/wiki/Marathon_world_record_progression

doi:10.6342/NTU202210012

space and the human professional player's decision-making and action space at the low and high level. Our argument is not (neither is it our place) to say that dominance of AI is inevitable, or that every complex human endeavor can and should be deciphered. Instead, we simply point out that breaking down or uncovering at least some elementary building blocks of a problem domain has generally led to an improvement of skill and efficiency, sometimes culminating in advanced AIs that either beat the humans that they learned from, or get close to it.

After considering these tremendous achievements in other domains, we look back at our "interactive debugging" field, and we realize that **we have not made much progress** in decades. Sure, tools, strategies and methodology for testing, system design and many other aspects of the development process have progressed, but debugging processes and the debuggers we use seem to have barely changed in decades. What we know for sure is that, while the body of research has steadily grown, practitioners have adopted little to no novel tools [67] [81] [60] [92] [27]. Interestingly, strides made in *automatic* debugging also do not seem to translate to strides made in interactive debugging, possibly due to the fact that automatic debuggers are still vastly relying on brute-force discovery and manipulation of control and data flow, which is not feasible for human debuggers.

One important question is: why have there barely been any strides toward mastering, or even defining, the debugging process? We conjecture that a lack of a **debugging culture** might be a significant contributor to the dilemma. Software engineers generally have no inclination to discuss a codebase' property of debuggability, debugging processes or let alone skill or strategies, be it in companies or software engineering social media circles. Let's compare that to the culture surrounding RTS games, which are a much more social endeavor. Skilled players discuss, share or watch videos about strategies, behavior optimizations and important aspects of individual game mechanics all the time. Even strategy games that were built without a multiplayer component in mind, such as GeoGuessr[2], have attracted and nourished a vibrant competitive community, by simply allowing individuals or teams compete for accuracy and/or time. But in debugging, there is no celebrating of

[2] https://www.geoguessr.com/

doi:10.6342/NTU202210012

debugging skill, no "debugging competitions", and engineers don't seem to leave much thought to the honing of this particular set of skills, or relevant strategy. Unlike (most) games, software engineering (or, more generally, computer science) has dedicated educational institutions, and even there, debugging is entirely absent from the curriculum [90].

But there might be hope. Many of the challenges in RTS game AI, as outlined by Ontañón et al. [89], including learning, dealing with uncertainty, domain knowledge exploitation, task decomposition and more, very much apply to debugging (and probably most types of problem-solving domains) as well. This might hint at the possibility that, if we can decipher the RTS game problem domain, maybe we can also decipher debugging processes, skills and strategies? Discoveries by Böhme et al. [31] add credibility to that suspicion. They found a promising amount of consensus among practitioners in terms of (i) what constitutes a fault location, (ii) why a bug occurred and (iii) how it should be fixed. If there is this much consensus, maybe there are patterns and/or consensus belying the intricate steps throughout their processes as well? And if there is such consensus, maybe we can produce a cohesive definition of debugging, its elementary building blocks, and use that to start discussions on skill and strategy? This chapter is an attempt toward that goal.

3.1 The Dark Matter of Debugging

Debugging is an investigative process. The investigator of a bug, just like the investigator of a crime scene, needs access to data and information in order to track down the culprit. That sought-after information often includes the answers to questions as simple as "How often did A execute?", "From where did B get called?", "What are all the events that triggered C?" or "What was the value of D? Where did it come from?". Sadly, in practice, the cost of answering these questions can be frustratingly high. That is because the answers we seek are hiding in, what we call, the "**dark matter of debugging**", which is simply a fancy name for "**the dynamic execution behavior** of our applications". This dark matter comprises the set of all physical states of the machine(s) during the execution of the application(s), which generally cannot be directly observed or analyzed by the in-

doi:10.6342/NTU202210012

vestigator. Thus, the dark matter of debugging, just like dark matter in physics, is largely **invisible** and **intangible**.

On the plus side, and unlike dark matter in physics, we know that the computational states are real, and they CAN BE observed. The solution lies in **debuggers** and other specialized tools that are capable of making that dark matter more visible and more tangible, by recording relevant data and presenting it to the investigator. Sadly, debuggers have not notably matured over time, and aside from one particular implementation, which we call the "traditional debugger", not commonly used. This keeps investigators "in the dark".

Lacking better debuggers, investigators need to invest great amounts of time and effort to conceive of a plan and then execute it, just to answer a single debugging question. In simple cases, such as when asking "What is the value of expression X during its first execution?", the traditional debugger can be of help, and even print-debugging can generate the answer in possibly less than 30 seconds, depending on the speed of (re-)execution of the program.

But even slightly more complex questions might require a lot more effort to answer. For example, the traditional debugger does generally not keep state pertaining to arbitrary code locations. Thus, answering "How often did A execute?" or, the follow-up question "What was A's value during its 31st execution?" already goes beyond its capabilities. Ultimately, answers to all but the simplest questions require complex reverse-engineering endeavors, during which the investigator has to re-run the program multiple times, while engaging in a series of "information recovery strategies", such as manipulating input, breaking with the debugger or placing/removing print/log statements etc.

We thus conjecture that the immense cost of program comprehension and debugging can, to a large extend, be attributed to the cost of retrieving information from the dark matter. Its solution lies in (1) **better debuggers** to help investigators see and interact with the dark matter of debugging. But those tools will not help much, if the well-established lack of debugging understanding and education [90] is not also addressed. We thus posit that (2) **training/discussion/fostering of debugging skill, strategy, efficiency and culture**, as well as **program analysis** in general, are going to be paramount steps to not only help

doi:10.6342/NTU202210012

developers navigate the dark matter, but also to improve debugging tool adoption.

These deeply-rooted problems are further explored in this chapter: We proceed by introducing a new descriptive model of the debugging problem in §3.2. Then we briefly discuss the current state of interactive debuggers in §3.3. In §3.4, we propose a more rigorous approach to debugging case studies for tool evaluation, before summarizing in §3.5.

3.2 Debugging as a Search Problem

Katz and Anderson [54] posit that debugging has three components: (1) detecting of a bug, (2) locating the fault and (3) fixing it. However, like many other works, we use the term "debugging" to only refer to the second part, that is fault localization, and assume the bug to have already been detected. Furthermore, we use the term debugging as a synonym of "interactive debugging" or "manual debugging", as opposed to "automatic debugging".

Locating a bug (or "fault" or "root cause") is a problem-solving process that puts the "human debugger" in the position of an "investigator" or "detective". The investigator generally starts at the bug description and from there, takes steps, until all faulty locations in the code have been found. Ko [56] already noted that this process, and much of program comprehension in general, is a "search task". In the following we thus propose a descriptive model [111] that defines this process as a **search problem**.

3.2.1 The True Debugging State Space

We first start by defining the *true* state space, i.e. the states that exist in the real-world. Each such state is a DAG (illustrated in Fig. 3.1a):

1. The **nodes** of a true state are **locations** in **knowledge sources**, including the codebase itself. A knowledge source can consist of many locations. We assume locations to generally be hierarchical in nature. E.g. a function can be a location, while all nodes in its AST subtree could also be deemed locations. Some knowledge sources, such as people, are too complex to break down, and are thus represented as single

doi:10.6342/NTU202210012

black-boxed locations.

2. Every location has an associated **factoid set**, containing **facts**. We associate a measure of **tangibility** with facts. More tangible facts are generally easier to discover, recover and verify.

3. The edges comprise **location links**. It denotes a relationship between two individual locations that is relevant for the investigator in the context of the current debugging task. Investigators traverse a link by shifting focus from its source to its target location with the goal of uncovering more relevant facts or to move closer to the faulty locations. Each link is also a **fact**. Examples are given below.

Much of the investigator's work involves uncovering and traversing links (and uncover other facts to help find links), in an "incremental problem-solving process" [35] that hopefully, step-by-step, location-by-location, leads them to the final goal: discovery of all fault locations. Examples of true links include:

- **Causal links**[3] and closely related concepts (e.g. causal dependencies, models, reasoning) are commonly mentioned in the literature [127] [54] [116] [65] [56] [95], but lack clear definition. We consider causal links as those induced by data or control flow. Examples include the link between caller and callee (control flow), or links between reads and writes of the same variable (data flow). To better account for Katz and Anderson's "causal reasoning", links between code and output are also included in this category.

- Links between data sources and consumers, e.g. links between program input or database locations and any code referencing them.

- Links between two code locations that show up in the same full-text search.

- Links induced by domain knowledge, such as two non-adjacent functions rendering adjacent locations on screen.

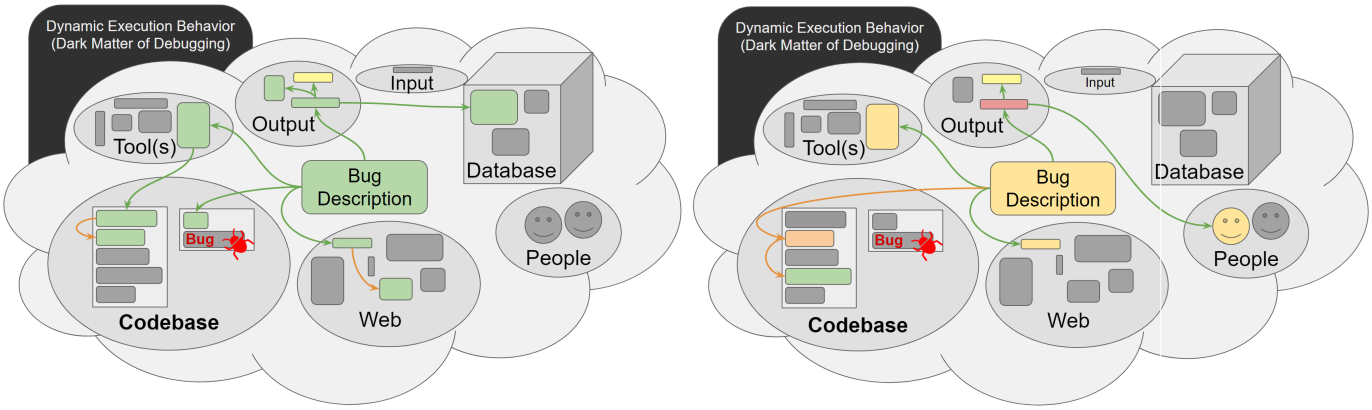
- Links between a line of code and relevant documentation or other web resource.

[3] This definition of "causal link" is different from a "causal relationship" in mathematics.

doi:10.6342/NTU202210012

- A link from some web resource to a comment in an old bug report explaining relevant logic of a third-party API etc.
- Link from a line of code to the person who wrote it or who is a suitable expert to ask about it.

3.2.2 The Perceived Debugging State Space



(a) Example of Good Comprehension (b) Example of Bad Comprehension

Figure 3.1: Two Examples of the Same True State, Perceived by Different Investigators

The interactive debugging process is complex and riddled with human error [59]. We thus differentiate between **true and perceived state space**. A perceived state is the state as perceived by the investigator at a given point in time, and only exists in their mental models. The rest of this work predominantly concerns itself with the perceived rather than the true state space.

Perceived states are mostly the same as true states, but diverge in a few crucial ways. We simplify by assuming that there is no divergence between true and perceived locations. The main divergence is in factoid sets. Perceived factoids can be true or false: we denote a **true** factoid a **fact**, while **false** factoids are **falsehoods** (which can also be thought of as misunderstandings). Perceived links are also factoids, and can thus also be true or false. The **perceived factoid set** encapsulates a measure of the investigator's **location comprehension**:

- **Good comprehension** means that a location's factoid set contains (i) many "rele-

doi:10.6342/NTU202210012

vant" facts (ii) and little to no falsehoods.

- **Bad comprehension** means the opposite. The location's factoid set comprise (i) few or no "relevant" facts, and (ii) many falsehoods.
- An **empty factoid set** means that a location is unknown.

Fig. 3.1 illustrates two examples of a snapshot in time of two different investigators investigating the same bug. The color of links and nodes represents **location comprehension**, varying from red (bad) to green (good), while gray means unknown. One investigator has better comprehension of their discovered locations than the other. Better comprehension can generally be attributed to a range of factors, including better (i) domain knowledge, (ii) familiarity with the source code, (iii) program analysis and debugging skill, but also (iv) better tools.

3.2.3 Goal Path: From Start Location to Goal Locations

The debugging process generally starts with the investigator comprehending the **problem description** or "bug description", which we also deem the **start location**. The set of **goal locations** are all locations that are faulty in terms of the given problem description.

Starting from the start location, the investigator wants to traverse any number of states in order to find a **goal state**. The goal state is any state where the set of all **goal locations** is known, while the factoid sets of all goal locations contain all facts that identify them as faulty, and no other factoid in contradiction with that. (Note that just because a faulty location is known does not mean that it has been identified as faulty.) State transitions correspond to individual iterations of the **debugging loop**, as we discuss in §3.2.6.

The last step of a successful debugging journey should ideally lead to uncovering the last of the relevant goal location facts, thereby "determining the adequacy of the emerging solution" [35]. We posit that this process should also have the side effect of producing at least one complete path of links from start to goal locations which we denote the **goal path**[4].

[4]For simplicity, we use the term "goal path". However, in case of multiple fault locations, there is not one but multiple goal paths, which can also be considered a "goal subgraph".

doi:10.6342/NTU202210012

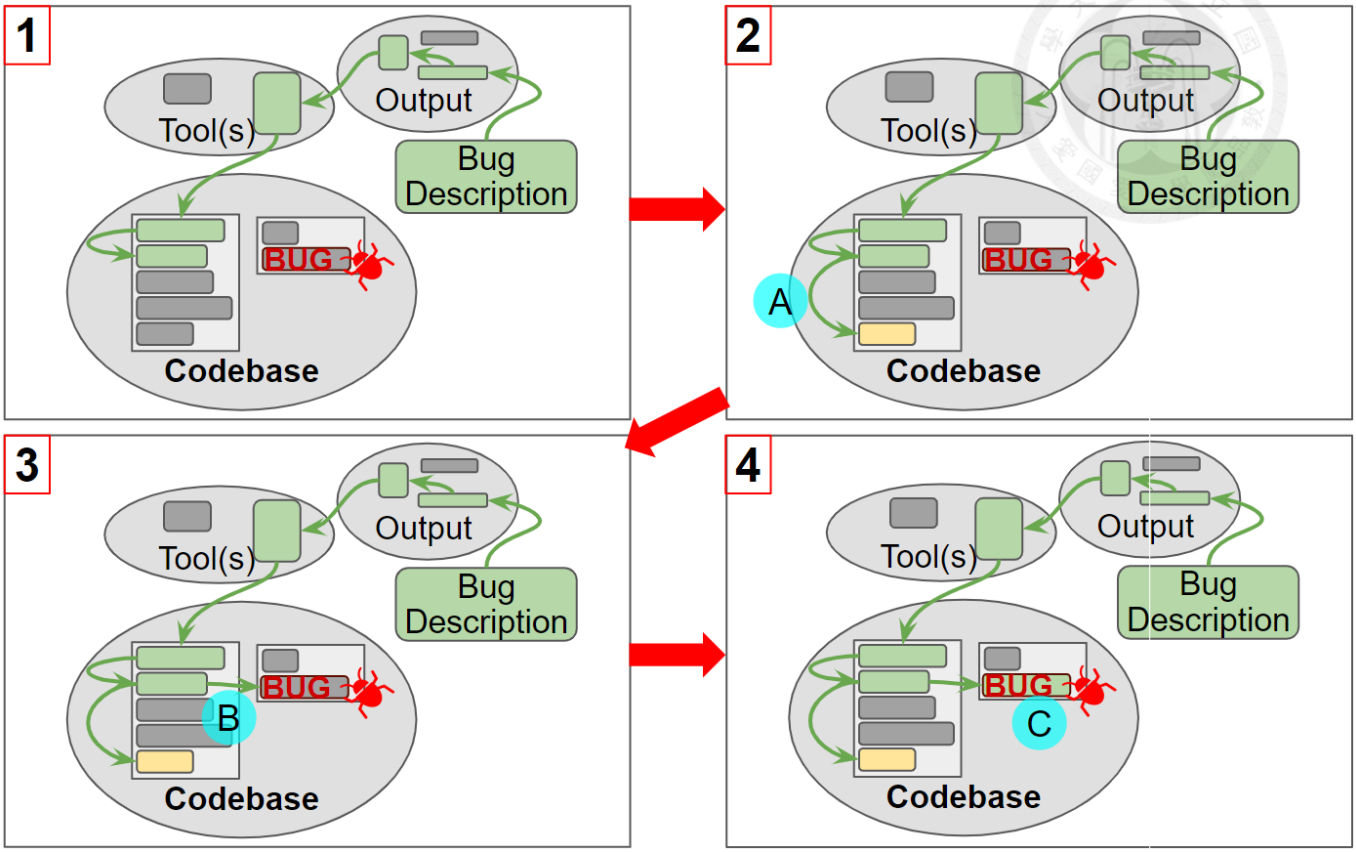


Figure 3.2: An Investigator's Final Few Steps Toward the Goal

Fig. 3.2 illustrates the final few steps of an investigator finding and verifying a bug. The example starts at an arbitrary state 1 which visually encodes the first part of the investigator's journey: From the visualization, we can guess that the investigator visited six locations by traversing five links thus far: They started by comprehending the "Bug Description". From there, they uncovered a link to some location in the output. They found something in the output that triggered them to look at the visualizations of some "Tool" (e.g. some debugger, a network monitoring tool etc.). There, they found information that leads them to some piece of code, followed by another piece of code.

From state 1, they transition to the next state 2: They find a new link A, traverse the link, and discover factoids of a new location as a result. In state 3, a new link B is discovered that takes the investigator to the fault location. In the final state 4, they uncover several more facts about the fault location C, including the fact that it is indeed faulty, thereby solving the fault localization problem.

A goal path marking this solution is visible in the final step 4. It starts at the "Bug

doi:10.6342/NTU202210012

Description", and ends in the fault location, labeled "**BUG**".

3.2.4 On Knowledge Sources

Davies [35] points out that "programming behavior can be understood only with reference to the interactions between multiple knowledge sources". We consider the **primary knowledge source** to be the codebase itself, or, if the bug is not in the code, all knowledge sources that contain the faulty locations. That is because if one had no access to any other knowledge source, a sufficiently skilled investigator can still find the target location, while the same cannot be said knowledge sources that do not contain the faults.

We also note that the **dynamic behavior of a particular execution** of the target application(s) is ***not*** a knowledge source because its information is not accessible directly. In fact this undesirable property of being not directly observable or interactable upon, earned it our nickname "the dark matter of debugging", as discussed in §3.1. Instead, the developer needs to reveal its secrets indirectly, by uncovering facts from locations in available knowledge sources, such as the code itself, program output or debuggers.

3.2.5 Debugging Actions

If debugging is a search problem, then the traversal of its state space shall be undertaken by selecting and executing actions from a set of all potentially useful actions, which we denote the **action set**.

The **main goal** of any individual action is to find and add facts to, while removing falsehoods from the perceived factoid sets. A non-exhaustive list of examples of meaningful debugging actions follows:

- **Think**: considering what is already known, trying to understand complex relationships, making a plan, building mental models, using domain knowledge to reason about a particular location, etc. This type of action is probably the most difficult to break down further or encode in a meaningful way.
- **Read**: read source code, documentation etc.

doi:10.6342/NTU202210012

- **Navigate**: move between locations, generally across location links.
- **Search**: compose and enter a query to elicit more locations, e.g. in the IDE, a search engine or database.
- **Skim and scan**: A common reading technique that allows rapidly reading multiple adjacent locations in order to extract or find important information or patterns.
- **Elicit run-time information**, i.e. exploring the "dark matter of debugging", through debugging tools, code edits, program output etc. This type of high-level action usually involves several smaller actions.
- **Ask**: e.g. a colleague, relevant experts, on online forums etc.

Actions can be hierarchical, in that some higher level actions (or "action plans") require performing multiple (lower level) actions.

Some actions can have **side effects** on the state. Examples of generally side-effect-free actions are those that are primarily cognitive, such as thinking, reading, skimming and scanning. Certain physical actions, such as a small code edit, on the other hand, can potentially cause an arbitrary amount of changes during a single state transition. Agans [17] thus recommends to only change "one thing at a time", and to "Use a Rifle, Not a Shotgun" when debugging. Our interpretation of this is: pick and execute actions surgically, rather than serendipitously, and be mindful of their side effects on the debugging state. Following Agans' recommendation assures greater stability of the state between transitions, thereby reducing the chance of previously accumulated facts turning into falsehoods.

3.2.6 The Debugging Loop

The debugging process is an "incremental process" [35]. Our model proposes that the investigator attempts to repeatedly execute actions in order to transition from the start to the goal state, primarily by uncovering and then making use of crucial facts at relevant locations.

Uncovering facts involves asking and answering questions [100] [114] [56] [113] [63] [43], as well as other forms of **hypothesis generation and verification** [30] (or, infor-

doi:10.6342/NTU202210012

mally, exercising "trial and error" [77]). Hypothesis verification adds perceived factoids, while refutation removes them. Perceived factoids can generally be interpreted as hypotheses with a high measure of confidence. Once a hypothesis has been generated, its confidence can be increased through hypothesis verification: a perceived fact is a verified correct hypothesis, while a falsehood is a verified incorrect hypothesis.

An individual iteration of the **debugging loop** causes a **state transition**. We have already illustrated an example of multiple iterations of the debugging loop in §3.2.3 (depicted in Fig. 3.2). Any such iteration generally involves the following steps:

1. **Decide on a known location**, or set of locations `L`.
2. **Generate hypotheses** (e.g. by iterating answers to a question, drawing conclusions, "having a hunch", or through other means) `h` about `L`.
3. **Decide** on an action `a` that can help verify or refute `h`.
4. **Execute** `a`, thereby creating/changing/removing factoids, which leads to an updated factoid set `f` for `L`.
5. **Update `L`'s perceived factoid set** by setting it to `f`.

In real-world debugging loops, not all steps are taken all the time. E.g., when following a linear path, a location does not need to be decided upon. Instead, the next location is picked trivially: the investigator enters (or "decides on", or "focuses on") the first location, discovers a link, follows link, enters new location, and then proceeds from there. Sometimes the debugging loop might be **recursive**. For example, in order to generate or refine one hypothesis `h`, an investigator might choose to execute several iterations of the debugging loop that uncovers more factoids contributing to `h`. Likewise, we have pointed out that one action might actually be the execution of multiple actions. As we will discuss next, even uncovering a single fact can require multiple iterations of the debugging loop.

3.2.7 Fact Tangibility

To better reason about facts and their role in the debugging process, we introduce the informal concept of **fact tangibility** as a measure of **fact difficulty**:

doi:10.6342/NTU202210012

> Fact tangibility encodes the inverse of the amount of cognitive and physical effort required by the investigator to uncover a fact.

High tangibility thus implies that a fact can be easily uncovered, while low tangibility implies that it is difficult to obtain.

Facts of **high tangibility** can be **easily** revealed, e.g. because code structure, the IDE or tools at hand allow uncovering them through a single action. Examples include (i) the target of a `write` operation (i.e. the LHS of the operation in many types of languages), (ii) links between a code location that reads or writes a variable and the location where it is defined, as well as (iii) links between variables of the same name, which can be revealed through the commonly available (and generally easy-to-use) full-text search feature of IDEs.

On the other hand, less tangible facts are **difficult** to find because uncovering them requires a lot of effort. For example, the link between one code location writing an object, and another in a far-away file (large physical distance), executing a lot later (large temporal distance), that reads the same object's clone (only indirect data flow) might be mostly **intangible**. Many of the most intangible facts are "hidden" in the *dark matter of debugging*[§3.1]. In absence of novel debugging tools, these types of facts might require a lot of planning, as well as sweat and tears, to uncover.

Uncovering a single fact pertaining to some **target location** requires one or multiple steps of the debugging loop (§3.2.6) on its own. We define the **path-to-fact** as any path spanning from the target location to the location that finally helps uncover the fact. The locations on the path-to-fact are **fact locations**. Contributors to tangibility thus include (but are not limited to) the following:

- The path-to-fact only spans over a **small amount of locations**.
- All fact locations are **clearly visible**, e.g. a button that does not require scrolling on screen or other types of physical movement to be viewed.
- All actions required to uncover the fact can be **easily performed**, e.g. clicking a button (in contrast to asking a colleague who is not available).

doi:10.6342/NTU202210012

- All actions have a **low cognitive barrier** to entry, e.g. entering an obvious search-term (in contrast to searching for a complex regex).
- All fact locations are not obstructed by **(visual) clutter**, e.g. picking from one of three search results (in contrast to a search result that needs to be found in a list of over a hundred matches).

As an example, a fact of type "first value of an expression or symbol name" usually is highly tangible, even if one uses the traditional debugger. In this case, the investigator needs to perform three simple actions, all involving a small set of clearly visible and physically close locations: (1) click to set a breakpoint, (2) run the program with the traditional debugger enabled and (3) read the value from the debugger UI.

On the other hand, facts regarding "a particular –rather than the first– execution of a statement, upholding condition `C`" can be arbitrarily more difficult to obtain when using only traditional tools, because these tools can only encode very simple conditions. If `C` cannot be encoded by the tool or the code itself, then that type of fact, might become highly intangible, i.e. hard to obtain, when using that set of tools. As an example, consider facts where `C` encodes reachability [62] between the given and some set of far-away target locations. When using traditional tools, these types of facts are very likely intangible. In contrast, if one was to use LaToza's Reacher [64], they could become very tangible.

3.2.8 Using Goal Paths to Measure Bug Difficulty

Since the goal of the debugging process is to uncover a goal path, it stands to reason that its discovery is strongly related to debugging difficulty. Breaking down the path into its links and then applying the equivalence between fact tangibility and difficulty just established, gives rise to a new conjecture:

> Fault localization of a bug is difficult if all its goal paths are long and have multiple links of low tangibility. Conversely, it is easy, if there exists at least one short goal path containing only high tangibility links.

doi:10.6342/NTU202210012

This result of our argumentation has support in the literature: Eisenstadt [38] found and Perscheid et al. [92] later verified that the main contributor to bug difficulty is the distance/chasm between symptom and root cause. However, they failed to define this terminology. We believe that relevant distance metrics include the following:

1. Temporal distance (also mentioned by [38] [92])
2. Distance in files and folders
3. Distance in trace
4. Call graph depth
5. Goal path distance

We believe (albeit without definitive proof) that our new difficulty measure based on goal path length and tangibility could be a more accurate indicator of bug difficulty than just "distance" or "chasm size". However, our measure does not account for difficulty in bug detection, as is the main contributor to difficulty in Heisenbugs or other types of bugs that cannot be reliably reproduced. It is worth noting that in all these definitions, including ours as well as Eisenstadt's and Perscheid's, the bug needs to have been found before its difficulty can be measured.

While this conjecture allows for an intuitive measure of fault localization difficulty, it inherently depends on the tools being used, since they can have a strong effect on tangibility (see §3.2.7 for examples). In order to allow for comparison of goal path difficulty, this variability would have to be eliminated, e.g. by normalizing the set of tools that can be used. The conjecture could then be tested by correlating our measure of goal path difficulty with well-established measures of debugging efficiency, i.e. (i) accuracy and (ii) speed.

3.3 Interactive Debuggers

We proceed by briefly contextualizing the role of interactive debuggers and their architecture, as it pertains to the interactive debugging process. We note that the term "interactive

doi:10.6342/NTU202210012

debugger" is a bit of a misnomer, since it does not do any debugging on its own. Instead, it is generally used to describe a tool that aids the software engineer in prodding and analyzing dynamic program behavior. As it turns out, debugging is not even the exclusive goal of using a debugger. Maalej et al. [81] found that a majority of their study's participants used debuggers for general program comprehension (not just fault localization) tasks. We thus suggest that "dynamic analysis tool" or "comprehension support tools" might be a more accurate label; but since that does not roll off the tongue all too smoothly, we decided to stick with the conventional nomenclature. We shall use the terms "debugger" and "interactive debugger" interchangeably.

Somewhat confusingly, and despite a vast plethora of research on new debugging tools, the ambiguous phrase "the debugger" is used by practitioners and researchers alike (e.g. [81] [70] [98] [16]) *NOT* to refer to any type of debugger, but specifically to what we refer to as **"the traditional debugger" (TD)**. That is the debugger that exists, in one variation or another, for every modern programming language, and made it into most modern development environments. We abbreviate a common feature set among TD implementations as **BVCF**: **B**reakpoints can be placed on individual lines or statements to stop the program right before execution reaches the breakpoint location. Once stopped, nearby **V**alues and the **C**all stack can be inspected. **F**orward stepping allows executing one statement at a time. We claim that this feature set has barely evolved since its inception which Lewis [73] credits to "DDT" in 1961, over 60 years ago. Lieberman [76] likewise claimed 25 years ago that debugging did not evolve in the 30 years prior.

doi:10.6342/NTU202210012

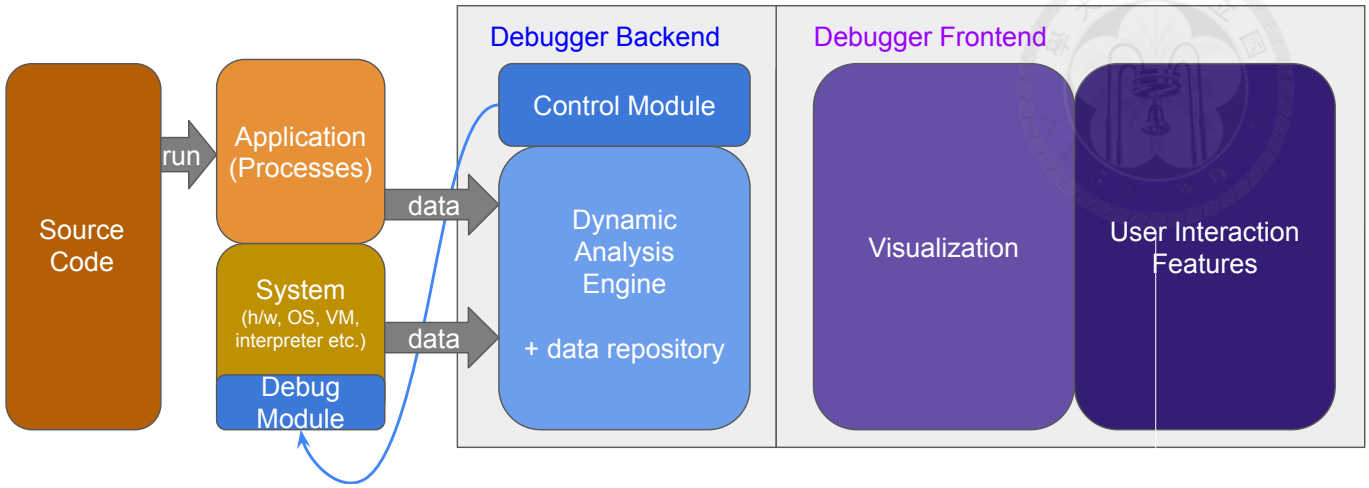


Figure 3.3: Interactive Debugger Architecture

We suggest that the **primary design goal** of interactive debuggers should be to improve fact tangibility of some categories of facts pertaining to some categories of locations. According to our interpretation, Lieberman [76] calls on researchers to find better ways of increasing tangibility of "dark matter facts" by alleviating the "cognitive task of **relating the static description embodied in the code to the dynamic behavior** of the program" and "use [better and faster] graphics to help the programmer visualize the behavior of the program". Eisenstadt [38], likewise, seems to express a wish for an increase in tangibility when describing how "**computable relations should be computed**" and "**displayable states should be displayed**" "on request rather than having to be deduced by the user".

Fig. 3.3 illustrates the general architecture of interactive debuggers: Debuggers usually employ dynamic (and sometimes static) analysis to help developers uncover facts. They (i) record execution data, (ii) provide (ideally powerful) queries upon that execution data through the dynamic analysis engine, in order to (iii) visualize important facts and (iv) make them interactive.

Several crucial properties of debuggers need to be fine-tuned to enhance said tangibility, including, but not limited to: ease-of-use, the expressiveness of visualizations, the power of interactions, the queries they support, and performance.

Interactive debugger research often focuses on making tangible, not just any type of facts, but **causal links** in particular, i.e. links between code locations that are induced by data or control flow; a task that the traditional debugger is notoriously bad at.

doi:10.6342/NTU202210012

3.4 Debugging Journeys

Having defined important properties of interactive debugging processes and debuggers, we now want to take a critical look at their evaluation.

The "debugging journey case study" format is a common choice of evaluation methodology in debugging research. However, we argue that it can easily fail to fulfill reader expectation, due to a lack of clearly defined goals, structure and process. In fact, we were unable to find a general definition of this type of case study, despite being commonly used. Just like software engineering case studies in general, they are often considered inferior to other evaluation methodology. Beck et al. [26] go so far as to denoting any form of evaluation "not necessarily involving users" as "rather lightweight".

In their 2017 study, Böhme et al. [31] discovered that there is consensus among practitioners when it comes to debugging results, specifically, bug locations, causes and bug fix approach. Even though the authors collected data and reported on (i) tools used, as well as (ii) "debugging strategies" of the study participants, comparison of the details of the adopted processes were absent. They seemed to lack a methodology that would allow comparing the debugging approach of one individual with one another. If they had had such methodology, we envision, they could have uncovered potential for consensus regarding the debugging process in and of itself, rather than only its outcomes.

In this section, we thus propose an important type of methodology in the interactive debugging field, with the goal of, ultimately, properly defining and allowing for communicating and, even comparing, individual instances of the debugging process.

3.4.1 Classifying Debugging Journeys

We define a debugging journey as a (written) account of an "investigator" engaging in a debugging or program comprehension task, with the primary goal of demonstrating a tool's novelty and prowess. The investigator is usually a core architect of the tool to be investigated, and thus has intricate knowledge of the tool's capabilities. We also assume that the investigator has expert-level program analysis and debugging skills, pertaining to the types of programs and bugs that the tool is aimed to help debug.

doi:10.6342/NTU202210012

Debugging journeys are the result of a debugging case study. Case studies are considered to be uncontrolled and involving an independent subject [102]. In debugging journeys, the program can be considered the independent subject, and its buggy execution uncontrolled. Wohlin et al. [133] goes so far as to suggest that, in software engineering research, the label "case study", in many instances, is inaccurate, and should be replaced with "small-scale evaluation".

However, we agree with the terminology chosen by many authors, and consider it a type of case study. It is worth noting that Flyvbjerg's "Five Misunderstandings About Case-Study Research" [42] can also all be applied to debugging journeys.

3.4.2 Problems with Debugging Journeys

We identify three major problems with debugging journeys:

Pbl. 1: The fields of "interactive debugging" or "interactive debuggers" are not well-defined, making it sometimes difficult to understand the **goals** of the debugging journey to begin with.

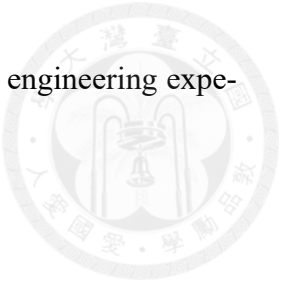
Pbl. 2: Aside from goals, debugging journeys are lacking **structural guidelines**. This unnecessarily adds to the already high difficulty involved in comprehending real-world debugging journeys. Comprehension can be challenging even for skilled software engineers (let alone for researchers who rarely, if ever, write code), as it requires following an investigator's (possibly long) chain of reasoning. It is further hampered by the reader's lack of prior knowledge of (i) the codebase, (ii) the newly introduced tool, or even (iii) advanced language semantics. This can leave readers at a disadvantage.

Pbl. 3: We find that debugging journeys often do not follow **due process**. This is not too surprising, considering that Shaw [111] 20 years ago put forward that, unlike many other sciences, software engineering research lacks "guidance" and that "software engineering researchers rarely write explicitly about their paradigms of research and their standards for judging quality of results." We find this to still hold true today, at least as it pertains to interactive debugging case studies.

In the following, we aim to address each of these problems, based on our proposed

doi:10.6342/NTU202210012

model of debugging (§3.2), as well as 20 years of individual software engineering experience.



3.4.3 Pbl. 1: Debugging Journey Goals

We define three types of goals:

G1: In interactive debugger research, the goal of case studies usually is to highlight how the given tool makes solving specific subclasses of debugging problems easier. As we posit in §3.3, debuggers achieve that goal by improving fact tangibility. We thus recommend that the debugging journey should always **show how the tool makes certain types of facts more tangible**.

G2: The locations and links traversed during the debugging journey should form a **goal path**, so as to assure that the described debugging process is complete and self-contained. This is also a pre-cursor for **G3**.

G3: A debugging journey should be **repeatable**. Slezák defines repeatability as "the closeness of the agreement between independent results obtained with the same method on the identical [material], under the same conditions" [117]. If tool, code and system needed to execute the journey are publicly available, any reader should, ideally, be able to run the program, and record (or use a previous recording of) the same bug. Barring potential complications arising from non-determinism, they should then be able to follow the debugging journey step-by-step, and get to the same results. We would like to note that we have not found any discussion of repeatability of the debugging processes in the literature. However, the consensus in debugging outcomes found by Böhme et al. [31] makes us hopeful that independent software engineers might also find consensus in goal path traversal, hence allowing for this kind of repeatability. One of the most beautiful things to come from repeatability is of course the potential for intellectual discussion ensuing, if a (sufficiently skilled) reader has questions regarding or even disagrees with a particular step of the proposed goal path.

doi:10.6342/NTU202210012

3.4.4 Pbl. 2: Debugging Journey Structural Guidelines

To address the lack of structure and process in debugging journeys, we propose an explicit set of guidelines and recommendations. We propose five structural guidelines:

S1: Goal **G2** is to provide a complete goal path. As described in §3.2.6, this implies **starting** at the **problem description**, and **ending** with the verification (or at least discovery) of the **fault location(s)**. Upon comprehending the problem description, some crucial early step would take the investigator to start using the tool. We deem it essential for the reader to understand this very step, and thus recommend making explicit the investigator's decision and rationale for using the tool, in the given context.

S2: In case of bugs, the **problem description** should adhere to software engineering best practices, and thus at least contain (i) expected outcome, (ii) actual outcome (standing in clear contrast to the expected outcome), (iii) relevant program input and output, as well as (iv) relevant steps to reproduce. Reproduction can often be achieved by providing a code sample or the context of execution of the target application or system. If a third-party description is used, and it does not contain this information, it might be useful for the investigator to make explicit, how they were able to reproduce the bug, as well as their interpretation of the problem statement, due to its crucial role in the debugging process [45].

S3: In the interest of brevity, we recommend only detailing the **shortest goal path**. A debugging journey, ideally, is only a **simplified approximation** of a real-world debugging scenario, designed to satisfy the goals laid out in §3.4.3. As any seasoned practitioner can certainly attest, the complete debugging process can contain many missteps and dead-ends, which are not necessarily interesting or relevant to those goals. If not omitted, steps that are not part of the shortest goal path should at least be clearly labeled as such, so as to allow the reader to stay focused. At the same time, we caution authors to be mindful of research ethics. The "shortest path" constraint is not supposed to be an excuse to carelessly discard negative results. For example, if facts were discovered that were not as tangible as expected (or even claimed), a discussion that explores the reasons behind the intangibility might be warranted.

doi:10.6342/NTU202210012

S4: **Clear steps**. We recommend **making explicit all important decision-making processes, and the knowledge and information they are based on**. To better facilitate understanding of this highly complex process, we recommend making the entire debugging loop explicit by answering the following questions for each step (if answer is not trivial):

1. If not following a linear path, explain which **known location** is picked and why?
2. What are the **relevant known facts** of that location?
3. What **questions** are asked? What **conclusions or hypotheses** are drawn? When acting on a **hunch**, destructuring should be attempted.
4. What **actions** are taken as a result?
5. What is learned (i.e. how is the **factoid set modified**) as a result of all of the above?

S5: For the same reason as **S4**, **important locations should (ideally) be visualized** (if they are visualizable) and shown in context. It is difficult to reason about unknown code. It is even more difficult to reason about code that one does not see. The same applies to non-code locations. For example, in case of goal-path locations in the tool, visualizations can help convince the reader that those locations are as relevant and as tangible as advertised, without having to trust the word of the author. If an important debugging step or location is only vaguely described, or worse, omitted, rather than shown explicitly and concretely, study repeatability suffers.

3.4.5 Pbl. 3: Debugging Journey Process Guidelines

We further provide a list of **process guidelines** to help enhance readability, and also uphold "criteria", "key characteristics" and suggested structure of a "plan" for case studies, as outlined by Runeson et al. [102]:

P1: A debugging journey should be written with a suitable **target audience** in mind. Any sufficiently interested and skilled reader should be given a fair chance to comprehend or even repeat it. We thus recommend writing debugging journeys for readers who

doi:10.6342/NTU202210012

are software engineers at the (sufficiently high end of the) grad student skill level, with sufficient domain-level knowledge (of the domains that are involved in the goal path).

P2: **Selection strategy**. Like in other types of tool case studies, it should be explained how a bug or problem was selected, and why it is (likely to be) relevant to the tool's analysis.

P3: **Data Collection** should commence in a "planned and consistent manner". For simplicity, we recommend the investigator first solve the bug, while being mindful of the main steps taken and decisions cast. Afterwards, in a second or third re-iteration of the process, identify and simulate only the shortest goal path. It might be easier to defer recording of minor details of the decision-making processes, results thereof, and visualizations of locations, until after the shortest goal path has been established. Having a complete video of the first and complete attempt is useful to help remind the investigator of crucial problems they had to solve in the process.

P4: **Threats to validity** should be "addressed in a systematic way". Runeson et al. [102] recommends being mindful of threats to four types of validity: (i) construct validity, (ii) internal validity, (iii) external validity and (iv) reliability. We note that (iv) reliability has generally been attained once Goal **G3** (*repeatability*) has been verifiably achieved. This can easily be implemented by, for example, inviting others to repeat the results.

P5: The debugging journey should be sufficiently **complete**. The reader should *not* be required to read anything but the debugging journey itself (and the work it is embedded in) to evaluate its goals. Prior knowledge of the codebase, or intricate details of the tool should not be required. Actively reducing assumptions on prior knowledge can help make the case study more convincing by lowering the cognitive barrier to entry. To comply with both, requirements of brevity and completeness, in some cases, the authors might want to provide the complete debugging journey as appendix or external artifact, while only providing a summary in the text.

P6: Runeson et al. [102] also recommends **triangulation**. Of the different triangulation methods, data triangulation can usually be achieved relatively easily, by collecting

doi:10.6342/NTU202210012

data from multiple sources, e.g. collecting multiple debugging journeys, each of different bugs, and, ideally, of different codebases.

In our own experience, these guidelines serve well as checklists for debugging-related case studies. Moreover, we want to note that we found it useful to destructure a debugging journey into multiple parts and subsections, often including: (i) **bug description** and some basic explanation of the target application, (ii) one or more **preliminary analyses** that help the reader understand relevant parts of the target codebase and important aspects of its behavior, without cluttering the journey itself, (iii) the brief, concise **debugging journey** only detailing the shortest goal path, with no side-tracking. The exploratory nature of this type of case study often gives rise to (iv) **further revelations** that go beyond the debugging journey itself, which can be touched on in follow-up analysis (sub-)sections.

3.4.6 A Checklist for Debugging Journeys

For the reader's convenience, Tbl. 3.1 summarizes all goals, as well as structural and process guidelines in a single checklist.

Table 3.1: Debugging Journey Checklist

(a)

G1	Fact Tangibility
G2	Goal Path
G3	Repeatable
S1	Start & End
S2	Problem Description
S3	Shortest Path
S4	Clear Steps

(b)

S5	Locations Visualized
P1	Target Audience
P2	Selection Strategy
P3	Data Collection
P4	Threats to Validity
P5	Completeness
P6	Triangulation

3.4.7 Evaluating Debugging Journeys

In this section, we have identified three major problems in debugging journey case studies, and provided a checklist of guidelines to address these problems. In a final step, we want

doi:10.6342/NTU202210012

to test this checklist on several existing case studies that, at least partially, match our expectation of "debugging journeys". To that end, the Debugging Journey checklist is applied to case studies of Reverb [88] and Extravis [34].

But before that, we want to express our admiration of both these works. Our goal is not to judge scientific integrity, their contributions or the overall quality of the work, and neither is it our place. Rather, this study should solely serve to help understand our definition of a "debugging journey", what expectations we propose it should have, how these expectations were met, and, in part, how we would apply our framework to meet all our expectations. We do not claim our definitions or the checklist to be perfect by any means. On the contrary, we hope to use this opportunity to invite others to provide feedback, so as to strive for more robust and integral debugging case studies in the future. That is also one of the reasons as to why we decided not to evaluate the process items (**P1**~**P6**) of the checklist.

Evaluation results are summarized in Tbl. 3.2. A checkmark (✓) indicates that the goal or recommendation has been fulfilled, while a half checkmark (✓̸) indicates partial fulfillment. An item-by-item explanation of that interpretation follows, first of Reverb and then of Extravis.

Table 3.2: Debugging Journey Checklist Evaluation (R: Reverb, E: Extravis)

	Item	R	E
G1	Fact Tangibility	✓	✓
G2	Goal Path	✓̸	✓
G3[5]	Repeatable	✓̸	✓
S1	Start & End	✓	✓
S2	Problem Description	✓	✓
S3	Shortest Path	✓̸	✓
S4	Clear Steps	✓̸	✓
S5	Locations Visualized	✓	✓

doi:10.6342/NTU202210012

Evaluating Debugging Journey#1

Reverb [88] is a wide-area debugger from 2019 that captures and visualizes data-flow across multiple network endpoints and even through database operations. The following is an application of our checklist's goals and structural guidelines on the "Bug Diagnosis Case Study" in their §5.3. A breakdown of the checklist items follows:

(G1) The case study helps understand how **fact tangibility** of value provenance and wide-area causality have been increased. One concern we would like to express though is that data flow visualizations can clutter easily, which leads to decreased tangibility. Explicitly addressing issues of clutter could be advantageous, considering the authors have artificially reduced clutter by cropping the visualization and removing "more obvious data flows". This does *not* mean that Reverb is unable to help deal with clutter. In fact, their §2.3 touts Reverb's ability to utilize program slicing.

(G2) While many important steps of the debugging process have been made explicit, some essential steps of the **goal path** are missing in the investigative account that describes how the investigator uncovers the fault location (explained below).

(G3) Reverb does not seem to be available for public use. It is also not explained how the visualization was generated. Independent of that, incompleteness of the goal path likely makes the debugging journey only partially **repeatable**.

(S1) The **start location** (i.e. the bug description), as well as the **end location**, (i.e. the faulty location ③ `sheet.clear_range;` on L509 of `socialcalc3.js`), are both clear.

(S2) The **problem description** is clear.

(S3) The actual goal path is not complete, but the discussed steps (to the best of our interpretation) are all crucial to the investigation, thereby partially achieving the **shortest path** guideline.

(S4) We find that many important locations, as well as the investigator's interpretation of their role, are presented clearly. However, some **steps of the debugging loop** are missing, and other steps are out of order. As mentioned above, steps that deal with clutter were omitted. Moreover, the locations and their facts are explained in order of execution,

doi:10.6342/NTU202210012

but, as per the "goal path" goal **(G2)**, the steps should be provided in order of discovery. In the given order, many important questions about the debugging process with this tool remain unanswered, including: Did the investigator, in fact, uncover the locations in linear order from ① to ⑨? Or did they employ backward reasoning and first landed at the symptomatic location ⑨? If so, how did they find it in the visualization? If not, which location was uncovered first, and how? What was the next step after having discovered the first location etc.?

(S5) Lines of code are shown in the relevant data flow nodes, thereby reducing the necessity of explicitly providing code. The **visualizations** depicted in the screenshot are readable and show many relevant nodes and edges of the data flow graph across three applications. We find, it would have been useful to see a complete and zoomed-out version of the graph (without any removals), and/or the initial view, in order to better understand the visualization's ability to deal with clutter, and the investigator's interactions with it.

Evaluating Debugging Journey#2

Extravis [34] is an interactive debugging tool from 2008 with over 100 citations (as of 2022). It is one of the earlier works to capture and visualize high-level patterns in inherently large and noisy call graph recordings. It does not feature a debugging journey, but in §6, a closely-related "feature location" case study is performed. A breakdown of the checklist items on that study follows.

(G1) Extravis specializes in visualizing and thus increasing the **fact tangibility** of call graph patterns. Successful interpretation of such pattern visualizations require an investigator to have sufficient skill, strategy and/or practice. While that is an inhibitor to overall tangibility improvement, that improvement is noticeable regardless, and, we believe, sufficiently illustrated in the case study. Patterns and their discovery are explained. The role of zoom and filter features in dealing with clutter and noise are also addressed.

(G2) This case study has a total of six goal locations (1 creation of a new drawing + 5 figures drawn). A **goal path** to all six has been established.

(G3) Since the provided download link does no longer work, and we were not able to

doi:10.6342/NTU202210012

find another copy of Extravis, we were not able to actually repeat the case study. However, we estimate that it is very likely **repeatable**, since the goal path is sufficiently complete.

(S1) The descriptions of the six target features mark the explicit **start locations**. The location of the symbol names of all target features mark the **end locations**.

(S2) The **problem description** is clear.

(S3) No unnecessary steps seem to have been taken, therefore the case study seems to demonstrate a **shortest path**.

(S4) The **steps seem clear**. However, it is not explained how Fig. 3 reveals a call toward `contrib.TriangleFigure` (we are guessing through tooltips?). Another small issue we found was with Fig. 9: Explanations in §3.4 indicate that zooming should cull the call graph in both views, but it is not clear why after zooming in on the last two figures, the calls to the first three figures are still visible in Fig. 9.

(S5) Visualizations of most steps and their locations are available. Some missing visualizations include: (i) trace filtering mentioned in §6.3 and (ii) the role that tooltips played in identifying call sites.

This concludes our tests of the debugging journey checklist on two related case studies in the literature. As per our interpretation of debugging journeys, we find that the main difference between the two case studies is in completeness of the goal path. While the Reverb study only provides a single view of their tool, the Extravis study provides multiple views, and explains relevant locations in each view, the debugging steps taken throughout each view, as well as the reasoning that takes the investigator from one to the next.

3.5 Summary

In this chapter, we propose a descriptive model of the debugging process based on knowledge sources, locations, facts, fact tangibility, location links, falsehoods, actions. We propose the debugging loop as the all-encompassing decision-making process that drives it all. We (i) define the goal path as a unique type of artifact that is the result of solving a debugging task, (ii) explain debugging difficulty in terms of goal path and fact tangibility, and (iii) explain the role of any type of interactive debugger, not just the traditional

doi:10.6342/NTU202210012

debugger, in increasing said tangibility. We explicitly discuss how the generally least tangible types of facts are hiding in the "dark matter of debugging". Despite currently only being in a high-level proposal state, we believe, this model can already help better explain debugging processes and debuggers. In the future, we hope to be able to add debugging strategies, skill and scaffolding to that list. For example, we see one potential future application of the model in using goal paths to aid signposting when developing learner material.

We further used the model to better inform debugging case study methodology in form of "Debugging Journeys". First tests make us feel confident that our definition of "Debugging Journey" and its checklist can prove valuable in assessing case study completeness. We even find that the checklist improves our ability to read existing case studies, as it aids compartmentalizing the individual aspects of the study, such as the types of facts that are increased in tangibility, the type of locations that the tool helps with, as well as the linear traversal of the goal path to comprehend the debugging solution.

doi:10.6342/NTU202210012

doi:10.6342/NTU202210012

Chapter 4

Dbux: An Omniscient Debugger and Integrated Debugging Environment for JavaScript

4.1 System Overview

As of 9/2022, Dbux consists of 64.7 kloc. Its three stages are implemented in four core and four supplementary modules, as depicted in Fig. 4.1. Three shared library modules (`dbux/common`, `dbux-node-common` and `dbux-graph-common`) are not shown.

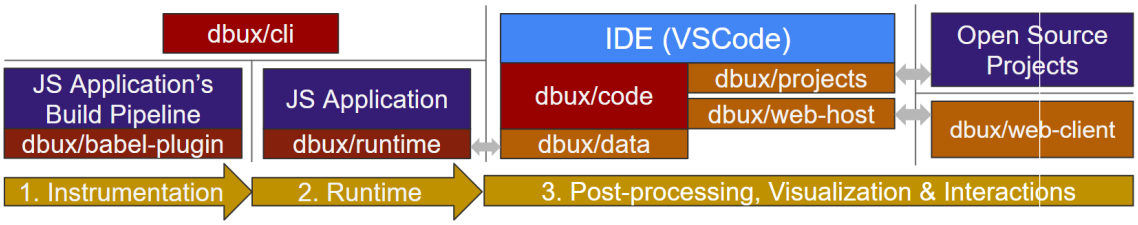


Figure 4.1: The Main Modules of Dbux

The responsibilities of the four core components of Dbux are as follows:

- `dbux/babel-plugin` instruments the target application and injects the `dbux/runtime` to enable data collection. As the name suggests, it requires to be added as a plugin to a Babel [13] build.

doi:10.6342/NTU202210012

- `dbux/runtime`, when injected into a target application, records the application's execution trace and sends it to a server in real time.
- `dbux/code` is the one-click-installable Dbux VSCode extension, available on the VSCode Marketplace [1]. Upon activation, it starts the "Runtime Server" which waits for the execution data produced by `dbux/runtime`. When received, it *post-processes* the data and provides low- and high-level program analysis queries through the `dbux/data` module. Data is processed and presented as soon as it is received, meaning that applications can be debugged while they are still running.
- `dbux/cli` is to Dbux, what the NYC cli tool is to the coverage reporter Istanbul [1], that is: a convenient command line tool that makes it easier for developers to execute a JS application with Dbux enabled, without having to prepare a build pipeline. Instead, it uses `@babel/register` [2] to inject `dbux/babel-plugin` on the fly.

The `dbux/code` extension makes use of three more modules:

- `dbux/projects` provides the *Dbux Projects and Practice feature* (§4.6).
- `dbux/web-host` and `dbux/web-client` provide "Dbux's Webview component system": a custom light-weight RPC component layer for rendering and controlling VSCode Webviews. They are used for the Call Graph (§4.5), Dbux-PDG (§5) and Dbux-ACG (§6).

4.2 Dbux Data Collection

In the following, we briefly touch on several primary types of data collected by Dbux.

4.2.1 Call Graph Data Collection

Modern JS engines are driven by a single-threaded **event loop** dispatching events from multiple queues. These queued events include all user-requested JS events, such as the program's entry point, event handler callbacks and `async` function stacks interrupted by

[1] `https://istanbul.js.org/`
[2] `https://babeljs.io/docs/en/babel-register`

doi:10.6342/NTU202210012

`await`. JavaScript is non-preemptive and single-threaded, thereby guaranteeing that once dequeued, a script keeps executing until the call stack has no more user code on it, or it cedes control with a root-level `await`, before another event is dequeued. This means that non-interruptable functions are always ensured to execute to completion, without any interruption. We use that property to capture the entire call graph on a single timeline.

We employ a **shadow stack** to track all function executions with an initial `push` and a concluding `pop`, which are respectively injected at the beginning and end of every function block. The call graph is then built according to the following rule set:

1. We refer to a function, file, script tag as a "static execution context", or short: **static context**.

2. Given a static context `f`, we denote its i'th execution f_i. f_i is an "execution context", or short: **context**. Each context starts when it is pushed onto and ends when popped from the stack. When i does not matter, we refer to `f`'s context as f.

3. If in context f_i, some function g is called, then, for some j, g_j is a child —or **callee**— of f_i, and f_i is a parent —or **caller**— of g_j.

Function interruptability and, more generally, asynchrony are addressed in §6.

4.2.2 Event Capture

Using the shadow stack, it is easy to determine which context any traced event belongs to.

That data is recorded in the `dbux/runtime` module, mostly in the `RuntimeMonitor` class which contains event hooks for all types of syntax, and the `builtins` folder which is responsible for monkey-patching built-ins. When an event is recorded, it is annotated with its `staticId`, that is the id of a static event record that represents its code location, as well as its `contextId`.

Data flow recording is discussed in §5.1.2.

doi:10.6342/NTU202210012

4.2.3 Trace Filtering

Two of the biggest problems of omniscient debuggers are performance and clutter (or "noise"). One simple method of mitigation is to simply not record parts of the application, configured either by code location, or by time of recording. Pothier et al. [97] call these techniques spatial scoping and temporal scoping, respectively.

Dbux supports several regex-based methods to filter packages, modules and files[3]. Recording of individual functions or statements can also be configured via comments. While the application is live, results shown in the call graph can be paused, resumed and cleared[4].

4.3 Visualization & Interaction

We proceed with an overview of Dbux's main features:

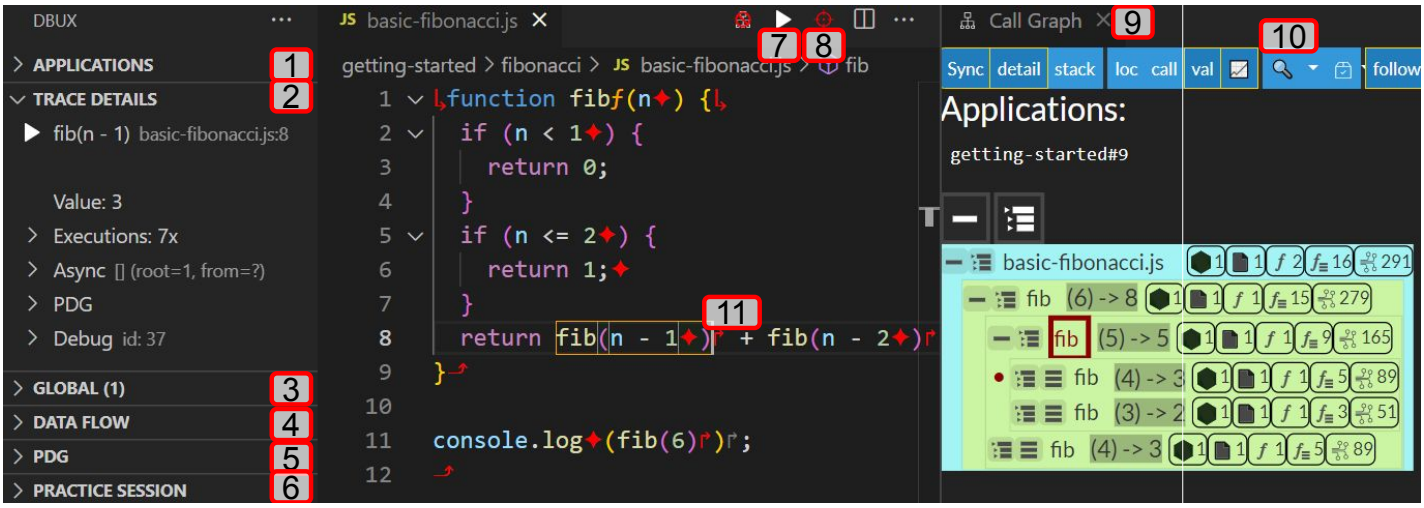


Figure 4.2: Dbux Overview (Legend Below)

Table 4.1: Legend for Dbux Overview

1	The "Applications view" allows viewing, toggling and managing of all recorded applications.

[3]Trace Filtering: https://domiii.github.io/dbux/guides/runtime-trace-filtering

[4]Call Graph Pause, Resume & Clear: https://domiii.github.io/dbux/features/call-graph#pauseresumeclear

doi:10.6342/NTU202210012

2	The “Trace Details view” (§4.3.1) allows inspecting low-level information, such as event values and all executions of a line of code, as well as navigation of events in order of execution and along the call graph.
3	The “Global view” (§4.3.2) lists/summarizes some global events and statistics.
4	The “Data Flow view” (§4.3.3) lists all events of currently selected value or memory address through the entire execution history.
5	The “PDG view” allows running a PDG sample and viewing its PDG using a single click (more in §5).
6	The “Practice Session view” is part of Dbux-Projects (§4.6) and provides a range of tools to easily practice on open-source codebases and bugs.
7	The “Run with Dbux” button ▶ that executes the currently opened `*.js` file in Node.js with Dbux enabled.
8	The “Select button” ⌖ is available if the keyboard cursor is placed on a statement or expression that has at least one recorded trace. When pressed, will select the recorded trace of the innermost statement or expression. Relevant events of the selected piece of code can now be inspected in the Trace Details view. Press multiple times to toggle through nested traces. If another trace is already selected, will select the one that is “closest” to that.
9	The “Call Graph” 品 (§4.5) acts as a map of all function executions.
10	The “Search Tools” allow searching runtime data by their string representation. The user can choose between: (1) executed lines of code, (2) executed function names or (3) runtime values.
11	“Code decorations” (§4.4) helps understand which code actually executed and how.

The feature list in Tbl. 4.1 is not entirely complete. Most notably, we will discuss Dbux-PDG (§5) and Dbux-ACG (§6) in their respective chapters later. Also, to increase usability, Dbux provides over 50 VSCode commands and its own CLI tool.

In the rest of this section, we briefly introduce the smaller features of this list, while the two bigger features are explained in following sections: (i) The Call Graph in §4.5 and

doi:10.6342/NTU202210012

(ii) Dbux-Projects in §4.6.

4.3.1 Trace Details

If a piece of code is selected, relevant information about all its events (executions) can be inspected in this view. It provides five major features:

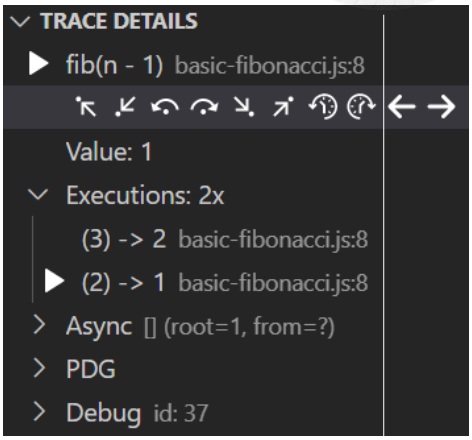


Figure 4.3: Trace Details View

Navigation↷: Navigate the executed trace, forward and backward in time, and up and down the call graph.

Value: If the currently selected trace event is an expression with a value that is not `undefined`, that value (or value hierarchy, in case of reference types) will be rendered here. It also has buttons to (i) render the value in a new VSCode window and (ii) go to the value's origin trace (see Data Flow for more info).

Executions: A piece of code might have executed multiple times. This node lists all of them. One can select and investigate them individually by clicking. Since this can be a long list, they can be grouped by several different criteria through the group button on the Executions node itself.

Async: If the selected value is a promise or executed as part of an asynchronous CGR, this node allows inspecting relevant ACG edges and PromiseLinks, which we will discuss in §6.

Debug: Lists relevant raw data associated with the selected event.

4.3.2 Global View

The Global View provides an overview over certain types of global events and statistics:

Errors: Lists all detected errors in the program. Errors are identified by interrupted stack frames or throw statements. Each (synchronous part of any) interrupted stack contributes one node to this list. Each node represents the first (deepest) recorded interrupted frame of that stack. The children of each node are all other interrupted stack frames on the

doi:10.6342/NTU202210012

same stack. If there are recorded errors, the user select one from the list or click the Error button to navigate to and select the first recorded error location in code. From there the user can investigate everything related to the error, including the error object itself, if it is available. If there are multiple errors, the user can click the button multiple times to toggle through all of them.

Console: Lists all calls of `console` methods. This allows the user to quickly navigate to the relevant place in the execution that printed something. This is to address the first step of what Katz and Anderson refer to as "causal reasoning" [54]: uncovering a location link from the output into the relevant place in the code.

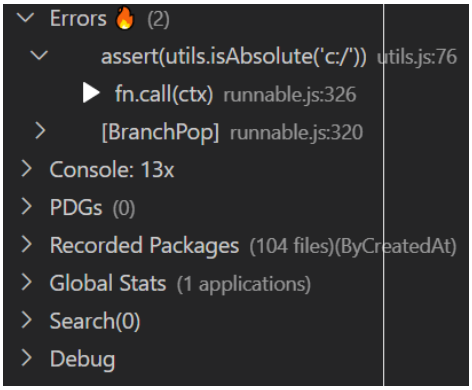


Figure 4.4: Global View

PDG: The global PDG node allows inspecting the structure of the currently active PDG (more in §5).

Recorded Packages: Lists all dependencies, packages, modules and their files.

Global Stats: Lists trace record statistics by function and package. The information here can be used to understand fine-tune the trace filter. The strategy generally is: if a package produces a lot of data, but is not (currently) of use, ignoring it can help improve Dbux's performance significantly.

Search: After a search has been started, results are listed here.

4.3.3 Data Flow View

If the currently selected trace event has a value, this view lists all events that (1) either contain the same value, (2) or reference the same memory address. These two data flow modes can be toggled through a button at the top of the view. Another button allows toggling between reads only, writes only or both. The first entry in this list is always the creation of the value or initial value assignment of a memory address (or, at

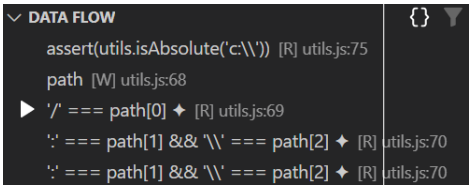


Figure 4.5: Data Flow View

doi:10.6342/NTU202210012

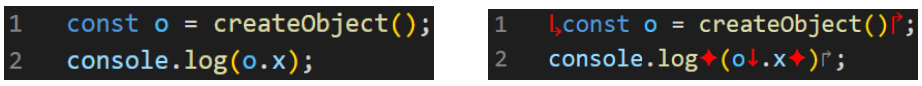
least, the first recording thereof).

4.3.4 Enabling Dbux

Before a user can analyze a target application, they must first **execute it with Dbux enabled**. This means, the application must be instrumented by `dbux/babel-plugin` and injected with the `dbux/runtime`. This can be achieved in four different ways:

- The "Run with Dbux" button ▶ that executes the currently opened `*.js` file in Node.js with Dbux enabled.
- Using the Dbux CLI directly (it is used by the "Run" button under the hood) allows for more advanced configuration.
- In order to enable Dbux for frontend and other bundled or non-Node applications, the user needs to manually integrate the `dbux/babel-plugin` into the project's build pipeline[5].
- Dbux is automatically enabled when running applications from the "Practice Session view" (§4.6).

4.4 Code Decorations



(a) Without Code Decorations (b) With Code Decorations

Figure 4.6: Code Decorations Example of Code w/ three Function Calls

"Code decorations" are used to highlight the currently selected expression or statement, and help visualize recorded execution behavior of the code. It generally allows the user to differentiate between lines of code that executed vs. those that have not, as well as certain properties of that execution.

We find "call graph indicator decorations" particularly useful. Consider the example in Fig. 4.6: How many functions were called in these two lines of code? Even without

[5] `https://domiii.github.io/dbux/guides/build-pipeline-integration`

doi:10.6342/NTU202210012

decorations, we can count two function calls: (1) `createObject()` was called on L1. Even though, this function call is quite clear without decorations, it is further enunciated by the red arrow at the end the line in Fig. 4.6b. (2) `console.log()` was called on L2, enunciated by the gray arrow at the end of that line in Fig. 4.6b.

However, as it turns out, these are not the only function calls. On L2, the red, downward-pointing arrow in the middle indicates that `o.x` triggered a non-obvious step down the call graph, which means that `o.x` is either a getter or triggered some other type of trap, which lead to the execution of an unknown function somewhere. This is a common example of "dark matter": Invisible and (often intangible) execution behavior. An arbitrary amount of code (with possible side effects!) was executed from code (`o.x`) that looks too innocuous to warrant an investigation.

Finally, we note that the arrow on L2, behind `console.log()` is gray (unlike the arrow at the end of L1). That is because the function call itself (and its entire call stack ancestry) was not recorded. Its node (and its ancestors) will not show up in the call graph, and we cannot step into it. This happens either (as in this case) if the function is built-in (or "native"), or if the function is not recorded due to *trace filtering*§4.2.3.

4.5 The Call Graph (CG)

The "Call Graph" (CG) is a dynamic call graph and represents a global type of "map" of the entire execution history, specifically high-level control flow events. Call graphs are commonly used in profilers to help developers identify performance bottlenecks [46]. They capture the caller-callee relationship between executed files and functions.

The CG's vertical dimension is time: lower is later. A vertical layout mimics the familiar visual flow of the source code itself. Vertical layouts also avoid horizontal stacking of text, and are generally a preferred format for reading. Node colors are assigned pseudo-randomly. Same color means same static context (same function or file).

doi:10.6342/NTU202210012

4.5.1 Call Graph Example: fibonacci

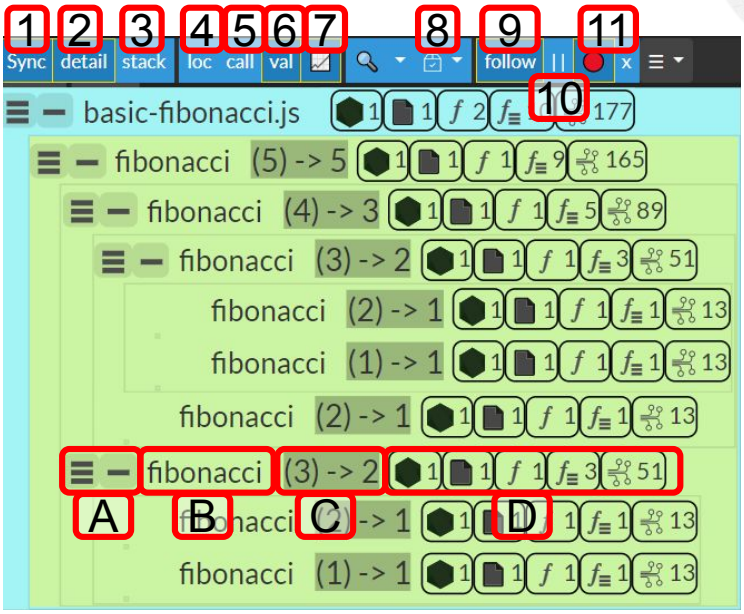


Figure 4.7: CG of `fibonacci(5)`

Fig. 4.7 demonstrates a CG of the `fibonacci` function with `val` and `stats` enabled. It demonstrates the power of the CG to serve as a complete recursion tree of any particular execution of a recursive function. Each row represents one node, that is one recorded file or function execution (i.e. one "context"), indented to the right of their parents. Nodes with children have two of three buttons A to (1) expand only the node itself, (2) collapse the node or to (3) expand the node's entire subtree. Behind the buttons, each node displays its name B, as well as location (**loc**) and/or value (**val**) C, if enabled. When clicking the name, the start of the function declaration is selected and shown in the code editor. Lastly, if enabled, five types of **stats** are summarized D.

4.5.2 Call Graph Settings

Fig. 4.7 demonstrates an example call graph with the toolbar at the top. From left to right, the toolbar buttons have the following functionality:

doi:10.6342/NTU202210012

Table 4.2: CG Settings

1	**Sync**: Toggle between sync and async mode. Sync mode displays the CG, while Async mode switches to the Asynchronous Call Graph (ACG) instead. The ACG is discussed in §6.
2	**detail**: Toggle "detail mode". This currently does not do anything for the CG. However, in Async mode, disabling details visually compacts the graph. This is used to better expose high-level patterns between CGRs. One can more easily see high-level patterns when disabling details and then zooming out.
3	**Stack**: Toggles the Asynchronous Call Stack (ACS). This is discussed in §6.6.1.
4	**loc**: Toggle showing the location (file and line number) for each node.
5	**call**: Toggle showing the caller for each node. The caller can be clicked to select (and move to) the caller in code.
6	**val**: Toggle showing the values of each node's call expression in form of `(arguments) → returnValue`.
7	**Stats**: Toggle whether to show five types of statistics about each node's entire subtree: amount of different modules from which the subtree's functions were called, amount of files, amount of functions, amount of function executions (actual node count) and amount of trace events.
8	**Filter**: Opens a menu where one can configure a trace filter, similar to the one discussed in §4.2.3, but applied to the call graph. When enabled, all filtered nodes are merged into "Hole nodes", thereby largely reducing clutter of unwanted function calls.
9	**follow**: Toggle "follow mode". When enabled, the call graph will automatically open all ancestors of, highlight and scroll to the context node of the currently selected trace event.
10	**thin**: Toggle "thin mode". When enabled, reduces indentation space. Useful for rendering large CGs more compactly.

doi:10.6342/NTU202210012

11	**Pause/resume** and **clear**: These are recording tools. The pause/resume button allows toggling whether to keep updating the CG when new events are recorded. The clear button hides all previously recorded nodes.

4.6 Dbux-Projects: Project Playground and Practice Sessions

In addition to being an omniscient debugger, Dbux also hosts Dbux-Projects: a debugging "playground" that makes it easy for the user to use Dbux on popular open source projects. The main purpose of Dbux-Projects is to re-produce a specific execution scenario (which we call "exercise") at a single button click. The user first picks from a list of projects, and then from a list of exercises associated with each such project. The projects are currently hand-picked and curated by the Dbux maintainers (us).

Once an exercise has been chosen, a single button click will (1) clone the codebase from an open source repository, (2) switch to a given branch or tag if necessary and/or apply patches, and then (3) run the given exercise with Dbux enabled. An exercise can be anything, such as a failing test case that illustrates a bug, or any Node.js or web application.

The `dbux/projects` module aims to **simplify project and exercise curation**. To add a new project, one only needs to add a new `Project` file to the `projects` folder, register it, and provide some basic information, such as (i) the repository name or address, (ii) commit hash, branch or tagname, as well as (iii) execution parameters. If bundling is necessary, the API provides a simplified way of providing `webpack` build parameters. A frontend application will automatically be served and started in the browser, while non-frontend applications are, by default, executed with Node.js. Tools for wrapping Jest and Mocha, as well as easily providing their parameters are also provided. Each project can have an arbitrary number of exercises, which encapsulate customized start parameters, in order to, for example, (i) target a specific buggy version or (ii) change inputs. Exercises can either be hard-coded or provided separately in an exercise file. Exercise files can also

doi:10.6342/NTU202210012

be dynamically reloaded.

As of 9/2022, these are the GitHub repositories of all currently curated projects that were confirmed as working (i.e. Dbux was manually verified to run successfully on them): `webpack/webpack`, `real-world-debugging/todomvc-es6`, `socketio/socket.io-client`, `sequelize/sequelize`, `gothinkster/web-components-realworld-example-app`, `node-fetch/node-fetch`, `trekhleb/javascript-algorithms`, `BugsJS/hexo`, `BugsJS/express`, `BugsJS/eslint`, `pandao/editor.md`, `petkaantonov/bluebird`, `caolan/async`, `gabrielecirulli/2048`.

4.7 Debugging Journey: Express#1

In this section, we set out to investigate how and whether Dbux increases *fact tangibility*§3.2.7 when confronted with real-world bugs.

4.7.1 BugsJs

We selected several bugs from BugsJs [49], a benchmark bug database, containing 453 real-world bugs from 10 popular open-source Node.js projects which all use the Mocha testing framework. Each bug is isolated, usually affects only few lines of code, and has at least one test case that fails before the bug is fixed and succeeds once it is fixed. From it, we have selected **express#1** for the following case study.

4.7.2 Express

Express is a web application framework for Node.js, commonly used as an HTTP server. It is relatively small with 9 kloc, but also among the most popular JavaScript repositories on Github[6] with 25+M downloads per week[7].

[6] As of 9/2022, Express is found on the second page of Github's JavaScript topic page, sorted by star count: `https://github.com/topics/javascript`.

[7] `https://www.npmjs.com/package/express`

doi:10.6342/NTU202210012

4.7.3 Bug Description

```
OPTIONS should only include each method once
Error: expected 'GET,PUT' response body, got 'GET,PUT,GET'
```

Figure 4.8: Test Result (from Console Output) for Express#1

Relevant Express bugs from BugsJs have been added to Dbux-Projects §4.6. **Reproducing** thus only requires a single button click.

Expected Result: The test case should pass.

Actual Result: The test case did not pass, and instead, as Fig. 4.8 shows, reported the error message: "`Error: expected 'GET,PUT' response body, got 'GET,PUT,GET'`".

4.7.4 Debugging Journey: Short Version

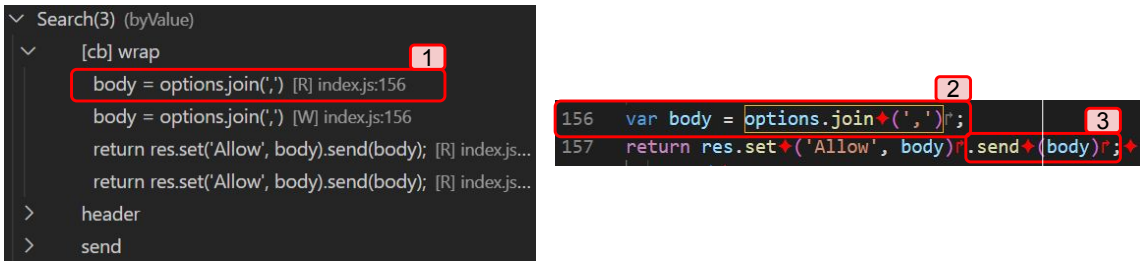


Figure 4.9: All Locations (sans Output) of the Short Version's Goal Path

In the short version, we can find the bug by traversing only 3 major locations, and 6 steps. Most of the goal path is visualized in Fig. 4.9:

1. The test output tells us that express sent the incorrect response `GET,PUT,GET`.
2. We proceed to use Dbux's search function to search for that unwanted (yet materialized) run-time value. We see the results in Fig. 4.9 (left). There are a total of 16 results in 3 functions, only 5 of which are Writes.
3. Of the search results, the first function contains assignment `body = options.join(',')` 1 . Domain knowledge renders this line promising: the naming and semantics sug-

doi:10.6342/NTU202210012

gest that this constructs the `body` (which is a `string`) of the HTTP response, from an array called `options`.

4. Clicking that statement 1 takes us to the code 2 (right).

5. We check the *value*[§4.3.1] of the expression `options.join(',')` 2, and it is indeed the incorrect result string `GET,PUT,GET`. This seems to be the bug.

6. The next line enhances our suspicion that this is the right code, again due to domain knowledge: `send(body)` 3 seems to send out the HTTP body.

We thus feel we have enough evidence and decide to fix this bug: instead of sending all of `options`, we build `body` from a new array that only contains unique values of `options` instead. This makes the test pass.

We conclude that the "search by value" feature makes this type of bug almost trivial. As we posit in §3.2.7, low bug difficulty correlates to length and high tangibility of the goal path, from the starting point (output message) to the faulty line of code. In our next attempt, we aim to answer the question: What if we did not have that feature? Would the bug still be that easy?

doi:10.6342/NTU202210012

4.7.5 Debugging Journey: Longer Version

Figure 4.10: All Locations of the Longer Version's Goal Path

Let's compare the previous journey to another one, but without the "search by value" feature. It is important to note that this constraint drastically decreases the tangibility of important output message facts, specifically: this time there is no direct link from the output message to the search feature (which in the previous journey links directly to the faulty line). As a result, we have to take an entirely different approach. The goal path is visualized in Fig. 4.10:

- We start with a common strategy: look at the test case. For that, we first need to find it. Luckily, the test case is a CGR, visible on the call graph 1 . Clicking it takes us to its code (top right).
- Scanning the test case reveals that it calls an `options` function 2 . This is promising because, from the test output, we know that the problem is with `OPTIONS`. The gray arrow *decoration*[§4.4] next to it indicates that its synchronous call stack only contains library code and none of our own code. It thus cannot contain the bug (assuming that the bug is in our code and not in a library). This often implies that the call does not execute our code directly, but rather schedules it asynchronously.

doi:10.6342/NTU202210012

- But this is still our best lead. We thus decide to use the "search by context" feature to search for executions of any function containing the word `options` instead. There are 4 results: 1 test file and 3 executions of the `Router._options` function.
- Clicking any of the `Router._options` functions 3 takes us to its code (not shown). The function does not look too interesting. We are deep in the call stack.
- Navigating up the call stack to the caller takes us to `options.push.apply(options, route._options());` 4 .
- Using the Trace Details → Executions feature reveals that this was called 3 times, and the values (in order) were: `GET`, `PUT`, `GET` 5 . This is suspicious, since it seems to be the data that also made it into the final erroneous output. We now want to track this data to any locations where its used.
- The result value of that call is getting pushed into an array called `options` 6 .
- We thus choose to trace down the `options` array, using the *Data Flow View*[§4.3.3] (bottom right).
- The last occurrence in that view 7 again looks promising, for the same reason it looked promising when we saw it in Location 3 of the shorter journey above.
- Clicking it, again brings us to the buggy line 8 .

We verify the bug using the same method as in the first journey's Step 5.

We can conclude that, due to a small change in fact tangibility, our approach has changed drastically. We assess that not only were more steps required, but decision-making involved in some of those steps is also more complex, requires more effort and a more complex strategy. Or in short: this journey has a longer goal path with individual fact tangibility decreased. We thus conclude that the journey without the "search by value" feature is more difficult, due to a decrease in fact tangibility.

4.7.6 Express: Other Bugs

We now skim through a few more Express-related examples to highlight some of Dbux's features.

doi:10.6342/NTU202210012

Code decorations assisted us in finding several bugs. For example, in bug Express#2, the HTTP protocol was used when it should have been HTTPS. Fig. 4.11 demonstrates the augmented red downward arrow on line 86, telling us that `req.protocol` is a getter function, and not just a variable. We could navigate to the getter function with just two clicks, where we then found the bug.

```
80  it('should respect X-Forwarded-Proto', function (done) {
81    var app = express();
82
83    app.set('trust proxy', 1);
84
85    app.use(function (req, res) {
86      res.end(req.protocol);
87    });
88
89    request(app)
90    .get('/')
91    .set('X-Forwarded-Proto', 'https')
92    .expect('https', done);
93  })
```

Figure 4.11: Express#2 Test Case

The call graph visualizer and its error highlighting assisted us in fixing bug Express#14. Fig. 4.12 shows how the call graph implicitly highlights the error stacktrace.

4.8 Summary

In this work we have presented Dbux, a first multi-resolutional code-level Integrated Debugging Environment (IDbE). Thanks to Dbux-Projects, in just a few steps, we were able to easily reproduce all previously prepared bugs. We have illustrated some of Dbux's vast feature set, and how much of an impact the difference in tangibility of one type of fact can have on the overall debugging process.

This chapter shall serve as a foundation for the following two chapters. While successful studies of the following chapters primarily focus on newly introduced features that go beyond what has been discussed in this chapter, they also speak to the usefulness and

doi:10.6342/NTU202210012

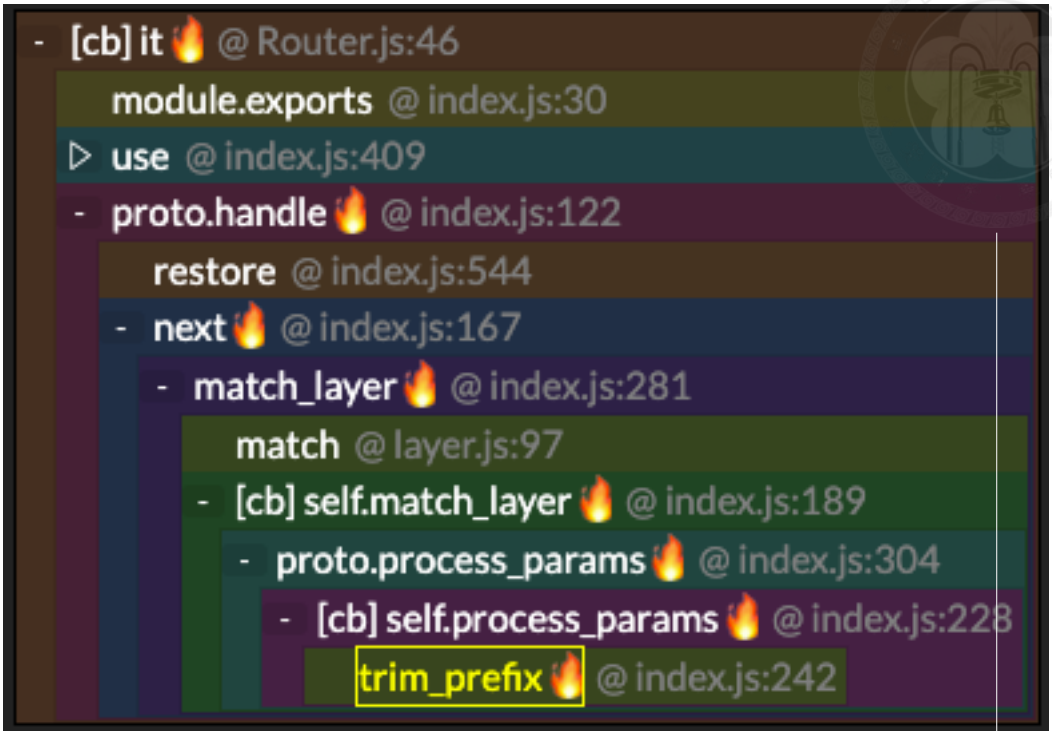


Figure 4.12: Express#14 Call Graph Excerpt

resilience of Dbux itself.

Performance and scalability issues are often cited as the biggest problems for omniscient debuggers. However, as the author of ZStep 95 [77] aptly pointed out: "Even in an extremely large program, where keeping a complete history is infeasible, judicious testing can often isolate a fragment of the code which is not too large to run ZStep on". This not only applies to ZStep, but to any omniscient debugger. Nevertheless, we admit that Dbux and many of its omniscient peers are certain to falter when confronted with bugs in resource-intensive applications with high event frequency, such as games, especially if those bugs cannot be easily reproduced in a test case or short test run.

doi:10.6342/NTU202210012

doi:10.6342/NTU202210012

Chapter 5

Dbux-PDG: Revealing Hidden Properties of Data Structures and Algorithms

Comprehending of complex programs, such as those of data structures and algorithms (DSA), requires iterative build up and traversal of complex mental models. When looking at or working on one part, it is easy to omit, misconstrue or forget related parts [59].

To aid developers in this difficult process, Algorithm Visualizations (AV) have become a popular tool to help with learning and understanding DSA, but they commonly suffer from two major shortcomings: (i) every problem requires their own AV to be carefully constructed and curated. Even if one was to find a sufficiently closely related AV of the problem at hand, (ii) AVs target the high conceptual level, while many bugs and comprehension tasks involve the implementation level.

Having identified a clear lack of tools in this space, we designed Dbux-PDG to allow users to stay oriented while seeking answers to questions about data flow and data dependencies between memory addresses, statements and control blocks. This first version of Dbux-PDG has been designed to help learners understand and interact with DSA implementations.

In its fully summarized form (e.g. Fig. 5.4a), Dbux-PDG only shows a selected func-

doi:10.6342/NTU202210012

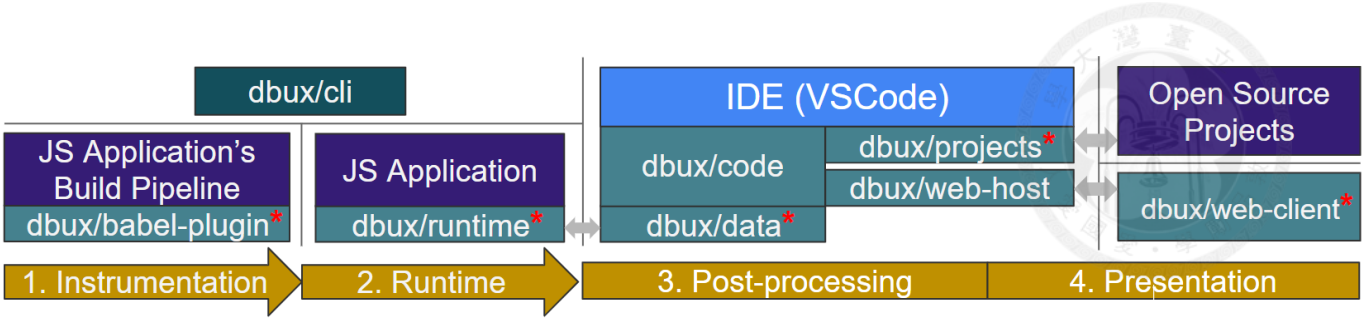


Figure 5.1: Dbux Architecture: modified modules have been annotated with a red asterisk.

tion's inputs, outputs and their dependency relationships. Many bugs become obvious on this level already. To understand causality, the user can further expand (and collapse) nested control flow blocks, all the way down to the statement level.

Dbux-PCG is implemented as part of Dbux. It thus currently only supports JavaScript programs. While JavaScript might not be the first choice for algorithms at the competitive level, it is worth noting that JavaScript is among the most popular programming languages [119]. Furthermore, as of summer 2022, the `javascript-algorithms` (JSA) repository is among the top 20 most-starred GitHub repositories of all time. All of our tools are open source and available on Dbux's GitHub repository [104].

§5.1 explains how and what kind of data is collected. Dbux-PDG is formally introduced in §5.2. Visualization and interaction design are discussed in §5.3. §5.4 investigates Dbux-PDG's use on several types of algorithms. We show test results of confronting 94 of JSA's algorithms in §5.6. Future Work & Limitations are explored in §5.7, before concluding in §5.8.

5.1 PDG Data Collection in Dbux

In this work, we build a modified, dynamic version of a Program Dependency Graph (PDG) [41]. There are three structural deviations from the original PDG. (i) We employ dynamic rather than static analysis, and thus a different set of construction algorithms (see §5.2). (ii) The nodes of our PDG comprise memory locations (and their values), rather than statements. (iii) As explained in §5.2.5, several types of dependencies are omitted.

Dbux-PDG is implemented as a modification of Dbux [104], as indicated in Fig. 5.1. Data recording happens at the language level. Thus, any target application needs to be

doi:10.6342/NTU202210012

"executed with Dbux enabled".

5.1.1 Recording Control Flow

The original PDG [41] captures controlled statements in "region nodes" (a notation proposed by Allen [20] to refer to control flow graph nodes). We denote such nodes as **control blocks** instead.

In Dbux, a shadow stack tracks all function executions with an initial `push` and a concluding `pop`, respectively at the beginning and end of the function block. This allows all events to be assigned a `contextId` representing their file or function execution. We consider all other control statements and expressions (e.g. `if`, `switch`, ternary expressions, loops etc.) to have one or more decision events controlling a statement or statement list. A control block captures the execution of such controlled statement or statement list. Dbux-PDG records three types of control blocks:

1. Dbux already recorded *function* executions.
2. For `if` statements, we added tracing of push and pop (start and end) events of the `if` block and annotated them with the statement's `staticId`. The decision expression event is also annotated. To remove clutter, we group nested `if` statements into one, with all ids set to that of the group's root.
3. Loops require multiple strategies. Basic `for` and `while` loops are relatively easy to deal with, similar to a repeated `if`. `for-of` is the most complicated, since it requires more complex instrumentation and also monkey-patching of iterators.

`switch`, `for-of`, `for-in` and `do-while` were cut for time, since we only came across `switch` in one of our test subjects (see §5.6), while these loop types did not make any appearance.

5.1.2 Data Flow: Events and Nodes

Data flow is recorded in a new data flow node (**DFNs**) data structure. DFNs can have (i) type, (ii) memory address, (iii) input nodes and (iv) value.

doi:10.6342/NTU202210012

There are four **types** of DFNs: `Read`, `Compute`, `Write` and `Delete`.

Like Jalangi [108], we encode any **memory address** in either one of two categories: (i) variable access or (ii) object/array field access (e.g. `a[f()]`, `o.x`). Access of the same variable is stored in form of a `declarationId`, while field access is stored by the object's DFN id and the `prop` integer (array) or string (object).

We identify two types of **data dependencies**: (i) `Write` and `Compute` DFNs depend on their **input nodes**. E.g. input nodes of `a + b` are the DFNs of `a` and `b`, while the input nodes of the branching assignment expression `a ||= b` are the DFN of either `a` or `b`. (ii) A `Read` DFN depends on the last `Write` or `Delete` of the same memory address.

Primitive **values** are stored as-is. For reference values, an id is stored. For reference values, we maintain index data structures for tracking all field access of any props of the same object. This allows reconstructing the object as it was during any point during the execution.

5.1.3 Recording Data Flow

In order to record **DFNs**, `RuntimeMonitor` and the `builtins` folder (see §4.1) were modified to create DFNs which track all computations and memory access.

Event records and DFNs observe a 1-to-many relationship. E.g. `i++` has 2 DFNs: (i) the DFN that represents the event's own value, a `Read` of `i`'s old value, followed by (ii) a `ComputeWrite` of `i`'s new value. Furthermore, every valued event (e.g. any recorded expression) is annotated with the id of the DFN that represents its own value.

Data collection in Dbux requires modifying the `dbux/runtime` and `dbux/data` modules. The logic of built-ins (native API functions and classes, e.g. `Array`, `Map`, `Math` etc.) needs to be captured explicitly, since their effects are invisible to instrumentation. Their data flow is captured using a mixture of monkey-patching [94] and post-processing. Currently, only a small set of built-ins are traced. While most data flow is recorded during the runtime phase, some data flow, such as that of function parameters and return values, is resolved in post-processing.

In post-processing, all DFNs accessing a specific memory address are assigned a

doi:10.6342/NTU202210012

unique `accessId`, and all DFNs recording the same value of same origin a unique `valueId`. This allows tracking all access of a given memory address, as well as movement of a value throughout the execution. Storage of `Map` and `Set` involve extra steps to serialize referential keys.

We note that while this lowest layer, the one responsible for capturing program dependency events, is inherently language-specific, most parts of Dbux-PDG are language-agnostic.

5.2 Dbux-PDG Construction and Summarization

This section explains PDG construction and summarization, as well as the role of the WatchSet in both. It concludes with a discussion of and the reasons behind missing dependencies.

5.2.1 The WatchSet and PDG Bounds

Before a PDG is constructed, the user needs to select a **WatchSet**: a set of watched events the user has expressed interest in. The general idea is to let the user pick any recorded events in the code editor, and then construct a PDG that spans between these arbitrary points in the program execution. For now, the WatchSet simply marks the inputs and outputs of a user-selected function execution of interest. This makes Dbux-PDG suitable for investigating algorithm implementations, as well as the data structures employed therein. We also use the WatchSet to determine the boundaries of the PDG, which for now are the boundaries of the selected function execution. We denote PDGNodes that are part of the WatchSet as `watched`.

5.2.2 PDG Construction

The PDG comprises ten types of **PDGNode**s – six types representing ControlBlocks and four types of data PDGNodes. Edges represent data depedencies.

A **ControlBlock**'s children are the PDGNodes of events that happened inside the

doi:10.6342/NTU202210012

block, including nested ControlBlocks. Children are ordered by time of execution. There are six types of ControlBlock nodes: (1) The single root. (2) Function blocks. (3) If statements. (4) Loops. (5) Loop iteration nodes are inserted as children of loops when the loop is (re-)entered. (6) Groups of built-in *higher-order functions* (HoFs): to increase readability, we create an extra control block node to group all callback executions of the same HoF call (e.g. `find`, `map` etc.).

There are four types of **data PDGNodes**:

1. Value nodes represent a value that was moved or a computation.
2. Delete nodes represent single deletes of object fields, e.g. as induced by the `delete` statement or `Array.pop`.
3. Snapshot nodes represent complete or partial snapshots of reference type objects and their contents. They have children of types Snapshot, Value, Delete and RepeatedRef.
4. RepeatedRef nodes represent circular references inside nested Snapshots.

For PDG construction, a temporary `controlStack` data structure maintains the stack of all currently active ControlBlocks. The PDG is constructed in the following high level steps:

1. Compute the bounds of the WatchSet (see §5.2.1).
2. Iterate all recorded events `e` within the bounds of the WatchSet in order of execution.
3. If `e` marks the start or end of a ControlBlock: push or pop the `controlStack`.
4. Iterate all DFNs `dn` of `e` –
5. Add data PDGNode `n` for `dn`. Determine its type. Build snapshots greedily, using the field access index (see §5.1.2).
6. Add `n` to the ControlBlock at the top of the `controlStack`.
7. After a data PDGNode has been added, find the PDGNodes representing the DFN of its data dependencies (see §5.1.2), and add edges from them. Note that the order-of-execution constraint ensures that its dependencies have already been added.

doi:10.6342/NTU202210012

PDG construction further involves bookkeeping and heuristics to: (i) construct node labels, (ii) reduce clutter from long read chains, (iii) track and connect watched nodes, (iv) reduce snapshot size and (v) determine connectedness (see §5.3.4).

5.2.3 Summarization View Construction

The summarized graph is a separate graph from the original PDG. Note that this graph might contain back edges, while the original PDG does not. It uses the original PDG and the set of user-selected summary modes of all ControlBlock nodes as its input.

Every node is annotated with a `summaryMode`. The `summaryMode`s of visible ControlBlocks are explained in §5.3.3. Internally, two more modes, `Show` and `Hide`, are used. Watched nodes are always shown as-is.

For each summarized ControlBlock, a `nodeSummary` object stores three types of `SummaryNode`s, representing (i) variables and (ii) snapshot roots that are written inside, but accessed after the block, as well as (iii) the return node in case of non-void function blocks. Internally, two index data structures help looking up nodes by `declarationId` or a reference type's `valueId`. During construction, a temporary `nodeRouteMap` is maintained to map each (possibly hidden) node to its `reroutes`: a set of visible nodes that its children should be connected to in its stead.

We differentiate between full and shallow summaries of a block. A full summary summarizes the entire subgraph, while a shallow summary only summarizes the nodes of the block sans descendants. Summarization is performed when the user changes the summary mode of a node. It involves the following steps. Several steps for dealing with shallow summaries are omitted for clarity:

1. Remove all edges.
2. Propagate summary mode changes throughout all ControlBlocks (using DFS). E.g. when a node's `summaryMode` is set to `ExpandSelf`, that of its descendants is set to `Hide`.
3. Iterate over each original node `n` using DFS (this mirrors order of execution) –

doi:10.6342/NTU202210012

4. If `n` is a visible ControlBlock and it is fully or shallow summarized: Build its `nodeSummary` and `SummaryNode`s.
5. If `n` is visible: add it to `nodeRouteMap` with key and `reroutes` both set to itself.
6. If `n` is hidden: find its `SummaryNode` in the nearest summarized ControlBlock's `nodeSummary`. If found, add an entry to `nodeRouteMap` with key set to `n` and `reroutes` set to that `SummaryNode`.
7. If `n` is hidden and has no `SummaryNode`: for each original ancestor PDGNode `from`: lookup `from`'s `reroutes` in the `nodeRouteMap` and add them to `n`'s `reroutes`. This has the effect of multicasting all incoming nodes of a hidden node to all its outgoing nodes.
8. If `n` is visible or has `SummaryNode`: for each original ancestor PDGNode `from`: lookup `from`'s `reroutes` in the `nodeRouteMap`. For each such rerouted node `r`, add a new edge from `r` to `n` or its `SummaryNode`.

The PDG is rendered using the set of original nodes, `SummaryNode`s, summarized edges and `summaryMode`s. During rendering, original nodes are iterated in order, but only visible nodes are rendered. When encountering a summarized ControlBlock, its `SummaryNode`s are rendered instead of its original children.

5.2.4 Summarization Example

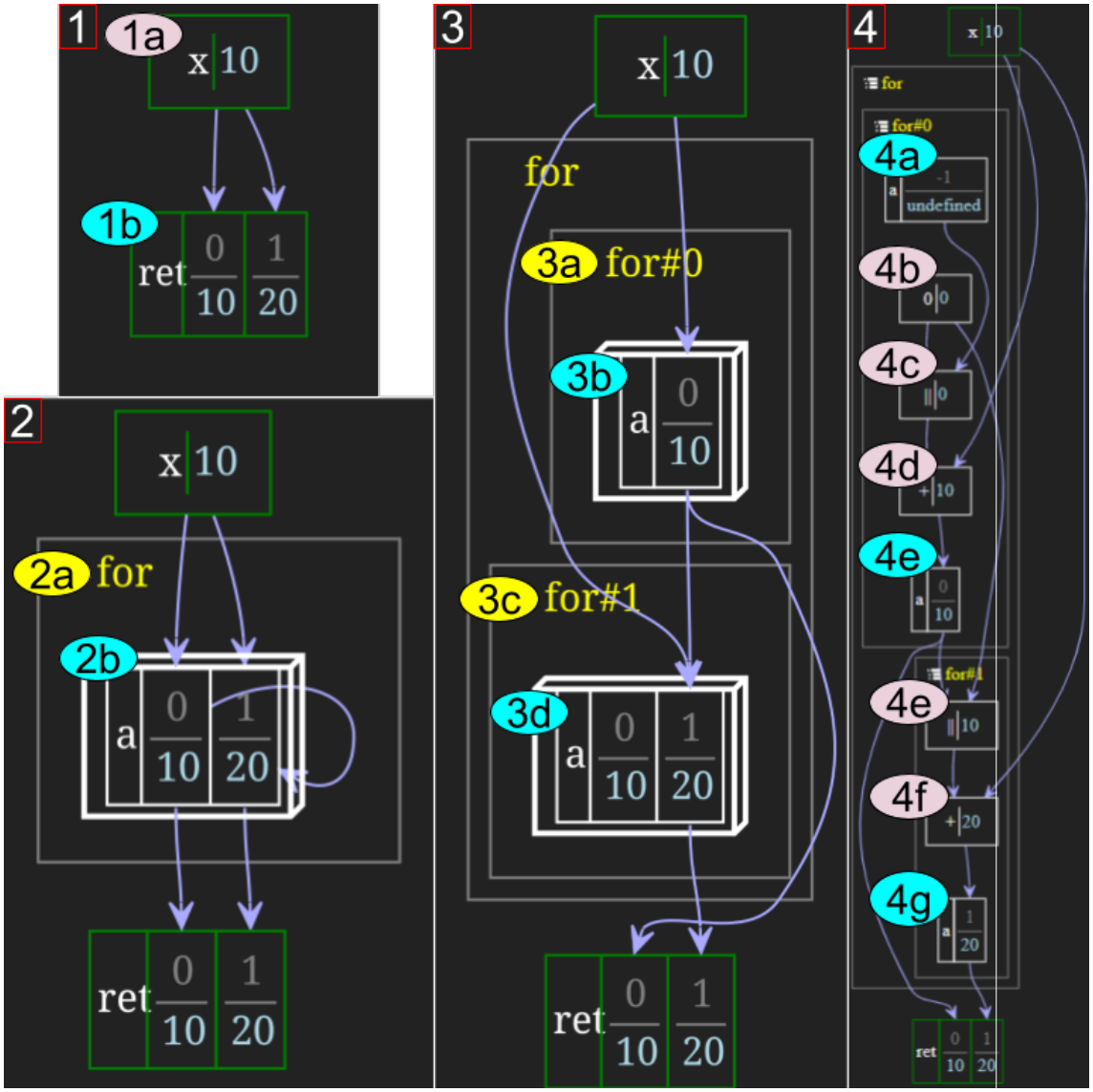
Fig. 5.2b illustrates summarization in multiple steps. Its code (Listing 5.2a) executes a short `for` loop. Initially, the PDG renders 1: the fully summarized view (top left). In it, we only see the watched input value node 1a and output snapshot node 1b.

After expanding the root, summarization gets recomputed, which leads to 2 (bottom left). Now, the fully summarized `for` loop ControlBlock 2a is visible. It only contains one summary snapshot 2b, rendering the final state of array `a`. `a` is the only object or variable that is written to inside and accessed after the entire ControlBlock's subgraph. `i`, for example, does not get accessed after the block and thus does not get summarized. Edges indicate, among other things, that `a[1]` depends on `a[0]` and `x`.

doi:10.6342/NTU202210012

```
function main(x) {
  var a = [0], n = 2;
  for (let i = 0; i < n; ++i) {
    a[i] = (a[i - 1] || 0) + x;
  }
  return [...a];
}
main(10, 100);
```

(a) Example Code



(b) The Same PDG in Four Summary Modes

Figure 5.2: Summarization Example

doi:10.6342/NTU202210012

Expanding the `for` block leads to 3 (middle). Now, the `for` block is only shallow-summarized. It does not contain any summarizable memory addresses and thus does not show any summary nodes. Its individual iterations 3a and 3c are now fully summarized. Each iteration renders its final version of the summarized snapshot `a` 3b and 3d.

The fully expanded PDG is shown in 4 (right). It displays one constant and four computations as value nodes 4b, 4c, 4d, 4e and 4f. Access of the non-existing index (`a[-1]`) and the two array write operations are shown as partial snapshots 4a, 4e and 4g.

5.2.5 Discussion: Missing Dependencies

Dbux-PDG has different goals from the original PDG [41]. While the original strived for completeness, Dbux-PDG strives to help the user cutting through the noise of dynamic execution data. Here, we explain how, as a result of that goal, several types of dependencies were omitted in Dbux-PDG, and the severity of problems that would arise from trying to implement them:

(i) Control flow decision dependencies can cause a lot of noise. For example, a simple loop induces several edges per iteration between often relatively uninteresting data points, such as the iteration variable, a constant and/or the length of an array. That is why we currently ignore branch decision nodes (denoted as "predicate nodes" in the original text [41]). The summarization system would require several adjustments, if not entirely new layers to keep the graph readable after inserting this many edges.

(ii) We currently do not trace expression-level branches (e.g. ternary, logical and optional chaining operators), because they would add many small blocks, and thus quickly overwhelm the user with an unwanted level of detail. Necessity and usefulness of, as well as solutions to address these "small blocks" require further investigation.

(iii) Some algorithms, or parts of algorithms, have more control dependencies than others. For example, most sorting algorithms can be well comprehended through data dependencies alone. At the same time, most search algorithms move little data, but involve a more complex network of branching decisions. If all control dependencies are visual-

doi:10.6342/NTU202210012

(a) A hand-drawn AV of `rain-terraces`.

(b) Dbux-PDG View of `rain-terraces` (r) and its Code (l)

Figure 5.3: Dbux-PDG and AV of `rain-terraces` with the same input, side-by-side.

ized, new tools to help the user focus on specific subsets of dependencies thus might be required.

(iv) A less commonly discussed form of memory access dependency has also been omitted. Specifically, the DFN of `f()[g()]` depends on the DFNs of `f()` and `g()`. In the original work [41], this is denoted by the `select` operator. Again, we do not render these dependencies because more innovations would be necessary to keep such a PDG sufficiently small and readable.

5.3 Visualization & Interaction Design

Fig. 5.3 serves as a first motivating example, comparing the usefulness of a hand-drawn AV with Dbux-PDG: the AV (left) demonstrates the `rain-terraces` problem with a given input. In it, each element of the input array is rendered as a stack of gray blocks. Each value is represented twice: (i) the number below each stack, and as (ii) the height of each stack. The problem is expressed visually: compute the amount of blue (water) tiles, contained by those gray blocks.

When compared to the Dbux-PDG on the right, the AV helps understand the problem statement, while Dbux-PDG does not. Both visualize the input array. Dbux-PDG also shows the final result, 4 (in the `ret` node, bottom). The AV shows that array index 1

doi:10.6342/NTU202210012

and 3 each contribute exactly 2 to the final sum 4, while the PDG shows that this happens during iterations 1 and 3 of the for loop, respectively. The PDG also shows which input array elements contributed to each iteration. Moreover, it allows for further inspection and exploration.

The sceenshot captures the moment where the user hovers over the second `res` node. It has a red background and its edges are highlighted. Clicking it takes the user to the location of its last Write in the code (L18). We posit that the PDG cannot fully replace, but complement AVs, while creation comes at a fraction of the cost.

5.3.1 Layout

The PDG is rendered with `d3-graphviz`, a JS wrapper for Graphviz [39], on top of *Dbux's Webview component system*[§4.1].

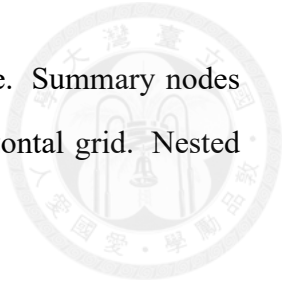
We use Graphviz's vertical `digraph` layout. The reason for the vertical choice is the same as for the CG.

In our experiments, we found thin, tall graphs perferrable to wide or square graphs because it allows the user to follow an edge along a single dimension. Thin graphs can lead to longer edges which is why they go against the general recommendation of having the aspect-ratio of the overall graph shape match that of its container [28]. However, it felt significantly harder to follow edges in a graph that is wider than its container, where edges extend in arbitrary, rather than only the vertical directions. Furthermore, unlike the originally proposed PDG [41], Dbux-PDG captures dynamic, not static, data and control flow. That imposes an order of execution by time on all recorded nodes. It is thus (generally) useful to think of the graph as a timeline. Verticality visually reinforces that timeline. These insights motivated us to provide an "enhanced verticality" feature which is explained in §5.3.2.

All nodes of the same ControlBlock are grouped into the block's own rectangular region which we refer to as subgraph or cluster as per the Dot language specification[1]. Sibling subgraphs are assured not to overlap.

[1] `https://graphviz.org/doc/info/lang.html#subgraphs-and-clusters`

doi:10.6342/NTU202210012



Watched nodes have a green outline, while other nodes are white. Summary nodes have a 3D border. The contents of snapshots are rendered in a horizontal grid. Nested snapshots are connected with gray edges (without arrowheads).

5.3.2 Enhanced Verticality

As explained in §5.3.1, verticality can have a strong positive effect on the readability of the PDG. That is why "enhanced verticality" is enabled by default. It is implemented by adding heavy invisible edges between every two data or summary nodes in render order (nested nodes of snapshots excluded). That works because in Graphviz's `digraph` layout, heavier edges increase verticality.

Adding edges to the PDG can have a super-linear impact on performance due to the simplex optimization problem belying Graphviz's graph layout implementation [44]. While in case of most DSA programs with small input size, the drop in performance can be acceptable, in some scenarios, things might slow down considerably. We found that in case of the Dijkstra algorithm performed on a graph with 8 vertices and 12 edges, rendering of the fully expanded PDG takes about 0.2s without and 0.35s with enhanced verticality on an i7 laptop.

5.3.3 ControlBlocks and Summary Modes

ControlBlocks are rendered as rectangular blocks with solid white outline and their yellow label in the top left. Clicking the label expands or collapses it. When hovering over it, a menu appears with seven summary mode buttons, allowing even greater control. The root's seven summary mode buttons are in the blue toolbar at the top (parts of which can be seen in Fig. 5.3b).

The first and last summary mode fully collapse and expand the node, respectively. The second mode is called `ExpandSelf`, from the left sets the node to expand only the node itself, but summarize all ControlBlock children. The following buttons expand by 1 to 4 more levels. In these modes, the deepest level is always fully summarized, while all ControlBlocks at higher levels are shallow-summarized. Shallow summarization only

doi:10.6342/NTU202210012

summarizes nodes of the same ControlBlock, excluding its descendants.

5.3.4 "Connected" and "Param" Modes

To reduce clutter, "Connected" Mode is enabled by default. It only shows nodes that have a data dependency on the final output and hides everything else. When dealing with bugs of omission, this mode often needs to be disabled. The "param" button controls whether parameter nodes are shown. We find it useful to disable parameters by default. They can be enabled to help users understand function parameters and where they come from. However, especially in recursive algorithms, they can take up a lot of extra space.

5.3.5 Choosing a Level of Detail

The massive amount of events generated even by simple applications presents a major design challenge standing in the way of widespread adoption of novel developer tools such as this. Visualizing everything can help identify high-level patterns, but makes it inherently difficult to investigate causality. That is why Dbux-PDG provides a multi-resolutional solution. Here, we provide some recommendations as how to navigate it.

The user starts at a fully summarized view, where only data flow between the watched nodes are visible. While the user can choose to expand the full graph, in most scenarios, this view would obstruct comprehension due to the immense amount of detail presented at once. Instead, when looking for details, the user can first investigate the summarized top-level of control flow blocks and expand individual blocks on demand, all the way down to the individual statement level. The user can (and probably should) do this incrementally, one step at a time, as they are honing in on a particular point of interest, or collapse blocks to regain overview of high level patterns. When trying to understand a specific node or block, we recommend considering edge complexity. When encountering too many or crossing edges on a summarized node, expanding the node (or rather its ControlBlock) helps disentangle those edges, as the concentrated subset of summarized edges would be redistributed among multiple nodes.

doi:10.6342/NTU202210012

Table 5.1: Comparison of Seven of JSA's Nine Sorting Algorithms.

name	con	block	sup
Bubble	✓	✓	
Heap	✓	✓	✓
Insertion	✓	✓	
Merge	✓	✓	✓
Counting	✓	✓	✓*
Selection	✓	✓	
Shell	✓	✓	

5.4 Case Study: Sorting

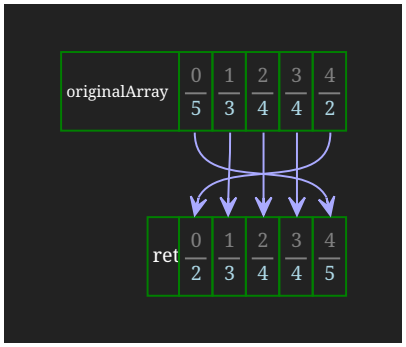
In this section, we pit Dbux-PDG against JSA's nine sorting algorithms because they are a staple in most algorithm classes and textbooks and generally a popular choice for teaching algorithm fundamentals.

As can be seen in Tbl. 5.2, we were not able to run all of JSA's sorting algorithms successfully. Specifically, CountingSort, RadixSort and QuickSort were using `unshift`, `reduce` and `shift`, respectively. These are built-ins that we have not properly traced yet. When manually testing these algorithms, we found that CountingSort is visualized correctly, despite its use of untracked built-ins, and has thus been included in this study. That is because PDG construction can reverse-engineer (or "guess") some data flow by tracking object identity. RadixSort and QuickSort, on the other hand, have incorrect results and were thus excluded. In order to assess the accuracy of Dbux-PDG, we asked three questions of each algorithm:

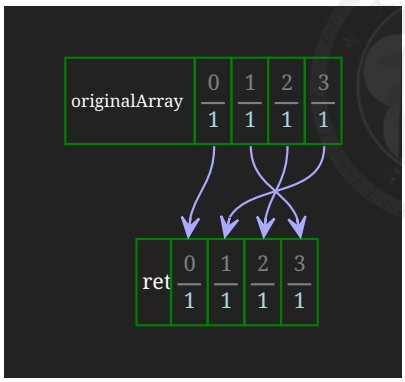
- **(con)** *Are all input and output values connected correctly?*
- **(block)** *Are block summaries correct: do blocks show up and are understandable?*
- **(sup)** *Are support data structures and their operations observable?*

Tbl. 5.1 summarizes the results. For each algorithm, we selected at least four non-empty inputs from the JSA Jest test cases. We then manually inspected and verified correctness of nodes and edges. To illustrate some of the results (and to save space), we use a custom input sample that is somewhat representative of but also shorter than the samples already in JSA.

doi:10.6342/NTU202210012



(a) The fully collapsed PDG of the seven sorting algorithms: they all look exactly the same.



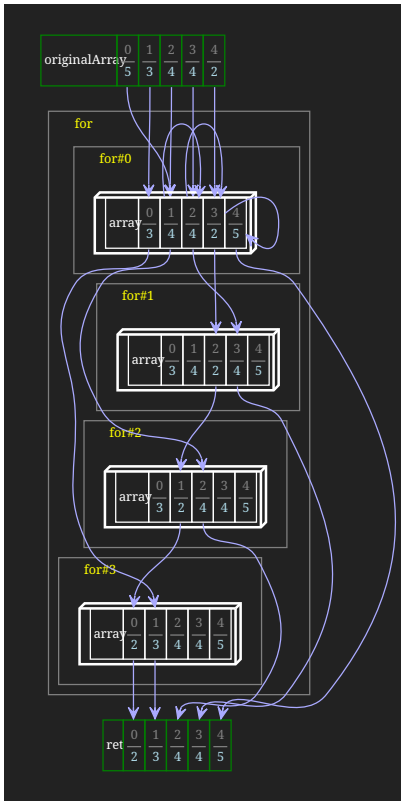
(b) **HeapSort**'s unstable sorting shuffles equal values.

Figure 5.4

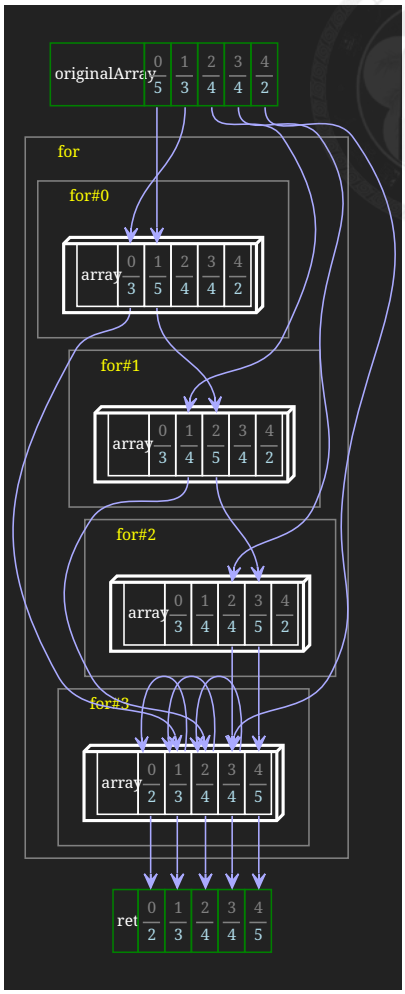
The first column **con** captures whether Dbux-PDG can accurately connect inputs and outputs of these sort algorithm implementations in fully summarized form. Fig. 5.4a demonstrates the fully summarized PDG of *all* algorithms: given this particular sample input, all sorting algorithms are visualized exactly identical in fully summarized form. This speaks to the great visual consistency of our approach, and also data flow visualization, in general.

The column **block** concerns itself with whether Dbux-PDG can accurately capture control flow and place data nodes in the right place. We used the same visual verification process as for **con**, but instead of focusing on the fully summarized view, we switched to views with collapsed and expanded control blocks. Fig. 5.5a and 5.5b compare BubbleSort and InsertionSort at expansion level 1: It can be seen that BubbleSort does most of its work in the first iteration of its nested loop, while only performing a single swap in each following iteration. InsertionSort does the opposite and performs most swaps on its tail. In both cases, the contribution of all single-swap iterations toward the final result can be grasped on first sight. However, the busy iterations have too many edges to easily understand all steps involved. In this case, the user can simply click the group label to expand it. In doing so, the iteration block summarizing four swaps would morph into four inner loop iterations, each with a single swap. The expanded loop block visualizes each step of the algorithm separately, thereby disentangling the edges, while taking up more space.

doi:10.6342/NTU202210012



(a) In **BubbleSort**, at each iteration of the outer loop, the largest remaining value is "bubbled" to its final position. It performs most swaps in the first iteration, followed by multiple single-swap iterations.



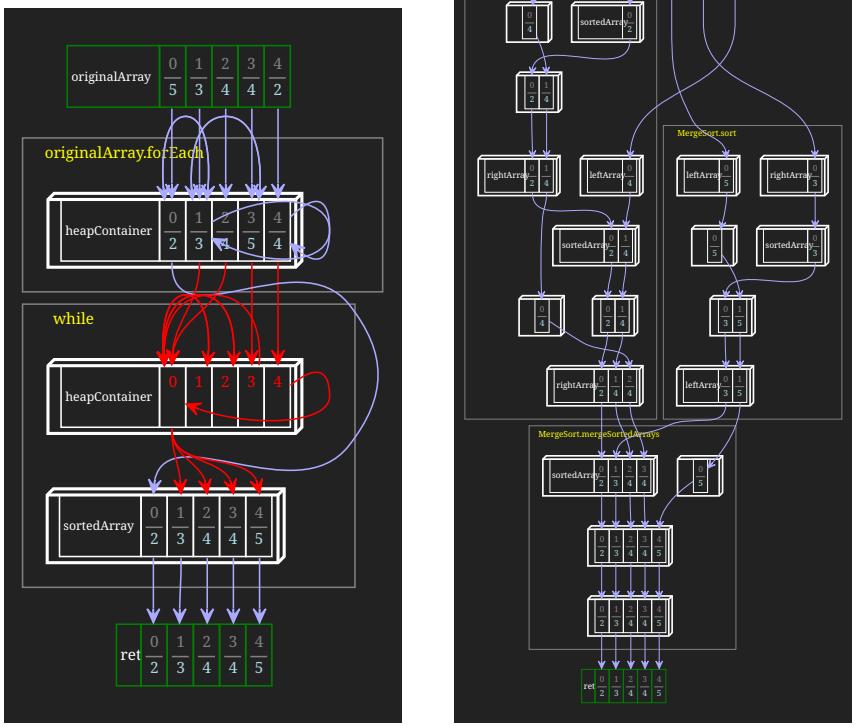
(b) During **InsertionSort**, the first three iterations each only perform a single swap. Incoming and outgoing edges indicate where each changed value came from and where it goes.

Figure 5.5: BubbleSort vs. InsertionSort

Of the seven algorithms, three (Heap, Merge and Counting) make use of support data structures. As indicated by the column **sup**, we found them to be represented accurately. CountSort has an asterisk because its `buckets` are not visible with default settings. The user first needs to turn "connected mode" (see §5.3.4) off to see them.

Fig. 5.6a and 5.6b compare HeapSort and MergeSort. In HeapSort, the heap's backing array (called `heapContainer`) indicates that JSA's HeapSort implementation is not in-place. It has two phases: (i) at the top, an `Array.forEach` loop marks the first phase, during which the heap is constructed. (ii) Underneath it, the `while` loop marks the second

doi:10.6342/NTU202210012



(a) **HeapSort** happens in two phases: heap construction followed by heap deconstruction

(b) **MergeSort** is a classical example of a divide-and-conquer algorithm.

Figure 5.6: MergeSort vs. HeapSort

phase. During the `while` loop, the minimal element is repeatedly queried and removed from `heapContainer`, before being appended to the `sortedArray`. The `while` loop continues until the heap is empty and the result array fully constructed.

MergeSort's JSA implementation (Fig. 5.6b) is not in-place, and instead creates new arrays recursively. They are visible as `leftArray`, `rightArray` and `sortedArray`. When the user expands any of the blocks at the top, the recursive nature of the algorithm reveals itself.

In case of MergeSort, more interesting properties can be explored: For Fig. 5.6b, we decided to turn "enhanced verticality" (see §5.3.2) off. As a result, its divide-and-conquer property, as well as its potential for parallelization, are immediately obvious. The two

doi:10.6342/NTU202210012

blocks at the top mark the “divide phase”, and the final block at the bottom the “conquer phase”. Furthermore, Dbux-PDG reveals how the two blocks of the “divide phase” are not sharing any data dependencies. This important property makes MergeSort an excellent candidate for parallelization.

When “enhanced verticality” is turned on, the blocks of the “divide phase” would appear on top of instead of next to each other, making these properties less obvious. This highlights two important features of Dbux-PDG: (i) it naturally accentuates important algorithmic properties and (ii) it has the potential to make algorithm implementations “fun” to interact and play around with.

Of the seven sorting algorithms, three (Bubble, Insertion, Merge) are stable. **Stability** is a core property of sorting algorithms. While Dbux-PDG cannot be used to prove stability, it can visualize instability in some samples. For example, in Fig. 5.4b, HeapSort’s lack of stability can be inferred from the fact that it shuffles identical values. However, finding the right sample is not always easy. CountingSort, for example, did not reveal its lack of stability in any of the input samples.

doi:10.6342/NTU202210012

5.4.1 Finding Out Why

Dbux-PDG not only visualizes HeapSort's lack of stability. It can also help the user find out the reason why. Fig. 5.7 shows parts of HeapSort for input `[1, 1, 1]`. In the `forEach` block, the heap is constructed. Values are added to the heap's backing array `heapContainer` in order. In `while#0`, it can be seen that the first output value is also the first input value. However, the big arrow in the middle indicates where a shuffle occurs. When the first value is removed from the heap, it is replaced with the third value, instead of the second value, as indicated by the red edge. It is this third value that gets stored in the second output position, thereby violating stability. This demonstrates how the PDG can be used to determine that HeapSort's lack of stability stems from the Heap's property of not maintaining input order.

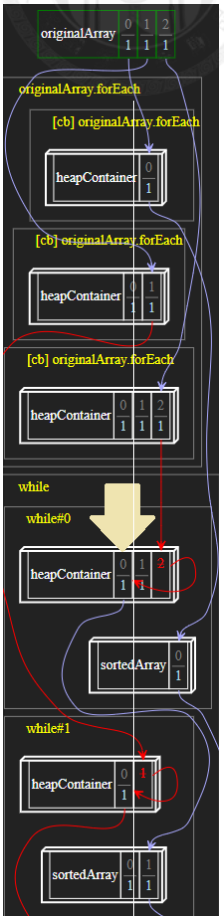
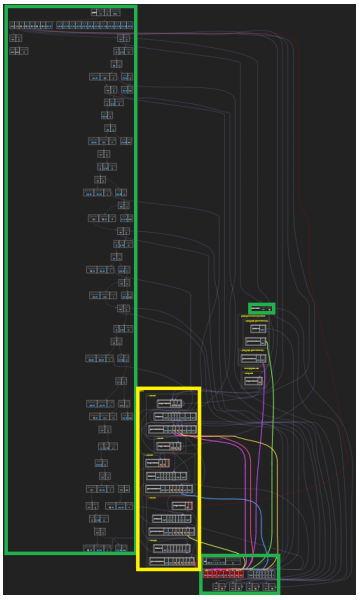


Figure 5.7: Why is HeapSort Unstable?

5.5 Case Study: Dijkstra

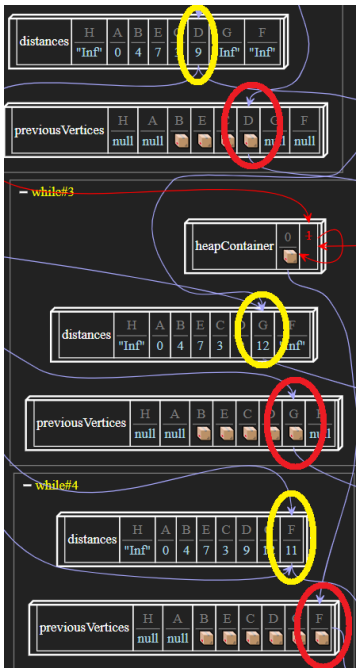
Next, we take a look at a staple graph algorithm: in one JSA sample, `dijkstra` is used on a graph with eight vertices (`A`, `B` …) and 12 edges. Its inputs are the `graph` itself and a `startVertex`. It has two outputs: (i) a `distances` map that contains each vertex's distance to `startVertex`, as well as (ii) a map of `previousVertices` that, for each vertex, contains the vertex that preceded it during shortest-path traversal, thereby allowing reconstruction of all paths.

doi:10.6342/NTU202210012



(a) Zoomed out view of fully summarized Dijkstra execution.

(b) The first iteration finds three vertices and the second none.



(c) The last three iterations each find one vertex.

Figure 5.8: Dijkstra in three parts.

Fig. 5.8a shows the fully summarized view. Outlines were enhanced for clarity. The `while` loop in the middle is outlined in yellow. Three watched nodes are visible, outlined in green. Because the `graph` snapshot (the big green box on the left) is rather large, we need to zoom out to see it in entirety. The graph snapshot is tall because we

doi:10.6342/NTU202210012

are building snapshots in a DFS manner, while BFS would produce more horizontal snapshots. `startVertex` (value is `A`) is the watched node in the top right. The return value snapshot (`ret`) is in the bottom middle. In this screenshot, the user is hovering over the returned `previousVertices` snapshot, inside the return snapshot. This highlights the snapshot in red, as well as all data flow edges from the corresponding vertices that were moved into it.

Fig. 5.8b shows the first iteration of the outer `while` loop. The `while` loop continuously pulls from and adds to a `PriorityQueue`. The queue is not visualized in the PDG because there is no data flow directly involving it. Instead, it internally uses a heap which shows up at the top of each iteration in form of its internal `heapContainer` backing array. We see the same `heapContainer` appear in MergeSort (Fig. 5.6a) because both algorithms use the same heap data structure. In the first iteration, the `startVertex A` was pulled from the queue. Its neighbors are dequeued and added to `distances` and `previousVertices` correspondingly. From Fig. 5.8b, we can see that its neighbors are `B`, `C` and `D`, clocking in at distances 4, 7 and 3 respectively. It might first seem confusing that `A` itself is not making an appearance. That is because it is rendered as a parcel (label generation still needs some improvement). However, we can find out that it is indeed `A` from clicking it, which selects the corresponding event in the code editor. From here, we can easily query its value through Dbux's other tools. Lastly, Fig. 5.8b also reveals a small bug in data flow tracking: it looks like `A` was first moved to `previousVertices.B`, and then from there to `previousVertices.C` etc., even though they all came from the queue (i.e. `heapContainer`).

A second and last look at the `while` loop suffices to wrap up the entire algorithm. We have eight vertices. `A` is the starting point. `H` is not connected, which we can see from the fact that its distance is `Inf`. `B`, `C`, `D` were found in the first iteration. That leaves three: `D`, `G` and `F`. In Fig. 5.8c we see the remaining three iterations contribute each one vertex to the final result. The three vertices are highlighted by red circles, and their respective distances by yellow circles. We ask the reader: how difficult do you find this visualization to read? We hope this helps refresh the reader's knowledge of Dijkstra.

doi:10.6342/NTU202210012

5.6 Quantitative Study

We conclude our evaluation with a study of Dbux-PDG on the `javascript-algorithms` (JSA) repository [126] in order to test its accuracy and resilience on a diverse range of algorithms.

5.6.1 javascript-algorithms

With 140+k GitHub stars, JSA is among the 20 most-starred GitHub repositories of all time. It hosts over 100 algorithms in 11 folders, some with multiple implementations. Many of the algorithms make use of a diverse range of data structures, which are also part of the repository. This makes JSA a great testbed for algorithm visualization tools such as Dbux-PDG.

In order to easily test individual algorithms, we added JSA to the project collection of the `dbux/projects` module (see Fig. 5.1). Clicking a test case, opens the test location in the VSCode code editor. Clicking an algorithm sample opens its PDG. If it was not previously installed, it will clone and install it. If it was not previously executed, it will first ask to run the test case and export the application file. Running most tests with Dbux can take around 10s, while importing feels instantaneous.

5.6.2 Data Collection & Results

We are particularly interested in three parameters, as expressed in the columns of Tbl. 5.2: The **success** column counts algorithms where the Dbux-PDG was constructed successfully and can comprehend important data flow connections, as well as all main control flow blocks in at least two root summary modes (see §5.3.3). The **allModes** column counts algorithms where the above holds, and all seven root summary modes are working correctly. It does not count samples where an error occurs when changing modes (i.e. during summarization), or where one summary mode shows obvious visual problems, such as nodes that were connected in some mode ending up disconnected in another. Lastly, the **con** (short for "connected") column counts how many tested algorithms had their input

doi:10.6342/NTU202210012

correctly connected to their output.

Data collection was performed by automatically generating all PDGs and then engaging in best-effort manual verification of at least three (if available) non-empty "good" samples per algorithm. We define a "good" input set as one with good coverage[2]. Since we have not (yet) added a coverage reporter to help estimate coverage, we again resorted to a best-effort approach. Test cases with empty inputs or those that do not involve an algorithm function with parameters and return value, are excluded.

This data collection process can deal with most circumstances, but some difficult cases require closer attention: Some algorithms have more than one variant (e.g. backtracking and dynamic programming). In that case, we check both, but if either one succeeds, we count them as a success. The `mathbits` group contains 16 bit-related algorithms. We did not include most of them since they are trivial one-liners, despite having manually verified that each one of them is indeed accurately captured by Dbux-PDG. Instead, we only include the five of them that involve at least a loop. We have come across an issue where the PDG works as expected when used inside VSCode, but fails during export. In that case, they might not be fully available in the gallery, but we do count them as a success (e.g. `power-set`). Since Dbux-PDG does not currently deal well with strings, the entire category of strings as well as all other string-based algorithms, such as `caesar-cipher` are automatically categorized as failed.

To explain **con**: in case of certain classes of algorithms, input will generally not be connected to output. That is due to Dbux-PDG's missing dependencies (see §5.7). For example, in case of many generative algorithms, such as `Matrix.zeros`, `pascal-triangle` or `fibonacci`, the inputs might only represent the parameters of generation. These are used in branch decisions, but are themselves not written to the output. In this case, the fully summarized view does not show any connections. The user needs to open up the graph by at least one level, where data flow dependencies of the generative process can be seen, despite not being connected directly to the input. The same applies to search problems. E.g. in `binary-search`, while **con** is false, the traversed indices are visible

[2] `https://en.wikipedia.org/wiki/Code_coverage`

doi:10.6342/NTU202210012

Table 5.2: JSA: Algorithm Results by Folder

category	total	success	allModes	con
cryptography	4	0	0	0
graph	13	8	6	8
image-processing	1	0	0	0
math	31	22	21	15
ml	2	0	0	0
search	4	3	1	1
sets	12	9	5	6
sorting	9	7	7	7
statistics	1	1	1	1
string	8	0	0	0
uncategorized	9	5	4	4
	94	59%	48%	45%

and connected, and thus categorized a success. However, when important connections between data structures are missing (e.g. `fourier-transform`), we do not count it as a success.

Tbl. 5.2 summarizes the results of this study. Of the over 100 algorithms, 94 were included. A 59% success rate for the baseline algorithm and 48% for everything, including summarization is a good start but also leaves room for improvement.

5.6.3 The Dbux-PDG Gallery

We host visual results in the online Dbux-PDG gallery [105]. This should allow anyone to take a first look at Dbux-PDG without having to install VSCode first. The main page lists all algorithms by folder. For most failed results, a simple failure message is provided. For all included samples that executed, multiple PDG visualizations were recorded in Dot format and rendered with `d3-graphviz`. In the gallery, most interaction features (see §5.3) are not available, except for switching between pre-recorded summary modes, pan and zoom.

doi:10.6342/NTU202210012

5.7 Limitations and Future Work

This first version of Dbux-PDG was specifically developed to help comprehend DSA, for the purpose of aiding comprehension and learning. It thus works well on short-running programs with high data flow density. On the other hand, the tool is currently not optimized for **system-wide debugging**. Complex systems employ asynchrony, system calls as well as inter-procedural data flow, all of which the Dbux-PDG does not currently address.

Performance is always a problem with omniscient debuggers. Aside from graph rendering, major performance bottlenecks are generally caused by Dbux rather than Dbux-PDG. In our experiments, we found that most samples ran within one minute, often much faster, especially with caching enabled. The rendering and summarization systems were real-time responsive in all samples we tried.

While the PDG's summarization feature significantly improves understanding of arbitrary dynamic PDGs, **clutter** is still likely to be one of its biggest enemies. It still negatively impacts comprehension, especially in case of loops with many iterations.

As discussed before, not all **syntax and built-ins** are fully traced yet. We also discussed the lack of **missing dependency edges** in §5.2.5. Furthermore, during summarization, control blocks are removed but control dependency edges are not yet inserted in their stead, thereby hiding those control dependencies.

While we already implemented instrumentation of **ES6 destructuring** assignments and parameters to trace their data flow, it is currently buggy. Due to time constraints, we decided to use Babel to convert this type of syntax for our experiments instead.

For this type of research, **user studies** are imperative, and as of now, future work of this project. These studies should investigate whether learners can use Dbux-PDG to (i) answer questions about the program as well as certain algorithmic properties, and (ii) locate bugs in incorrect algorithm implementations.

doi:10.6342/NTU202210012

5.8 Summary

In this work, we present a first version of Dbux-PDG, a system that aims to help learners comprehend and interact with the data flow of DSA implementations. In its fully summarized form, it allows the user to focus entirely on the connections between inputs and outputs. Multiple layers of summarization allow expanding and collapsing regions of interest, revealing the data flow between different variables, memory addresses, functions, loops and, if so desired, individual statements. Integration with a popular IDE and its source editing tools allows the user to locate the code responsible for data flow operations.

We showed how Dbux-PDG can be used on various sorting algorithms and Dijkstra. Important algorithmic properties, such as sorting stability, parallelization potential or changes in data structures over time are commonly part of computer science classes, but generally intangible, and thus difficult to understand by the learner. Dbux-PDG allows interactively exploring these properties. In some cases, the user is enabled to further explore the reason behind some of these properties, which can make things more fun and increase learner engagement.

We conclude that Dbux-PDG shows potential for aiding beginners in DSA program comprehension tasks. However, in order to be used for learning tasks, the right samples need to be hand-picked first, which, as we see in case of CountingSort, is not always a trivial task.

While JS is not the most popular choice for DSA, much of Dbux-PDG is language-agnostic and can be extended to support a greater variety of target languages.

Lastly, the quantitative study shows some first promising results in terms of accuracy and resilience. The testing system provides a rich testing environment for further algorithm tool development, and also allows users to reproduce any of the samples at the click of a button.

doi:10.6342/NTU202210012

doi:10.6342/NTU202210012

Chapter 6

Dbux-ACG: Revealing Concurrency in Asynchronous JavaScript

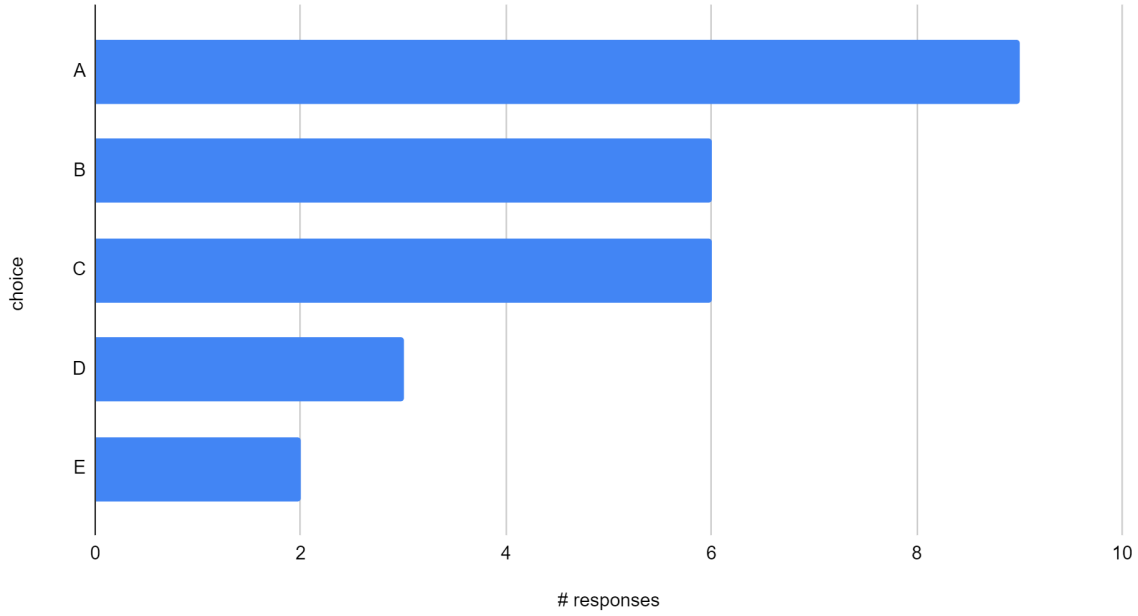


Figure 6.1: Survey Results: What type of programming problems are the most difficult to deal with? (A) Asynchronous behavior (setTimeout; setInterval; Process.next; promise; async/await etc.) (B) Third-party APIs (e.g. Node API, Browser API, other people's libraries, modules etc.) (C) Programming logic (D) Syntax (E) Events.

Dealing with asynchrony (or concurrency, in general) is hard. During a workshop in summer 2020 that introduced Dbux to 20 TAs of a local JavaScript Bootcamp provider, we asked the participants what type of bugs they found most difficult to deal with. The results are shown in Fig. 6.1: ten participants filled out our survey. The top choice for "program-

doi:10.6342/NTU202210012

ming problems" (multiple choice) was "asynchronous behavior" with 9 votes, while the second place only received 6. But asynchrony makes life hard not just for novices, but also catches even seasoned developers off-guard, especially when dealing with unfamiliar code, and proper tools are lacking.

The difficulty arises from the fact that code as innocent-looking as `A; f().then(B);` can be a rabbit hole. A developer might find it cumbersome to determine what happens between `A` and `B` due to (i) arbitrary nesting of promises in `f` and (ii) the difficulty of tracing down forks in `f`. That difficulty is further amplified by the fact that existing debugging tools have very limited support to help deal with asynchrony at all. Perscheid et al. [92] points out "that debugging parallel applications is especially difficult" and posited that they "may require specialized tools and methods not yet available". For example, developers complain how Node.js does not even offer complete asynchronous stacktraces[1] (we will discuss this particular problem further in §6.6.1).

It has become apparent to us that better tools are needed to assist the developer in the inherently difficult endeavor that is comprehension and debugging of asynchronous control flows. We thus propose the novel Asynchronous Call Graph (ACG). The goal of the ACG follows our paradigm of revealing the "dark matter" of debugging (§3.1): It shall make concurrent control flow visible and interactive to the observer.

While the traditional **dynamic call graph** captures the caller-callee relationship between executed files and functions [46], the ACG captures control flow where there is no direct caller: at the roots. As explained in §4.5, Dbux already has a dynamic call graph, with all CGRs vertically stacked on a linear timeline. The ACG, uses additional asynchrony data, to uncover the more meaningful CHAIN and FORK relationships between CGRs.

6.1 Background: Asynchronous JavaScript

JavaScript's non-preemptive property ensures that each context runs to completion without interruption, unless it encounters `await` or `yield` expressions. It uses several types of

[1] `https://github.com/nodejs/node/issues/36126`

doi:10.6342/NTU202210012

asynchronous semantics to deal with the inherent parallelism of modern computers, where multiple system calls (especially IO requests) can be queued and processed at the same time.

6.1.1 Asynchronous Callbacks

Asynchronous callbacks are the oldest solution: in this case, an asynchronous operation is implemented as a function that takes another function as parameter. The parameter function is called a "callback" and will be called upon success or failure of the asynchronous operation. For many types of operations (such as I/O operations), it is assured that the callback will be called exactly once.

A slightly different use-case of callbacks are *event handlers*. JS applications can react to user or system events, such as a mouse click or socket network events, by registering callbacks. These types of callbacks can be invoked an arbitrary amount of times.

6.1.2 Callback Hell

The main issue with callback-based asynchrony is titled "callback hell"[2]. This describes a particularly hard-to-read code structure that manifests itself in form of deeply-nested functions and appears as a side effect of trying to chain multiple operations, i.e. scheduling the next operation after the previous has finished.

6.1.3 Promises

Promises were introduced in ES2015 [4] as a more readable solution to deal with asynchrony. It still uses callbacks, but makes it a lot easier to chain or nest asynchronous operations with them.

Aside from readability, promises also help deal with errors. Whereas the callback-based solution requires manually propagating errors, promises **propagate errors automatically**. A single error handler can be placed to catch all errors of a promise chain.

[2] `http://callbackhell.com`

doi:10.6342/NTU202210012

6.1.4 Promise Chaining

Promise chaining[3] is the solution to callback hell (§6.1.2):
One can call `then(fulfillHandler[, failureHandler])`, `catch(fulfillHandler)` and/or `finally(settleHandler)` on a promise, to ensure execution of the handler callback upon fulfillment, rejection or either of the two, respectively. They each return a new promise, upon which the chain can be continued.

6.1.5 Promisification

Promisification is the process of using the promise constructor to wrap asynchronous callback operations into promises. We refer to an asynchronous callback scheduled from within the context of a promise constructor's executor function as a **"promisified CB"**.

The Promise constructor takes an executor function which in turn is provided two parameters: the `resolve` and `reject` functions which are to be called to settle the promise. The executor function is called synchronously from the constructor.

Promisified callbacks, do not automatically propagate errors. However, an error can be manually lifted into the promise chain by passing it to the `reject` function.

6.1.6 Promise Groups

The promise grouping primitives `Promise.all`, `Promise.allSettled`, `Promise.any`, `Promise.race` allow waiting for a set of multiple promises to fulfill some condition.

6.1.7 Async Functions

Async functions are a more recent addition to the ECMAScript Language Specification (i.e. JavaScript's specification), introduced in ES2017 [5]. They provide syntactic sugar for promises. When an async function is called, the runtime environment creates a new promise which is also the value of the function call. The call value promise fulfills when the async function has run to completion, and, `Promise.resolve(returnValue)` is

[3] `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_promises#chaining`

doi:10.6342/NTU202210012

fulfilled. The promise rejects when the function throws or `Promise.resolve(returnValue)` rejects. Async functions execute synchronously until an `await p` expression is encountered. `await p` interrupts control flow of the function. The async function execution is scheduled to resume from the paused `await` once `Promise.resolve(p)` fulfills, or to throw, when it rejects.

6.2 ACG Fundamentals

The partial ordering relationship between control regions make DAGs an attractive choice for control flow representation, first employed by Allen [20]. The ACG is thus also modelled as a DAG. The ACG is a variant of the dynamic *call graph*[§4.5] where:

1. The nodes of the ACG are Call Graph Roots (CGRs) instead of function calls.
2. The edges of the ACG are **CHAINs and FORKs** (§6.3), derived from asynchronous scheduling semantics, which we mostly capture in form of **Asynchronous Events** (§6.2.2) and **PromiseLinks** (§6.2.5).

6.2.1 Call Graph Roots (CGRs)

Let's recall that the nodes of a *dynamic call graph*[4.5] comprise *all* contexts (i.e. all function executions). Its directed edges span from a caller node to their callees. But what about the "root contexts", i.e. the contexts that have no caller? The traditional call graph does not bother to answer this question. This is where the ACG comes in.

We define **two types of Call Graph Roots (CGRs)**, in terms of the JS engine's *event loop*[4.2.1]. We amend the original call graph definition (§4.5) correspondingly, in order to properly record all CGRs:

1. **Real root contexts**. Any context f_i of a function `f` is a **real context**. It is considered a real Call Graph Root (CGR), if it has no parent caller. This implies that f_i was **directly invoked by the JS engine's event loop**, or in other words, the synchronous call stack is empty. For example, the code `setTimeout(f)` schedules `f` to be

doi:10.6342/NTU202210012

invoked by the event queue at a later point in time. Its context `f` will be a CGR. It is worth noting that in our implementation, not all contexts might be recorded. In that case, the shadow stack is empty, while the JS engine's actual stack might not be. f_i is thus considered a real CGR, if it is the first recorded context without a recorded parent on the shadow stack.

2. **Async function continuation context**: `Async` functions need special attention due to their property of interruptibility. Given an async function h, we refer to its i^{th} execution as h_i. h_i is a **real context**. When executed, we add a **virtual context** h_i^0 as a child to h_i and push it onto the shadow stack. Any expression `await p` tells the scheduler to **interrupt** the current control flow of h until `Promise.resolve(p)` has been settled. Upon interruption, the current virtual context is popped from the shadow stack, but h_i stays on. Once `Promise.resolve(p)` has settled, h_i (and its asynchronous stack) is re-queued and its execution continues upon the next tick of the event queue. When continuing for the k'th time, a new virtual context h_i^k is added as child of h_i. We denote the virtual context h_i^k as the **async function continuation** of the interrupted real context h_i. Since a real context of an async function h_i has the property of spanning multiple CGRs, the ACG ignores it, and solely reasons about its children h_i^k instead. We deduce two rules for the virtual child contexts of async function contexts to be CGRs: (i) The virtual context h_i^0 is considered a CGR, iff h_i has no parent. (ii) Each virtual context $h_i^k, k > 0$ is always a CGR.

We denote the entry point CGR as `R0`. Furthermore, we define two types of overloads of the `cgr(x)` primitive:

1. Given some CGR `r`, `cgr(x) = r = cgr(r)` holds for all contexts and events `x` executed after `r` is pushed onto and before it is popped from the stack.
2. Given a promise `p`, `cgr(p)` denotes the "first CGR" of `p` as defined in §6.2.3.

Lastly, a quick note on built-in asynchronous operations: Since we capture AEs at the language level, built-in asynchronous events and other effects are invisible to us. Hence,

doi:10.6342/NTU202210012

calling a built-in function does not add any CGRs on its own.

6.2.2 Asynchronous Events (AE)

ACG edges are induced by low-level asynchronous scheduling semantics, which we capture in the form of **Asynchronous Events (AEs)**. Any AE `e` has at least two properties:

1. The scheduler `sched(e)` is the event that scheduled `e`.
2. The completion CGR `to(e)` is the CGR that executes upon its completion. Once `to(e)` executed, AE `e` and `to(e)` have a unique 1:1 relationship, which is why we may refer to them as "an AE's CGR" or "a CGR's AE".

We currently capture three types of JavaScript asynchronous semantics: (i) async await, (ii) Promise and (iii) callback semantics. Other languages (such as C#, Java and Rust) support similar semantics. We define their corresponding AEs as follows:

1. An AE `e` of type **CB** is scheduled by a call to an uninstrumented function `f` which takes at least one "callback" argument `cb` of type `function`, e.g. `f(cb)`. `to(e)` is the executed callback context *`cb`*.
2. An AE `e` of type **AWAIT** is scheduled by an `await` expression inside an async function's context *`f`*. `to(e)` is the virtual context representing *`f`*'s asynchronous continuation, starting right after the `await` expression.
3. An AE `e` of type **THEN** is scheduled via `q = p.THEN(f[, g])`, for some promise `p`. We use `THEN` to represent `then`, `catch` or `finally`. We refer to the context of its handler function *`f`* or *`g`* as *`thenCb`*. In this case, `to(e) = cgr(p) = f`.

6.2.3 Promises and CGRs

A CGR is defined as "belonging to", or "owned by" a promise `p`, if the CGR is directly scheduled via `p`:

In case of THEN-type AEs `e`: `p = q.THEN(f)`, we know that `f = cgr(p)` and `p` is ensured to always have exactly one CGR. This is a definition and also a requirement for

doi:10.6342/NTU202210012

the ACG to be accurate. This is only ensured if the Promise implementation of the runtime environment adheres to the A+ specification [14] (specifically requirement 2.2.4), as V8 (and thus many modern browsers and Node.js) as well as other JS engines do. We briefly analyze and discuss the ACG's behavior when this requirement is not fulfilled in §7.4.2.

In case of AWAIT-type AEs `e` and given `p = (async function g() { A; await x; B; await y; C; })()`, `p` is the promise returned by the async function call `g()`. Asynchronous continuation CGRs (e.g. `cgr(B)` and `cgr(C)`) both belong to `p`. In this case, `cgr(p)` is `cgr(A)`, iff g has no parent, and `cgr(B)` otherwise.

In case of promisified CBs, the asynchronously executed callback's CGR also belongs to its promise. E.g., in `p = new Promise(r => setTimeout(() => r(), delay))`, the CGR representing the execution of `setTimeout`'s callback, `() => r()`, is also `p`'s unique CGR. Note that the executor function itself cannot be a CGR since it is executed synchronously with the promise constructor call.

If a promise `q` dynamically (transitively) nests another promise `p`, we also deem `q` to "transitively own" any of `p`'s CGRs.

Note that not all promises have CGRs. For example, `Promise.resolve(x)` has no CGR, and neither does the promise returned by `sleep()` in the first example of Listing 6.4.

6.2.4 Comparing Asynchronous Events

Fig. 6.2 illustrates three different implementations of a `send` function, each using one distinct type of **AE**. In all three cases, the resulting order of operations is:

`openFile` → `readFile` → `sendFile`.

As we can see from this example, the classic **callback**-based solution (right), unlike promises and async functions, requires repeated nesting of functions. That phenomenon is also commonly referred to as *callback hell*[§6.1.2]. Maybe somewhat counterintuitively, **promises** (middle) also involve callbacks, but alleviate the "callback hell problem" through *promise chaining*[§6.1.4]. **Async functions** (left) are generally the most concise

doi:10.6342/NTU202210012

```
async function send(fpath) {
  const file = await openFile(fpath);

  const cont = await readFile(file);


  await sendFile(cont);


  console.log('File sent!');
}
```

```
function send(fpath) {
  return openFile(fpath).
    then(function (file) {
      return readFile(file);
    }).
    then(function (cont) {
      return sendFile(cont);
    }).
    then(function () {
      console.log('File sent!');
    });
}
```

```
function send(fpath, cb) {
  openFile(fpath, function (file) {

    readFile(file, function (cont) {


      sendFile(cont, function () {
        cb && cb();

        console.log('File sent!');
      });
    });
  });
}
```

Figure 6.2: Three implementations of `send`: `openFile` → `readFile` → `sendFile`

of the three, leading to the least lines of code.

6.2.5 Promise Nesting & PromiseLinks

Next, we define the **promise nesting** relationship between a promise `outer` and a promise or set of promises `inner`. We say that `outer` nests `inner`, if settling of `outer` depends on settling of `inner`. In most cases, that is except for **promise grouping**, `outer` is ensured to settle after `inner` has settled. We capture most of these relationships in **PromiseLinks**. In ES2022, we identify 11 types of promise nesting, which we sort into 6 categories:

1. The three promise chaining methods (`Promise.then`, `Promise.catch`, `Promise.finally`) return a promise `outer`. Nesting can be achieved by returning an `inner` promise from inside the `thenCb`.

2. The expression value of an async function call is a newly created promise `outer`. Inside the async function, an arbitrary number of **await expressions** (`await inner;`) can delay settling of `outer`.

3. The async function call promise `outer` also (after having waited for all `await` expressions) nests the **async function's return value**.

4. **Promisification** also allows for promise nesting. Creating a promise via the promise

doi:10.6342/NTU202210012

constructor creates an `outer` promise (§6.1.5). An `inner` promise can be nested by passing it as an argument to the executor's `resolve` or `reject` callback functions.

5. `Promise.resolve`, `Promise.reject` are used to wrap any value or promise `inner` into another promise `outer`.

6. `Promise.all`, `Promise.allSettled`, `Promise.any`, `race` are **promise group** mechanics, allowing for one-to-many synchronization between promises. In this case, the single promise `outer` can nest zero or more `inner` promises (e.g. `outer = Promise.all(inner)`). Each such mechanism has a different synchronization on its nested set `inner` to resolve `outer`. At least one of `inner` are ensured to have settled before `outer` (unless `inner` is empty).

We deem the first four categories, (1) through (4), as **dynamic nesting**: `outer` has its own CGRs. Settling of `outer` can be delayed by dynamically deciding to nest a new or previously existing promise inside `outer`'s CGR(s). The last two categories, (5) and (6) create new `outer` promises, but nesting is not dynamic, and they do not have their own CGRs.

We define a promise `p` as **chained-to root**, iff `p`, or some other promise `q` that transitively nests or chains `p`, is nested dynamically, at root level. We note that the "at root level" constraint is not always ensured. For example, dynamic nesting category (2) does not necessarily nest at root level in case of the first `await` in an async function. Furthermore, categories (2) and (3) do not nest at root level if no previous `await` expression executed in the same async function and the async function execution itself is not a CGR.

6.2.6 ACG Data Collection in Dbux

We had to make several changes to Dbux data collection, in `dbux/babel-plugin`, `dbux/runtime` and `dbux/data` (see §4.2). We modified the `RuntimeMonitor`, and added a **PromisePatcher** as well as a **CallbackPatcher** to capture **AEs** and **PromiseLinks**.

The PromisePatcher monkey-patches all relevant Promise functions. It is responsible

doi:10.6342/NTU202210012

for recording most PromiseLinks, while nesting of async/await required dedicated instrumentation and separate changes to RuntimeMonitor.

The goal of the CallbackPatcher is to keep track of and patch all callbacks. It not only records CB-type AEs for documented system APIs, but also any other uninstrumented function that takes function-typed arguments. This includes Node-API[4] native addons which are commonly used in database APIs such as *sequelize*, among others. Before a function is executed, the CallbackPatcher checks if it has an override, and if so executes that function instead. If the target function has no override, and itself is not instrumented, the CallbackPatcher selectively replaces callback arguments: if an argument is an instrumented function, it will be replaced with a proxy wrapping the original callback. The proxy keeps track of the scheduler↔ callee relationship and forwards all trapped events (e.g. calls, constructor invocations, gets or sets of properties etc.) to it as well.

Since AEs encode semantics of an asynchronous operation, every AE comprises multiple events, each contributing relevant data. We record that data in form of `AsyncEventUpdates` from all three modified entities: RuntimeMonitor, PromisePatcher and CallbackPatcher.

6.3 CHAIN and FORK

This section discusses the ACG's edges and relevant rulesets. Edges are primarily built from *AEs*§6.2.2 and *PromiseLinks*§6.2.5. Given an AE `e`, an edge spans from `from(e)` to `to(e)`. The core problem of ACG construction is this:

> Given a newly recorded CGR `to(e)`, find `from(e)`, and categorize the edge between them into CHAIN or FORK.

CHAINs are to convey a serial flow of events, while FORKs imply a lack thereof. We first try to find a CHAIN. If not found, a FORK with `from(e) = cgr(sched(e))` is inserted instead. In order to find CHAINs, we partition all AEs into two groups with different rulesets **RS**:

[4] `https://nodejs.org/api/n-api.html`

doi:10.6342/NTU202210012

Listing 6.1: CHAIN vs. FORK: `f` has two AEs `E1` and `E2`. `E2` is always a CHAIN, but `E1` might be CHAIN or FORK, depending on the caller. Assume that the example codes ex1-4 are at root-level.

```
async function f() {
  FA
  await 0;  // E1
  FB
  await 0;  // E2
  FC
}

// ex1: E1 is FORK
A; f(); B;

// ex2: E1 is CHAIN
A; await f(); B;

// ex3: E1 is FORK
await g();
function g() { f(); }

// ex4: E1 is CHAIN
let p; h(); await p;
function h() { p = f(); }
```

1. **`RS-P`** is the "promise ruleset". It guides the connectivity rules of all AEs that involve promises. This is explained in §6.3.1.

2. **`RS-CB`** is the "callback ruleset". It concerns itself with all remaining AEs, i.e. non-promisified CB-type AEs. This is explained in §6.3.2.

Lastly, we define **virtual thread** (short: thread) and **chain-subgraph** as maximal path and maximal subgraph consisting only of CHAINs, respectively. We denote a chain-subgraph starting at `cgr(x)` as `G(x)`.

6.3.1 RS-P: Promises

`RS-P` concerns itself with "promise AEs", that are AWAIT- and THEN-type of AEs, as well as promisified CBs. Given a promise AE `e`, `from(e)` is determined by searching along the promise graph. To that end, it (i) searches up and down *promise chains*§6.1.4, (ii) unravels *promise nesting*§6.2.5 relationships, and in case of AWAIT, looks for (iii) CGRs of

doi:10.6342/NTU202210012

```
async function f() {
  FA
  await 0;
  FB
}
```

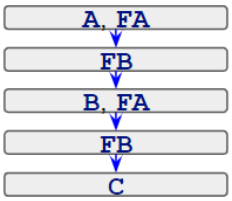
(a) `f`

```
A
await f();
B
await f();
C
```

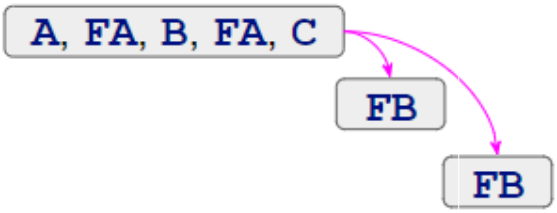
(b) Ex.: Serial ACG

```
A
f();
B
f();
C
```

(c) Ex.: Concurrent ACG



(d) The serial ACG has CHAINs



(e) The concurrent ACG has FORKs

Figure 6.3: These examples illustrate how ACGs use CHAINs and FORKs to make serial and concurrent control flows explicit. CGRs are labeled with lists of all their executed events.

awaited promises, or, if not found, previous CGRs of the same async function. Listing 6.2 illustrates these three types of semantics in three examples. This rule only applies to "new" nested promises, as we will explain in §6.5.1.

The `RS-P` ruleset generally establishes a CHAIN if (1) the owning promise of `to(e)` is *chained-to root*[§6.2.5], or, (2) in async functions if a promise is nested via nesting type 2 or 3 and an `await` has already executed before it.

As an example, let us consider `q = p.then(f).then(g)`. Promise chain semantics imply: $\{R_0 \rightarrow \texttt{G(cgr(p))} \rightarrow \texttt{G}(f) \rightarrow \texttt{G}(g)\}$. The last two edges are ensured to be CHAINs. However, the designation of the first edge depends on whether `q` is chained-to its root *R0*.

Fig. 6.3 shows two more examples. Listing 6.3 further illustrates the "first CHAIN problem" as part of the ruleset in four examples.

We note that CHAINs induced by the `RS-P` rule set propagate errors, meaning that a single error handler can be placed to catch all errors of the same thread. These AEs induce FORKs, instead of CHAINs, iff `to(e)` is the first CGR of a promise chain and all its nesting promise chains.

doi:10.6342/NTU202210012

```
// ex1: promise chain
p
  .then(() => A)
  .then(() => B);

// ex2: promise nesting
p.then(() => {
    A;
    return Promise.resolve()
      .then(() => B);
  });

// ex3: async function
p = (async function f() {
  await ...;
  await A;
  B;
})();
```

Listing 6.2: **CHAIN examples**: Given some promise `p` and non-promise expressions `A` and `B`, all three examples produce an ACG with at least one CHAIN between `cgr(A)` and `cgr(B)` due to promise nesting and chaining semantics.
.

6.3.2 RS-CB: Callbacks

`RS-CB` concerns itself with pure "callback AEs", that are non-promisified CB-type AEs. In absence of heuristics, language semantics provide insufficient information to determine asynchronous control flow semantics of non-promisified CBs reliably. We thus have to resort to two *heuristics* to determine whether they are CHAINed:

- **(h1)** CHAIN all CGRs of the same event handler and
- **(h2)** CHAIN recursive callbacks.

This type of CHAIN does not propagate errors. We have not yet found a better, or more general solution to categorize edges between non-recursive nested callbacks into CHAIN or FORK.

6.3.3 Promisification

After discussing the basics of the rulesets, let's look at two edge case scenarios. The first is *promisification*[§6.1.5]. Promisification adds some complications because it can take

doi:10.6342/NTU202210012

```
async function f() {
  FA
  await 0;  // AE1: ?
  FB
  await 0;  // AE2: CHAIN
  FC
}

// ex1: AE1 is FORK
A; f(); B;

// ex2: AE1 is CHAIN
A; await f(); B;

// ex3: AE1 is FORK
await g();
function g() { f(); }

// ex4: AE1 is CHAIN
let p; h(); await p;
function h() { p = f(); }
```

Listing 6.3: 4 CHAIN vs. FORK Examples: `f` has two AEs `AE1` and `AE2`. `AE2` always induces a CHAIN, but whether `AE1` induces CHAIN or FORK depends on the caller. Assume that the example codes ex1-4 execute at root-level and independent of one another.

any AE, wrap it into a promise, and because of that, modifies its asynchronous control flow semantics.

Sometimes (e.g. in sequelize's `retry-as-promised` library), the Promise constructor is misused to wrap other promises, including async function calls, into a Promise constructor, which generally only adds unnecessary complexity. That is why, for now, we only focus on the most common, and most practical use-case of promisification: wrapping callback-based asynchronous operations into Promises.

Many built-in asynchronous operations in JavaScript follow a common convention for callback signatures. By that convention, the callback function must have two parameters: the first is the `result` of the operation which is provided in case of success, and the second is an `error` which, if provided, indicates that the operation failed. Often, a generic `promisify` primitive, like the one in Listing 6.4, is employed to automatically convert such conventional callback-based asynchronous operation into one that returns a promise. In this case, the asynchronously executed callback's CGR also belongs to its promise. E.g., in `p = new Promise(r => setTimeout(() => r(), delay))`, the CGR repre-

doi:10.6342/NTU202210012

```
// ex1: sleep is setTimeout promisified.
function sleep(delay) {
  return new Promise(
    resolve => setTimeout(resolve, delay)
  );
}

// ex2: generic promisification primitive.
function promisify(operation) {
  return (...args) => {
    new Promise((resolve, reject) => {
      operation(...args, (result, error) => {
        if (error) {
          reject(error);
        }
        else {
          resolve(result);
        }
      });
    });
  };
}
```

Listing 6.4: The Promise constructor takes a single "executor" function argument with two parameters: the `resolve` and `reject` functions are called to settle the promise at a later point in time.

senting the execution of `setTimeout`'s callback, `() => r()`, is also `p`'s unique CGR. The executor function itself cannot be a CGR since it is executed synchronously with the promise constructor call.

6.3.4 Promise Groups

Promise groups (promise nesting category 6 in §6.2.5) can cause "multi-chains", i.e. multiple CHAINs going into and coming out of a CGR.

An example of that is shown in Fig. 6.4: on L10, the result of a `Promise.all` call is chained to its CGR (starting at L9 with `await start()`). It has three nested `inner` promises. The first and final CGRs of the first two nested promises, are chained against their predecessor on L9 (`g1`, `h1`) and successor on L14 (`g2`, `h2`), respectively, causing multiple chains coming out of and going into those CGRs.

doi:10.6342/NTU202210012

6.4 Visualization & Interactions

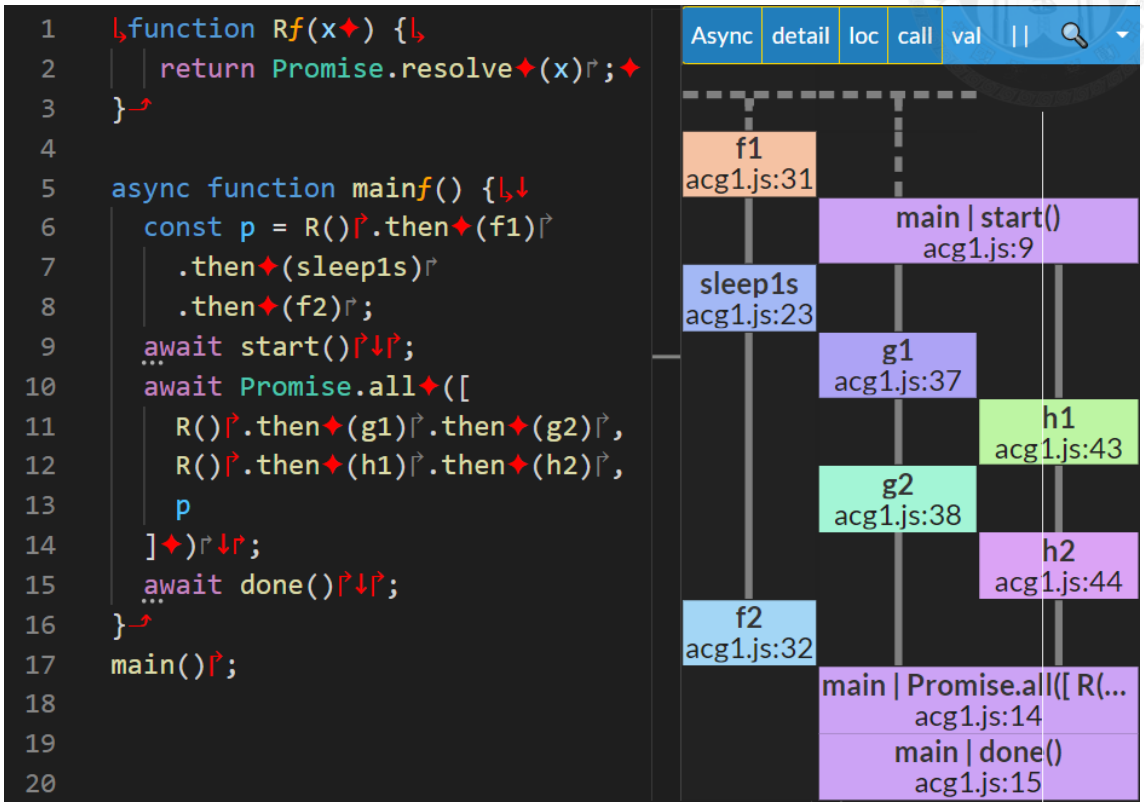


Figure 6.4: Screenshot of a sample program (left) and its ACG (right).

Fig. 6.4 illustrates a piece of sample code, and the ACG of its execution. To keep things consistent, the ACG's (just like the CG's and the PDG's) vertical dimension is time: lower is later. The horizontal axis represents *virtual threads*[§6.3]. The nodes represent CGRs, connected by CHAINs or FORKs. CHAINs are solid vertical lines, while FORKs are dashed non-vertical. Node labels are customizable. The user can use toolbar buttons to choose to display code location, function names and/or selected data values. The first toolbar button allows toggling between ACG and Dbux's original (synchronous) *call graph*[§4.5]. When an executed line of code or expression is "selected" in the editor, "follow mode" automatically highlights and pans to the CGR containing it.

In the sample ACG (Fig. 6.4), we see all CGRs, connected by either CHAIN or FORK edges. In the code (left), the red solid border around the 'done()' call expression indicates that it is *selected*[§4.3]. The yellow solid border around the `await Promise.all(...)` CGR node indicates that the selected event in the code is part of that CGR, because *follow*

doi:10.6342/NTU202210012

mode[§4.5.2] is enabled.

Some **concurrency patterns** are visible in this sample. For example, we can see that `f2` followed `f1`, `g2` followed `g1` and `h2` followed `h1`. The distinction between CHAIN and FORK made several types of concurrency explicit. Specifically, the three separate columns indicate that `f`, `g` and `h` executed concurrently. There are only two FORKs, which are leading from the entry point `basic1.js` to `f1` and the first CGR of `main`, while the rest is CHAINed.

6.5 Inter-Thread Synchronization (with Promises)

We have not yet discussed one crucial aspect of concurrency in JavaScript: inter-thread synchronization. To the best of our knowledge, the literature, in general, has not yet explored this topic in detail.

We focus our discussion on synchronization between promises, rather than all types of threads. If we wanted to solve all of inter-thread synchronization, inter-callback synchronization (that is synchronization between CGRs of `RS-CB`-*type AEs*[§6.3.2]) would also have to be solved. Due to the lack of explicit structure of callback-based asynchrony, that solution would require advanced data-flow analysis, whose level of difficulty would (most likely) warrant its own research. While all of inter-callback synchronization is thus out of scope, we briefly discuss its pre-cursor, inter-thread data-flow analysis, in §6.7.

JavaScript's primary inter-promise synchronization mechanism is *dynamic nesting*[§6.2.5] of shared promises, in order to emulate the semantics of common synchronization primitives. In the following, we exemplify several such primitives and their respective ACGs.

We note that our preliminary synchronization implementation allows the ACG to render red, dashed border around CGRs of the promise that the currently selected CGR synchronizes against, if it is on a different thread.

doi:10.6342/NTU202210012

6.5.1 Nesting “old” vs. “new” Promises

When it comes to the behavior of dynamic nesting, one deciding factor is whether the nested promise is “new” or “old”. This discussion also seems absent from the literature, despite the crucial role this plays in inter-promise synchronization. We define a nested promise as “new” when it was created in the same CGR that it is nested in, and “old”, if it was created in a previous CGR.

We have discussed in §6.3.1, that when a “new” promise is dynamically nested, its CGRs (and those of its transitively nested promises) will be CHAINed into the same thread as the promise it is nested in.

However, when an “old” promise `p` is dynamically nested, we do not consider it part of the nesting thread. Instead, we consider the following possibilities:

- `p` was intentionally nested for inter-promise synchronization purposes (as discussed below).
- `p` might have been nested too late, possibly causing unwanted race conditions.

The last possibility is concerning: if the promise rejects before nesting occurred, default error handling might have already caused unwanted side effects. For example, in Node@15+, an unhandled rejection (in the absence of a global rejection handler) leads to termination of the application. We thus decided that CGRs of “old” promises should never be assumed to be part of the nesting thread, and treated as a case of inter-thread synchronization instead.

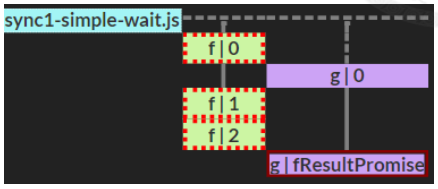
Next, we introduce three types of inter-thread synchronization mechanisms between promises.

6.5.2 Shared Wait

Fig. 6.5 illustrated a “shared wait”: a promise `q` dynamically nests an “old” promise `p`. Note that any amount of promises can wait at the same time, or over time; we thus call it “shared”.

doi:10.6342/NTU202210012

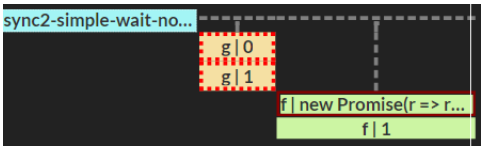
```
const p = f();
async function f() {
  await 0;
  await 1;
  await 2;
}
const q = (async function g() {
  await 0;
  await p;
})();
```



(a) Code for “shared wait” sample (b) ACG for “shared wait” sample

Figure 6.5: “Shared Wait”: `g` waits for and then finishes after `f`

```
let resolve;
async function f() {
  await new Promise(
    r => resolve = r
  );
  await 1;
}
async function g() {
  await 0;
  await 1;
  resolve?.();
}
f(); g();
```



(a) Code for Wait/NotifyAll sample (b) ACG for Wait/NotifyAll sample

Figure 6.6: Wait/NotifyAll: `f` “resumes” only after `g` called `resolve`

6.5.3 Wait/NotifyAll

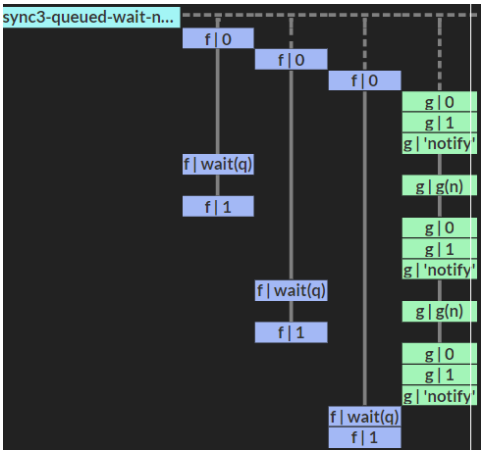
The Promise constructor allows for more fine-grained control. Through it, the Promise `p`’s `resolve` (and `reject`) functions can be accessed and called arbitrarily. Calling `resolve` fulfills the promise and thus notifies any promise currently dynamically nesting `p`.

In the example of Fig. 6.6, `f` waits for `g` to notify it before continuing. This is achieved by using a previously stored `resolve` function as an external `notify` operation. If `f` never executed or if it was already notified, `g`’s `resolve?.()` call becomes a no-op.

doi:10.6342/NTU202210012

```
const q = [];
function notify(queue) {
  const next = queue.shift();
  if (next) { next(); }
}
function wait(queue) {
  return new Promise(
    r => queue.push(r)
  );
}
async function f() {
  await 0;
  await wait(q);
  await 1;
}
async function g(n) {
  if (--n) { await g(n); }
  await 0;
  await 1;
  await 'notify';
  notify(q);
}
f(); f(); f(); g(3);
```

(a) Code for queued wait/notify sample



(b) ACG for queued wait/notify sample

Figure 6.7: Queued wait/notify: each call to `notify` resumes one waiting `f`

6.5.4 Queued Wait/Notify

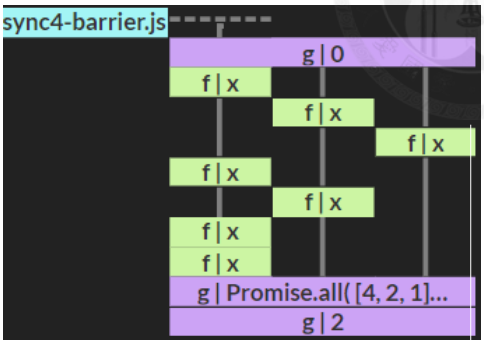
The previous "Wait/NotifyAll" primitive only provides a single `resolve` operation, which "notifies" *all* waiting promises at once[5]. However, often times, we want our "Wait/Notify" implementations to be able to only notify one waiting thread/promise at a time. Sometimes, we have the added wish for the longest-waiting thread to be notified first.

The pair of `wait(queue)` and `notify(queue)` functions in Fig. 6.7 allows for just that. The recursive `g` function calls notify three times in a row, thereby waking up one waiting `f` function at a time. Our current ACG implementation does not yet render these synchronization relationships when selecting a CGR.

[5]Similar to pthread's `broadcast`, Java's `Object.notifyAll` and C#'s `Monitor.pulseAll`.

doi:10.6342/NTU202210012

```
async function f(x) {
  for (; x; --x) {
    await x;
  }
}
(async function g() {
  await 0;
  await Promise.all(
    [4, 2, 1].map(x => f(x))
  );
  await 2;
})();
```



(a) Code for Barrier sample (b) ACG for Barrier sample

Figure 6.8: Barrier: `g` waits for all instances of `f` to complete

6.5.5 Barrier

JavaScript's built-in *promise grouping*[§6.1.6] mechanisms make an explicit barrier primitive easy to implement. In the following example, `g` waits for all three executions of `f` to complete, each waiting 4, 2 and 1 ticks, respectively:

doi:10.6342/NTU202210012

6.6 The Extended ACG Feature Set

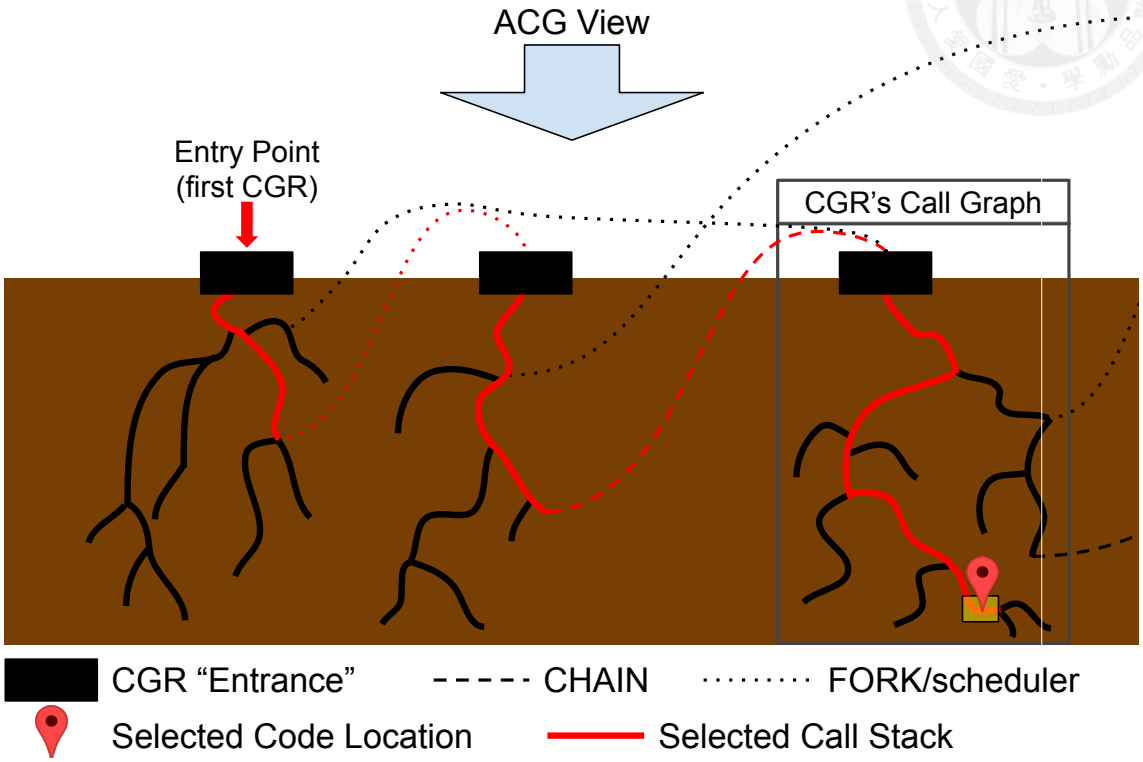


Figure 6.9: Mineshaft Analogy: Execution of an Application

While the ACG itself provides a high-level bird's eye overview over the "roots" of the execution, it is insufficient for most real-world debugging scenarios. Here, we thus introduce several smaller features of the "Extended ACG Feature Set" to complement the ACG in revealing previously intangible asynchronous facts and links.

To better understand the need for an extended feature set, consider the analogy depicted in Fig. 6.9: The synchronous call graph of an individual CGR is one mineshaft, an entangled, long-winded network of paths, stretching deep below the surface. The execution of the program corresponds to depth-first traversal of all CGRs, one-by-one (except for async functions, whose call graphs have suspended subgraphs which will execute at a later point in time). An asynchronous application is actually a complex network of these "mineshafts". The ACG depicts all mineshaft entries on a surface-level map, as well as certain types of connections between them, but **their underground networks are hidden from view**.

At the same time, the Dbux *Trace Details View*[§4.3.1] is the lowest-level flickering flash-

doi:10.6342/NTU202210012

light that illuminates the nearby vicinity of the currently **selected code location** deep inside the mineshaft (i.e. deep inside the synchronous call graph).

We now introduce two more features to help make sense of the "hidden underground networks":

6.6.1 The Asynchronous Call Stack (ACS)

In Fig. 6.9, the Asynchronous Call Stack is visible as the red path that spans from the entry point of the application to the selected code location.

Lack of a proper ACS in error reporting (outside the traditional debugger) has been lamented by developers[6]. One (relatively simple) way to remedy the situation is used (among others) by the Bluebird and longjohn libraries: the former monkey patches promise methods, and the latter well known callback scheduler functions (such as `setTimeout`) in order to keep track of partial stacktraces crossing asynchronous boundaries and stitch them together when required. Sadly, none of those methods handles all sources of asynchrony, and thus, error stack reporting remains incomplete in the absence of debuggers.

We found no discussions of how the ACS supports program comprehension in the literature, aside from a mention by Vilk [128] which neglects to account for promises or async functions (but does mention other DOM-related scheduling semantics).

Luckily, as of 2022, some traditional debugger implementations produce a complete asynchronous call stack, so this is not an unsolved problem. In our tests, Chrome seemed to support all three asynchronous scheduling mechanisms, while Firefox also supports them, but with incorrect labeling. Our ACS implementation produces very similar results to theirs:

(1) We take the entire call stack up to the current CGR's context `r`. (2) Then, we look up the scheduling context `sched(r)`, and (3) repeat from there. But this is not good enough, as we will discuss next.

[6] e.g. `https://github.com/nodejs/node/issues/36126`

doi:10.6342/NTU202210012

6.6.2 Links to All Asynchronous "Parents"

It is important to note that what we refer to as the "ACS" is not actually linear since there can be multiple competing interpretations of the parenting relationships between CGRs. First, consider the synchronous call graph: this call graph is actually a tree where each callee only has one caller, making the synchronous call stack linear and unambiguous. However, when facing asynchrony, an individual CGR can be interpreted as having several types of parents. In the following, we discuss 4 possible types of parent CGRs `t`, given some child CGR `r`:

1. The scheduler CGR `t = cgr(sched(r))`. The scheduler observes a 1:1 relationship with its schedulee, making this the common (and very practical) choice for the ACS.

2. The "CGR that logically executed before `r`" relationship comprises the edges of the ACG. Here, `r` is connected to its previous CGR `t` on the same thread via a CHAIN, if it exists. Else `r` is connected via FORK to its scheduler `t`.

3. Inter-thread synchronization causes two more types of links between CGRs where there is no FORK nor CHAIN between them. The first type is *Shared Wait*§6.5.2. In this case, `r` is the CGR of some promise `p` which dynamically nests an old promise `q`, where `t` is the last CGR transitively nested by `q`.

4. Multiple inter-promise synchronization mechanisms, such as *Wait/NotifyAll*§6.5.3 and *Queued Wait/Notify*§6.5.4, create promises `p` via the Promise constructor and call `resolve` from some other CGR `t` that does *not* belong to the nested promise. Any CGR `r` of a promise `q` nesting `p` now has a dependency on `t`, despite `t` belonging to neither `p` nor `q`.

The extended ACG feature set provides the tools to find all four types of links when any code inside either `r` or `t` is selected.

doi:10.6342/NTU202210012

6.7 Inter-Thread Data Dependency Detection

In this final excursion of this chapter, we want to show the possibilities that can come from combining the powers of the ACG and PDG, and, more importantly, their underlying data structures and APIs.

Let us first recall that, as explained in §6.5, the ACG currently only captures inter-thread synchronization between promises. The key to solving the synchronization problem in general lies in inter-thread data flow. We posit:

> A CGR `r` is identified as synchronizing against CGR `t` if there is no path from `t` to `r`, and there is a variable `v` which is last written by `t`, before it is read by `r`.

We thus propose a small Dbux modification for identifying inter-thread data dependencies, as a pre-cursor to solving the overall synchronization problem. This first prototype finds and displays any matching candidate variable `v` that is written by one and read by another thread afterwards, without enforcing the other constraints. It is less than 100 lines of code, and uses the new ACG's and PDG's data structures, their APIs, as well as some other high-level Dbux primitives.

Output results are ordered by time of first read. Naturally, the algorithm could be extended to identify atomicity violations and other data races.

6.7.1 Inter-Thread Data Dependency Detection: Results

We have tested the detector on several toy problems. The first sample tests the barrier synchronization mechanism for callbacks, as depicted in Fig. 6.10. The results display only `this.waiting`, as expected.

We also tried the detection feature on all three producer-consumer programs (see §7.1). Results are shown in Fig. 6.11. It displayed all expected shared variables. The inter-thread data dependencies of the first two versions (AWAIT and THEN) are identical, even in the order of access (but with different line numbers). The CB variant, in contrast, does not list the (non-existing) `{consumer,producer}queue` used by the other two. The most

doi:10.6342/NTU202210012

```
class CBBarrier {
  constructor(n) {
    this.n = n;
    this.waiting = [];
  }
  enter(notify) {
    this.waiting.push(notify);
    const { waiting, n } = this;
    if (waiting.length === n) {
      setTimeout(() => {
        this.waiting.forEach(
          cb => cb()
        );
      }, 100);
    }
  }
}
```

(a) `CBBarrier` class

```
function f1(id) {
  console.log('f1', id);
  barrier.enter(f2.bind(null, id));
}
function f2(id) {
  console.log('f2', id);
}

let barrier;
(function main() {
  barrier = new CBBarrier(3);

  setTimeout(() => f1('A'), 200);
  setTimeout(() => f1('B'), 400);
  setTimeout(() => f1('C'), 600);
})();
```

(b) Sample Code

```
f1 A
f1 B
f1 C
f2 A
f2 B
f2 C
```

(c) The Console Output

Figure 6.10: The Callback Barrier Sample

interesting result was that it displayed one shared variable for all three versions that we did not know about before: a "`pool`" variable in the `seedrandom` library. Clicking it takes us to the code, allowing us to find out this is a shared global "entropy pool" array, used by the random number generator (RNG). The RNG is called by each agent when computing the amount of "work" (in ticks) for each item.

6.8 Summary

In this chapter, we presented a new method to automatically find an asynchronous ordering relationship between CGRs in form of CHAINs and FORKs, leading to automatic detection of "virtual threads". We used these concepts to construct the ACG, an interactive high-level debugging and program comprehension model and tool.

An extended ACG feature set to help uncover more types of asynchrony-induced

doi:10.6342/NTU202210012

(a) Version 1: P-C AWAIT (b) Version 2: P-C THEN (c) Version 3: P-C CB

Figure 6.11: Shared Inter-thread Data in the Three P-C Implementations

causal links has also been proposed and implemented. Several inter-thread synchronization mechanisms and their effect on the ACG have been illustrated. A novel mini-extension showcases how simple it is to combine APIs of PDG and ACG to uncover even more high-level facts, specifically inter-thread data dependencies.

We note that the ACG shares most of Dbux's own **limitations**. Since it is an Omniscient Debugger, performance is always a major limitation. For now, we can report that, if instrumentation is cached, running our real-world sample programs took between a few seconds to about one minute on an i7 laptop. Furthermore, the ACG is not yet handling asynchronous scheduling induced by dynamic imports, nor workers. Dbux-ACG and Dbux in general are also incapable of assisting in analyzing many types of infinite loop or starvation bugs, such as hapi bug#3347[7], which is also listed by NodeCB [130]. That is because these types of bugs would also starve the network queue, making it impossible for the `dbux/runtime` module to send out any data for analysis.

We believe, this work does not only have the potential to contribute to the understanding of asynchrony in JavaScript, but also other languages, such as C#, Java and Rust, which boast semantics very similar to async/await as well as promises. We proceed to evaluate the ACG in the next chapter.

[7] `https://github.com/hapijs/hapi/issues/3347`

doi:10.6342/NTU202210012

Chapter 7

Evaluation of Dbux-ACG

In this chapter we evaluate the ACG in three steps:

1. Explore asynchronous control flow patterns in a classic concurrent computing problem (§7.1).
2. Use *Debugging Journeys*[§3.4] to test the ACG's utility in dealing with two real-world bugs (§7.2, §7.3).
3. Test whether we can successfully produce ACGs of a multitude of real-world projects (§7.4).

7.1 Case Study: The Producer-Consumer Problem

This section investigates the ACG's ability to capture and visualize concurrency properties of multiple implementations of a classic problem in concurrent computing: the **Producer-Consumer (P-C) problem**.

7.1.1 The P-C Problem

The P-C problem is a simulation with two types of agents: **producers** and **consumers**, each with one action, `produce` and `consume` respectively. All agents share a global resource: a **queue** with limited **capacity** to store **items**. Producers contend for space in the queue to produce one item at a time. Once produced, an item is added to the queue.

doi:10.6342/NTU202210012

Conversely, consumers contend to consume one item at a time. Once consumed, the item is removed from the queue. When an agent cannot perform their action, they must **idle**.

The difficulty lies in managing the queue, subject to the following **constraints**: Multiple producers must not produce the same item, and multiple consumers must not consume the same item. When the queue is full, producers have to idle until there is more space, and when the queue is empty, consumers must idle until there is another item.

Production and consumption times are simulated using a seeded random number generator to assure determinism between consecutive runs. All implementations share the same simulation parameters:

1. There are five agents, 2 producers (Pi) and 3 consumers (Cj). They are started in the following order: P1, C1, C2, C3, P2
2. A total of 9 items are to be produced and consumed.
3. P1 produces 6 items, P2 produces 3 items, while C1, C2, C3 each consume 3 items.
4. `produce` takes 3 to 5 ticks, while `consume` takes 3 to 7 ticks.
5. The queue has a capacity of 2.

7.1.2 P-C Implementations

We implemented the problem three times[1], each using one type of AE exclusively:

1. A: only uses AWAIT-type AEs scheduled via async/await.
2. T: only uses THEN-type AEs scheduled via promises and the `then` method.
3. C: only uses CB-type AEs scheduled via `setImmediate`.

The async version A is implemented by scheduling each next step through a root-level await, in a simple for-loop, which leads to the desired CHAINs. We note that only asnyc/await semantics allows for a non-recursive (non-stack-based) asynchronous loop.

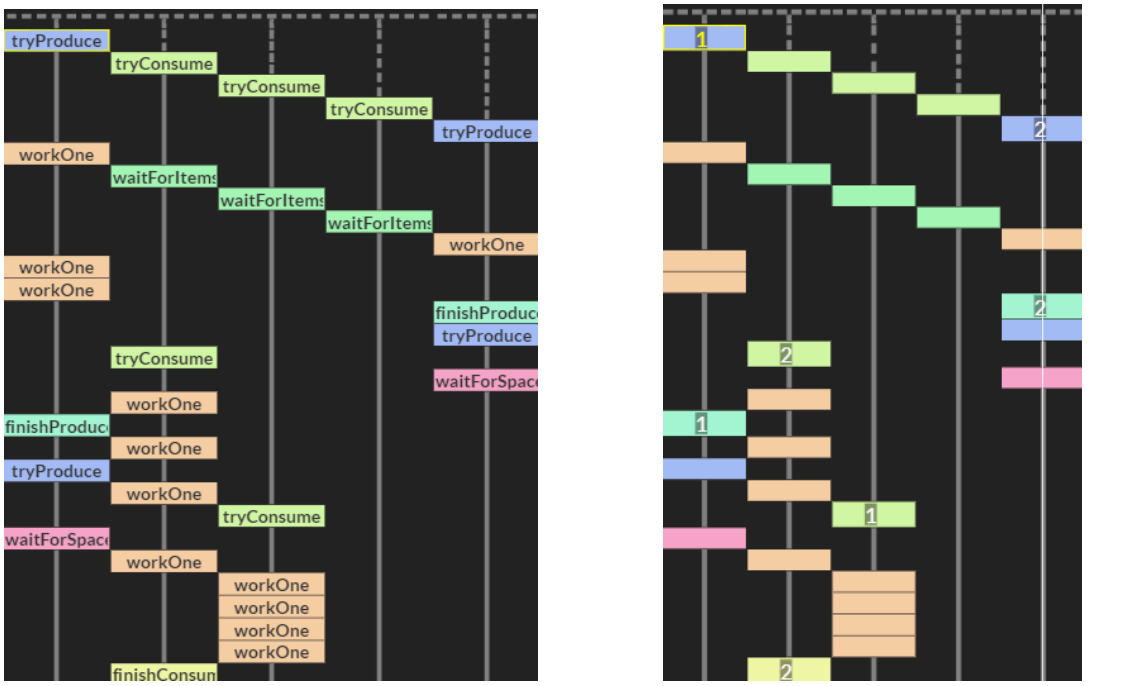
[1]Source code: `https://github.com/Domiii/dbux/blob/master/pub/shared_samples/producer_consumer`

doi:10.6342/NTU202210012

The promise implementation T uses recursive promise nesting. A and T both use two separate queues for the queued wait-notify mechanism from §6.5.4. Producers wait for space on one queue, which is notified by consumers upon consumption. Consumers wait for items on another queue, which is notified by producers when an item is produced.

The callback implementation C uses `setImmediate` to schedule callbacks recursively. This triggers the first *callback chain heuristic*[§6.3.2]. That heuristic requires the root function to always be the same, which is why all nodes have the same exact color. Also, unlike the other implementations, it must use a busy-wait loop, leading to many "empty ticks".

7.1.3 Tracking Agents and Items



(a) View 1: CGR names (b) View 2: using `val` to render work items

Figure 7.1: Two different views of the T THEN-only ACGs

We start by taking a closer look at the THEN implementation, how it works, and what the ACG reveals:

Agents, Threads and CGR names: Fig. 7.1a illustrates the ACG of the THEN-only Implementation T (no annotations, no zoom; cropped for space) with CGR names as labels. Since there is only THEN-type asynchrony, each shown CGR (except for the entry

doi:10.6342/NTU202210012

point) corresponds to the execution of a `thenCb`. Five columns are rendered, each corresponding to one *virtual thread*[§6.3]. We can verify from the CGR names that each thread only contains function calls pertaining to a producer (`tryProduce`, `waitForSpace` and `finishProduce`) or a consumer (`tryConsume`, `waitForItems` and `finishConsume`). Furthermore, each thread captured exactly one agent on a single timeline. The order of threads also matches the order in which agents were started (see parameter 1): P1, C1, C2, C3, P2 — From the CGR names, we know that the first and last thread only produce items, while the middle three only consume.

How to track items: In Fig. 7.1b, we *enabled* `val`[§4.2] to have CGR labels represent the item "used" by an agent, at the start and end of the `produce` and `consume` operations, while all other CGRs are empty. For that, (i) we made sure that the P-C implementation calls a function `useItem(item)` on a given item whenever it started or finished production or consumption. We can then (ii) *select*[§4.3] the `item` inside the `useItem` function. (iii) The first value of that item will then be shown in each CGR.

Tracking items: This view thus allows us to understand exactly which item starts and ends being produced and consumed when, and on which thread. For example, we can see how both producers (left- and right-most columns) start producing their first items, `1` and `2`, on their respective first CGRs. We can also see that item `2` finishes producing first (second occurrence of `2` in right-most column), and subsequently, how the first consumer (second column) starts consuming item `2` right after it was produced; ditto second consumer (third column) consuming item `1` a few ticks later.

doi:10.6342/NTU202210012

7.1.4 Comparing Consume/Produce Patterns with the ACG

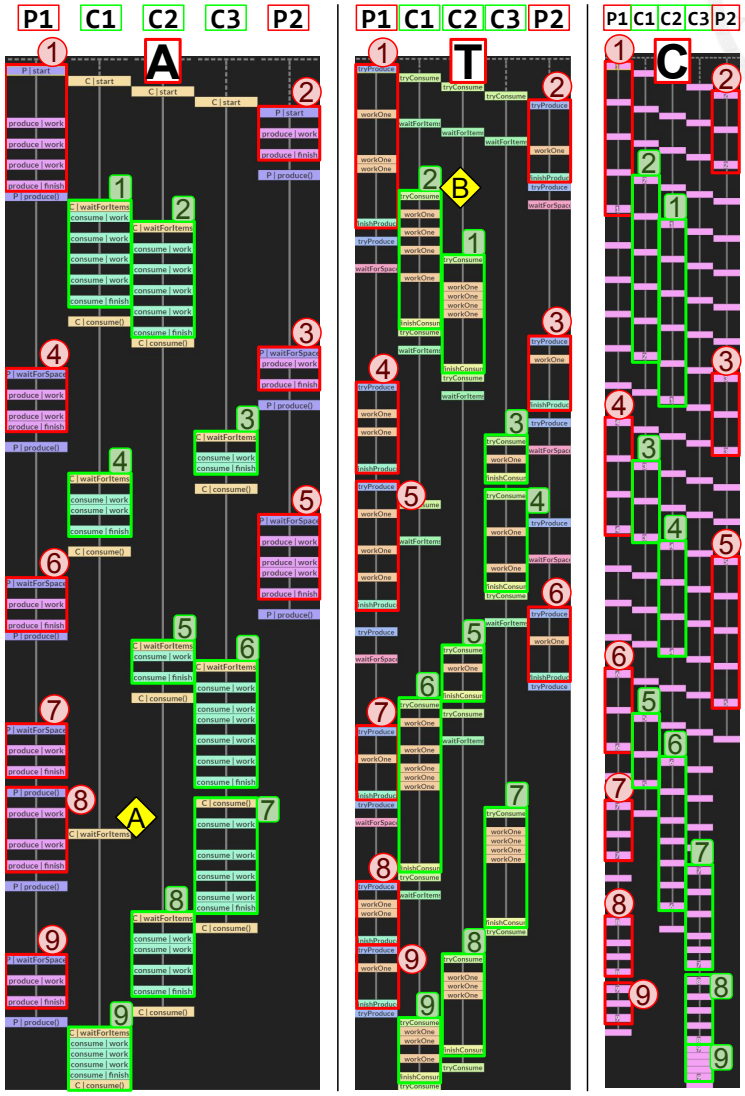


Figure 7.2: Zoomed Out ACGs of the three P-C Implementations: A, T, C

Fig 7.2 shows the annotated ACGs of all three implementations (zoomed out; not cropped): A is the implementation that only uses async/await, T is THEN and C is callback-based asynchrony only. Each thread is labeled with the agent it represents, near the top.

We used the trick introduced in §7.1.3 to manually track the items in all three implementations. In the Figure, we manually annotated all **produce** ticks of one item with a red frame, and the item's id in a red circle (e.g. ①) next to it. Likewise, **consume** ticks of an item are surrounded with a green frame with its item id next to it in a green rectangle (e.g. 1).

This view allows us to track production and consumption of all items. It thus also

doi:10.6342/NTU202210012

allows us to check for race conditions, and, subsequently verify that all P-C constraints (see §7.1.1) have been met: there is no visible violation. All nine items do get produced and consumed. Each item is consumed after production has finished, and there are no duplicate productions or consumptions. Producers idle when the queue is full and consumers idle when it is empty.

7.1.5 Visualizing Concurrent Computing Pitfalls with the ACG

Lastly, we want to highlight two interesting phenomena/pitfalls that are visible on the ACG. They are visibly marked in Fig. 7.2:

Pitfall A: **Notify does not ensure that work is available** (or: "waking up" takes time). Producers call `notify` after having finished producing an item, to wake up an idling consumer. In this case, C1 was woken up, probably by P1 (we cannot tell from the ACG alone) which just finished producing item 7. But waking up takes time. By the time C1 was actually ready, the queue was empty again. That is because C3 already finished consuming its previous item and moved on to consuming item 3 . This "unnecessary wake-up" is visible in the ACG in form of C1's lonely tick, after it idled for a while, and before keeping on idling. As it turns out, we found the authors of the `async` library to have fallen into this pitfall, as we will discuss in §7.2. The pitfall can generally be avoided by adding a conditional branch to check whether constraints on the state (such as "work being available") are still upheld after having woken up, and before trying to just blindly start working.

Pitfall B: **Order of start time of asynchronous operations is not the same as order of end time**. Since asynchronous operations have a start and end time, their ordering is not as simple as it is for instant operations. In this case, P1 started producing item 1 before P2 started on 2. But the ordering of the end of operations is reversed. That is why, C1 first picks up 2 , before C2 starts consuming 1 .

doi:10.6342/NTU202210012

7.1.6 Discussion

It is important to note that this case study is contrived. The different P-C implementations had to be carefully crafted in a specific way in order to produce such clean results. In the real world, one agent might not always (i) precisely comprise a singular thread, (ii) its CGR labels might not be as readable, or (iii) the `val` tool cannot always allow such a clean interpretation of important data points.

But despite being subjected to such limitations, we find that this case study still shows the potential of the ACG, if not at least as a tool for learners to study asynchronous patterns and pitfalls. We now move on to real-world scenarios.

7.2 Debugging Journey: async Bug #1729

We now provide an account of how we used the ACG on two real-world bugs in multiple codebases without prior assumptions or knowledge of said codebases, in form of *debugging journeys*§3.4. We start with issue #1729 of the `async` repository.

7.2.1 Bug Description

We manually selected issue #1729[2] from open issues on the `async` repository in summer 2021. It stood out to us because it looked like (and turned out to actually be) an open order-violation bug in a popular JS library.

[2] `https://github.com/caolan/async/issues/1729`

doi:10.6342/NTU202210012

```
await start();
const q = queue( // create queue
  async (task) => {
    await sleep(10);
    console.log('Task Done');
  });

q.push([]);     // trigger bug
q.push([1, 2]); // add tasks

await q.drain(); // wait
console.log('All Done');
```

(a) `main` function

```
All Done
Task Done
Task Done
```

(b) output

Figure 7.3: Async-js bug #1729

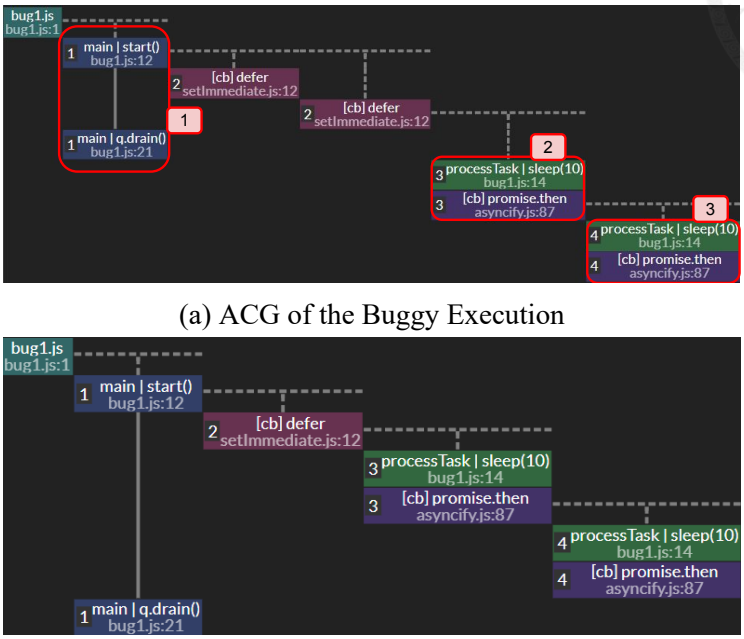
We **reproduced** the bug by copying the code from the issue description, as shown in Listing 7.3a. We did make some minor modifications to accommodate for the library's author's recommendation of not mixing callback- and promise-based asynchrony.

Expected Result: First, the message `'Task Done'` should be printed after each task (twice). The message `'All Done'` should be printed ***last***.

Actual Result: The **output** is shown in Listing 7.3b – `'All Done'` is printed ***before*** the two `'Task Done'` messages.

doi:10.6342/NTU202210012

7.2.2 Preliminary Analysis



(a) ACG of the Buggy Execution

(b) ACG without Bug Trigger (L9)

Figure 7.4: ACG Comparison: Buggy vs. Not Buggy

Let us take a quick first look at the bug-triggering code in Listing 7.3a. We take note of the three high level operations of the code sample: (i) It first creates an asynchronous queue `q` (L2). (ii) Two tasks are pushed to `q` (L9). Finally, (iii) it waits for `q` to finish the two tasks via `await q.drain()` (L11).

The ACG of the execution is shown in Fig. 7.4a. The bug is visible: the last CGR of the `main` function 1 occurred before (i.e. rendered higher than) the CGRs of the two task CGRs 2 and 3 . Even without log messages, we are able to see the bug from here. For clarity, we also provide the correct ACG in Fig. 7.4b, where the bug-triggering line (L9) was removed. Here, the order between `main` and the individual tasks is correct, i.e. the `main` function finishes after both tasks.

Clicking the seemingly misplaced `main | queue.drain()` node 1 takes us to the `await queue.drain()` expression in the code, in L11. From a quick look at the code, we infer that, (i) in order for `'All Done'` to be printed first, the previous `await` must have not waited long enough. We can thus infer that (ii) the promise returned by `queue.drain()` (which is being `await`ed) settled before the tasks completed. That

doi:10.6342/NTU202210012

however, defies the expectation of `drain`, according to the official API documentation. This shows another strength of the ACG: clicking the misplaced node took us directly to the most suspicious line of code. We thus want to start by finding out why the promise of `q.drain()` (L11) settles too early.

7.2.3 Debugging Journey: Goal Path

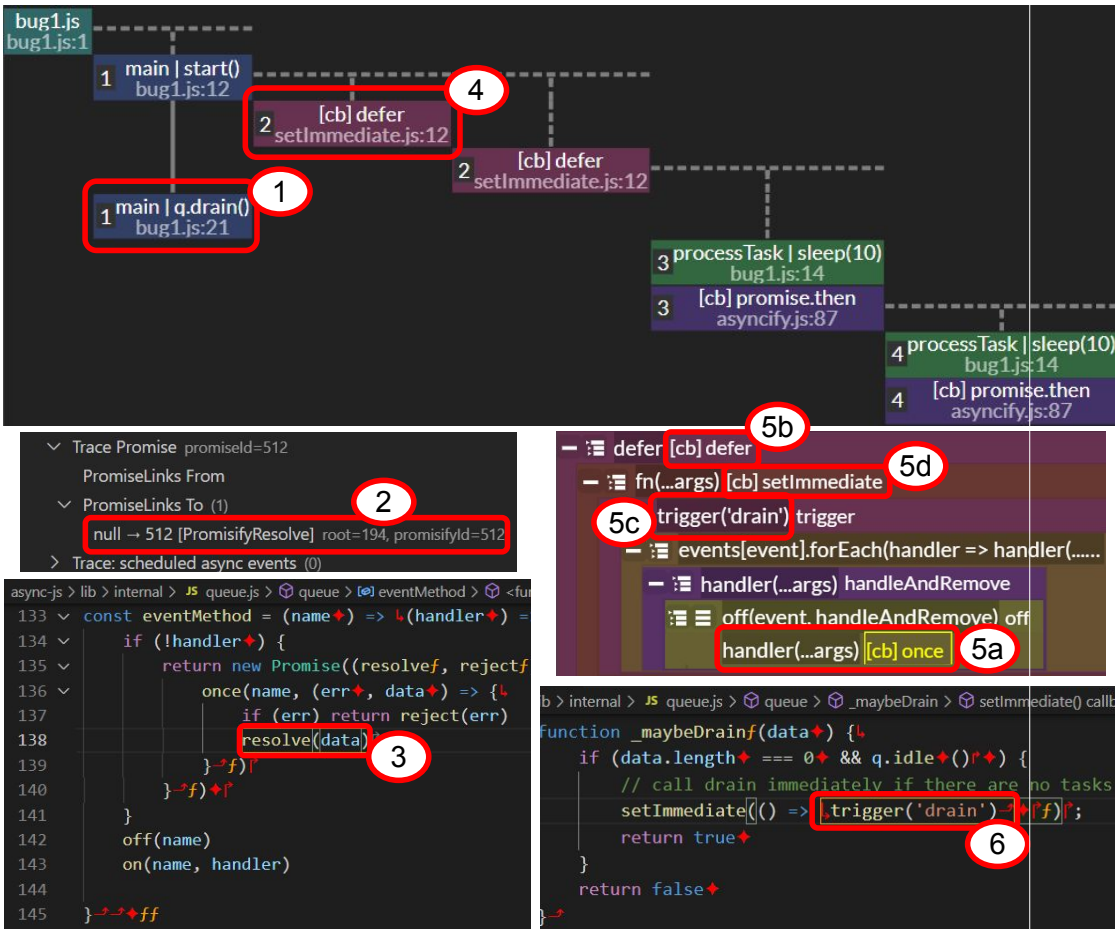


Figure 7.5: All Locations of the Goal Path

Fig. 7.5 visualizes the complete goal path of the debugging journey in 6 major locations. We want to find out why the `q.drain()` promise triggered too early. To that end, we can select the expression in the code, and then use the new "Promise Link" feature ② in *Trace Details*§4.3.1 to jump to the event that settled it. This is an example of asynchronous link type 4 explained in §6.6.2. Clicking the link takes us to the `resolve` call ③ of a promisified callback, created inside a generic event handler wrapper, called `once`.

doi:10.6342/NTU202210012

Now that we have jumped an arbitrary distance to another location in the unknown codebase, we lost our bearings. We thus want to **gain back orientation**. We use the ACG to learn that we are now in the first "`[cb] defer`" CGR (4). Its incoming edge indicates that its FORKed from and executed right after the `main | start()` CGR. This helps us gain an overview, but to actually understand our location in the code, the call stack proves more useful. We thus switch from ACG back to the original Synchronous Call Graph (see §4.5), which also acts as the synchronous call stack. We are in the `once` callback (5a). Its CGR (5b) is visible at the top. Upon scanning the relatively short call stack inside the CGR, we find the next location of interest: the `trigger('drain')` call (5c) (inside a `setImmediate` callback (5d)). This is likely what we are looking for: it might be the code that actually triggers the (too early!) `drain` event. Clicking that call takes us to its corresponding line of code (6), inside the `_maybeDrain` function. This is the bug.

This bug is caused by the first pitfall mentioned in §7.1.5: `setImmediate` is only supposed to be called in a given state (i.e. the queue being empty). In this particular bug, that state is encoded in the condition of the `if` statement right above it. However, due to lack of atomicity, when the callback gets executed at a later point in time, that constraint is not ensured to hold, which causes the bug. One possible solution is to explicitly check again whether the required state constraint still holds after event activation and before committing the requested operation. That is also the solution we have detailed in our bug report[2].

7.2.4 Inter-Thread Synchronization

Lastly, we take a look at why, after FORKing away from, asynchronous control flow ultimately gives control back to the `main` function's second CGR (1). That second CGR of the `main` function (1) is CHAINed to the first CGR because it is the second `await` in an async function, and its nested promise contains no CGRs.

More specifically, `queue.drain()` returns a promise which is nested to that CGR (1). The promise is then resolved (once the queue has "drained") from within a `once` callback at (3), but that callback execution is not a CGR. Instead, the `once` callback is

doi:10.6342/NTU202210012

called from a CGR of a `setImmediate` callback ⑥, which was already FORKed due to the *RS-CB*[§6.3.2] ruleset of `setImmediate`.

That means that the promise does not own nor nest any CGRs and thus is not able to CHAIN other CGRs into the `main` function thread. However, the second `main` function CGR ① is synchronizing against that promise, and thus against the CGR that settles the promise, which is ④. This is thus an example of the *Wait/NotifyAll*[§6.5.3] synchronization mechanism explained earlier.

7.2.5 Discussion

This first real-world case study shows how the extended ACG feature set can be used to quickly find "relatively easy" bugs in asynchronous JavaScript code.

While using the ACG is not always as easy as in this example, we found that it generally provides great value to the analysis process. It aided us in the following ways:

- It visualizes the discrepancy in ordering of CGRs that are symptomatic of the bug. Even without any print or logging, the bug is detectable.
- With a single click, we were able to follow asynchronous link type 4.
- Orientation is provided: after moving to unknown code, we still know where we are on the "map" and on the "timeline", both visualized by the ACG.
- When trying to understand the structure of the CGR that we currently find ourselves in, the synchronous call graph (which also encodes the synchronous call stack) allows us to understand where we are in the "hidden underground networks" (but, in this case, we could have used the ACS to the same effect).

In the next subsection, a more complex scenario is investigated.

7.3 Debugging Journey: Sequelize Bug #13554

In this second case study, we explore `sequelize`, a popular ORM library with asynchronous database operations. We set out to follow up on the already fixed atomicity vio-

doi:10.6342/NTU202210012

lation bug #1831[3] from 2014: back then, the composite `findOrCreate` operation did not use a transaction to protect atomicity between its two child operations, `find` and `create`. That has been fixed since. However, when we tested multiple concurrent `findOrCreate` operations in the most recent version, we discovered two new problems that we also reported on their GitHub page[4]. This bug is not only convoluted but also happening in a much bigger codebase. Hence, this, unlike the previous, section features multiple preliminary analyses, followed by a narrowing down of the debugging goals, before actually starting the debugging journey.

7.3.1 Bug Description

```
try {
  await sequelize.sync();
  await Promise.all([
    User.findOrCreate({
      where: { id: 1 },
      defaults: { x: 1 }
    }),
    User.findOrCreate({
      where: { id: 1 },
      defaults: { x: 2 }
    })
  ]);
}
catch (err) {
  console.error('####### FAIL', err);
}
```

(a) Simplified Pseudocode for the sequelize bug

```
Executed (default): DROP TABLE IF EXISTS `users`;
Executed (default): DROP TABLE IF EXISTS `users`;
Executed (default): CREATE TABLE IF NOT EXISTS `users`[...]
Executed (default): PRAGMA INDEX_LIST(`users`)
Executed ([...]240008): BEGIN DEFERRED TRANSACTION;
Executed ([...]240008): BEGIN DEFERRED TRANSACTION;
Executed ([...]240008): ROLLBACK;
####### FAIL [...] SQLITE_ERROR: cannot start a transaction within a transaction
  at Query.formatError (...\sequelize\...\query.js:403)
  at Query._handleQueryResponse (...\sequelize\...\query.js:72)
  at Statement.afterExecute (...\sequelize\...\query.js:238) {
    [...]
    sql: 'BEGIN DEFERRED TRANSACTION;'
}
Executed ([...]240008): SELECT `id`, `name`, `age`[...]
Executed (default): INSERT INTO `users` [...]
Executed (default): COMMIT;
```

(b) Output (with some omissions)

Figure 7.6: Sequelize Bug Code & Output

We **(re-)produced** the bug by using our own code, as shown in Listing 7.3a, with the SQLITE `dialect`.

[3] https://github.com/sequelize/sequelize/issues/1831
[4] https://github.com/sequelize/sequelize/issues/13554

doi:10.6342/NTU202210012

Expected Result: Multiple concurrent `findOrCreate` operations should work as expected, and with atomicity guarantees. However, after seeing the error message ("`SQLITE_ERROR: cannot start a transaction within a transaction`"), we tamed our expectation. As it turns out, SQLITE does not support multiple transactions at a time on the same connection. We thus expect instead that `sequelize` should either be able to serialize the operations, using an async queue (e.g. the one we looked at in §7.2), or at least deal with the error gracefully, and provide a helping hand in resolving it.

Actual Result: The obvious problem is that the second `findOrCreate` call errors out. However, upon further investigation, we found more relevant results, as discussed below.

7.3.2 Preliminary Analysis 1: Understanding the FORKs

We started by running the sample code of Listing 7.6a. The resulting ACG is shown in Fig. 7.7, on the left.

We proceeded with assessing the application's real degree of concurrency. The two concurrent `findOrCreate` calls make us think that it should ideally only have two threads (we do not imply that number of threads is not the same as degree of concurrency).

At first inspection, we found it curious that despite primarily being connected by ASYNC- and THEN-type AEs, the ACG, depicted in Fig. 7.7 (left), boasted 14 FORKs, despite having an assessed degree of concurrency of only two. We would expect less.

Upon closer investigation (i.e. manually checking all FORKs), we found that 10 of these 14 FORKs were due to the `retry-as-promised` library. The many purple CGRs (labeled with `retry`) following many FORKs are visible on the left. As it turns out, the FORKs occur because `retry-as-promised` employs a type of promisification that the ACG currently does not properly recognize (as briefly hinted at in §6.3.3). The library promisifies an async function call, which is an unnecessary promisification since async function calls already produce a promise.

Fig. 7.7 (right) shows the same program after editing 13 lines of that library to remove the unnecessary promisification. In the new version, only 4 FORKs remain. The resulting

doi:10.6342/NTU202210012

ACG captures asynchronous control flow much more clearly, with less clutter. For example, the initialization routine before line 2 is now accurately represented by a single thread.

7.3.3 Preliminary Analysis 2: High-level Deconstruction

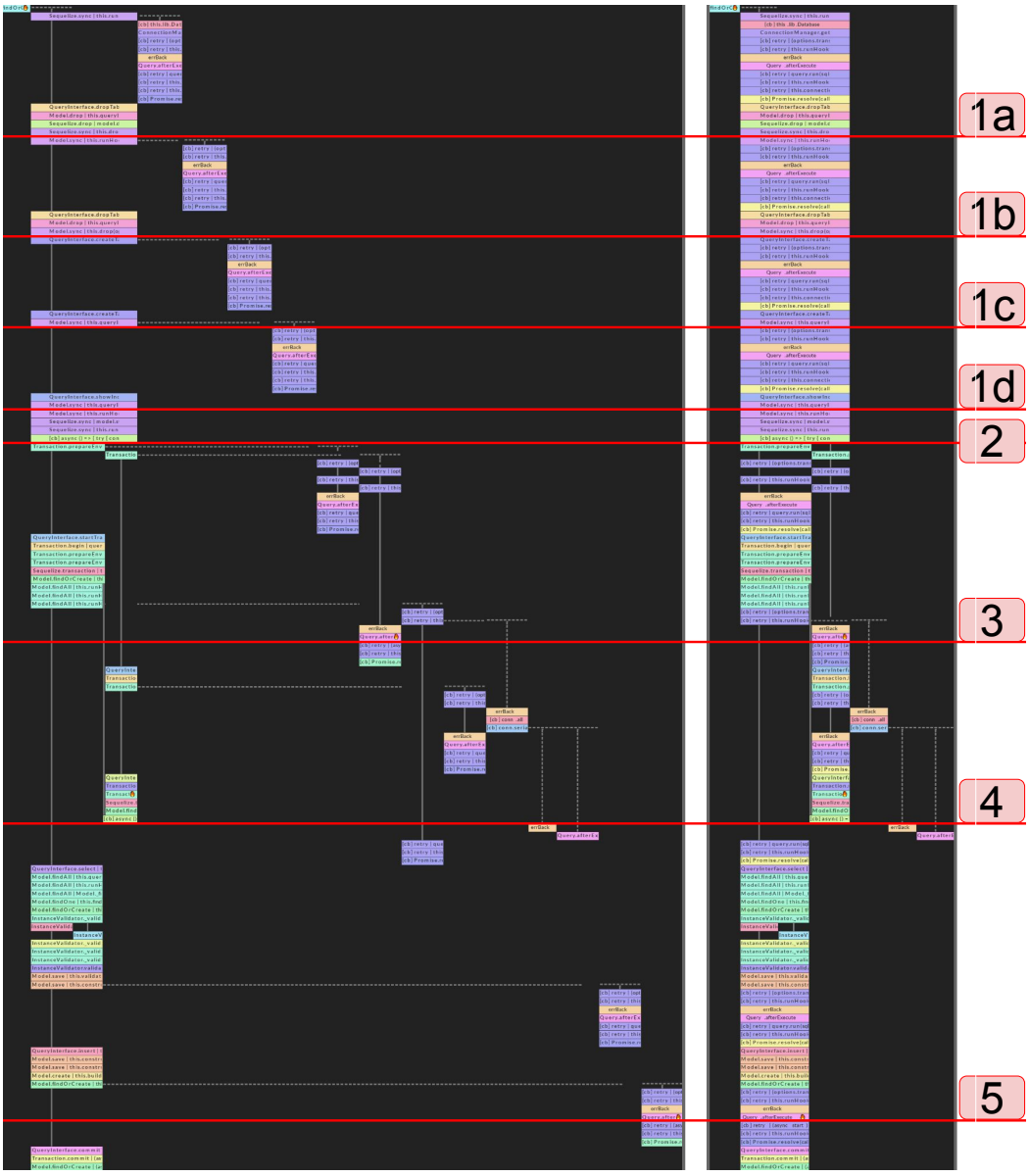


Figure 7.7: The ACGs of the two sequelize implementations. The right-hand side removed an unnecessary promisification in the `retry-as-promised` library.

Both ACGs, depicted in Fig. 7.7, have 141 CGRs, more than ten times the size of the previous sample. In face of such complexity, we assess that a little bit of planning and gaining an initial overview can vastly improve the debugging experience moving forward.

doi:10.6342/NTU202210012

We thus started by first getting a sense of the high-level structure of the asynchronous operations, so that we will be able to stay oriented during the debugging journey. To that end, we start by deconstructing some high-level "landmarks" from the ACG:

The CGR colors between the four lines 1a-d have a somewhat repeating pattern. Upon clicking into the relevant nodes, one can quickly learn that each of these groups represents a query to the database. These four queries are part of sequelize's initialization routine: `sequelize.sync()` which starts in the entry point CGR and concludes at horizontal line 2.

Right below that line, both ACGs widen: clicking each of the top nodes tells us that these are the two `findOrCreate` operations starting in parallel.

There is a flame icon on the CGR at horizontal line 3. Clicking it allows verifying that this marks the occurrence of the first bug that is also visible in the output (Listing 7.6b): the second `findOrCreate` operation tries to create a second transaction while there is already a transaction active, but SQLITE does not support multiple active transactions on the same DB connection.

Line 4 marks the settling of the `Promise.all` promise (let's call it `p`). However, that is not the end of the application's journey. The ACG helps us see through a classic misconception of asynchronous JavaScript: `Promise.all` (unlike `Promise.allSettled`) does not wait until all its nested promises have settled. Instead, if any nested promise failed, the result promise `p` settles early, and all remaining unsettled promises are left "dangling", without anyone awaiting their settling.

In this code, the failure of the second `findOrCreate` operation leads to `p` settling early, and the first `findOrCreate` operation, which has not yet concluded, "dangling". That is why we can see its thread growing past line 4.

Line 5 marks another CGR with a flame icon: here, **the first `findOrCreate` operation errors out**, despite the output **not reporting** the error.

Our preliminary inspection of the ACG thus allowed us to gain a general sense of the high-level operations, and their representation in the ACG. Furthermore, we found that line 5 marks an unknown error that was not reported in the output. We proceed to

doi:10.6342/NTU202210012

investigate that error.

7.3.4 Preliminary Analysis 3: Analyze Unknown Error

Figure 7.8: Investigating the Hidden COMMIT Error: Complete ACG (left); zoomed in partial ACG (top right); ACS (bottom right)

Fig. 7.8 details our investigation of the previously unknown error (visible at line 5 , Fig. 7.7).

After clicking the flame icon in the ACG (1), the *Value feature*[§4.3.1] reveals the error message: "`SQLITE_ERROR: cannot commit - no transaction is active`". Because this error is reported as part of the first `findOrCreate`'s thread, we know that this is the first transaction being unable to `COMMIT`. However, when checking the database, we see the first operation's data inserted. We thus hypothesize that the operation was successful, but the transaction did not survive, meaning that the atomicity guarantees of 2014's fix (#1831) did not apply. The suspicion is further confirmed by the fact that the query log reveals the `COMMIT` action's connection id to be `default` (last line in Listing 7.6b), which indicates that it was executed outside a transaction.

We thus now re-define the bug: **atomicity guarantees are violated**.

After having found the main issue, we want to check if the `Transaction.commit`

doi:10.6342/NTU202210012

function itself needs repair. Fig. 7.8 shows how we could easily find and inspect it: After having clicked the flame icon in the ACG ①, we open the ACS. The error-throwing function is at the bottom ②. A quick scan of the stack reveals the start of the `commit` operation ③. Clicking it takes us there, and also selects the corresponding CGR in the ACG ④. The code is simple: send the query to the database, then call `cleanup()`. There is nothing suspicious here warranting a fault location. To explain: This last part (checking `Transaction.commit`) was a short tangent with no result, other than giving us a sense of comfort. We believe these type of tangents play an important role in the debugging process: "checking simple facts along the way".

7.3.5 Debugging Journey: Setting a new Goal

As we pointed out above, we cannot (easily) fix this bug s.t. both transactions can run concurrently, due to SQLITE not allowing more than one transaction at a time. We should instead aim to make sure that, at least, the second transaction's failure does not affect the first, and thereby making sure that the atomicity guarantees does not get violated. Our new **Expected Result** is thus for the `COMMIT` command to not fail.

doi:10.6342/NTU202210012

7.3.6 Debugging Journey: Goal Path

Figure 7.9: All Locations of the Goal Path

Fig. 7.9 visualizes the goal path of this debugging journey in 7 major locations. We start by assuming that the first error inside the **second** `findOrCreate` is a potential cause for the unwanted side effects. We thus proceed by clicking its flame icon in the ACG ①, which takes us to the location where the error was thrown. The ACS should reveal the operation that caused it. When opened, the error-throwing function ② is at the bottom. Scanning the stack reveals the `Transaction.begin()` function ③, inside the `Transaction.prepareEnvironment` function ④a. Clicking its caller button takes us to the relevant location in the code ⑤ and highlights the CGR in the ACG ④b.

A good look at the code reveals the problem. Even though the transaction fails to start, it sets the shared `connection.uuid` variable to itself in L121 ⑥. Since the trans-

doi:10.6342/NTU202210012

action never actually starts, it should not have this side effect. Finally, `rollback()` gets called in L128 ⑦. A quick investigation of the `rollback` function (not shown) confirms that it rolls back the currently active (i.e. the first, not this second) transaction. We thus conclude/hypothesize that L128 ⑦ causes the first `findOrCreate` operation to lose its atomicity guarantees. We thus implement the following solution:

1. Fix ⑥: The shared `connection.uuid` should only be set AFTER the transaction has started. So we move `this.connection.uuid = this.id;` to after `begin()`.
2. Fix ⑦: `ROLLBACK` should not be executed if the current transaction is not this transaction. So we protect the `rollback()` call with `if (this.connection.uuid !== this.id)`.

Re-running the program verifies that this fixes this bug. We note that this does not assure correctness of the bug fix in general, as we have not run further verifications. But at least this shall increase our confidence regarding the identified locations ⑥ and ⑦.

7.3.7 Discussion

In this study, we experienced several more benefits that the ACG brings to the debugging process:

- We posit that the ACG's ability to uncover the connections between the CGRs and their asynchronous events eases one major source of frustration of the investigative process. Being able to not only see, but also navigate directly to the code that created, triggered or is otherwise related to each CGR **simplifies comprehension of asynchronous control flow at the highest level**. Understanding it has become a more straight-forward, interactive and exploratory (rather than a purely cognitive) endeavor. This is the main reason behind us being able to identify and recognize important "landmarks", as explained in §7.3.3.
- The ACG **visualizes and contextualizes all errors** (by "putting it on the map"). If an error occurs that was not caught by user code, the JS engine should report it. However, that does not always happen. E.g. the COMMIT error (§7.3.4) is not re-

doi:10.6342/NTU202210012

ported because it is the second rejection from a promise wrapped with `Promise.all`. Muting of follow-up errors is one of multiple fallacies of `Promise.all`.

- We find that when investigating and jumping between different locations in the code, having access to a spatial layout of the entire execution makes it significantly easier to **recall** places we visited in unknown code, when compared to the alternative of only memorizing symbol names, file names and file layout.
- The **ACS** proves invaluable in quickly finding important functions up the call chain, which also can be interpreted as "triggers" of the events that are currently being inspected.

7.4 Quantitative Study

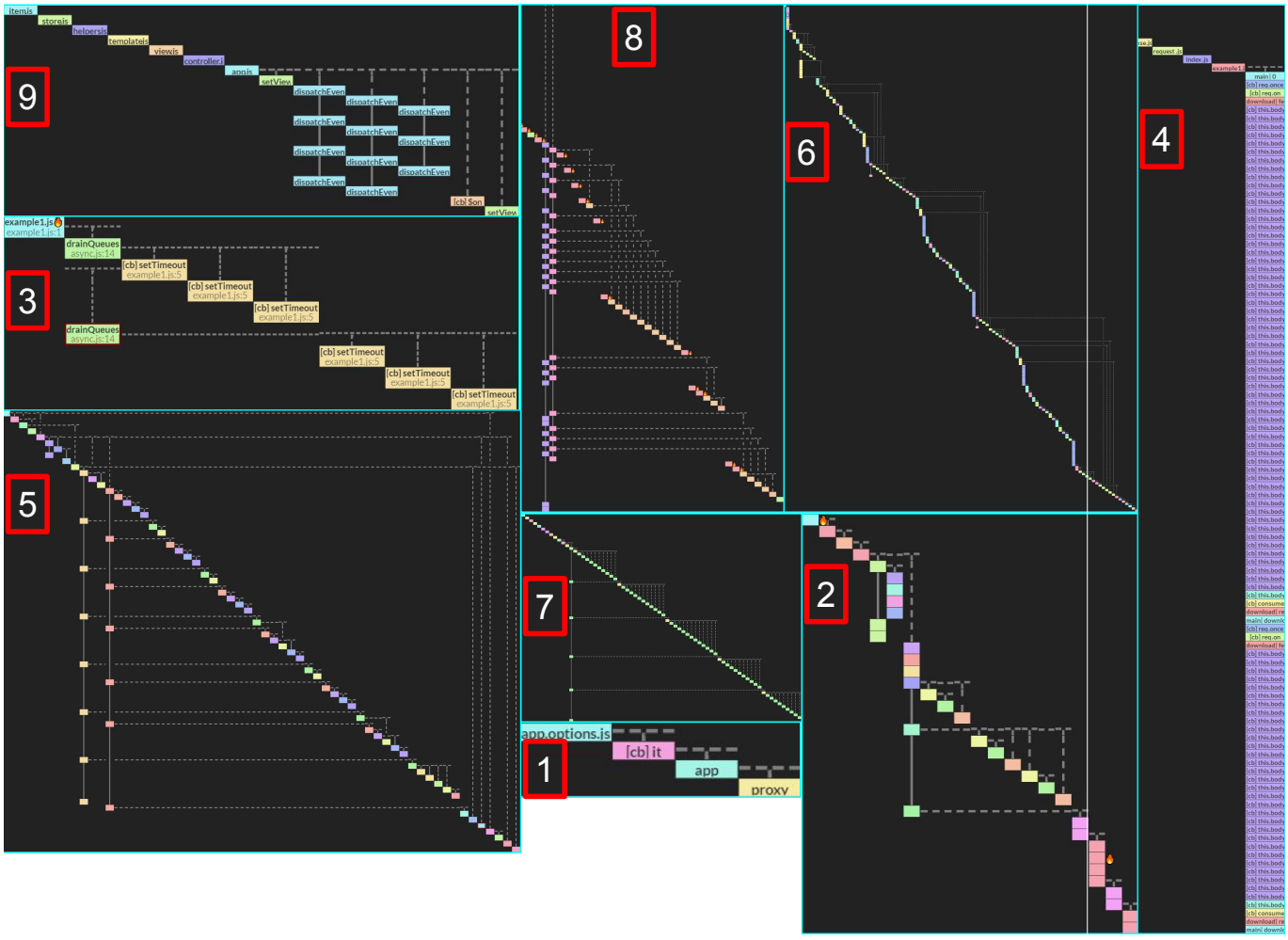


Figure 7.10: Nine out of the eleven Sample ACGs

doi:10.6342/NTU202210012

Table 7.1: **Projects** #: Number of depiction in ACG. L: size of the trace log (number of traced events). A: AWAIT AEs. T: THEN AEs. CB: CB AEs. C: CHAINs. F: FORKs. O: Orphan CGRs. TT: Total threads created.

#	project	L	A	T	CB	C	F	O	TT
1	express	6,772	0	0	3	0	3	1	4
2	hexo	775,329	0	13	22	16	19	1	20
3	bluebird	21,003	0	3	6	1	8	1	8
–	async-js	2,825	4	2	2	3	5	1	6
4	node-fetch	15,999	7	2	94	102	1	8	2
–	sequelize	519,893	109	10	23	138	4	1	6
5	socket.io	161,946	12	0	61	14	59	1	60
6	webpack	1,579,621	10	12	198	114	106	1	107
7	2048	56,669	0	0	65	5	60	10	61
8	Editor.md	164,249	0	0	60	32	28	30	29
9	todomvc-es6	4,355	0	0	14	8	6	7	7

We conclude our evaluation with a quantitative study of Dbux-ACG on eleven diverse real-world projects. The goal is simply to make sure that the ACG works and is (mostly) correct. Best effort manual inspection verifies that, indeed, these ACGs are all correct, barring some small rendering issues.

Numerical results are summarized in Tbl. 7.1. All ACGs are depicted in Fig. 7.10 (already presented ACGs of async (§7.2) sequelize (§7.3) are omitted). Some are scaled more than others. Some have smaller ACGs have `details` enabled. Some have some file threads cropped.

Projects were selected as follows: We selected the first two entries of Tbl. 7.1, express and hexo, from BugsJs [49]. The next six were selected because they are popular open source projects which we knew also were inherently asynchronous, and run on Node.js. Bluebird was specifically selected for a brief case study, detailed in §7.11. The next three were found by searching GitHub for popular frontend projects. For every project, we either used an existing bug (hexo, express, sequelize, async) or constructed a simple example application. In case of frontend applications, we also manually interacted with the UI to trigger event handlers. The ACG investigated for sequelize was the one that followed editing the `retry-as-promised` library (see §7.3.2).

doi:10.6342/NTU202210012

7.4.1 Characteristics

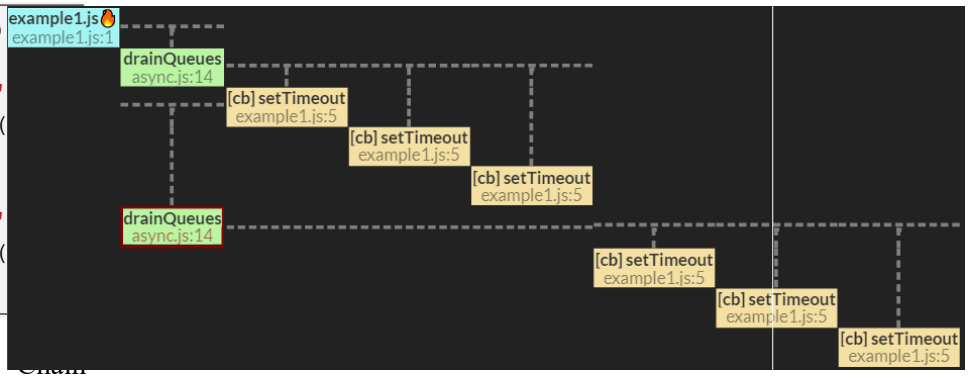
We briefly use Tbl. 7.1 to understand the characteristics of execution behavior of the selected samples:

`webpack`, `hexo` and `sequelize` are the three "biggest" samples (i.e. of the given samples, these had the most recorded events).

We also note that async functions (AWAITs) are rarely seen. Despite its semantically more attractive counterparts already being part of the ECMAScript standard for years, asynchronous callbacks are still commonplace. In fact, callbacks are the primary asynchronous scheduling mechanism in nine out of the eleven projects, while only sequelize predominantly uses async functions.

7.4.2 Bluebird: Testing ACG Robustness

```
Promise.resolve()
  .then(() => {
    console.log('
    return sleep(
  })
  .then(() => {
    console.log('
    return sleep(
  });
```



(a) Code of Promise Chain (executed three times)

(b) Result ACG of Bluebird Sample (`detail` on)

Figure 7.11: Bluebird Code and Result ACG

We believe that much of the utility of developer tools comes from their reliability and robustness. To that end we challenged the ACG to cope when confronted with a different promise implementation, despite being fine-tuned for Promise/A+ compliance. To that end, we tested the ACG on Bluebird, (despite its authors, these days, discouraging its use).

Bluebird also adheres to the Promise/A+ standard. However, if we treat it as a normal library and not as "platform code", the A+ requirement [2.2.4] (a requirement that ACG depends on for accuracy) is violated.

doi:10.6342/NTU202210012

In our test, we produced three promise chains in parallel (see Listing 7.11a). They all have two CGRs, each returning a new promise sleeping for 100ms (using Bluebird's Promise constructor). The expected ideal outcome would be that the ACG can capture the promise chains, and thus renders each of the three promises on their own thread. However, the result shows that all promise CGRs are FORKed off Bluebird's singular driver function thread instead.

The first thread following the entry point captures the two invocations of the Bluebird's queue driver function `drainQueues`. The fact that they are rendered on the same thread is actually caused by a small visual bug. In fact, the second call to `drainQueues` was forked from the first `drainQueue` CGR's child (in column 3). Each of the two nodes forks three CB-type (which should be THEN) edges to the timer activation CGR. Moreover, the ACS from inside the second `drainQueues` CGR is inaccurate. It's scheduler CGR has been omitted from the stack. We conclude that the ACG can deal with other promise specs, but accuracy suffers and non-critical bugs occur.

Furthermore, we can selectively disable tracing of Bluebird's (or any library's) implementation, which is equivalent to treating it as platform code. We did so for hexo (which uses Bluebird). In this case, the basic promise semantics are captured correctly. We note that even in this case, non-standard promise methods (such as `map` and `each`) are still not chained, and treated as CB-type AEs instead.

7.5 Summary

We find that the ACG and its extended feature set has proven an invaluable addition to debugging and understanding inherently asynchronous codebases. More specifically, it supported the debugging process in the following ways:

1. The ACG generally aids asynchronous control flow comprehension.
2. The ACG makes some order-violation bugs visually obvious.
3. The ACG visualizes and visually contextualizes errors, even if they were not reported.

doi:10.6342/NTU202210012

4. The ACG aids orientation while striding through unknown code.
5. The ACG visualizations aid recall, allowing to more quickly remember relevant code or context beyond traditional methods.
6. The ACS (and the also synchronous call graph) prove useful to understand the network of control flow “hidden beneath the surface”.
7. The extended ACG feature set uncovers and allows traversing all types of asynchronous links (see §6.6.2).

On the flip side, and unsurprisingly so, we discovered that navigating and reading a complex ACG requires more skill than a smaller one. However, we conjecture that such skill can be practiced effectively, and we also conjecture that it not only makes it easier to use the tool, but can also enhance the developer’s ability to think about and comprehend more complex asynchronous control flow of their own or even unknown codebases. User studies are still going to be necessary to more properly evaluate real-world utility of the ACG, and Dbux in general.

doi:10.6342/NTU202210012

doi:10.6342/NTU202210012

Chapter 8

Conclusion

8.1 Summary of Work Completed

This work is an attempt at discussing, and making better sense of the essence of interactive debugging, one of the most difficult and costly endeavors that software engineers engage in on a daily basis. To that end, we propose a descriptive model of the debugging process based on knowledge sources, locations, facts, fact tangibility, location links, falsehoods, actions. We propose the debugging loop as the all-encompassing decision-making process that drives it all. We (i) define the goal path as a unique type of artifact that is the result of solving a debugging task, (ii) explain debugging difficulty in terms of goal path and fact tangibility, and (iii) explain the role of any type of interactive debugger, not just the traditional debugger, in increasing said tangibility. That model was further used to better inform debugging case study methodology in form of "Debugging Journeys".

In this work we presented Dbux, a first multi-resolutional code-level Integrated Debugging Environment (IDbE). Thanks to Dbux-Projects, in just a few steps, we were able to easily reproduce all previously prepared bugs. We illustrated some of Dbux's vast feature set, and how much of an impact the difference in tangibility of one type of fact can have on the overall debugging process.

We presented a first version of Dbux-PDG, a system that aims to help learners comprehend and interact with the data flow of DSA implementations. Integration with a popular IDE and its source editing tools allows the user to locate the code responsible for data flow

doi:10.6342/NTU202210012

operations. Important algorithmic properties, such as sorting stability, parallelization potential or changes in data structures over time are commonly part of computer science classes, but generally intangible, and thus difficult to understand by the learner. Dbux-PDG allows interactively exploring these properties. In some cases, the user is enabled to further explore the reason behind some of these properties, which can make things more fun and increase learner engagement.

Moreover, we presented a new method to automatically find an asynchronous ordering relationship between CGRs in form of CHAINs and FORKs, leading to automatic detection of "virtual threads". We used these concepts to construct the ACG, an interactive high-level debugging and program comprehension model and tool. An extended ACG feature set to help uncover more types of asynchrony-induced causal links has also been proposed and implemented. Several inter-thread synchronization mechanisms and their effect on the ACG have been illustrated. A novel mini-extension showcases how simple it is to combine APIs of PDG and ACG to uncover even more high-level facts, specifically inter-thread data dependencies.

We find that the ACG and its extended feature set has proven an invaluable addition to debugging and understanding inherently asynchronous codebases.

8.2 Contributions

We conclude by once more summarizing the major contributions of this dissertation. This research has also resulted in two publications, which are marked in this list.

- A descriptive model of the interactive debugging process §3.2.
- Definition of "Debugging Journey" case study methodology §3.4.
- A first omniscient debugger for JavaScript (Dbux) §4.
- **(Published [107])** A Program Dependency Graph to reveal hidden properties of data structures and algorithms (Dbux-PDG) §5.
- **(Partially Published [106])** An Asynchronous Control Graph for revealing concurrency in JavaScript (Dbux-ACG) §6.

doi:10.6342/NTU202210012

8.3 Limitations & Future Work

In this final section, we discuss the limitations of this work as well as some of the (what we believe to be) important issues in the interactive debugging field.

Omniscient debuggers (ODs), such as Dbux, are complex beasts powered by huge datasets. This generally makes ODs a great solution for applications that generate small-to-medium-sized tracelogs. Big codebases, as well as real-time or high-performance applications, such as games or scientific applications with large inputs will bring them to their knees. While many **optimizations** have been proposed, they also often end up reducing the debugger's scope, and with it its ability to help answer global queries pertaining to arbitrary code locations and their executions. Building omniscient debuggers on top of replay debuggers (as proposed by Pothier et al. [96]) currently seems like one of the more promising avenues forward. To avoid having to collect data from all parts of the application, **out-of-core and partial methods** are going to be crucial so as to answer global queries and global visualizations in the fact of incomplete data.

In order for Dbux (and ODs in general) to become more widely useful, several other limitations need to be overcome. **Edit-and-continue** should allow making minor edits without the need to re-collect all data. In order to support **more systems/languages/IDEs**, Dbux needs to be more modular. For example, all data analysis queries are currently in a single file. Each query or query family must become its own module since many queries need to support variability (some sort of plug-n-play system) for different target systems or languages. Despite many novel debuggers aiming to make even **more types of facts tangible**, we assume that the quest to make all facts hidden in debugging's dark matter is never-ending.

While Dbux has been built with the goal of allowing any developer increase tangibility of many important types of facts, it's UI and UX design leaves some to be desired. Not only novices and intermediate learners, but even expert developers will likely require **proper onboarding and even tutorials/practice scaffolds** in order to make sufficient use of Dbux's wide feature set. That is due to the fact that today's developers generally do not learn about many important program analysis concepts (such as control flow, data flow,

doi:10.6342/NTU202210012

call graph concepts etc.) that lie at the core of this (and many other) novel debugging tools.

One of the most notable omissions of this work are **user studies**. That is due to our focus on debugging facts and *fact tangibility*[§3.2.7]. Our debugging journey case studies have been designed to allow an in-depth discussion of the steps and facts involved in an expert debugging solution of the given bugs using the tools proposed in this work. The debugging journey generally assumes that the investigator (1) has intricate knowledge of the tool's capabilities, and (2) is an expert in relevant program analysis and debugging skills (see §3.4.1). If this assumption holds, any solution found by participants of a user study would be unlikely to outperform the case study's solution, in terms of length and tangibility of the shortest goal path. That gives our case studies an additional use: they produce a "high quality" baseline solution, that can serve to anchor follow-up user studies.

We go so far as to propose that **more diverse types of debugging user studies** would benefit the field. For example, we want to not only measure user debugging outcomes, such as accuracy and speed, but also analyze the processes themselves, in terms of the goal paths and the crucial links involved. Such approach could allow for a more in-depth understanding of the individual steps and the intricacies of the interactive debugging process in much greater detail than previous studies.

Furthermore, we strongly believe that **debugging skill training** is an important next step to be taken. We envision a future where students can reliably practice and train their ability to uncover even the least tangible facts accurately and efficiently, if not downright surgically. Students of the future, if they so desire, should have access to the tools and be able to practice skills to allow them to find even the nastiest bugs in only a fraction of a time that it would take today's experts. We conjecture that a scaffolding for said training should at least have the following three ingredients:

- Training of *program analysis skills* to help students make better sense of their dark matter.
- Training of high-level reasoning, planning and problem-solving strategies, on scenarios of varying uncertainty, ranging from "everything is obvious" to "alone in the

doi:10.6342/NTU202210012

dark”.

- All of that should be supported by an immense corpus of program comprehension and debugging tasks inside real-world codebases, ranging in difficulty levels from “trivial” to “brutal”, made available to the learner at a single click, as we already have started maintaining in *Dbux-Projects*[§4.6].

Lieberman’s excerpt on the “debugging scandal” [76] called us to action. For 25 years, we have not done a good enough job heeding the call. I, for one, am hopeful that things are going to change. Maybe, it will not be another 25 years until interactive debugging is sufficiently supported by myriads of powerful, yet easy-to-use debuggers. I consider debugging a crucial pillar of software engineering that informs and interacts with all the other aspects that engineers engage in when building and safeguarding complex systems. I thus am looking forward to a future where debugging is not discarded as an annoying chore standing in the way of other software engineering endeavors, but rather celebrated as a craft in its own right.

doi:10.6342/NTU202210012

doi:10.6342/NTU202210012

Bibliography

[1] *Dbux —VSCode Extension*, (accessed in 10/2022). URL: https://marketplace.visualstudio.com/items?itemName=Domi.dbux-code.

[2] *Angular Augury*, (accessed in 11/2020). URL: https://augury.rangle.io/.

[3] *Chrome Dev Tools*, (accessed in 11/2020). URL: https://developers.google.com/web/tools/chrome-devtools.

[4] *ECMAScript 2015 Language Specification*, (accessed in 11/2020). URL: https://www.ecma-international.org/ecma-262/6.0.

[5] *ECMAScript 2017 Language Specification*, (accessed in 11/2020). URL: https://www.ecma-international.org/ecma-262/8.0.

[6] *NetBeans Visual Debugger*, (accessed in 11/2020). URL: https://netbeans.org/kb/docs/java/debug-visual.html.

[7] *React Dev Tools*, (accessed in 11/2020). URL: https://chrome.google.com/webstore/detail/react-developer-tools/fmkadmapgofadopljbjfkapdkoienihi?hl=en.

[8] *TensorBoard: TensorFlow's visualization toolkit*, (accessed in 11/2020). URL: https://www.tensorflow.org/tensorboard.

doi:10.6342/NTU202210012

[9] *Vue Dev Tools*, (accessed in 11/2020). URL: https://chrome.google.com/webstore/detail/vuejs-devtools/nhdogjmejiglipccpnnnanhbledajbpd?hl=en.

[10] *Matlab*, (accessed in 1/2021). URL: https://www.mathworks.com/products/matlab.html.

[11] *Redux Dev Tools*, (accessed in 1/2021). URL: https://github.com/reduxjs/redux-devtools.

[12] *Valgrind Memcheck*, (accessed in 1/2021). URL: https://www.valgrind.org/docs/manual/mc-manual.html.

[13] *Babel*, (accessed in 2/2022). URL: https://babeljs.io/docs/en/index.html.

[14] *Promises/A+*, (accessed in 2/2022). URL: https://promisesaplus.com/.

[15] E. E. Aftandilian, S. Kelley, C. Gramazio, N. Ricci, S. L. Su, and S. Z. Guyer. Heapviz, 2010. doi:10.1145/1879211.1879222.

[16] A. Afzal and C. Le Goues. A study on the use of ide features for debugging. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pages 114–117. IEEE, 2018.

[17] D. J. Agans. *Debugging: The 9 indispensable rules for finding even the most elusive software and hardware problems*. Amacom, 2002.

[18] S. Alimadadi, A. Mesbah, and K. Pattabiraman. Understanding asynchronous interactions in full-stack javascript, May 2016. doi:10.1145/2884781.2884864.

[19] S. Alimadadi, D. Zhong, M. Madsen, and F. Tip. Finding broken promises in asynchronous javascript programs. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–26, Oct 2018. doi:10.1145/3276532.

doi:10.6342/NTU202210012

[20] F. E. Allen. Control flow analysis, Jul 1970. doi:10.1145/390013.808479.

[21] M. V. Almeda, E. Kleinman, C. Jemmali, C. Ithier, E. Rowe, and M. Seif El-Nasr. Labeling debugging in may' s journey gameplay. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*. ACM, Feb 2020. doi:10.1145/3328778.3372624.

[22] K. Araki, Z. Furukawa, and J. Cheng. A general framework for debugging. *IEEE Software*, 8(3):14–20, May 1991. doi:10.1109/52.88939.

[23] K. Arulkumaran, A. Cully, and J. Togelius. Alphastar, Jul 2019. doi:10.1145/3319619.3321894.

[24] R. M. Balzer. Exdams. In *Proceedings of the May 14-16, 1969, spring joint computer conference on XX - AFIPS '69 (Spring)*. ACM Press, 1969. doi:10.1145/1476793.1476881.

[25] H. Banken, E. Meijer, and G. Gousios. Debugging data flows in reactive programs. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 752–763. IEEE, 2018.

[26] F. Beck, M. Burch, S. Diehl, and D. Weiskopf. A taxonomy and survey of dynamic graph visualization. *Computer Graphics Forum*, 36(1):133–159, Jan 2016. doi:10.1111/cgf.12791.

[27] M. Beller, N. Spruit, D. Spinellis, and A. Zaidman. On the dichotomy of debugging behavior among programmers. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, May 2018. doi:10.1145/3180155.3180175.

[28] C. Bennett, J. Ryall, L. Spalteholz, and A. Gooch. The aesthetics of graph visualization. In *CAe*, pages 57–64, 2007.

doi:10.6342/NTU202210012

[29] L. Bidlake, E. Aubanel, and D. Voyer. Systematic literature review of empirical studies on mental representations of programs. *Journal of Systems and Software*, 165:110565, Jul 2020. doi:10.1016/j.jss.2020.110565.

[30] R. Brooks. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18(6):543 – 554, Jun 1983. doi:10.1016/s0020-7373(83)80031-5.

[31] M. Böhme, E. O. Soremekun, S. Chattopadhyay, E. Ugherughe, and A. Zeller. Where is the bug and how is it fixed? an experiment with practitioners. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2017*. ACM Press, 2017. doi:10.1145/3106237.3106255.

[32] X. Chang, W. Dou, Y. Gao, J. Wang, J. Wei, and T. Huang. Detecting atomicity violations for event-driven node. js applications. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 631–642. IEEE, 2019.

[33] F. Cornelis, A. Georges, M. Christiaens, M. Ronsse, T. Ghesquiere, and K. Bosschere. A taxonomy of execution replay systems. In *Proceedings of International Conference on Advances in Infrastructure for Electronic Business, Education, Science, Medicine, and Mobile Technologies on the Internet*, 2003.

[34] B. Cornelissen, A. Zaidman, D. Holten, L. Moonen, A. van Deursen, and J. J. van Wijk. Execution trace analysis through massive sequence and circular bundle views. *Journal of Systems and Software*, 81(12):2252 – 2268, Dec 2008. doi:10.1016/j.jss.2008.02.068.

[35] S. P. Davies. Models and theories of programming strategy, Aug 1993. doi:10.1006/imms.1993.1061.

[36] J. Davis, A. Thekumparampil, and D. Lee. Node. fz: Fuzzing the server-side event-driven architecture. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 145–160, 2017.

doi:10.6342/NTU202210012

[37] F. Eichinger, V. Pankratius, P. W. Große, and K. Böhm. Localizing defects in multithreaded programs by mining dynamic call graphs. In *International Academic and Industrial Conference on Practice and Research Techniques*, pages 56–71. Springer, 2010.

[38] M. Eisenstadt. My hairiest bug war stories. *Communications of the ACM*, 40(4):30–37, Apr 1997. doi:10.1145/248448.248456.

[39] J. Ellson, E. Gansner, L. Koutsofios, S. C. North, and G. Woodhull. Graphviz—open source graph drawing tools, 2002. doi:10.1007/3-540-45848-4_57.

[40] A. T. Endo and A. Møller. Noderacer: Event race detection for node. js applications. 2020.

[41] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization, 1984. doi:10.1007/3-540-12925-1_33.

[42] B. Flyvbjerg. Five misunderstandings about case-study research. *Qualitative inquiry*, 12(2):219–245, 2006.

[43] T. Fritz and G. C. Murphy. Using information fragments to answer the questions developers ask. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE '10*. ACM Press, 2010. doi:10.1145/1806799.1806828.

[44] E. R. Gansner, E. Koutsofios, S. C. North, and K.-P. Vo. A technique for drawing directed graphs. *IEEE Transactions on Software Engineering*, 19(3):214–230, 1993.

[45] D. J. Gilmore. Models of debugging. *Acta Psychologica*, 78(1-3):151-172, Dec 1991. doi:10.1016/0001-6918(91)90009-o.

[46] S. L. Graham, P. B. Kessler, and M. K. McKusick. Gprof: A call graph execution profiler. *ACM Sigplan Notices*, 17(6):120–126, 1982.

doi:10.6342/NTU202210012

[47] S. Grissom, M. F. McNally, and T. Naps. Algorithm visualization in cs education, 2003. doi:10.1145/774833.774846.

[48] P. J. Guo. Online python tutor. In *Proceeding of the 44th ACM technical symposium on Computer science education - SIGCSE '13*. ACM Press, 2013. doi:10.1145/2445196.2445368.

[49] P. Gyimesi, B. Vancsics, A. Stocco, D. Mazinanian, A. Beszedes, R. Ferenc, and A. Mesbah. Bugsjs: a benchmark of javascript bugs. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. IEEE, Apr 2019. doi:10.1109/icst.2019.00019.

[50] S. Halim. Visualgo–visualising data structures and algorithms through animation. *Olympiads in informatics*, 9:243–245, 2015.

[51] C. Hughes, J. Buckley, C. Exton, D. O' Carroll, and S. Group). Towards a framework for characterising concurrent comprehension, Mar 2005. doi:10.1080/08993400500056522.

[52] H. Kang and P. J. Guo. Omnicode. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*. ACM, Oct 2017. doi:10.1145/3126594.3126632.

[53] V. Karavirta and C. A. Shaffer. Jsav: the javascript algorithm visualization library, 2013. doi:10.1145/2462476.2462487.

[54] I. R. Katz and J. R. Anderson. Debugging: An analysis of bug-location strategies. *Human-Computer Interaction*, 3(4):351–399, 1987.

[55] D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, May 2013. doi:10.1109/icse.2013.6606626.

[56] A. J. Ko. *Asking and answering questions about the causes of software behavior*. PhD thesis, Carnegie Mellon University, 2008.

doi:10.6342/NTU202210012

[57] A. J. Ko and B. A. Myers. Debugging reinvented. In *Proceedings of the 13th international conference on Software engineering - ICSE '08*. ACM Press, 2008. doi:10.1145/1368088.1368130.

[58] A. J. Ko and B. A. Myers. Finding causes of program output with the java whyline. In *Proceedings of the 27th international conference on Human factors in computing systems - CHI 09*. ACM Press, 2009. doi:10.1145/1518701.1518942.

[59] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks, Dec 2006. doi:10.1109/tse.2006.116.

[60] P. S. Kochhar, X. Xia, D. Lo, and S. Li. Practitioners' expectations on automated fault localization. In *Proceedings of the 25th International Symposium on Software Testing and Analysis - ISSTA 2016*. ACM Press, 2016. doi:10.1145/2931037.2931051.

[61] T. D. LaToza, D. Garlan, J. D. Herbsleb, and B. A. Myers. Program comprehension as fact finding. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering - ESEC-FSE '07*. ACM Press, 2007. doi:10.1145/1287624.1287675.

[62] T. D. LaToza and B. A. Myers. Developers ask reachability questions. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE '10*. ACM Press, 2010. doi:10.1145/1806799.1806829.

[63] T. D. LaToza and B. A. Myers. Hard-to-answer questions about code. In *Evaluation and Usability of Programming Languages and Tools on - PLATEAU '10*. ACM Press, 2010. doi:10.1145/1937117.1937125.

[64] T. D. LaToza and B. A. Myers. Visualizing call graphs. In *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, Sep 2011. doi:10.1109/vlhcc.2011.6070388.

doi:10.6342/NTU202210012

[65] T. D. LaToza, G. Venolia, and R. DeLine. Maintaining mental models, May 2006. doi:10.1145/1134285.1134355.

[66] J. Lawrance, C. Bogart, M. Burnett, R. Bellamy, K. Rector, and S. D. Fleming. How programmers debug, revisited: An information foraging theory perspective. *IEEE Transactions on Software Engineering*, 39(2):197–215, Feb 2013. doi:10.1109/tse.2010.111.

[67] L. Layman, M. Diep, M. Nagappan, J. Singer, R. Deline, and G. Venolia. Debugging revisited: Toward understanding the debugging needs of contemporary software developers. In *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*. IEEE, Oct 2013. doi:10.1109/esem.2013.43.

[68] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, Jun 2012. doi:10.1109/icse.2012.6227211.

[69] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. Genprog: A generic method for automatic software repair. *Ieee transactions on software engineering*, 38(1):54–72, 2011.

[70] D. Lehmann and M. Pradel. Feedback-directed differential testing of interactive debuggers, Oct 2018. doi:10.1145/3236024.3236037.

[71] R. Lencevicius, U. Hölzle, and A. K. Singh. Query-based debugging of object-oriented programs. *ACM SIGPLAN Notices*, 32(10):304–317, Oct 1997. doi:10.1145/263700.263752.

[72] S. Letovsky. Cognitive processes in program comprehension, Dec 1987. doi:10.1016/0164-1212(87)90032-x.

[73] B. Lewis. Debugging backwards in time. *arXiv preprint cs/0310016*, 2003.

doi:10.6342/NTU202210012

[74] X. Li, S. Zhu, M. d' Amorim, and A. Orso. Enlightened debugging, May 2018. doi:10.1145/3180155.3180242.

[75] T. Lieber, J. R. Brandt, and R. C. Miller. Addressing misconceptions about code with always-on programming visualizations. In *Proceedings of the 32nd annual ACM conference on Human factors in computing systems - CHI '14*. ACM Press, 2014. doi:10.1145/2556288.2557409.

[76] H. Lieberman. The debugging scandal and what to do about it. *Communications of the ACM*, 40(4):26–30, 1997.

[77] H. Lieberman and C. Fry. Zstep 95: A reversible, animated source code stepper. *Software Visualization: Programming as a Multimedia Experience*, pages 277–292, 1997.

[78] Y. Lin, J. Sun, Y. Xue, Y. Liu, and J. Dong. Feedback-based debugging. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, May 2017. doi:10.1109/icse.2017.43.

[79] M. C. Loring, M. Marron, and D. Leijen. Semantics of asynchronous javascript. In *Proceedings of the 13th ACM SIGPLAN International Symposium on on Dynamic Languages - DLS 2017*. ACM Press, 2017. doi:10.1145/3133841.3133846.

[80] A. Luxton-Reilly, E. McMillan, E. Stevenson, E. Tempero, and P. Denny. Ladebug: an online tool to help novice programmers improve their debugging skills. In *Proceedings of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education - ITiCSE 2018*. ACM Press, 2018. doi:10.1145/3197091.3197098.

[81] W. Maalej, R. Tiarks, T. Roehm, and R. Koschke. On the comprehension of program comprehension. *ACM Transactions on Software Engineering and Methodology*, 23(4):1–37, Sep 2014. doi:10.1145/2622669.

doi:10.6342/NTU202210012

[82] M. Madsen, O. Lhoták, and F. Tip. A model for reasoning about javascript promises. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–24, Oct 2017. doi:10.1145/3133910.

[83] K. Mangaroska, K. Sharma, M. Giannakos, H. Træteberga, and P. Dillenbourg. Gaze-driven design insights to amplify debugging skills: A learner-centred analysis approach. *Journal of Learning Analytics*, 5(3):98–119, 2018.

[84] W. Mayer and M. Stumptner. Evaluating models for model-based debugging. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. IEEE, Sep 2008. doi:10.1109/ase.2008.23.

[85] T. Michaeli and R. Romeike. Current status and perspectives of debugging in the k12 classroom: A qualitative study. In *2019 IEEE Global Engineering Education Conference (EDUCON)*. IEEE, Apr 2019. doi:10.1109/educon.2019.8725282.

[86] T. Michaeli and R. Romeike. Improving debugging skills in the classroom. In *Proceedings of the 14th Workshop in Primary and Secondary Computing Education on - WiPSCE' 19*. ACM Press, 2019. doi:10.1145/3361721.3361724.

[87] T. Naps. JhavÉ: Supporting algorithm visualization, Sep 2005. doi:10.1109/mcg.2005.110.

[88] R. Netravali and J. Mickens. Reverb. In *Proceedings of the ACM Symposium on Cloud Computing*. ACM, Nov 2019. doi:10.1145/3357223.3362733.

[89] S. Ontanon, G. Synnaeve, A. Uriarte, F. Richoux, D. Churchill, and M. Preuss. A survey of real-time strategy game ai research and competition in starcraft, Dec 2013. doi:10.1109/tciaig.2013.2286295.

[90] D. H. O' Dell. The debugging mind-set. *Communications of the ACM*, 60(6):40–45, May 2017. doi:10.1145/3052939.

doi:10.6342/NTU202210012

[91] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller. Evaluating and improving fault localization. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, May 2017. doi:10.1109/icse.2017.62.

[92] M. Perscheid, B. Siegmund, M. Taeumel, and R. Hirschfeld. Studying the advancement in debugging practice of professional software developers, Jan 2016. doi:10.1007/s11219-015-9294-2.

[93] F. Petrillo, Z. Soh, F. Khomh, M. Pimenta, C. Freitas, and Y.-G. Gueheneuc. Towards understanding interactive debugging. In *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, Aug 2016. doi:10.1109/qrs.2016.27.

[94] B. Pfretzschner and L. ben Othmane. Identification of dependency-based attacks on node.js, Aug 2017. doi:10.1145/3098954.3120928.

[95] G. Pothier and □. Tanter. Back to the future: Omniscient debugging, Nov 2009. doi:10.1109/ms.2009.169.

[96] G. Pothier and □. Tanter. Summarized trace indexing and querying for scalable back-in-time debugging, 2011. doi:10.1007/978-3-642-22655-7_26.

[97] G. Pothier, □. Tanter, and J. Piquer. Scalable omniscient debugging. *ACM SIGPLAN Notices*, 42(10):535 – 552, Oct 2007. doi:10.1145/1297105.1297067.

[98] S. Proksch, S. Amann, and S. Nadi. Enriched event streams. In *Proceedings of the 15th International Conference on Mining Software Repositories - MSR '18*. ACM Press, 2018. doi:10.1145/3196398.3196400.

[99] A. Rahman. Comprehension effort and programming activities, May 2018. doi:10.1145/3196398.3196470.

doi:10.6342/NTU202210012

[100] A. Ram. A theory of questions and question asking, Jul 1991. doi:10.1080/10508406.1991.9671973.

[101] T. Roehm, R. Tiarks, R. Koschke, and W. Maalej. How do professional developers comprehend software? In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, Jun 2012. doi:10.1109/icse.2012.6227188.

[102] P. Runeson and M. Höst. Guidelines for conducting and reporting case study research in software engineering, Dec 2008. doi:10.1007/s10664-008-9102-8.

[103] G. Salvaneschi and M. Mezini. Debugging for reactive programming, May 2016. doi:10.1145/2884781.2884815.

[104] D. Seifert and M. Wan. Dbux, (accessed in 10/2022). URL: https://github.com/Domiii/dbux.

[105] D. Seifert and M. Wan. Dbux-pdg gallery, (accessed in 6/2022). URL: https://real-world-debugging.github.io/dbux-pdg/gallery.html.

[106] D. Seifert, M. Wan, J. Hsu, and B. Yeh. An asynchronous call graph for javascript. In *2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 29–30. IEEE, 2022.

[107] D. Seifert, M. Wan, J. Hsu, and B. Yeh. Dbux-pdg: An interactive program dependency graph for data structures and algorithms. In *2022 Working Conference on Software Visualization (VISSOFT)*, pages 141–151. IEEE, 2022.

[108] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs. Jalangi: a selective record-replay and dynamic analysis framework for javascript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2013*. ACM Press, 2013. doi:10.1145/2491411.2491447.

doi:10.6342/NTU202210012

[109] C. A. Shaffer, M. L. Cooper, A. J. D. Alon, M. Akbar, M. Stewart, S. Ponce, and S. H. Edwards. Algorithm visualization, Aug 2010. doi:10.1145/1821996.1821997.

[110] E. Y. Shapiro. *Algorithmic program debugging*. Yale University, 1982.

[111] M. Shaw, Oct 2002. doi:10.1007/s10009-002-0083-4.

[112] G. Shu, B. Sun, T. A. Henderson, and A. Podgurski. Javapdg: A new platform for program dependence analysis, Mar 2013. doi:10.1109/icst.2013.57.

[113] J. Sillito, G. Murphy, and K. De Volder. Asking and answering questions during a programming change task. *IEEE Transactions on Software Engineering*, 34(4):434–451, Jul 2008. doi:10.1109/tse.2008.26.

[114] J. Sillito, G. C. Murphy, and K. De Volder. Questions programmers ask during software evolution tasks. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering - SIGSOFT '06/FSE-14*. ACM Press, 2006. doi:10.1145/1181775.1181779.

[115] J. Silva. A survey on algorithmic debugging strategies. *Advances in Engineering Software*, 42(11):976 – 991, Nov 2011. doi:10.1016/j.advengsoft.2011.05.024.

[116] R. G. Simmons. A theory of debugging plans and interpretations. In *AAAI*, pages 94–99, 1988.

[117] P. Slezák and I. Waczulíková. Reproducibility and repeatability. *Physiol Res*, 60:203–205, 2011.

[118] T. Sotiropoulos and B. Livshits. Static analysis for asynchronous javascript programs. 2019. URL: https://arxiv.org/abs/1901.03575, doi:10.48550/ARXIV.1901.03575.

[119] Stack Overflow. *Stack Overflow Developer Survey 2021*, (accessed in 2/2022). URL: https://insights.stackoverflow.com/survey/2021.

doi:10.6342/NTU202210012

[120] H. Sun, D. Bonetta, F. Schiavio, and W. Binder. Reasoning about the node.js event loop using async graphs, Feb 2019. doi:10.1109/cgo.2019.8661173.

[121] R. Suzuki, G. Soares, A. Head, E. Glassman, R. Reis, M. Mongiovi, L. D' Antoni, and B. Hartmann. Tracediff: Debugging unexpected code behavior using trace divergences. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, Oct 2017. doi:10.1109/vlhcc.2017.8103457.

[122] R. Tiarks and T. Roehm. Challenges in program comprehension. *Softwaretechnik-Trends*, 32(2):19–20, 2012.

[123] K. F. Tomasdottir, M. Aniche, and A. van Deursen. Why and how javascript developers use linters. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, Oct 2017. doi:10.1109/ase.2017.8115668.

[124] K. F. Tomasdottir, M. Aniche, and A. van Deursen. The adoption of javascript linters in practice: A case study on eslint. *IEEE Transactions on Software Engineering*, 46(8):863–891, Aug 2020. doi:10.1109/tse.2018.2871058.

[125] E. Tominaga, Y. Arahori, and K. Gondow. Awaitviz. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*. ACM, Apr 2019. doi:10.1145/3297280.3297528.

[126] Trekhleb, Oleksii. Javascript algorithms, (accessed in 6/2022). URL: https://github.com/trekhleb/javascript-algorithms.

[127] I. Vessey. Expertise in debugging computer programs: A process analysis. *International Journal of Man-Machine Studies*, 23(5):459 – 494, Nov 1985. doi:10.1016/s0020-7373(85)80054-7.

[128] J. Vilk. Righting web development. 2018.

doi:10.6342/NTU202210012

[129] J. Vilk and E. D. Berger. Bleak: automatically debugging memory leaks in web applications, Dec 2018. doi:10.1145/3296979.3192376.

[130] J. Wang, W. Dou, Y. Gao, C. Gao, F. Qin, K. Yin, and J. Wei. A comprehensive study on real world concurrency bugs in node.js, Oct 2017. doi:10.1109/ase.2017.8115663.

[131] M. Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–452, Jul 1982. doi:10.1145/358557.358577.

[132] A. Whitaker, R. S. Cox, S. D. Gribble, et al. Configuration debugging as search: Finding the needle in the haystack. In *OSDI*, volume 4, 2004.

[133] C. Wohlin and A. Rainer. Is it a case study?—a critical analysis and guidance, Oct 2022. doi:10.1016/j.jss.2022.111395.

[134] T. Zimmermann and A. Zeller. Visualizing memory graphs, 2002. doi:10.1007/3-540-45875-1_15.

[135] D. Zou, J. Liang, Y. Xiong, M. D. Ernst, and L. Zhang. An empirical study of fault localization families and their combinations. *IEEE Transactions on Software Engineering*, 2019. doi:10.1109/tse.2019.2892102.

doi:10.6342/NTU202210012